main.py

CODE WITH REPL.IT

FROM BEGINNER TO EXPERT THROUGH GUIDED TUTORIALS

repl.it

CONTENT BY RITZA

Programming walkthroughs: Coding with Python and Repl.it

Ritza

Contents

Welcome to Code with Repl.it

In a series of tutorials, you'll go from beginner to expert in coding with Repl.it. While these lessons are designed to be taken in order, they each make sense on their own too, so feel free to jump in wherever looks the most interesting to you.

Part 1 covers the basics of using Repl.it: how to create projects, work with files, use third party dependencies, do plotting and graphing, and use multiplayer to code as part of a team.

Part 2 covers more advanced Repl.it use. You'll see how to pull projects from GitHub and collaborate on open source software, build a game, and keep your code secure. You'll build a full web application using Test Driven Development (TDD), and find out how to be an elite hacker by using the shortcuts offered by Repl.it

Once you've completed Part 1 and Part 2, you'll be able to build nearly any project that you want, and deploy it for the world to use. If you're stuck for ideas, you can go through the examples given in Part 3, which consists of practical tutorials to build everything from web scrapers to chat bots.

Note that this set of lessons does not focus on teaching you to code, though we will explain some key concepts along the way. If you don't already know how to code, it's best to take this course in conjunction with a more traditional course. If you're not sure what to do next, jump right in and see if you can keep up. We're beginner friendly.

Part 1: The basics of Repl.it

In this section of the course, you'll learn the basic of Repl.it. But that doesn't mean you won't build some fun stuff along the way.

Tutorial 1: Introduction to Repl.it and using the IDE

Learn the basics of the Repl.it IDE. Why use an online IDE and what are all those different panes? Build a simple program to solve your maths homework.

Tutorial 2: Working with files using Repl.it

Computers were initially created to read and write files, and although we've come a long way files remain central to everything we do. Learn how to create them, read from them, write to them, and import and export them in bulk.

Tutorial 3: Managing dependencies with Repl.it

No one is an island, and if you build software you'll build it on top of existing modules that others have written. Here we show you how to work with other people's code in a variety of ways: in many cases all you need to do is import antigravity and fly away[1].

Tutorial 4: Data science: plotting and graphing

Data is only useful if it can be easily understood. Plots, charts, and graphs are the easiest way to know what's happening in the world around you. And did you know that data science is the sexiest job of the 21st century[2]. Follow along to plot every city in the USA and find out if richer people live longer.

Tutorial 5: Pair programming and using multiplayer

Did we mention that no one is an island? Coders don't have to work alone. You can invite your friends to code along with you, a technique used by beginners and experts alike. Learn how to code collaboratively, as if you were using a Google Doc.

Part 2: Advanced Repl.it use

Once you know the basics, it's time to build larger and more complicated projects and keep them secure.

Tutorial 6: Running projects from GitHub

Most open source software lives on GitHub and it's easy to take advantage of all of this free software by pulling code from GitHub to Repl.it and running it with one click. Some software needs to be configured in specific ways so you'll also learn how to modify what happens when you press that big green "run" button.

Tutorial 7: Building a game with PyGame

Do you want to develop games? Of course, you can do that with Repl.it to. We'll build a 2D juggling game using PyGame in this lesson and you'll learn more about graphics programming at the same time: sprites, physics, and more.

Tutorial 8: Can you keep a secret? What about from time travellers?

Have you been hacked? It's only a matter of time if you haven't. Learn how to keep your secrets safe, even when coding in public spaces. Pro tip: if you accidentally paste a password into your code and then remove it, others might still find it in your history, so you'll learn how to navigate that too.

[1]https://xkcd.com/353/
[2]https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

Tutorial 9: Introduction to TDD using PyTest

By this stage you'll have made a few mistakes. Learn the TDD way and how to write code that tests your other code to catch frustrating errors before they can hurt anyone. Repetitive jobs is what computers are best at after all.

Tutorial 10: Become an elite hacker with productivity hacks

Have you seen the Matrix? Learn to be the Neo of coding by getting more than one cursor, using keyboard shortcuts, and all of the other productivity features that Repl.it offers. You'll be soon producing more code in less time.

Tutorial 11: Keeping your data in check with the Repl.it database

Now that you are starting to build larger and more complicated applications, it is time to start using databases to keep your data clean and secure.

Tutorial 12: Repl audio - control (or create) your music with code

Find, download, play, and control the volume of your music, all in code. If that's not enough, create your own music too.

This is the part where you realize that the possibilities are endless while you learn how to control your music with code.

Part 3: Building your own projects

Once you've gone through everything, you might think "but what should I build?". It's a common problem and we've got you covered. Choose your favourite projects from a list (or turn on the coffee machine, order some pizza, and get through them all). Once you've gone through the step-by-step guides you can easily modify or extend the projects to make them your own.

Beginner web scraping with Python and Repl.it

Learn more about what web scraping is, how websites are built, and how to automatically scrape data from websites.

Building news word clouds using Python and Repl.it

Extending the beginner's web scraping tutorial, you'll build a more advanced scraper that extracts the plain text from news articles, stripping away the 'boilerplate' content, such as text in sidebars.

Building a Discord Bot with Python and Repl.it

Build an echo bot using the Discord API. Your bot will always respond with exactly what you send it, but you can customize it afterwards to do something more useful.

Building a Discord bot with Node.js and Repl.it

A NodeJS version of the Discord bot tutorial above. Even if you prefer Python, it's good to go through this one too to get experience with other languages.

Creating and hosting a basic web application with Django and Repl.it

Build a django web application and host it with Repl.it. You'll use geolocation a weather API to show the user their local weather forecast.

Building a CRM app with NodeJS, Repl.it, and MongoDB

Another web application, but using NodeJS instead of Django. This is a different application where you'll build a basic app to manage customer information.

Introduction to Machine Learning with Python and Repl.it

Build a machine-learning based text classifier. We skip the maths but show how you can use machine learning libraries to implement useful solutions without in-depth theoretical knoweldge.

Quicksort tutorial: Python implementation with line by line explanation

Whether you're applying for jobs or just like algorithms, it's useful to understand how sorting works. In real projects, most of the time you'll just call `.sort()`, but here you'll build a sorter from scratch and understand how it works.

Steganography with Python and Repl.it

Share secret messages with your friends by hiding them inside images with steganography, Python and Repl.it.

2D platform game with PyGame and Repl.it

Build a fun 2D platform game while learning all about Python game development. It's is easily expandible with endless possibilities so let your imagination run wild!

Understanding the Repl.it IDE: a practical guide to building your first project with Repl.it

Software developers can get pretty attached to their Integrated Development Environments (IDEs) and if you look for advice on which one to use, you'll find no end of people advocating strongly for one over another: VS Code, Sublime Text, IntelliJ, Atom, Vim, Emacs, and no shortage of others.

In the end, an IDE is just a glorified text editor. It lets you type text into files and save those files, functionality that has been present in nearly all computers since those controlled by punch cards.

In this lesson, you'll learn how to use the Repl.it IDE. It has some features you won't find in many other IDEs, namely:

- It's fully online. You can use it from any computer that can connect to the internet and run a web browser, including a phone or tablet.
- It'll fully manage your environment for building and running code: you won't need to mess around with making sure you have the right version of Python or the correct NodeJS libraries.
- You can deploy any code you build to the public in one click: no messing around with servers, or copying code around.

In the first part of this guide, we'll cover the basics and also show you how multiplayer works so that you can code alone or with friends.

Introduction: creating an account and starting a project

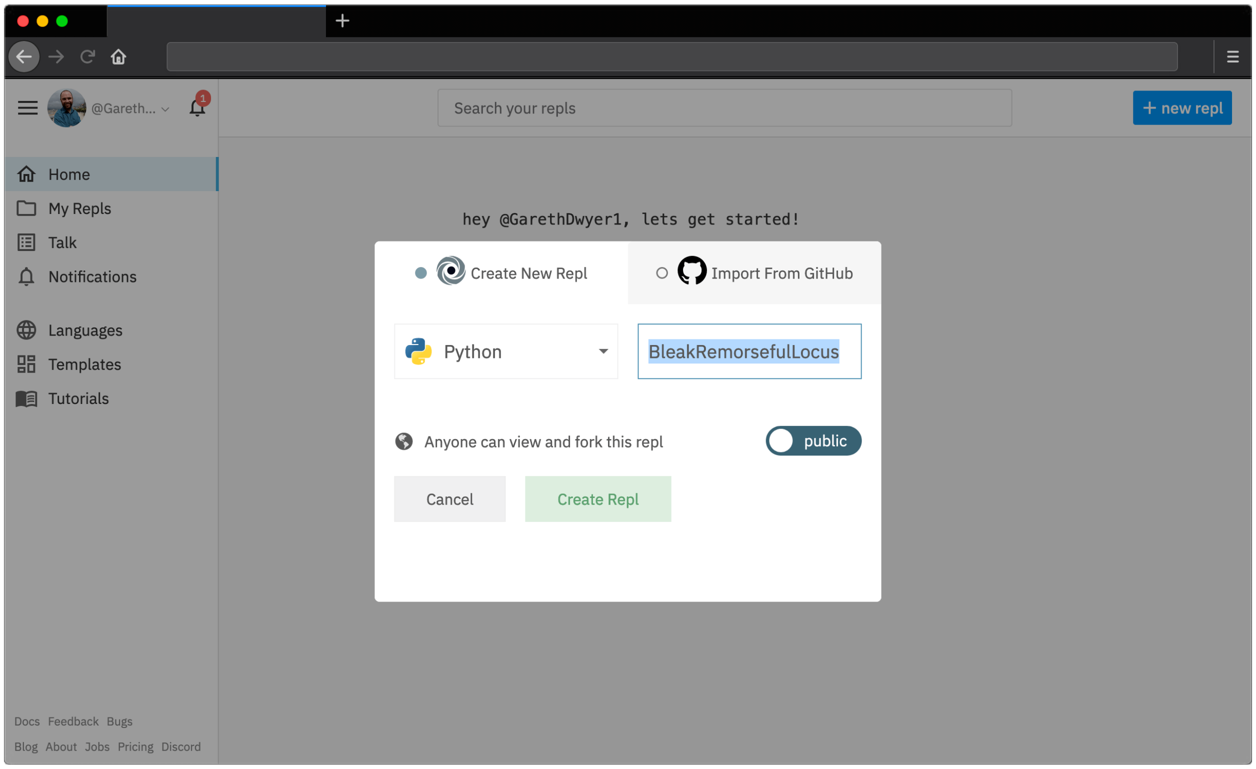
Although you don't need an account to use Repl.it (you can just navigate to repl.it[3] and press the "start coding" button), let's set one up in order to have access to all of the features.

Visit https://repl.it/signup[4] and follow the prompts to create a user account, either by entering a username and password or by logging in with Google, GitHub, or Facebook.

Once you're done, hit the `+ new repl` button in the top right. In the example below, we choose to create a new Python project. Repl.it will automatically choose a random name for your project, or

[3]https://repl.it/
[4]https://repl.it/signup

you can pick one yourself. Note that by default your repl will be public to anyone on the internet; this is great for sharing and collaboration, but we'll have to be careful to not include passwords or other sensitive information in any of our projects.



hey @GarethDwyer1, lets get started!

Python

BleakRemorsefulLocus

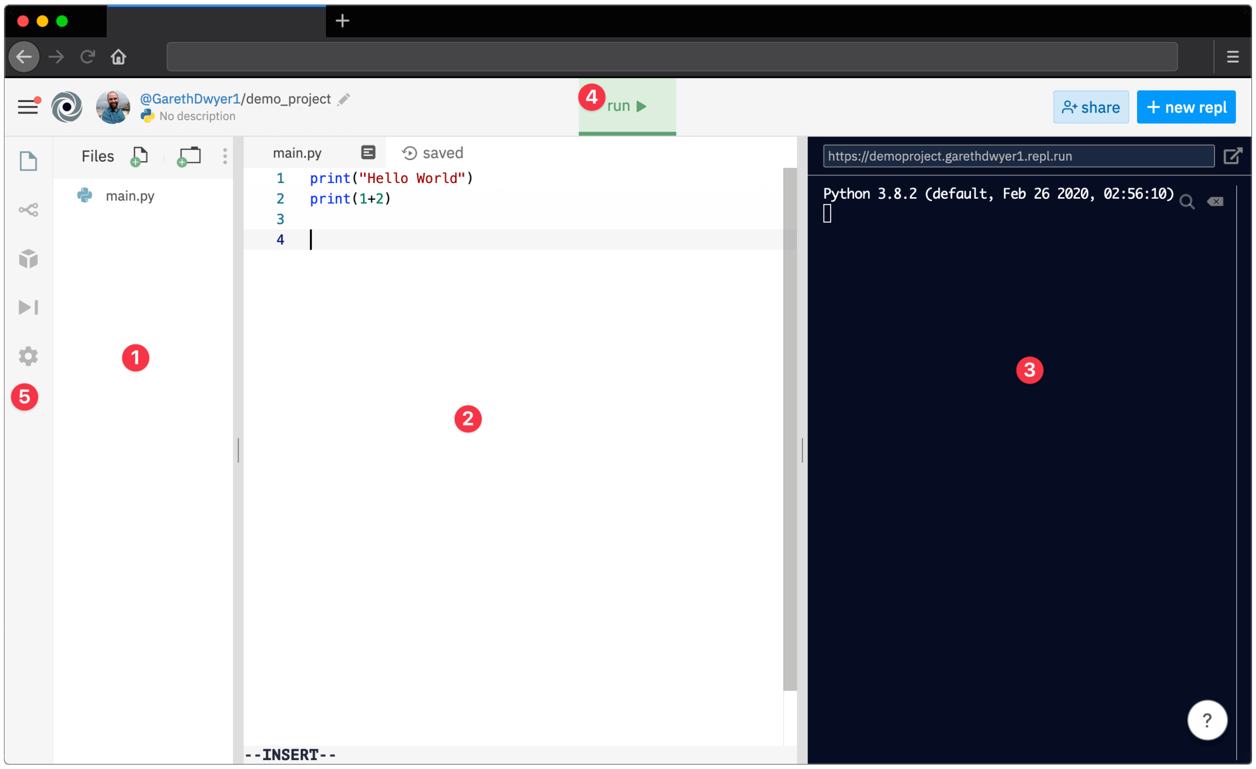
Anyone can view and fork this repl: public

Image 1: ***Creating a new Python project***

You'll also notice an "Import from GitHub" option. Repl.it allows you to import existing software projects directly from GitHub, but we'll create our own for now. Once your project is created, you'll be taken to a new view with several panes. Let's take a look at what these are.

Understanding the Repl.it panes

You'll soon see how configurable Repl.it is and how most things can be moved around to suit your fancy. However, by default, you'll get the following layout.


@GarethDwyer1/demo_project
No description
run
share
+ new repl
Files
main.py
saved
https://demoproject.garethdwyer1.repl.run
--INSERT--
1
2
3
4
5

```
print("Hello World")
print(1+2)
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

Image 2: *The Repl.it panes*

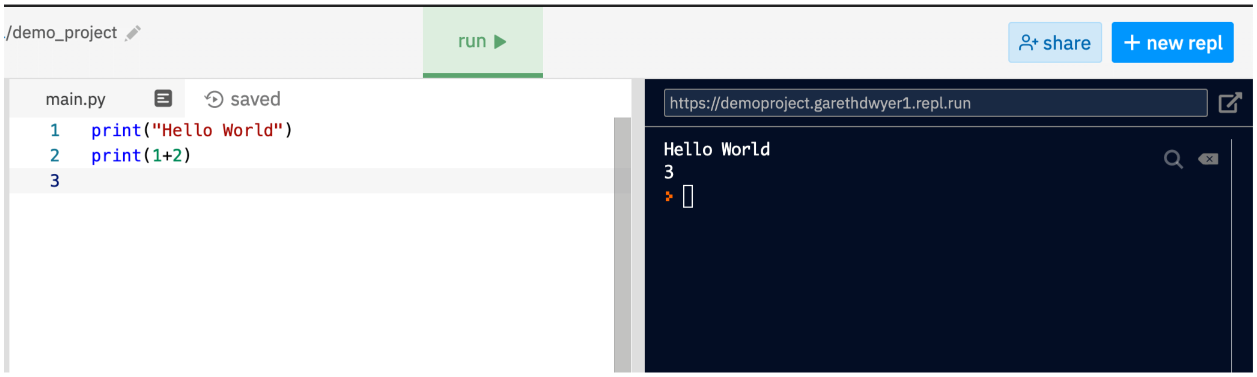
1. **Left pane: files and configuration**. This, by default, shows all the files that make up your project. Because we chose a Python project, Repl.it has gone ahead and created a `main.py` file.
2. **Middle pane: code editor**. You'll probably spend most of your time using this pane. It's a text editor where you can write code. In the screenshot, we've added two lines of Python code, which we'll run in a bit.
3. **Right pane: output sandbox**. This is where you'll see your code in action. All output that your program produces will appear in this pane, and it also acts as a quick sandbox to run small pieces of code, which we'll look at more later.
4. **Run button**. If you click the big green `run` button, your code will be executed and the output will appear on the right.
5. **Menu bar**. This lets you control what you see in the main left pane (pane 1). By default, you'll see the files that make up your project but you can use this bar to view other things here too by clicking on the various icons. We'll take a look at these options later.

Don't worry too much about all of the functionality offered right away. For now, we have a simple goal: write some code and run it.

Running code from a file

Usually, you'll enter your code as text in a file, and run it from there. Let's do this now. Enter the following code in the middle pane (pane 2), and hit the run button.

```python
print("Hello World")
print(1+2)
```



```python
print("Hello World")
print(1+2)
```

```
Hello World
3
>
```

Image 3: *Your first program*

Your script will run and the output it generates will appear on the right pane (pane 3). Our code output the phrase “Hello World” (it’s a long-standing tradition that when you learn something new the first thing you do is build a ‘hello world’ project), and then output the answer to the sum `1 + 2`.

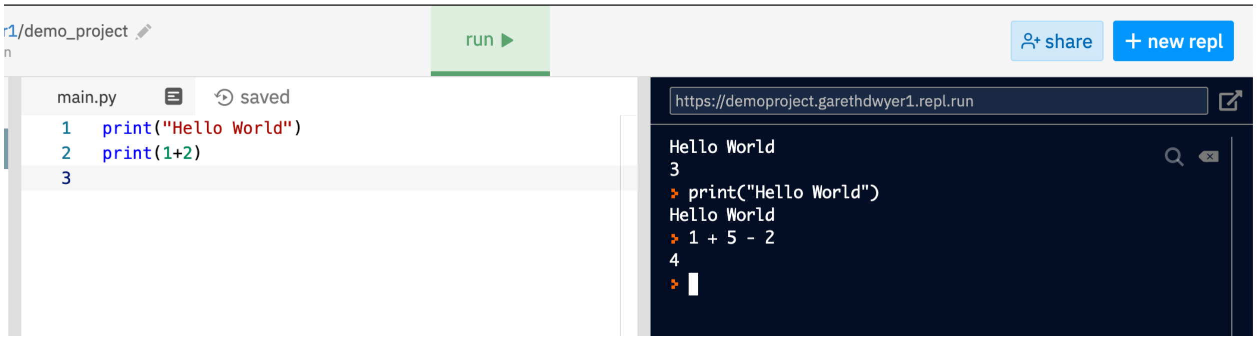
You probably won’t be able to turn this script into the next startup unicorn quite yet, but let’s keep going.

Running code from Repl.it’s REPL

In computer programming, a REPL is a read-eval-print loop[5], and a REPL interface is often the simplest way to run short computer programs (and where Repl.it got its name).

While in the previous example we saved our code to a file and then executed the file, it’s sometimes quicker to execute code directly.

You can type code in the right-hand pane (pane 3) and press the “Enter” key to run it. Take a look at the example below where we print “Hello World” again and do a different sum, without changing our code file.



```python
print("Hello World")
print(1+2)
```

```
Hello World
3
> print("Hello World")
Hello World
> 1 + 5 - 2
4
>
```

Image 4: *Running code from the REPL*

[5]https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

This is useful for prototyping or checking how things work, but for any serious program you write you'll want to be able to save it, and that means writing the code in a file like in our earlier example.

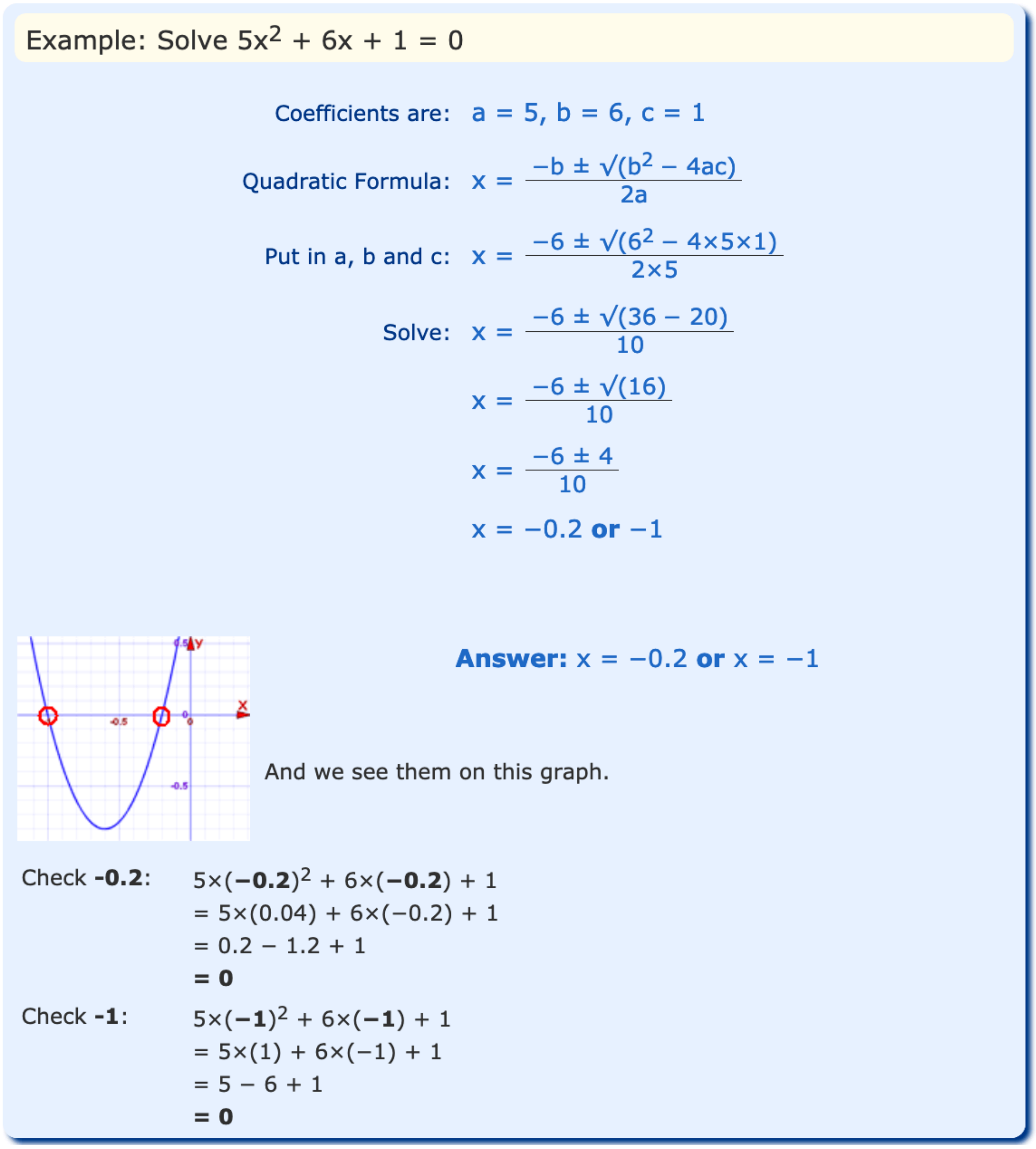
Adding more files to your software project

If you build larger projects, you'll want to use more than a single file to stay organised, grouping related code in different files.

So far, we've been using the 'main.py' file that was automatically created for us when we started the project, but we're not limited to this file. Let's add a new file, write some code in that, and import it into the main file for use.

As an example, we'll write code to solve quadratic equations[6]. If you've done this before, you'll know it can be tedious without a computer to go through all of the steps.

[6]https://www.mathsisfun.com/algebra/quadratic-equation.html



Example: Solve $5x^2 + 6x + 1 = 0$

Coefficients are: $a = 5, b = 6, c = 1$

Quadratic Formula: $x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$

Put in a, b and c: $x = \frac{-6 \pm \sqrt{(6^2 - 4\times5\times1)}}{2\times5}$

Solve: $x = \frac{-6 \pm \sqrt{(36 - 20)}}{10}$

$x = \frac{-6 \pm \sqrt{(16)}}{10}$

$x = \frac{-6 \pm 4}{10}$

$x = -0.2$ **or** -1

Answer: $x = -0.2$ **or** $x = -1$

And we see them on this graph.

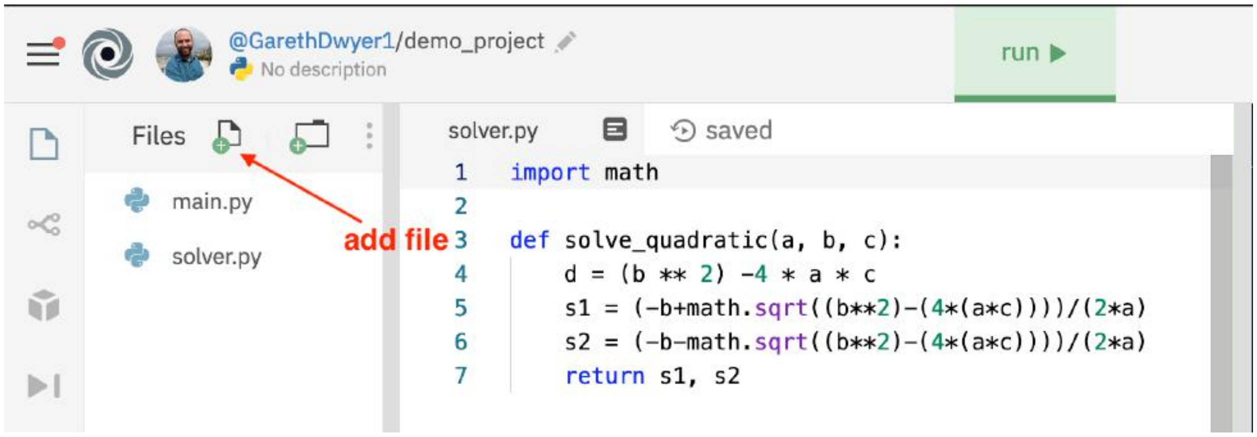
Check **-0.2**: $5\times(\mathbf{-0.2})^2 + 6\times(\mathbf{-0.2}) + 1$
$= 5\times(0.04) + 6\times(-0.2) + 1$
$= 0.2 - 1.2 + 1$
$\mathbf{= 0}$

Check **-1**: $5\times(\mathbf{-1})^2 + 6\times(\mathbf{-1}) + 1$
$= 5\times(1) + 6\times(-1) + 1$
$= 5 - 6 + 1$
$\mathbf{= 0}$

Image 5: *A quadratic equation example*

Let’s make Python do the repetitive steps for us by creating a program called “solver”. This could eventually have a lot of different solvers, but for now we’ll just write one: `solve_quadratic`.

Add a new file to your project by clicking on the `new file` button, as shown below. Call the file `solver.py`. You now have two files in your project: `main.py` and `solver.py`. You can switch between your files by clicking on them.



```
import math

def solve_quadratic(a, b, c):
    d = (b ** 2) -4 * a * c
    s1 = (-b+math.sqrt((b**2)-(4*(a*c))))/(2*a)
    s2 = (-b-math.sqrt((b**2)-(4*(a*c))))/(2*a)
    return s1, s2
```

Image 6: ***Adding a new file***

Open the `solver.py` file and add the following code to it.

```
import math

def solve_quadratic(a, b, c):
    d = (b ** 2) - 4 * a * c
    s1 = (-b + math.sqrt(d)) / (2 * a)
    s2 = (-b - math.sqrt(d)) / (2 * a)
    return s1, s2
```

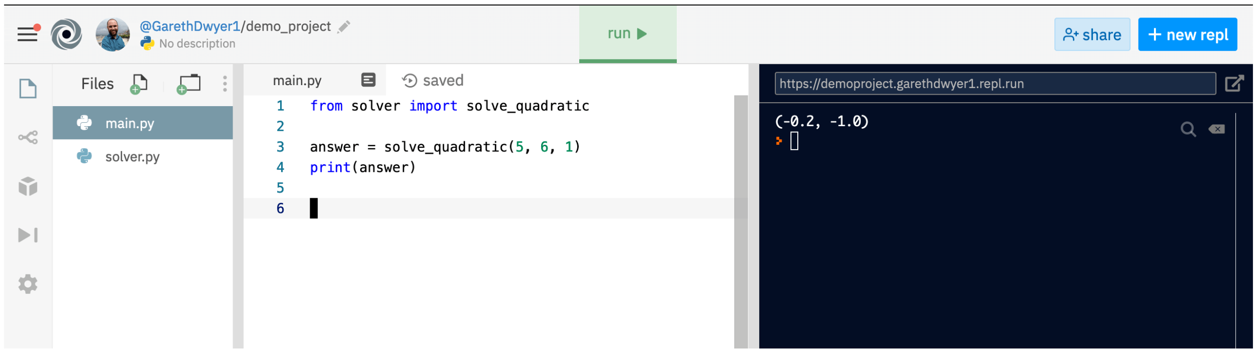
Note that this won't solve all quadratic equations as it doesn't handle cases where `d`, the discriminant, is 0 or negative. However, it'll do for now.

Navigate back to the `main.py` file. Delete all the code we had before and add the following code instead.

```
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)
```

Note how we use Python's import functionality to import the code from our new solver script into the `main.py` file so that we can run it. Python looks for `.py` (Python) files automatically, so you omit the `.py` suffix when importing code. Here we import the `solve_quadratic` function (which we just defined) from the `solver.py` file.

Run the code and you should see the solution to the equation, as shown below.


@GarethDwyer1/demo_project
No description
run ▶
share
+ new repl
Files
main.py
solver.py
main.py
saved
https://demoproject.garethdwyer1.repl.run

```python
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)

```

```
(-0.2, -1.0)
>
```

Image 7: *The solution to the quadratic equation*

Congratulations! You've written your first useful program.

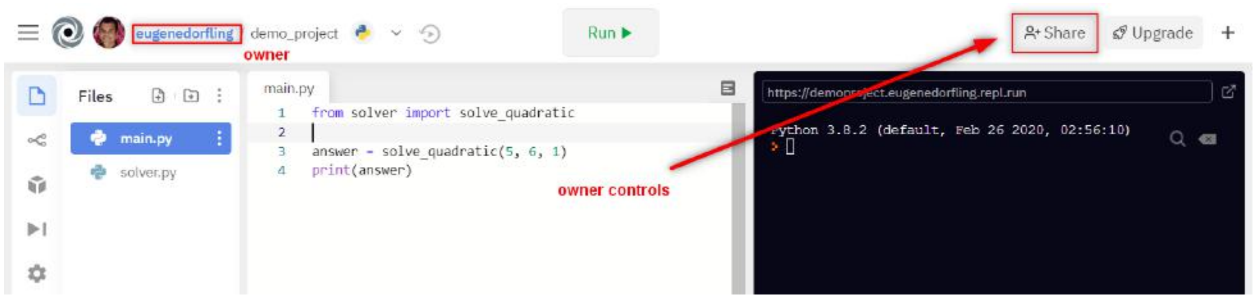
Sharing your application with others

Coding is more fun with friends or as part of a team. If you want to share your code with others, it's as easy as copying the URL and sending it. In this case, the URL is `https://repl.it/@GarethDwyer1/demoproject`, but yours will be different based on your Repl.it username and the project name you chose.

You can copy the link and open it in an incognito tab (or a different web browser) to see how others would experience your project if you were to share it. By default, they'll be able to see all of your files and code and run your code, but not make any changes. If someone else tries to make changes to your repl, it'll automatically get copied ("forked") to their account, or an anonymous account if they haven't signed up for Repl.it. Any changes your friends make will only happen in their copies, and won't affect your code at all.

To understand this, compare the three versions of the same repl below.

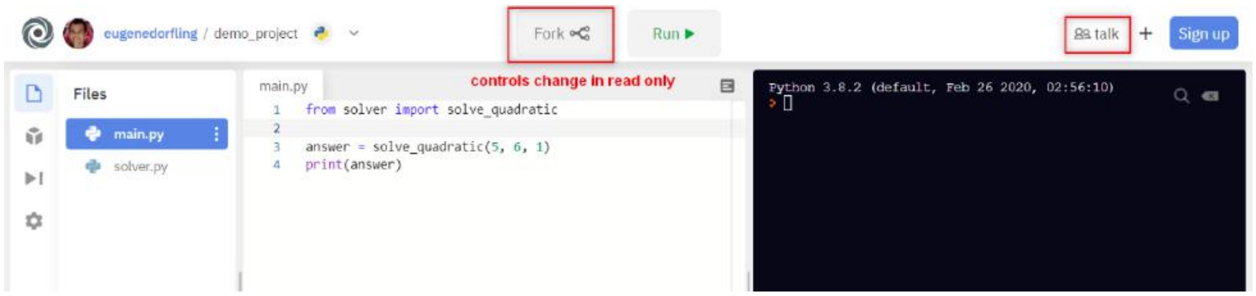
- As you see it, with all of the controls
- As your friend would see it, a read-only version
- As your friend would see it after forking it, on an anonymous account



```python
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

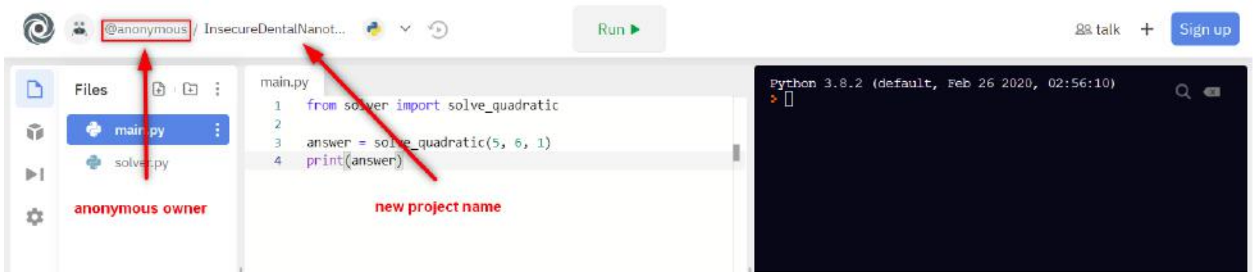
Image 8: *The owner's view of a repl*



```
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Image 9: ***A guest's 'read-only' view of a repl***



```
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

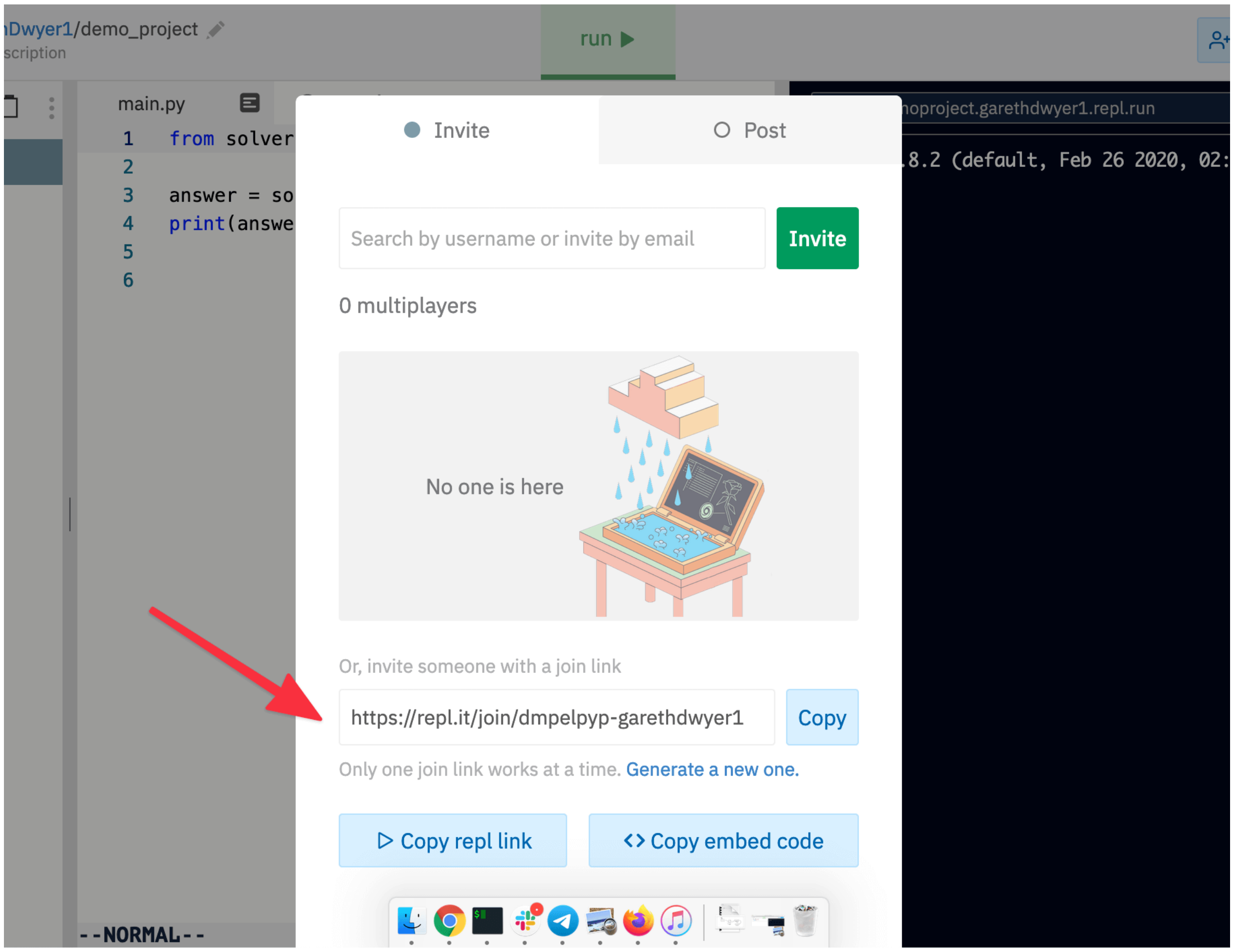
Image 10: ***An anonymous owner's view of a copy of a repl***

What does this mean? Because no one else can edit your repl, you can share it far and wide. But because anyone can read your repl, you should be careful that you don't share anything private or secret in it.

Sharing write-access: Multiplayer

Of course, sometimes you might want others to have write access to your repl so that they can contribute, or help you out with a problem. In these cases, you can use Repl.it's "multiplayer" functionality.

If you invite someone to your repl, it's different from sharing the URL with them. You can invite someone by clicking `Share` in the top right and sending them the secret link that starts with "https://repl.it/join". This link will give people edit access to your repl.

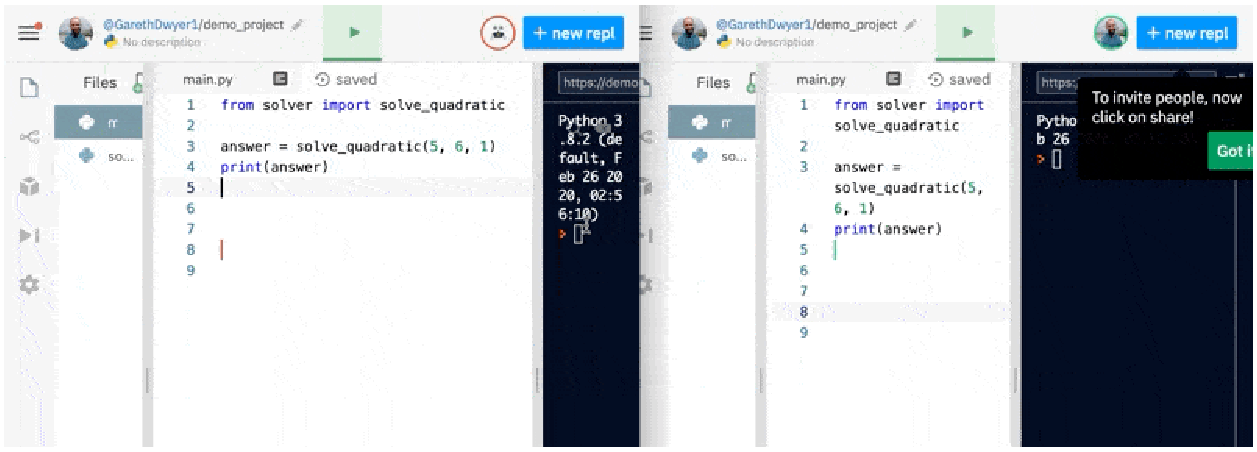

Dwyer1/demo_project
scription
run ▶
main.py
Invite
Post
Search by username or invite by email
Invite
0 multiplayers
No one is here
Or, invite someone with a join link
https://repl.it/join/dmpelpyp-garethdwyer1
Copy
Only one join link works at a time. Generate a new one.
Copy repl link
Copy embed code
oproject.garethdwyer1.repl.run
--NORMAL--

```
from solver
 
answer = so
print(answe
```

```
.8.2 (default, Feb 26 2020, 02:
```

Image 11: ***Inviting someone to your repl***

If you have a friend handy, send it to them to try it out. If not, you can try out multiplayer anyway using a separate incognito window again. Below is our main Repl.it account on the left and a second account which opened the multiplayer invite link on the right. As you can see, all keystrokes can be seen by all parties in real time.


@GarethDwyer1/demo_project
No description
+ new repl
Files
main.py
saved
rr
so...
To invite people, now click on share!
Got it

```
from solver import solve_quadratic

answer = solve_quadratic(5, 6, 1)
print(answer)
```

```
Python 3
.8.2 (de
fault, F
eb 26 20
20, 02:5
6:10)
>
```

Image 12: ***Using multiplayer***

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it from https://repl.it/@GarethDwyer1/cwr-01-quadratic-equations[7].

Where next?

You can now create basic programs on your own or with friends, and you are familiar with the most important Repl.it features. There's a lot more to learn though. In the next nine lessons, you'll work through a series of projects that will teach you more about Repl.it features and programming concepts along the way.

If you get stuck, you can get help from the Repl.it community[8] or on the Repl.it Discord server[9].

[7]https://repl.it/@GarethDwyer1/cwr-01-quadratic-equations
[8]https://repl.it/talk/all
[9]https://repl.it/discord

Working with Files using Repl.it

In this lesson, you'll gain experience with using and manipulating files using Repl.it. In the previous lesson you saw how to add new files to a project, but there's a lot more you can do.

Files can be used for many different things. In programming, you'll primarily use them to store data or code. Instead of manually creating files and entering data, you can also use your programs to create files and automatically write data to these.

Repl.it also offers functionality to mass import or export files from or to the IDE; this is useful in cases when your program writes data to multiple files and you want to export all of these for use in another program.

Working with files using Python

Create a new Python repl and call it `working-with-files`.

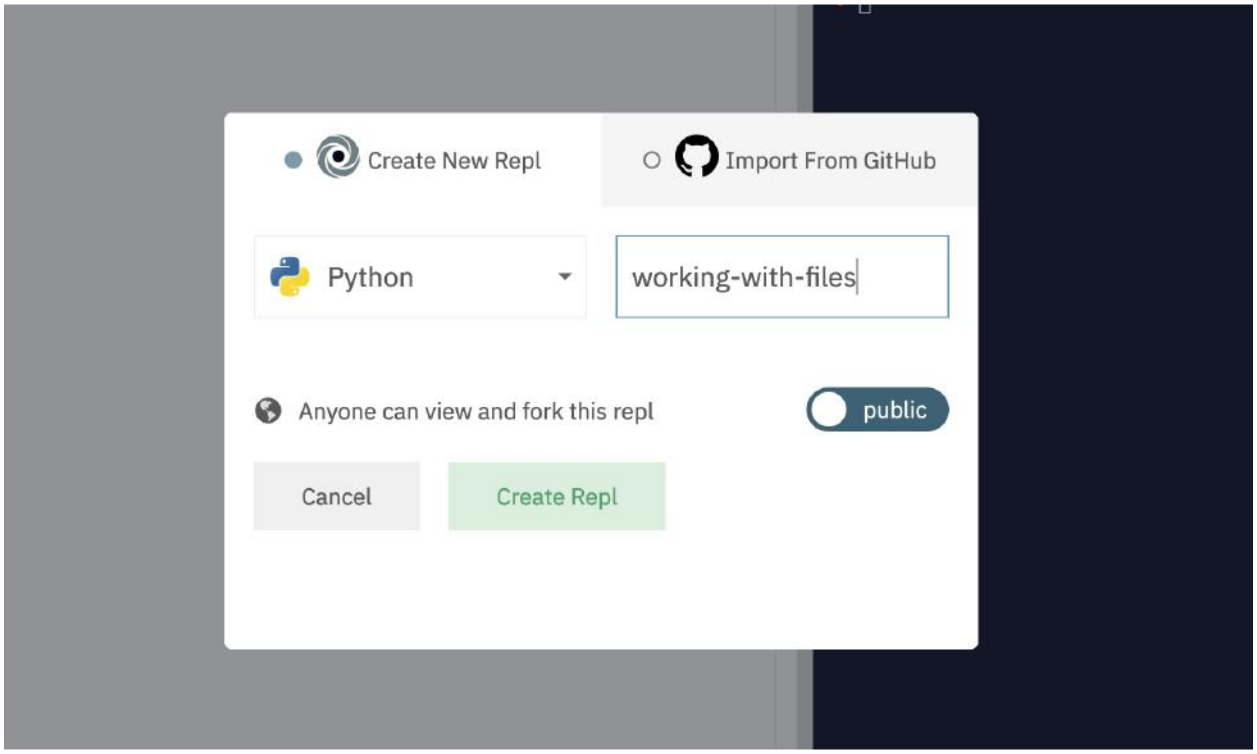
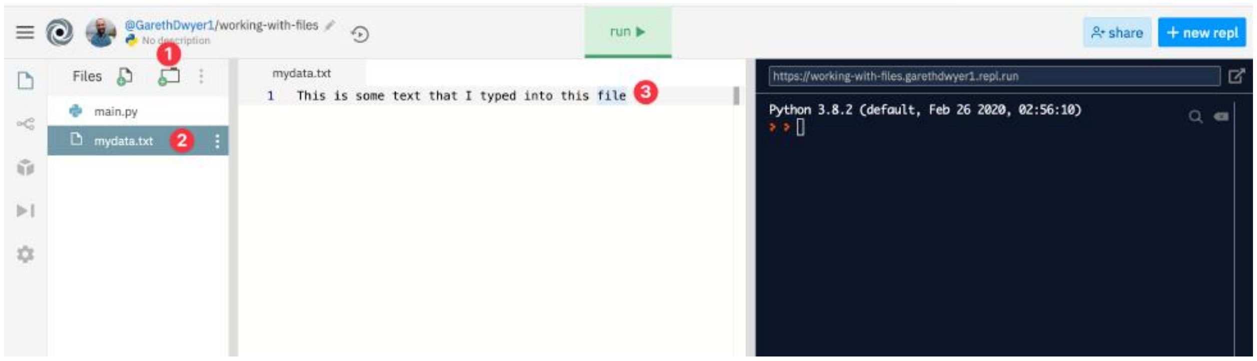


Image 1: Creating our files project

As before, you'll get a default repl project with a `main.py` file. We need a data file to practise reading data programmatically.

1. Create a new file using the `add file` button.
2. Call the file `mydata.txt`. Previously we created a Python file (`.py`) but in this case we are creating a plain text file (`.txt`).
3. Type a line of text into the file.

You should now have something similar to what you see below.


@GarethDwyer1/working-with-files
No description
run ▶
share
+ new repl
Files
main.py
mydata.txt
1
2
3
https://working-with-files.garethdwyer1.repl.run

```
This is some text that I typed into this file
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> >
```

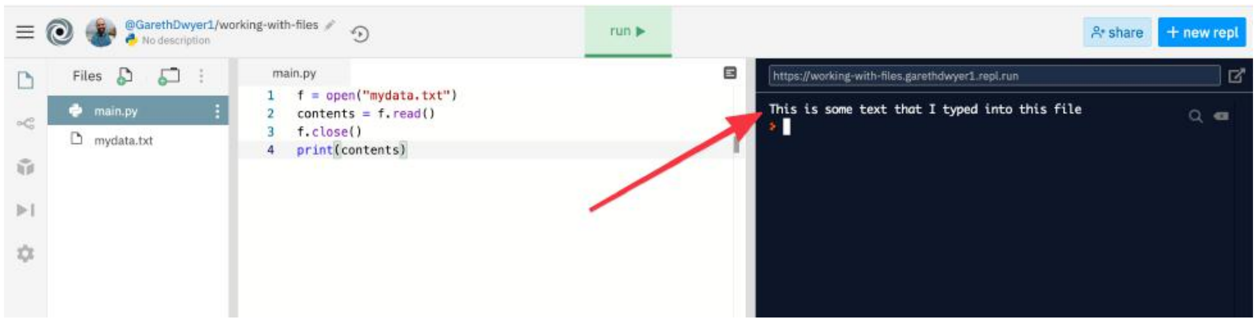
Image 2: Adding data manually to our text file

Go back to the `main.py` file and add the following code.

```python
f = open("mydata.txt")
contents = f.read()
f.close()
print(contents)
```

This opens the file programmatically, reads the data in the file into a variable called `contents`, closes the file, and prints the file contents to our output pane.

Press the `Run` button. If everything went well, you should see output as shown in the image below.


@GarethDwyer1/working-with-files
No description
run ▶
share
+ new repl
Files
main.py
mydata.txt
https://working-with-files.garethdwyer1.repl.run

```python
f = open("mydata.txt")
contents = f.read()
f.close()
print(contents)
```

```
This is some text that I typed into this file
>
```

Image 3: Reading data from a file and printing it out

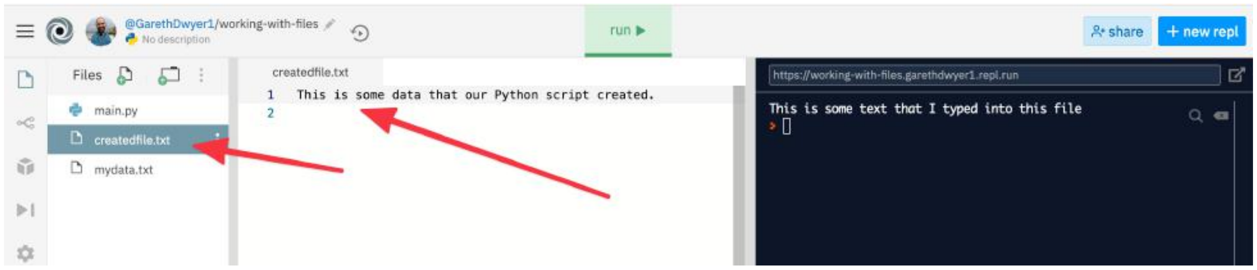
Creating files using Python

Instead of manually creating files, you can also use Python to do this. Add the following code to your `main.py` file.

```python
f = open("createdfile.txt", "w")
f.write("This is some data that our Python script created.\n")
f.close()
```

Note the `w` argument that we pass into the `open` function now. This means that we want to open the file in "write" mode. Be careful: if the file already exists, Python will delete it and create a new one. There are many different 'modes' to work with files in different ways which you can read about in the Python documentation[10].

Run the code again and you should see a new file pop up in the files pane. If you click on the file, you'll find the data that the Python script wrote to it, as shown below.



```
This is some data that our Python script created.
```

```
This is some text that I typed into this file
>
```

Image 4: Writing data to a file and viewing it

Building a weather logging system using Python and Repl.it

Now that you can read from files and write to them, let's build a mini-project that records historical weather temperatures. Our program will

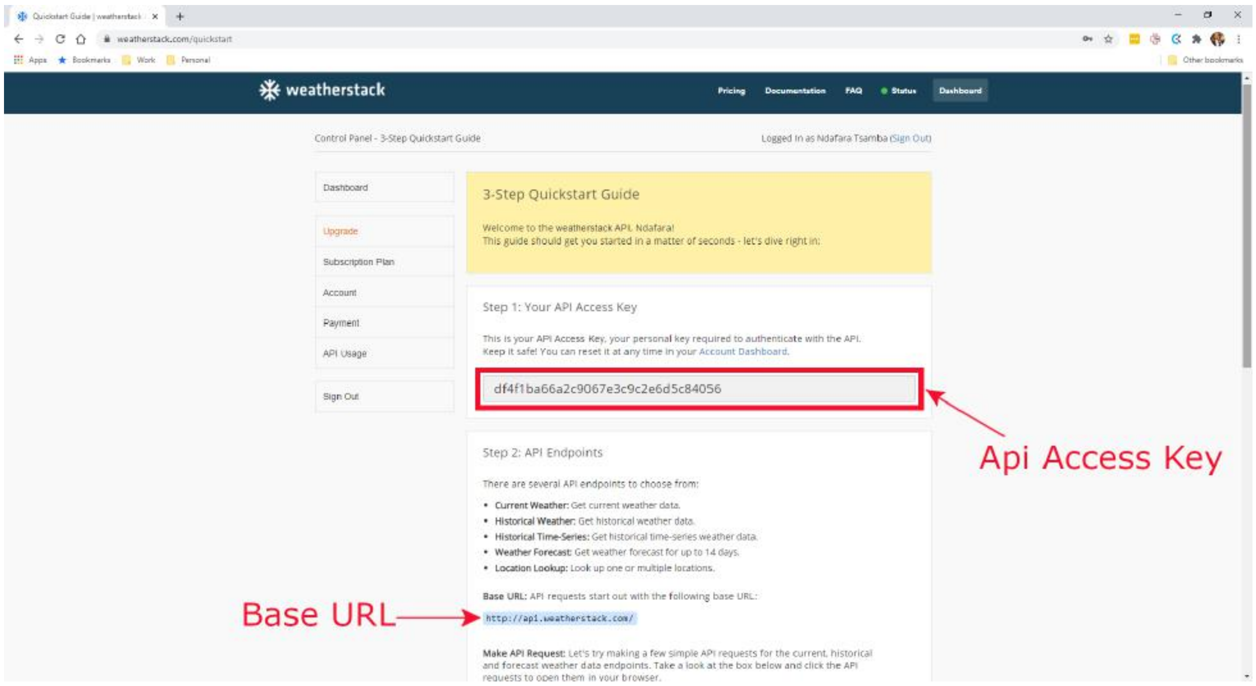
- Get the current temperature for a specified set of cities.
- Write this data to file with today's date.
- Print a summary of this data to the console.

Creating a WeatherStack Account and getting an API key

We'll get weather data from the WeatherStack API. You'll need to sign up at weatherstack.com[11] and follow the instructions to get your own access key. Choose the "free tier" option which is limited to 1000 calls per month.

After sign-up, you should see a page similar to the following containing the API access key.

[10]https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
[11]https://weatherstack.com



Control Panel - 3-Step Quickstart Guide

Logged In as Ndafara Tsamba (Sign Out)

3-Step Quickstart Guide

Welcome to the weatherstack API, Ndafara!
This guide should get you started in a matter of seconds - let's dive right in:

Step 1: Your API Access Key

This is your API Access Key, your personal key required to authenticate with the API.
Keep it safe! You can reset it at any time in your Account Dashboard.

df4f1ba66a2c9067e3c9c2e6d5c84056

Step 2: API Endpoints

There are several API endpoints to choose from:

- **Current Weather:** Get current weather data.
- **Historical Weather:** Get historical weather data.
- **Historical Time-Series:** Get historical time-series weather data.
- **Weather Forecast:** Get weather forecast for up to 14 days.
- **Location Lookup:** Look up one or multiple locations.

Base URL: API requests start out with the following base URL:

`http://api.weatherstack.com/`

Make API Request: Let's try making a few simple API requests for the current, historical and forecast weather data endpoints. Take a look at the box below and click the API requests to open them in your browser.

Image 5: *Getting an API key from WeatherStack*

You should keep this key secret to stop other people using up all of your monthly calls.

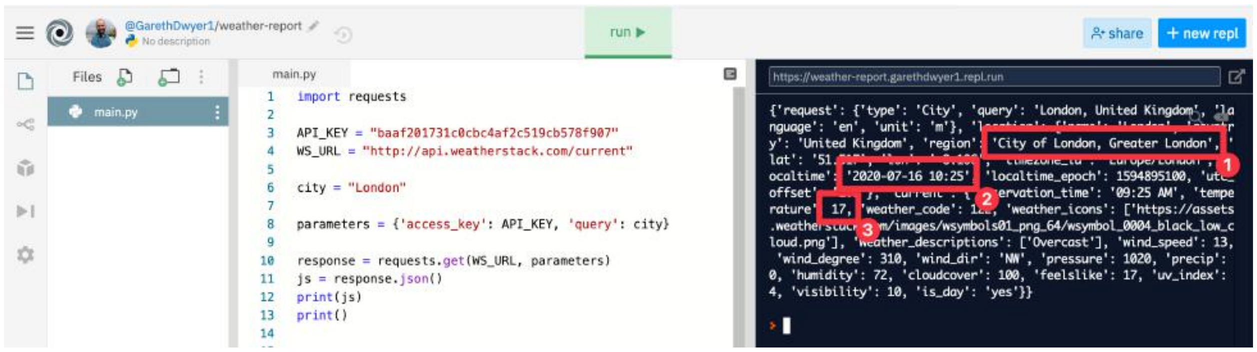
Creating our weather reporting project

You can continue to use the `working-with-files` project that you created earlier if you want to, but for the demonstration we'll create a new Python repl called `weather report`.

In the `main.py` file add the following code, but replace the string for `API_KEY` with your own one.

```python
import requests

# change the following line to use your own API key
API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

city = "London"

parameters = {'access_key': API_KEY, 'query': city}

response = requests.get(WS_URL, parameters)
js = response.json()
print(js)
print()
```

This code asks WeatherStack for the current temperature in London, gets the JSON[12] version of this and prints it out. You should see something similar to what is shown below.


@GarethDwyer1/weather-report
No description
run ▶
share
+ new repl
Files
main.py
https://weather-report.garethdwyer1.repl.run
1
2
3

```
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

city = "London"

parameters = {'access_key': API_KEY, 'query': city}

response = requests.get(WS_URL, parameters)
js = response.json()
print(js)
print()
```

```
{'request': {'type': 'City', 'query': 'London, United Kingdom', 'la
nguage': 'en', 'unit': 'm'}, '[illegible]
y': 'United Kingdom', 'region': 'City of London, Greater London', '
lat': '51.[illegible]
ocaltime': '2020-07-16 10:25', 'localtime_epoch': 1594895100, 'utc_
offset'[illegible]ervation_time': '09:25 AM', 'tempe
rature': 17, 'weather_code': 1[illegible], 'weather_icons': ['https://assets
.weather[illegible]m/images/wsymbols01_png_64/wsymbol_0004_black_low_c
loud.png'], 'weather_descriptions': ['Overcast'], 'wind_speed': 13,
 'wind_degree': 310, 'wind_dir': 'NW', 'pressure': 1020, 'precip':
0, 'humidity': 72, 'cloudcover': 100, 'feelslike': 17, 'uv_index':
4, 'visibility': 10, 'is_day': 'yes'}}

>
```

Image 6: *Getting the current weather in JSON format*

WeatherStack returns a lot of data, but we are mainly interested in

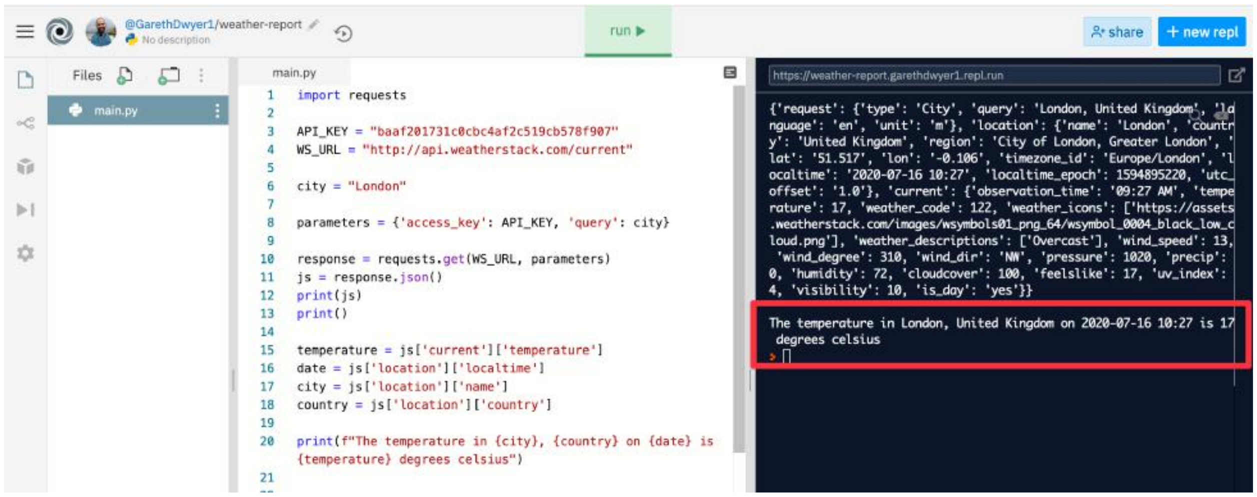
1. **The location**: To see if we found the correct London and not one of the 29 other places[13] called London.
2. **The date**: We'll record this when we save this data to a file.
3. **The current temperature**: This is specified by default in Celsius, but can be customised[14] if you prefer Fahrenheit.

Add the following code below the existing code to extract these values into a format that's easier to read.

```
temperature = js['current']['temperature']
date = js['location']['localtime']
city = js['location']['name']
country = js['location']['country']

print(f"The temperature in {city}, {country} on {date} is {temperature} degrees Cels\
ius")
```

If you run the code again, you'll see a more human-friendly output, as shown below.

[12]https://www.w3schools.com/js/js_json_intro.asp
[13]https://www.wanderlust.co.uk/content/londons-around-the-world/
[14]https://weatherstack.com/documentation


@GarethDwyer1/weather-report
No description
run ▶
share
+ new repl
Files
main.py
https://weather-report.garethdwyer1.repl.run

```python
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

city = "London"

parameters = {'access_key': API_KEY, 'query': city}

response = requests.get(WS_URL, parameters)
js = response.json()
print(js)
print()

temperature = js['current']['temperature']
date = js['location']['localtime']
city = js['location']['name']
country = js['location']['country']

print(f"The temperature in {city}, {country} on {date} is {temperature} degrees celsius")
```

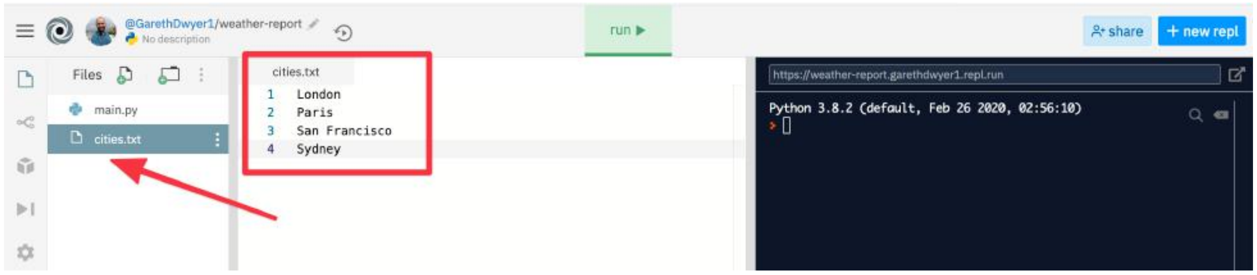
```
{'request': {'type': 'City', 'query': 'London, United Kingdom', 'la
nguage': 'en', 'unit': 'm'}, 'location': {'name': 'London', 'countr
y': 'United Kingdom', 'region': 'City of London, Greater London', '
lat': '51.517', 'lon': '-0.106', 'timezone_id': 'Europe/London', 'l
ocaltime': '2020-07-16 10:27', 'localtime_epoch': 1594895220, 'utc_
offset': '1.0'}, 'current': {'observation_time': '09:27 AM', 'tempe
rature': 17, 'weather_code': 122, 'weather_icons': ['https://assets
.weatherstack.com/images/wsymbols01_png_64/wsymbol_0004_black_low_c
loud.png'], 'weather_descriptions': ['Overcast'], 'wind_speed': 13,
 'wind_degree': 310, 'wind_dir': 'NW', 'pressure': 1020, 'precip':
0, 'humidity': 72, 'cloudcover': 100, 'feelslike': 17, 'uv_index':
4, 'visibility': 10, 'is_day': 'yes'}}

The temperature in London, United Kingdom on 2020-07-16 10:27 is 17
 degrees celsius
> 
```

Image 7: ***Seeing a human-readable summary of the data***

This is great for getting the current weather, but now we want to extend it a bit to record weather historically.

We'll create a file called `cities.txt` containing the list of cities we want to get weather data for. Our script will request the weather for each city, and save a new line with the weather and timestamp.

Add the `cities.txt` file, as in the image below (of course, you can change which cities you would like to get weather info for).


@GarethDwyer1/weather-report
No description
run ▶
share
+ new repl
Files
main.py
cities.txt
https://weather-report.garethdwyer1.repl.run

```
London
Paris
San Francisco
Sydney
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> 
```

Image 8: ***Creating the cities.txt file***

Now remove the code we currently have in `main.py` and replace it with the following.

```python
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

cities = []
with open("cities.txt") as f:
    for line in f:
        cities.append(line.strip())
print(cities)
```

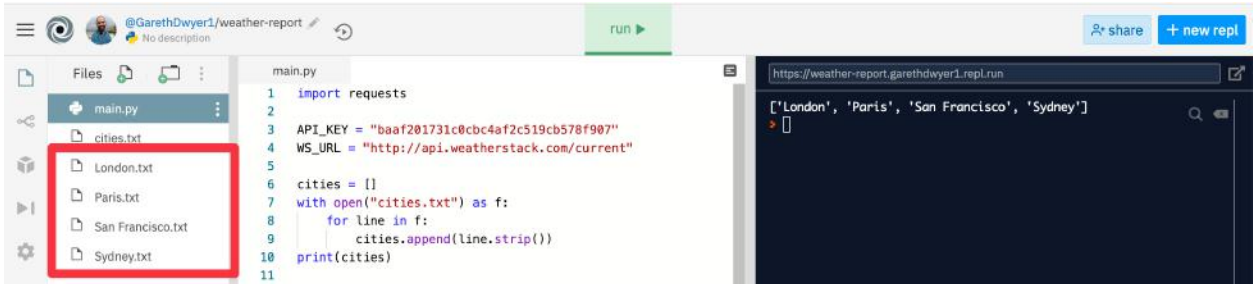
```python

for city in cities:
    parameters = {'access_key': API_KEY, 'query': city}
    response = requests.get(WS_URL, parameters)
    js = response.json()

    temperature = js['current']['temperature']
    date = js['location']['localtime']

    with open(f"{city}.txt", "w") as f:
        f.write(f"{date},{temperature}\n")
```

This is similar to the code we had before, but now we

- Read the city names from our `cities.txt` file and put each city into a Python list.
- Loop through the cities and get the weather data for each one.
- Create a new file with the same name as each city and write the date and temperature (separated by a comma) to each file.

In our previous examples we explicitly closed files using `f.close()`. In this example, we instead open our files in a `with` block. This is a common idiom in Python and is usually how you will open files. You can read more about this in the files section of the Python docs[15].

If you run this code, you'll see it creates one file for each city.



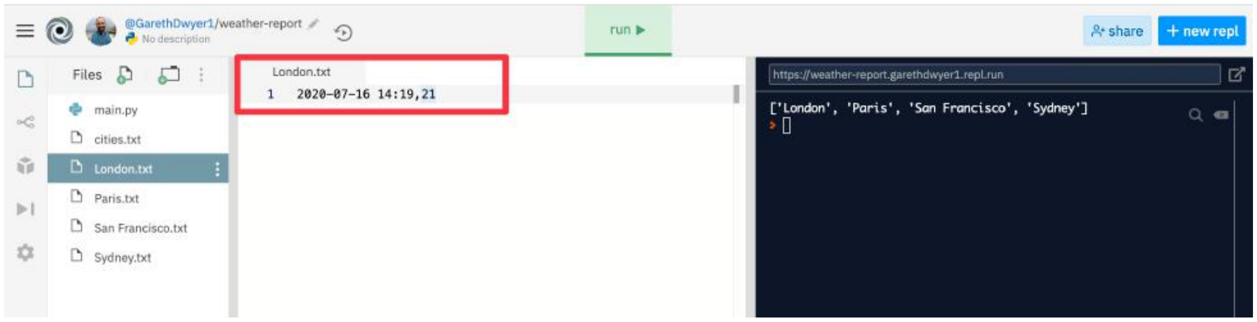
```python
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

cities = []
with open("cities.txt") as f:
    for line in f:
        cities.append(line.strip())
print(cities)

```

```
['London', 'Paris', 'San Francisco', 'Sydney']
> 
```

Image 9: *The script creates one file for each city*

If you open up one of the files, you'll see it contains the date and temperature that we fetched from WeatherStack.

[15]https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files



London.txt

```
2020-07-16 14:19,21
```

```
['London', 'Paris', 'San Francisco', 'Sydney']
>
```

Image 10: ***Example data recorded for London***

If you run the script multiple times, each file will still only contain one line of data: that from the most recent run. This is because when we open a file with in "write" mode (`"w"`), it overwrites it with the new data. Because we want to create historical weather logs, we need to change the second last line to use "append" mode instead (`"a"`).

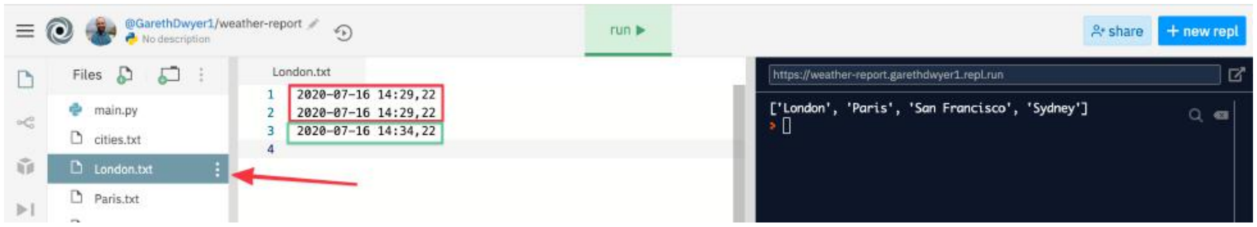
Change

```
    with open(f"{city}.txt", "w") as f:
```

to

```
    with open(f"{city}.txt", "a") as f:
```

and run the script again. If you open one of the city files again, you'll see it has a new line instead of the old data being overwritten. Newer data is appended to the end of the file. WeatherStack only updates its data every 5 minutes or so, so you might see exact duplicate lines if you run the script multiple times in quick succession.



London.txt

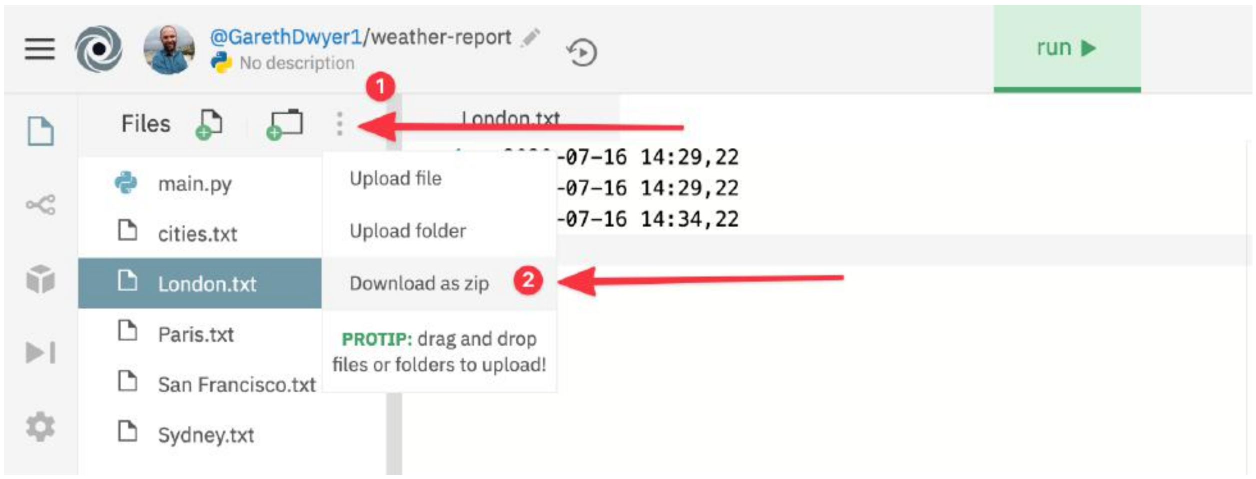
```
2020-07-16 14:29,22
2020-07-16 14:29,22
2020-07-16 14:34,22
```

```
['London', 'Paris', 'San Francisco', 'Sydney']
>
```

Image 11: ***Adding new data to the end of each file***

Exporting our weather data files

If you run this script every day for a few months, you'll have a nice data set that could be useful in other contexts too. If you want to download all of the data from Repl.it, you can use the `Download as zip` functionality to export all of the files in a repl (including the code and data files).



```
-07-16 14:29,22
-07-16 14:29,22
-07-16 14:34,22
```

Image 12: ***Downloading all of our files from Repl.it***

Once you've downloaded the `.zip` file you can extract it in your local file system and find all of the data files which can now be opened with other programs as required.

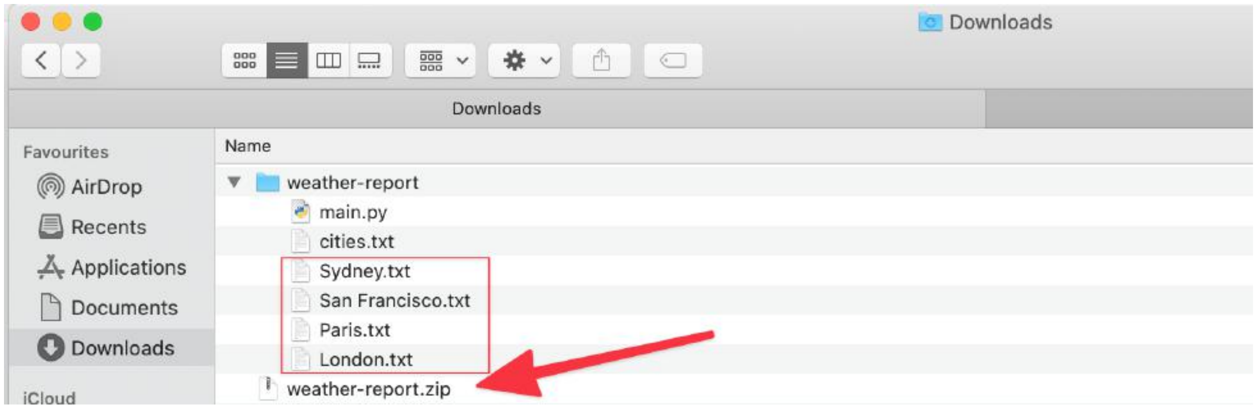
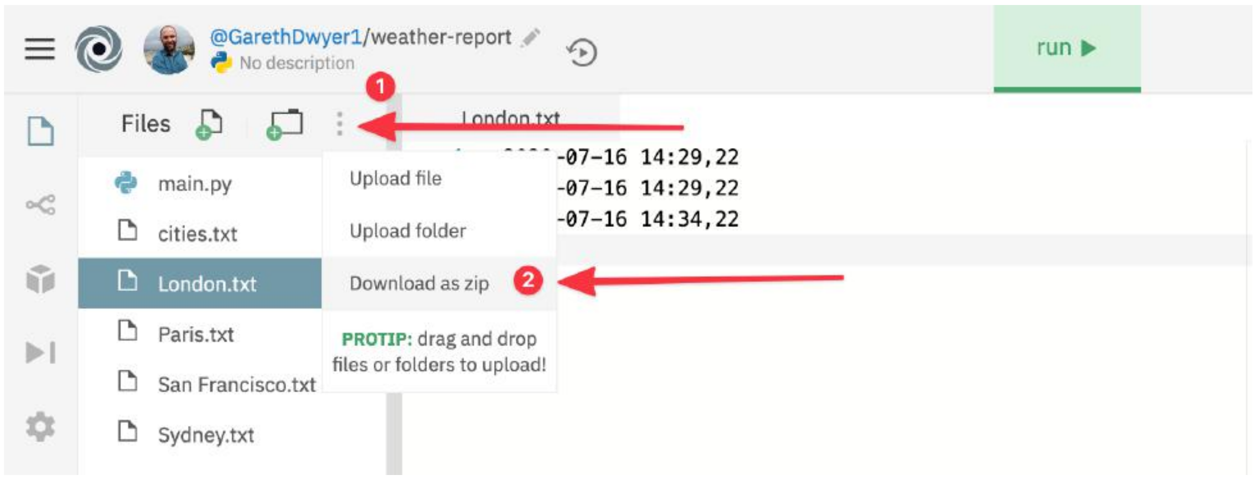


Image 13: ***The created data files on our local file system***

From the same menu, you can also choose `upload file` or `upload folder` to import files into your repl. For example, if you cleaned the files using external software and then wanted your repl to start appending new data to the cleaned versions, you could re-import them.

Repl.it will warn you about overwriting your existing files if you haven't changed the names.



```
-07-16 14:29,22
-07-16 14:29,22
-07-16 14:34,22
```

Image 14: ***Be careful about overwriting your precious data***

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it from https://repl.it/@GarethDwyer1/cwr-02-weather-report[16].

Where next?

That's it for our weather reporting project. You learned how to work with files in Python and Repl.it, including different modes (read, write, or append) in which files can be opened.

You also worked with an external library, `requests`, for fetching data over the internet. This module is not actually part of Python, and in the next article you'll learn more about how to manage external modules or dependencies.

[16]https://repl.it/@GarethDwyer1/cwr-02-weather-report

Managing dependencies using Repl.it

Nearly all useful programs rely to some extent on pre-existing code in various forms. The existing code that your code relies on is known as a dependency. You have already come across some dependencies in previous tutorials: you used the `math` module to calculate quadratic equations in the first tutorial and you used the `requests` module to fetch weather data in the second tutorial.

In the first tutorial, you also wrote the `solver.py` module, and imported this into the `main.py` file.

We can think of dependencies as falling into three broad categories:

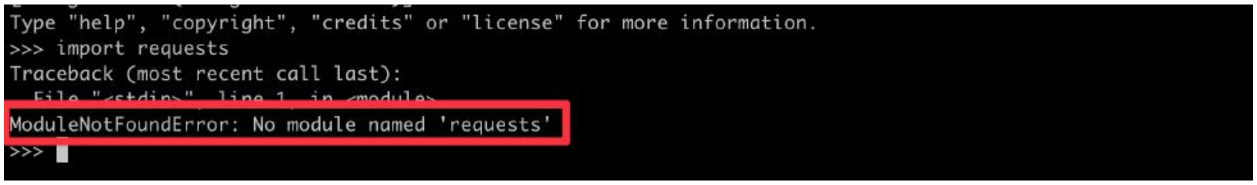
- Internal dependencies: other code that you or your organisation wrote and which you fully control, e.g. the `solver.py` file.
- Standard dependencies: code that exists as part of the standard language libraries, e.g. the `math` module.
- External dependencies: code that is written by third-party developers, e.g. the `requests` module.

In this tutorial, you'll gain more experience with all three categories of dependencies. Specifically, you'll write an NLP (natural language processing) program to analyse sentences, using spaCy[17], a third-party dependency.

Dependency management is a hugely complicated area, and there is a large ecosystem of related tools to help manage packaging and installing Python programs. We won't be covering all of the options and background, but you can read an overview of the different tools here[18].

Understanding Repl.it's magic import tool and the universal package manager

In nearly all programming environments, you have to explicitly install third-party dependencies. Let's say you wanted to use the `requests` library (which is not included in Python by default) on your local machine. If you try to import it, you would get a `ModuleNotFound` error, as shown below.



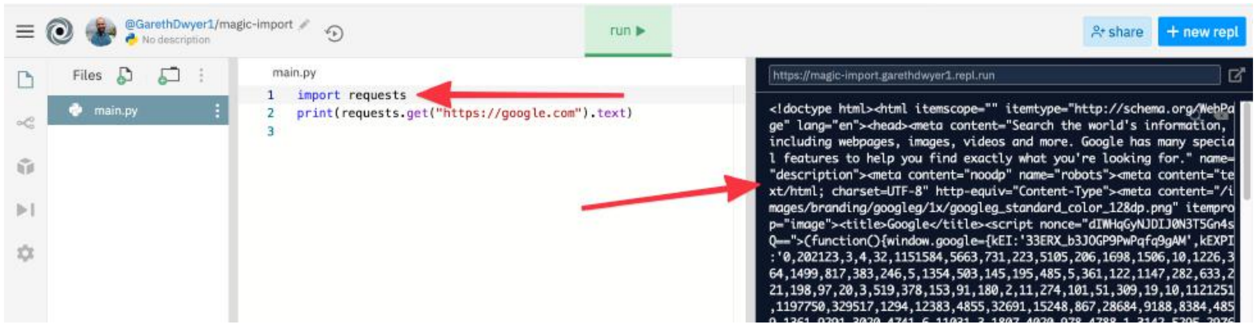
```
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
>>> 
```

Image 1: ***Trying to use a dependency without installing it***

[17]https://spacy.io/
[18]https://modelpredict.com/python-dependency-management-tools

In order to use this library, you would first have to install it using a command similar to `pip install requests`, and only then would the import statement run correctly.

Repl.it, by contrast, can often do the installation for you completely automatically, using the Universal Package Manager[19]. The moment you run the `import requests` line of code, the package manager will go find the correct package and install it, or in some cases Repl.it will even have pre-installed the package. Either way, your code will "just work".


@GarethDwyer1/magic-import
No description
run ▶
share
+ new repl
Files
main.py
https://magic-import.garethdwyer1.repl.run

```python
import requests
print(requests.get("https://google.com").text)
```

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPa
ge" lang="en"><head><meta content="Search the world's information,
including webpages, images, videos and more. Google has many specia
l features to help you find exactly what you're looking for." name=
"description"><meta content="noodp" name="robots"><meta content="te
xt/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/i
mages/branding/googleg/1x/googleg_standard_color_128dp.png" itempro
p="image"><title>Google</title><script nonce="dIWHqGyNJDIJ0N3T5Gn4s
Q==">(function(){window.google={kEI:'33ERX_b3JOGP9PwPqfq9gAM',kEXPI
:'0,202123,3,4,32,1151584,5663,731,223,5105,206,1698,1506,10,1226,3
64,1499,817,383,246,5,1354,503,145,195,485,5,361,122,1147,282,633,2
21,198,97,20,3,519,378,153,91,180,2,11,274,101,51,309,19,10,1121251
,1197750,329517,1294,12383,4855,32691,15248,867,28684,9188,8384,485
```

Image 2: ***Seamless external package use on a new repl***

This is super convenient, but sometimes you need more control. For example, you might need a specific version of a package, or the universal package manager might not be able to automatically install all of your dependencies. In these cases, you can use more advanced ways to install packages.

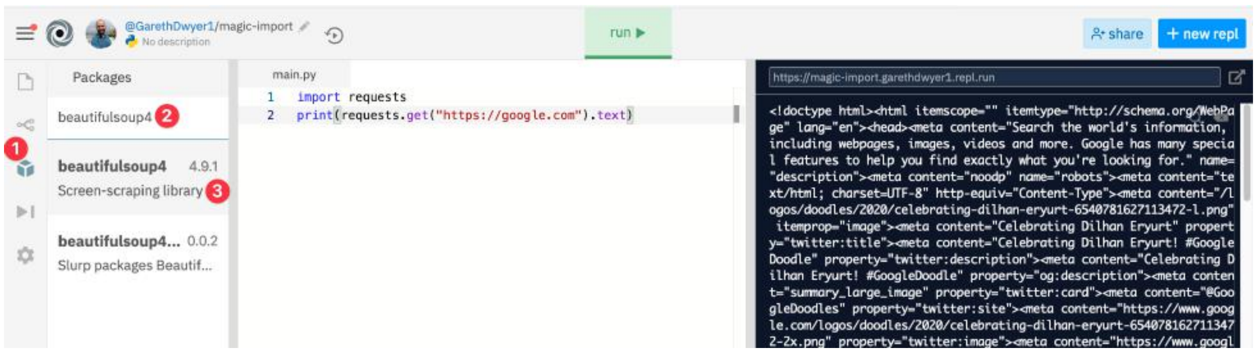
Installing packages through the GUI

If you're not sure exactly which package you need, you can use Repl.it's built-in package manager GUI to search for packages. In the example below, we are looking for a package called `beautifulsoup4`.

To use this, you need to

1. Click on the packages tab from the left toolbar.
2. Search for a package by typing in part or all of its name.
3. Select the package you want from the search results.

[19]https://blog.repl.it/upm

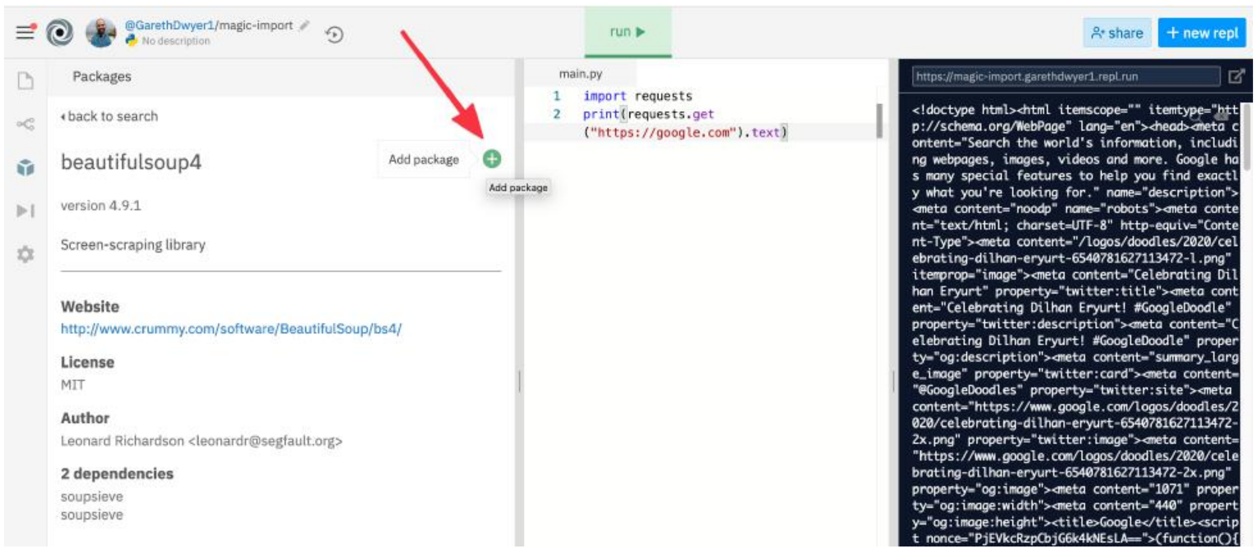


```
import requests
print(requests.get("https://google.com").text)
```

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPa
ge" lang="en"><head><meta content="Search the world's information,
including webpages, images, videos and more. Google has many specia
l features to help you find exactly what you're looking for." name=
"description"><meta content="noodp" name="robots"><meta content="te
xt/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/l
ogos/doodles/2020/celebrating-dilhan-eryurt-6540781627113472-l.png"
 itemprop="image"><meta content="Celebrating Dilhan Eryurt" propert
y="twitter:title"><meta content="Celebrating Dilhan Eryurt! #Google
Doodle" property="twitter:description"><meta content="Celebrating D
ilhan Eryurt! #GoogleDoodle" property="og:description"><meta conten
t="summary_large_image" property="twitter:card"><meta content="@Goo
gleDoodles" property="twitter:site"><meta content="https://www.goog
le.com/logos/doodles/2020/celebrating-dilhan-eryurt-654078162711347
2-2x.png" property="twitter:image"><meta content="https://www.googl
```

Image 3: *Using the GUI package manager to find a package*

This will take you to a page showing an overview and summary of the selected package. You can install it to your repl by using the + button, as shown below.



beautifulsoup4

version 4.9.1

Screen-scraping library

Website
http://www.crummy.com/software/BeautifulSoup/bs4/

License
MIT

Author
Leonard Richardson <leonardr@segfault.org>

2 dependencies
soupsieve
soupsieve

```
import requests
print(requests.get
("https://google.com").text)
```

```
<!doctype html><html itemscope="" itemtype="htt
p://schema.org/WebPage" lang="en"><head><meta c
ontent="Search the world's information, includi
ng webpages, images, videos and more. Google ha
s many special features to help you find exactl
y what you're looking for." name="description">
<meta content="noodp" name="robots"><meta conte
nt="text/html; charset=UTF-8" http-equiv="Conte
nt-Type"><meta content="/logos/doodles/2020/cel
ebrating-dilhan-eryurt-6540781627113472-l.png"
itemprop="image"><meta content="Celebrating Dil
han Eryurt" property="twitter:title"><meta cont
ent="Celebrating Dilhan Eryurt! #GoogleDoodle"
property="twitter:description"><meta content="C
elebrating Dilhan Eryurt! #GoogleDoodle" proper
ty="og:description"><meta content="summary_larg
e_image" property="twitter:card"><meta content=
"@GoogleDoodles" property="twitter:site"><meta
content="https://www.google.com/logos/doodles/2
020/celebrating-dilhan-eryurt-6540781627113472-
2x.png" property="twitter:image"><meta content=
"https://www.google.com/logos/doodles/2020/cele
brating-dilhan-eryurt-6540781627113472-2x.png"
property="og:image"><meta content="1071" proper
ty="og:image:width"><meta content="440" propert
y="og:image:height"><title>Google</title><scrip
t nonce="PjEVkcRzpCbjG6k4kNEsLA==">(function(){
```

Image 4: **Installing a package from the overview page**

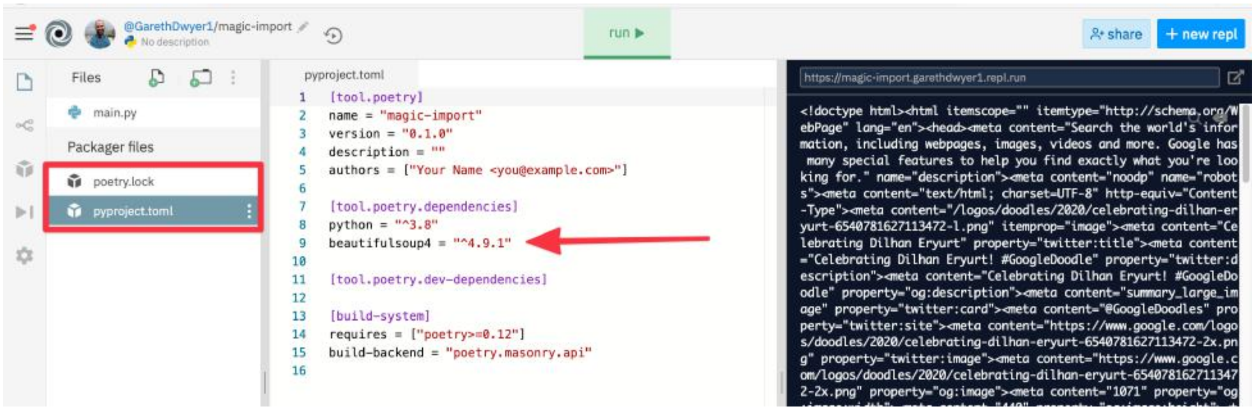
Once the package is installed, we can use it in our code. Run the example shown below to extract the "Google Search" text from the main button on the homepage.

```
import requests
from bs4 import BeautifulSoup

r = requests.get("https://google.com").text
soup = BeautifulSoup(r, "html.parser")

print([x.get("title") for x in soup.findAll("input") if x.get("title")])
```

This code uses the `requests` library to scrape the HTML from google.com and then uses the `beautifulsoup4` library to get the title of the button off the page and print it to the console.

Because `requests` is one of the most commonly used Python libraries, Repl.it probably installed it in a slightly different way from most packages. However, `beautifulsoup4` is less common and this will have been installed in the standard way using poetry[20].

If you go back to the files tab, you'll see two new files `poetry.lock` and `pyproject.toml` which were created automatically by the installer. Take a look inside the `pyproject.toml` file.



```
1   [tool.poetry]
2   name = "magic-import"
3   version = "0.1.0"
4   description = ""
5   authors = ["Your Name <you@example.com>"]
6
7   [tool.poetry.dependencies]
8   python = "^3.8"
9   beautifulsoup4 = "^4.9.1"
10
11  [tool.poetry.dev-dependencies]
12
13  [build-system]
14  requires = ["poetry>=0.12"]
15  build-backend = "poetry.masonry.api"
16
```

```
<!doctype html><html itemscope="" itemtype="http://schema.org/W
ebPage" lang="en"><head><meta content="Search the world's infor
mation, including webpages, images, videos and more. Google has
 many special features to help you find exactly what you're loo
king for." name="description"><meta content="noodp" name="robot
s"><meta content="text/html; charset=UTF-8" http-equiv="Content
-Type"><meta content="/logos/doodles/2020/celebrating-dilhan-er
yurt-6540781627113472-l.png" itemprop="image"><meta content="Ce
lebrating Dilhan Eryurt" property="twitter:title"><meta content
="Celebrating Dilhan Eryurt! #GoogleDoodle" property="twitter:d
escription"><meta content="Celebrating Dilhan Eryurt! #GoogleDo
odle" property="og:description"><meta content="summary_large_im
age" property="twitter:card"><meta content="@GoogleDoodles" pro
perty="twitter:site"><meta content="https://www.google.com/logo
s/doodles/2020/celebrating-dilhan-eryurt-6540781627113472-2x.pn
g" property="twitter:image"><meta content="https://www.google.c
om/logos/doodles/2020/celebrating-dilhan-eryurt-654078162711347
2-2x.png" property="og:image"><meta content="1071" property="og
```

Image 5: ***The `pyproject.toml` file lists all dependencies and their versions.***

In this case, line 9 says that our project relies on the `beautifulsoup4` package and needs at least version 4.9.1. If we look at the `beautifulsoup` page on PyPi[21], we'll see that the latest stable version is 4.9.1, so if this project is run in the future and there is a new version available, it will automatically use the updated package.

Building an NLP project using `spaCy`

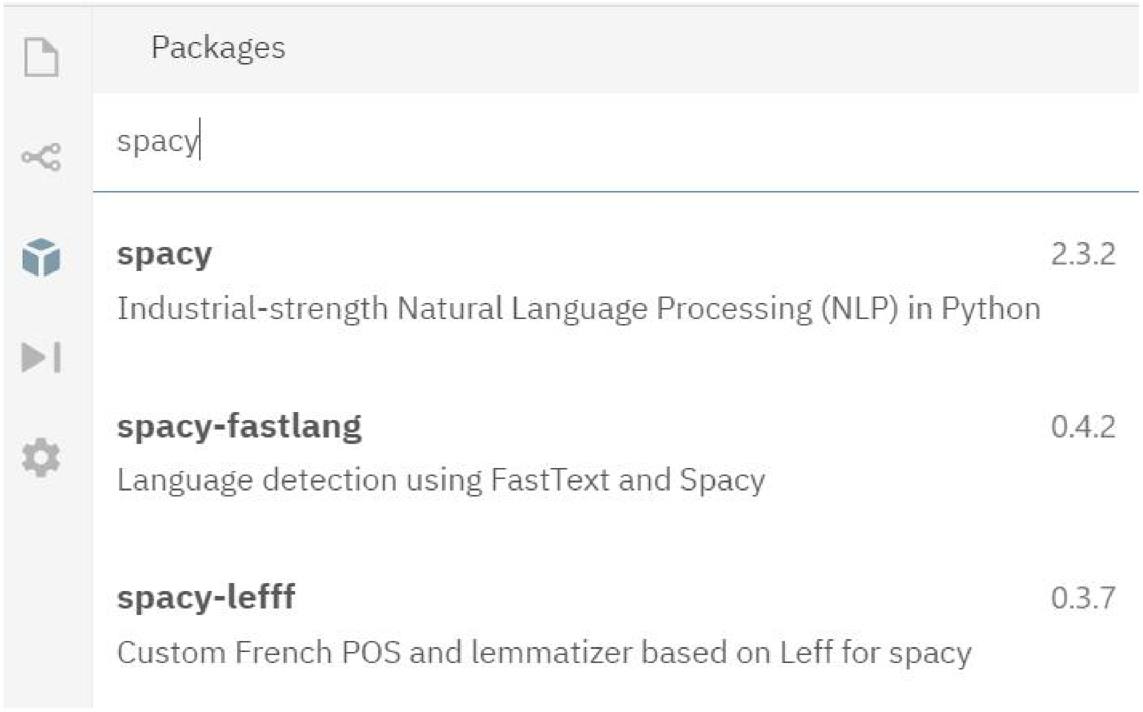
So far, we have installed packages that are easy for the Repl.it universal dependency manager to install automatically, behind the scenes. Some packages are more complicated though. `spaCy`[22], for example, is an NLP library that relies on a large external data file. When installing this library, you usually have to install this data file as a separate step.

Installing the `spaCy` language model

To get this to work on Repl.it, we'll have to manually modify the `pyproject.toml` file.

Create a new repl, `SpacyExample`, then click on the `Packages` icon and search for "spacy".

[20]https://python-poetry.org/docs/basic-usage/
[21]https://pypi.org/project/beautifulsoup4/
[22]https://spacy.io/


Packages

spacy

spacy 2.3.2
Industrial-strength Natural Language Processing (NLP) in Python

spacy-fastlang 0.4.2
Language detection using FastText and Spacy

spacy-lefff 0.3.7
Custom French POS and lemmatizer based on Leff for spacy

Image 6: ***We can access the*** `spaCy` ***package via the package index”***

Select the version at the top and hit the + button to add this package to your application. Once this is complete, head across to your `main.py` and enter the following code:

```python
import spacy
print(spacy.__version__)
```

This should output the version of `spaCy` that we are using, which means that `spaCy` has been added as a dependency correctly.

If you take a look at your `pyproject.toml` file now, you should see that it has specified `spaCy` as a dependency.

```toml
[tool.poetry]
name = "spacy-example"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]

[tool.poetry.dependencies]
python = "^3.8"
spacy = "^2.3.2"

[tool.poetry.dev-dependencies]
```

```

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

An important component of `spaCy` is a set of pretrained statistical models that support NLP. These do not come with `spaCy` by default, nor are they indexed on PyPi. One of these models is `en_core_web_sm`.

In your `main.py` file, replace your current code with the following:

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("The quick brown fox jumps over the lazy dog.")
for token in doc:
    print(token.text)
```

This code should simply break our short sentence into tokens (words), and print each one out.

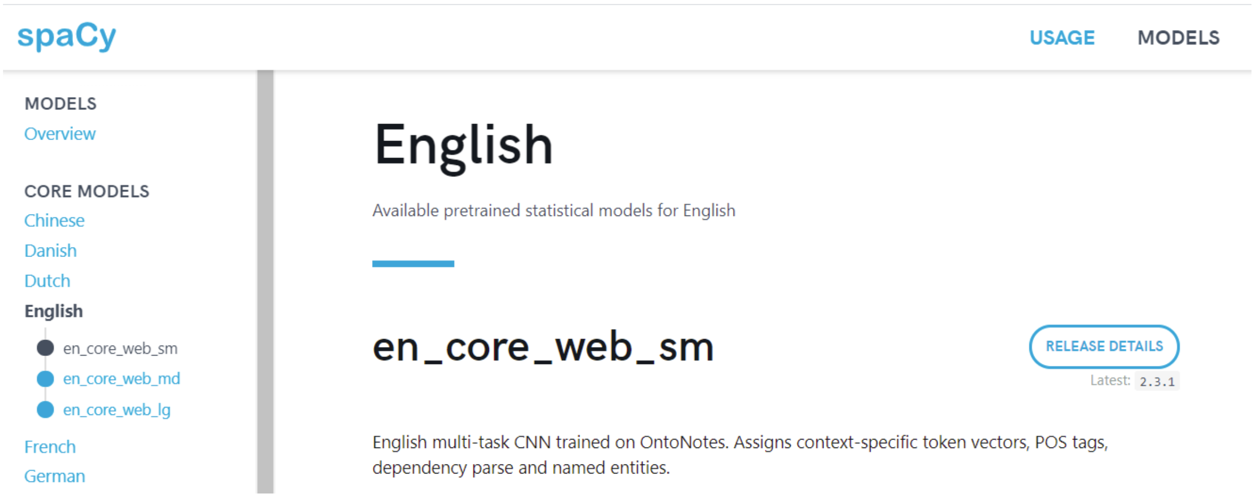
However, at this point, if you run your code you will get an error, as Python cannot find the `en_core_web_sm` model:

```
OSError: [E050] Can't find model 'en_core_web_sm'. It doesn't seem to be a shortcut \
link,
a Python package or a valid path to a data directory.
```

We will now explicitly tell our application how to access this dependency. To do this, we need to find where the model is stored online.

First, we need to find the `spaCy` documentation for this model. This can be accessed here[23].

[23] https://spacy.io/models/en



English

Available pretrained statistical models for English

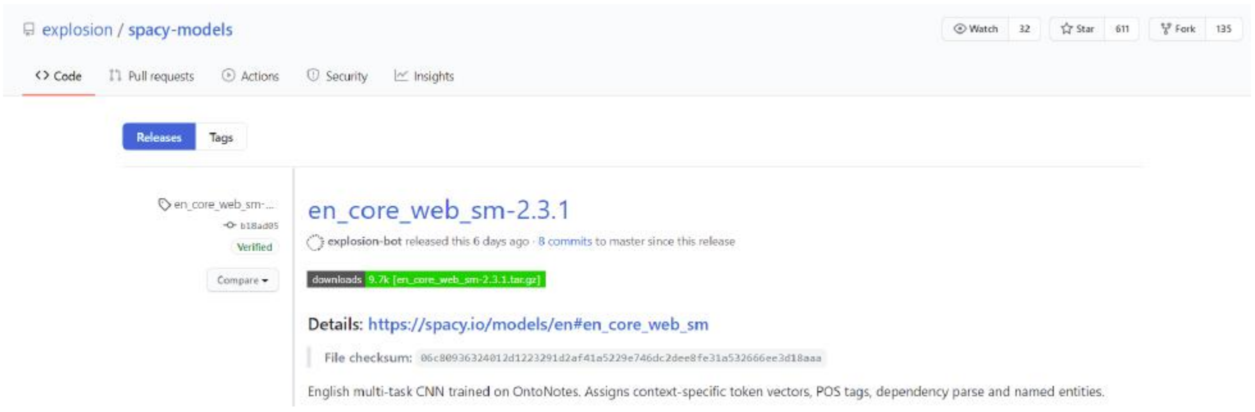
en_core_web_sm

Latest: 2.3.1

English multi-task CNN trained on OntoNotes. Assigns context-specific token vectors, POS tags, dependency parse and named entities.

Image 7: *The `spaCy` documentation gives us information about the model*

Selecting the `RELEASE DETAILS` button will guide us to where the model is stored online, on GitHub[24]. GitHub is a very common place to store code and related components online.



en_core_web_sm-2.3.1

explosion-bot released this 6 days ago · 8 commits to master since this release

downloads 9.7k [en_core_web_sm-2.3.1.tar.gz]

Details: https://spacy.io/models/en#en_core_web_sm

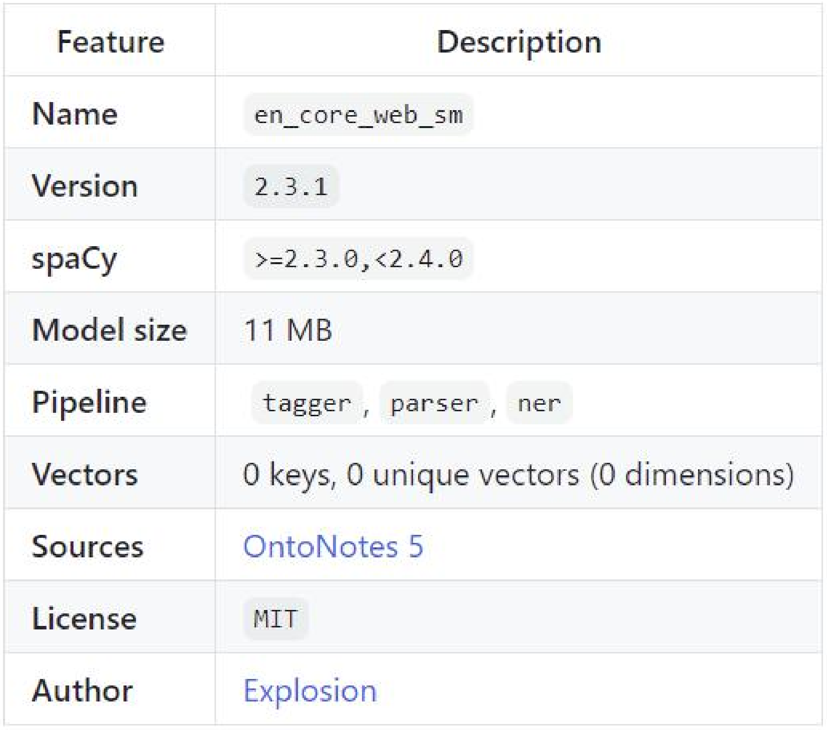
File checksum: 06c80936324012d1223291d2af41a5229e746dc2dee8fe31a532666ee3d18aaa

English multi-task CNN trained on OntoNotes. Assigns context-specific token vectors, POS tags, dependency parse and named entities.

Image 8: *This is the `en_core_web_sm` GitHub page*

The GitHub page also lets us know what version of `spaCy` is needed to make sure the model runs correctly.

[24]https://github.com/explosion/spacy-models/releases//tag/en_core_web_sm-2.3.1

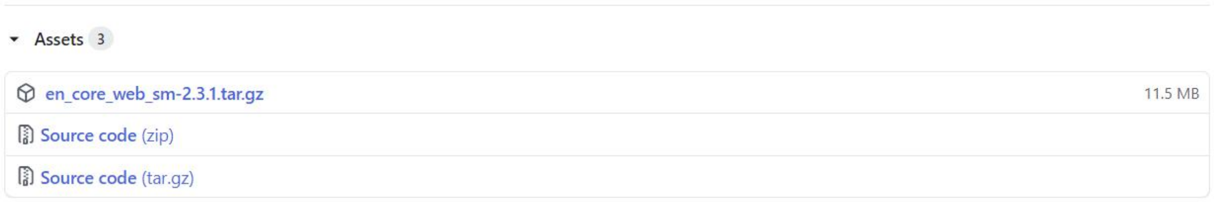


Feature	Description
Name	`en_core_web_sm`
Version	`2.3.1`
spaCy	`>=2.3.0,<2.4.0`
Model size	11 MB
Pipeline	`tagger`, `parser`, `ner`
Vectors	0 keys, 0 unique vectors (0 dimensions)
Sources	OntoNotes 5
License	`MIT`
Author	Explosion

Image 9: ***The GitHub page provides information about the requirements and features of the model***

Here we see that `spaCy` version should be greater than or equal to *2.3.0*, but less than *2.4.0*. We should make a note of this for later, so we can check that we have pinned an appropriate `spaCy` version.

If we scroll right to the bottom of the page, you will see an "Assets" section, and under this you will see the same `Package` icon we used in Repl.it with "en_core_web_sm-2.3.1.tar.gz" next to it. This is what we have been looking for: the file containing the model.



Assets 3

en_core_web_sm-2.3.1.tar.gz	11.5 MB
Source code (zip)	
Source code (tar.gz)	

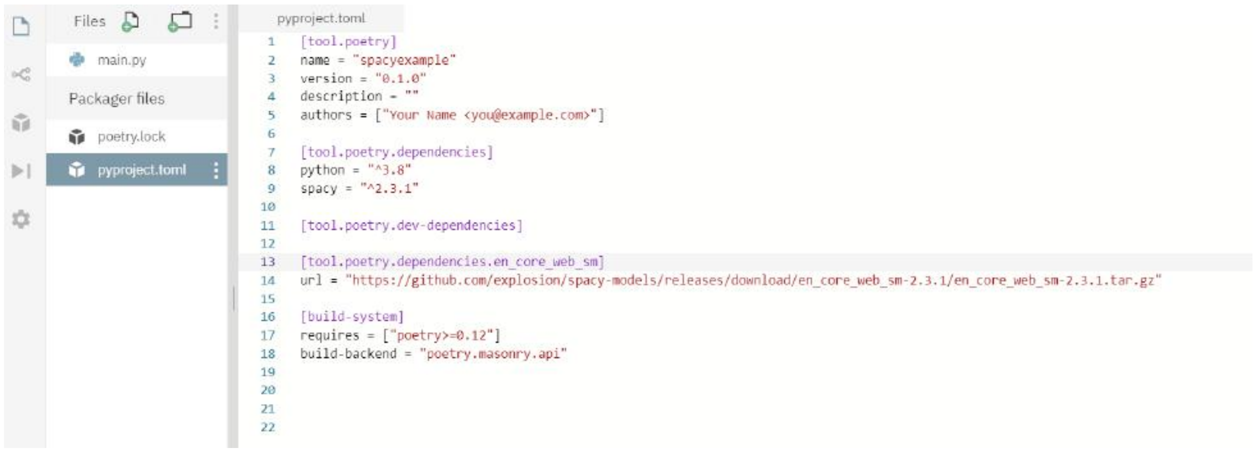
Image 10: ***The model can be found under the "Assets" heading***

Right-click on this file and select `copy link address`. We will need this shortly, as this is the URL of the file.

We now need to modify our `pyproject.toml` file in Repl.it. Open this file and add the following section to it

```
[tool.poetry.dependencies.en_core_web_sm]
url = "https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.\
3.1/en_core_web_sm-2.3.1.tar.gz"
```

The `url` should be the one that you copied from GitHub in the previous step. Your whole `pyproject.toml` file should now look like the one below.

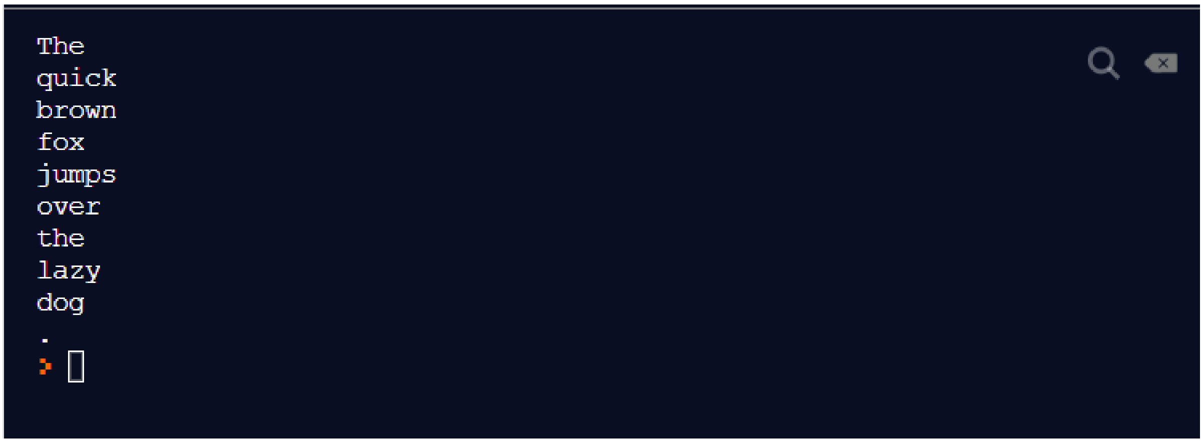


```
[tool.poetry]
name = "spacyexample"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]

[tool.poetry.dependencies]
python = "^3.8"
spacy = "^2.3.1"

[tool.poetry.dev-dependencies]

[tool.poetry.dependencies.en_core_web_sm]
url = "https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.3.1/en_core_web_sm-2.3.1.tar.gz"

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

Image 11: ***Modifying `pyproject.toml` to explicitly point to the model allows our application to find it and use it***

At this point that we should also check that we are using an appropriate version of `spaCy`. We are using version *2.3.2*, which is in the allowed range for the model release (>=2.3.0, <2.4.0) , so we do not need to modify this.

Finally, hit the `run` button. This will cause your configuration files to be updated and then will run your application. If everything has gone correctly, you should see the following in the output pane once it completes.



```
The
quick
brown
fox
jumps
over
the
lazy
dog
.
> 
```

Image 12: ***`spaCy` and the necessary components are all found as dependencies, so the application runs successfully***

Extracting names from headlines using `spaCy`

We've now seen how to install common packages like `requests` simply by importing them, how to find and install slightly more complicated packages like `beautifulsoup` using the GUI package manager, and how to manually install even more complicated packages like `spaCy` (which have their own dependencies) by manually writing sections of the `pyproject.toml` file.

Let's put everything together and use all three packages to extract people's names from today's headlines. We'll use the plaintext version of CNN at lite.cnn.com[25] as it's easier to extract text from.

Replace the code in your `main.py` with the following.

```
import spacy
import requests
from bs4 import BeautifulSoup
from collections import Counter

nlp = spacy.load("en_core_web_sm")
response = requests.get("http://lite.cnn.com/en")
soup = BeautifulSoup(response.text, "html.parser")

# https://stackoverflow.com/questions/1936466/beautifulsoup-grab-visible-webpage-text
[s.extract() for s in soup(['style', 'script', '[document]', 'head', 'title'])]
text = soup.getText()
doc = nlp(text)

names = []
for ent in doc.ents:
    if ent.label_ == "PERSON":
        names.append(ent.lemma_)

print("These people are in the headlines today")
print(Counter(names).most_common(10))
```

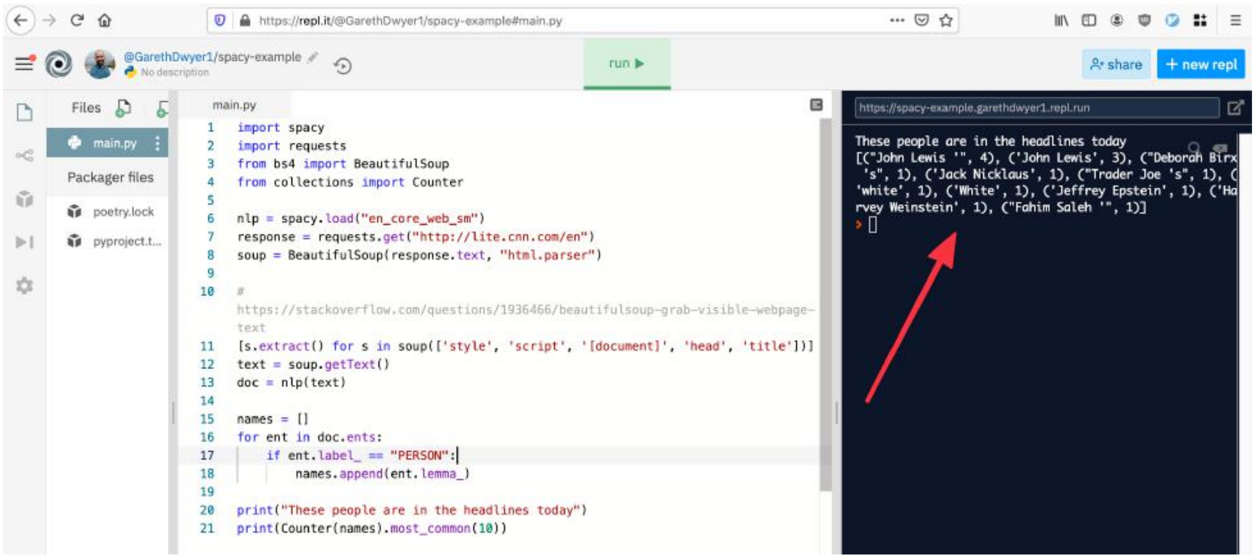
This pulls the HTML from the lite version of CNN, extracts the HTML, removes non-visible text such as CSS styles and JavaScript, and parses the resulting text using `spaCy`.

Then we loop through all of the named entities[26] that `spaCy` detects as part of its standard parse, and print out any that look like people.

If you run this code, you should see a list of people making headlines today. At the time of writing, John Lewis is mentioned in the most headlines. (Note that named entity recognition is a difficult

[25]https://lite.cnn.com
[26]https://en.wikipedia.org/wiki/Named_entity

task and here `spaCy` considers the possessive form `John Lewis'` to be a separate entity. We can see that John Lewis was mentioned a total of 7 times though.)


https://repl.it/@GarethDwyer1/spacy-example#main.py
@GarethDwyer1/spacy-example
No description
run ▶
share
+ new repl
Files
main.py
Packager files
poetry.lock
pyproject.t...
https://spacy-example.garethdwyer1.repl.run

```python
import spacy
import requests
from bs4 import BeautifulSoup
from collections import Counter

nlp = spacy.load("en_core_web_sm")
response = requests.get("http://lite.cnn.com/en")
soup = BeautifulSoup(response.text, "html.parser")

# https://stackoverflow.com/questions/1936466/beautifulsoup-grab-visible-webpage-text
[s.extract() for s in soup(['style', 'script', '[document]', 'head', 'title'])]
text = soup.getText()
doc = nlp(text)

names = []
for ent in doc.ents:
    if ent.label_ == "PERSON":
        names.append(ent.lemma_)

print("These people are in the headlines today")
print(Counter(names).most_common(10))
```

```
These people are in the headlines today
[("John Lewis '", 4), ('John Lewis', 3), ("Deborah Birx
 's", 1), ('Jack Nicklaus', 1), ("Trader Joe 's", 1), (
'white', 1), ('White', 1), ('Jeffrey Epstein', 1), ('Ha
rvey Weinstein', 1), ("Fahim Saleh '", 1)]
> 
```

Image 13: *Using `spaCy` to extract people in today's news.*

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it from https://repl.it/@GarethDwyer1/cwr-03-nlp-spacy[27].

Where next?

`spaCy` is a very powerful NLP library and it can do far more than simply extract people's names. See what other interesting insights you can automatically extract from today's news.

Now you can use the Repl.it IDE, write programs that use files, and install third-party dependencies. Next up, we'll be taking a look at doing data science with Repl.it by visualising data using `matplotlib` and `seaborn`.

[27] https://repl.it/@GarethDwyer1/cwr-03-nlp-spacy

Data science with Repl.it: Plots and graphs

So far, all the programs we have looked at have been entirely **text based**. They have taken text input and produced text output, on the console or saved to files.

While text is flexible and powerful, sometimes a picture is worth a thousand words. Especially when analysing data, you'll often want to produce plots and graphs. There are three main ways of achieving this using Repl.it.

1. Creating a front-end only project and using only JavaScript, HTML and CSS.
2. Creating a full web application with something like Flask[28], analysing the data in Python and passing the results to a front end to be visualised.
3. Using Python code only, creating windows using X[29] and rendering the plots in there.

Option 1 is great if you're OK with your users having access to all of your data, you like doing data manipulation in JavaScript, and your data set is small enough to load into a web browser. Option 2 is often the most powerful, but can be overkill if you just want a few basic plots.

Here, we'll demonstrate how to do option 3, using Python and `Matplotlib`[30].

Installing `Matplotlib` and creating a basic line plot

`Matplotlib` is a third-party library for doing all kinds of plots and graphs in Python. We can install it by using Repl.it's "magic import" functionality. `Matplotlib` is a large and powerful library with a lot of functionality, but we only need `pyplot` for now: the module for plotting.

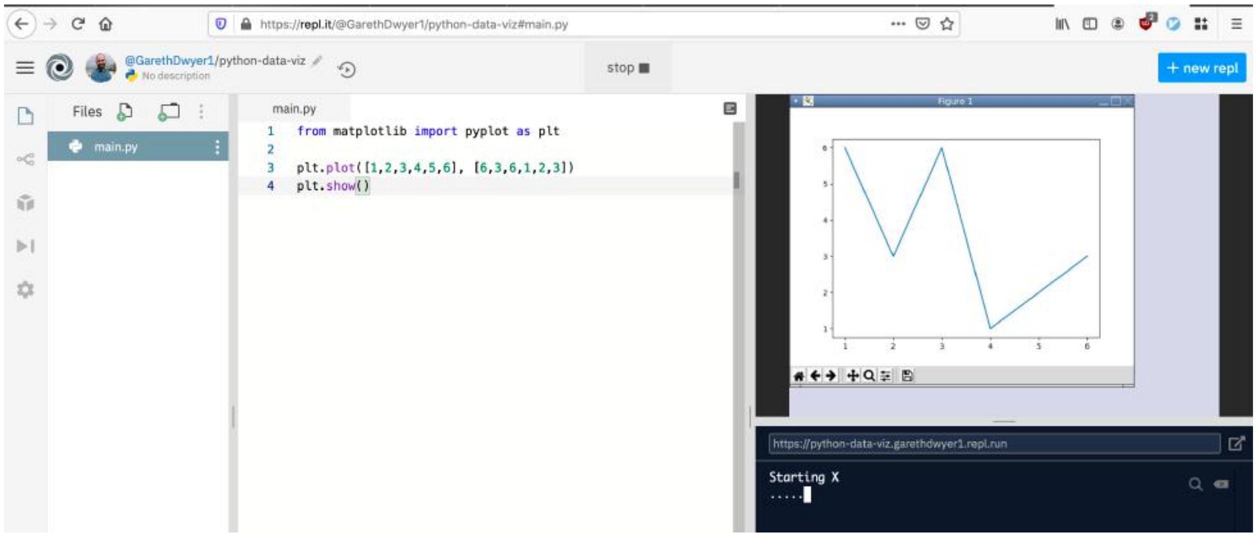
Create a new Python repl and add the following code.

```python
from matplotlib import pyplot as plt

plt.plot([1,2,3,4,5,6], [6,3,6,1,2,3])
plt.show()
```

[28] https://flask.palletsprojects.com/
[29] https://en.wikipedia.org/wiki/X_Window_System
[30] https://matplotlib.org/

There are many traditions in the Python data science world about how to import libraries. Many of the libraries have long names and get imported as easier-to-type shortcuts. You'll see that nearly all examples import `pyplot` as the shorter `plt` before using it, as we do above. We can then generate a basic line plot by passing two arrays to `plt.plot()` for X and Y values. In this example, the first point that we plot is `(1,6)` (the first value from each array). We then add all of the plotted points joined into a line graph.

Repl.it knows that it needs an `X` server to display this plot (triggered when you call `plt.show()`), so after running this code you'll see "Starting X" in the main output console and a new graphical window will appear.


https://repl.it/@GarethDwyer1/python-data-viz#main.py
@GarethDwyer1/python-data-viz
No description
stop
+ new repl
Files
main.py
Figure 1
https://python-data-viz.garethdwyer1.repl.run

```python
from matplotlib import pyplot as plt

plt.plot([1,2,3,4,5,6], [6,3,6,1,2,3])
plt.show()
```

```
Starting X
.....
```

Image 1: ***We can plot a basic line plot by passing in the X and Y values***

The `X` server is very limited compared to a full operating system GUI. Beneath the plot, you'll see some controls to pan and zoom around the image, but if you try to use them you'll see that the experience is not that smooth.

Line plots are cool, but we can do more. Let's plot a real data set.

Making a scatter plot of US cities by state

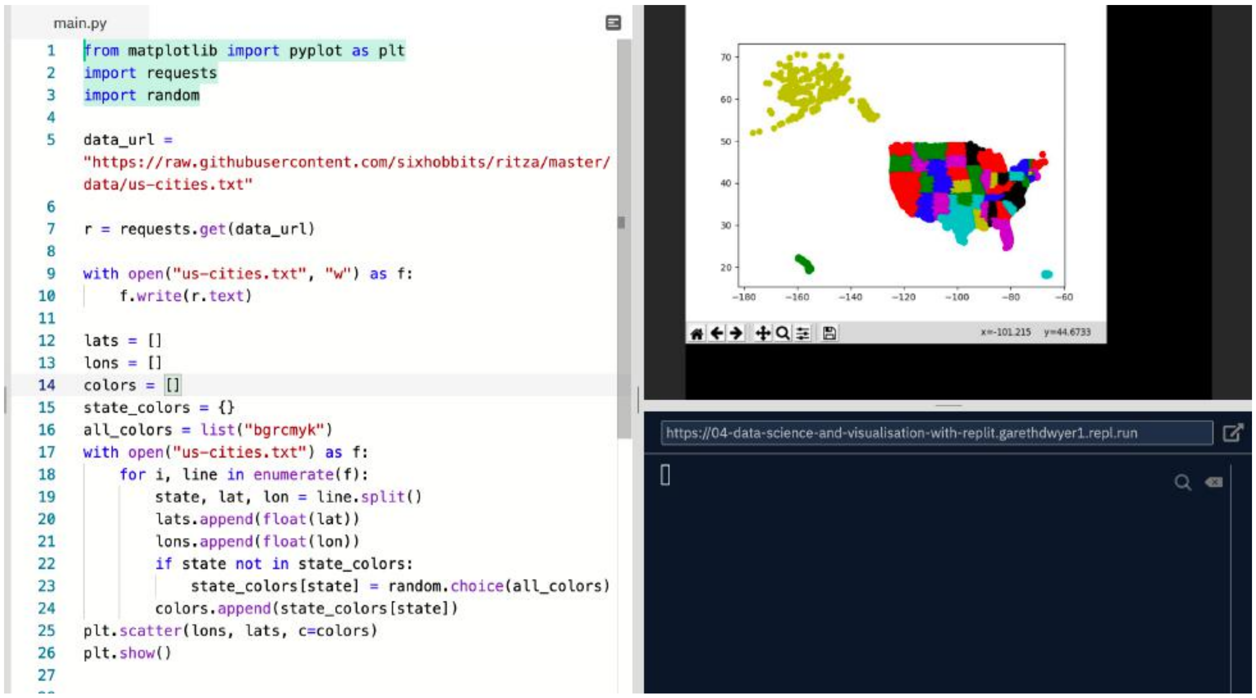
Scatter plots are often used to plot 2D data and look for correlations and other patterns. However, they can also loosely be used to plot geographical X-Y coordinates (in reality, the field of plotting geographical points is far more complicated[31]). We'll use a subset from the city data from simplemaps[32] to generate our next plot. Each row of the data set represents on city in the USA, and gives us its latitude, longitude, and two-letter state code.

To download the data and plot it, replace the code in your `main.py` file with the following.

[31]https://www.gislounge.com/what-is-gis/
[32]https://simplemaps.com/data/us-cities

```python
from matplotlib import pyplot as plt
import requests
import random

data_url = "https://raw.githubusercontent.com/sixhobbits/ritza/master/data/us-cities\
.txt"

r = requests.get(data_url)

with open("us-cities.txt", "w") as f:
    f.write(r.text)

lats = []
lons = []
colors = []
state_colors = {}

# matplotlib uses single letter shortcuts for common colors
# blue, green, red, cyan, magenta, yellow, black
all_colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

with open("us-cities.txt") as f:
    for i, line in enumerate(f):
        state, lat, lon = line.split()
        lats.append(float(lat))
        lons.append(float(lon))

        # we assign each state a random colour, but once we've picked
        # a colour we always use it for all cities in that state.
        if state not in state_colors:
            state_colors[state] = random.choice(all_colors)
        colors.append(state_colors[state])
plt.scatter(lons, lats, c=colors)
plt.show()
```

If you run this, you'll notice it takes a little bit longer than the six point plot we created before, as it now has to plot nearly 30 000 data points. Once it's done, you should see something similar to the following (though, as the colours were chosen randomly, yours might look different).


main.py
https://04-data-science-and-visualisation-with-replit.garethdwyer1.repl.run
x=-101.215 y=44.6733

```python
from matplotlib import pyplot as plt
import requests
import random

data_url = "https://raw.githubusercontent.com/sixhobbits/ritza/master/data/us-cities.txt"

r = requests.get(data_url)

with open("us-cities.txt", "w") as f:
    f.write(r.text)

lats = []
lons = []
colors = []
state_colors = {}
all_colors = list("bgrcmyk")
with open("us-cities.txt") as f:
    for i, line in enumerate(f):
        state, lat, lon = line.split()
        lats.append(float(lat))
        lons.append(float(lon))
        if state not in state_colors:
            state_colors[state] = random.choice(all_colors)
        colors.append(state_colors[state])
plt.scatter(lons, lats, c=colors)
plt.show()
```

Image 2: ***All the cities in the US plotted by state as a scatterplot***

You'll also notice that while it's recognisable as the US, the proportions are not right. Mapping a 3D sphere to a 2D plane is surprisingly difficult and there are many different ways of doing it.

More advanced plotting with `seaborn` and `pandas`

Plotting X-Y points is a good start, but in most cases you'll want to do a little bit more. `seaborn`[33] is a plotting library built on top of `Matplotlib` that makes it easier to create good-looking visualisations.

Let's do another scatter plot based on GDP and life expectancy data to see if people live longer in richer countries.

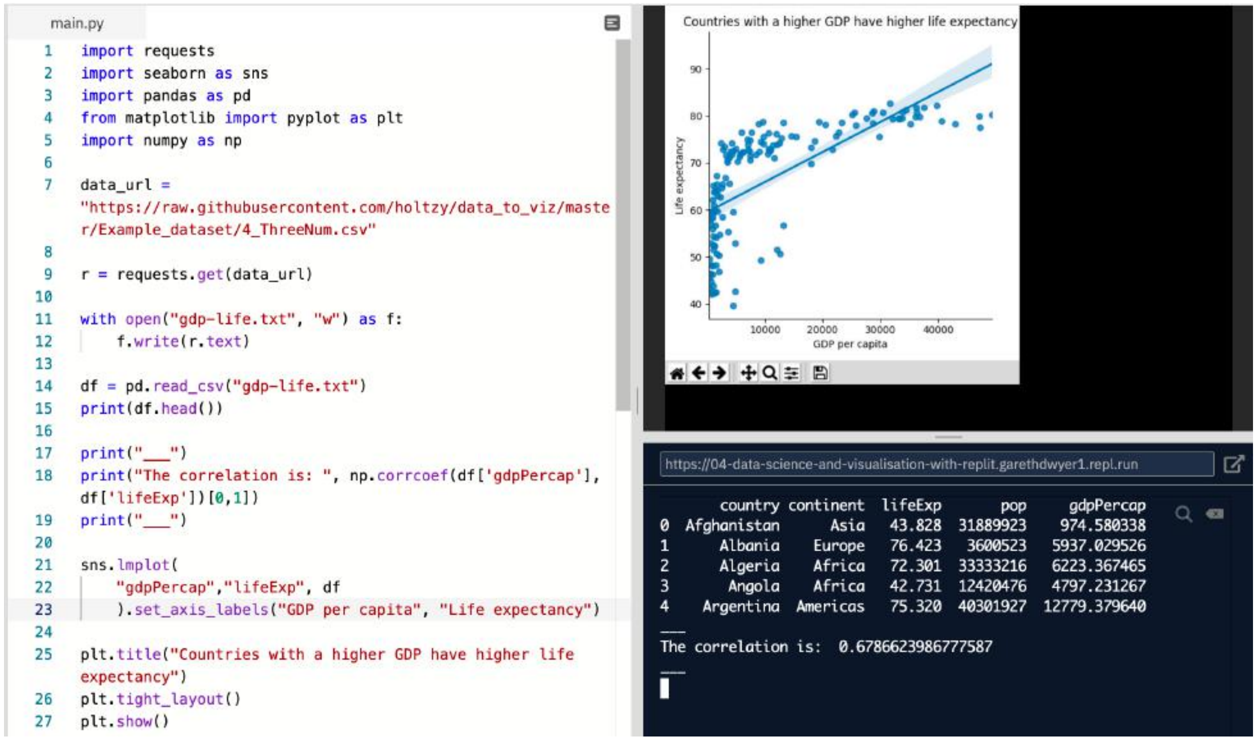
Replace the code in `main.py` with the following. Remember how we mentioned earlier that data scientists have traditions about how to import certain libraries? Here you see a few more of these "short names". We'll use `seaborn` for plotting but import it as `sns`, `pandas`[34] for reading the CSV file but import it as `pd` and `NumPy`[35] for calculating the correlation but import it as `np`.

[33]https://seaborn.pydata.org/
[34]https://pandas.pydata.org/
[35]https://numpy.org/

```
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url = "https://raw.githubusercontent.com/holtzy/data_to_viz/master/Example_data\
set/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df["gdpPercap"], df["lifeExp"])[0, 1])
print("___")

sns.lmplot("gdpPercap", "lifeExp", df).set_axis_labels(
    "Life expectancy", "GDP per capita"
)

plt.title("People live longer in richer countries")
plt.tight_layout()
plt.show()
```

If you run this, you'll see it plots each country in a similar way to our previous scatter plot, but also adds a line showing the correlation.

In the output pane below you can also see that the correlation coefficient between the two variables is 0.67 which is a fairly strong positive correlation.


main.py
Countries with a higher GDP have higher life expectancy
Life expectancy
GDP per capita
https://04-data-science-and-visualisation-with-replit.garethdwyer1.repl.run

```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url =
"https://raw.githubusercontent.com/holtzy/data_to_viz/maste
r/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'],
df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df
    ).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life
expectancy")
plt.tight_layout()
plt.show()
```

```
      country continent  lifeExp       pop     gdpPercap
0  Afghanistan      Asia   43.828  31889923    974.580338
1      Albania    Europe   76.423   3600523   5937.029526
2      Algeria    Africa   72.301  33333216   6223.367465
3       Angola    Africa   42.731  12420476   4797.231267
4    Argentina  Americas   75.320  40301927  12779.379640
___
The correlation is:  0.6786623986777587
___
```

Image 3: ***Using*** `seaborn` ***to create a scatter plot with a best fit line to see correlation***

Data science and data visualisation are huge topics, and there are dozens of Python libraries that can be used to plot data. For a good overview of all of them and their strengths and weaknesses, you should watch Jake Vanderplas's talk[36].

Saving plots to PNG files

While visualising data right after you analyse it is often useful, sometimes you need to save the figures to embed into reports. You can save your graphs by calling `plt.savefig()`. Change the last line (`plt.show()`) to

```python
plt.savefig("GDPlife.png")
```

Rerun the code. Instead of seeing the plot appear in the right-hand pane, you'll see a new file in the files pane. Clicking on it will show you the PNG file in the editing pane.

[36]https://www.youtube.com/watch?v=FytuB8nFHPQ

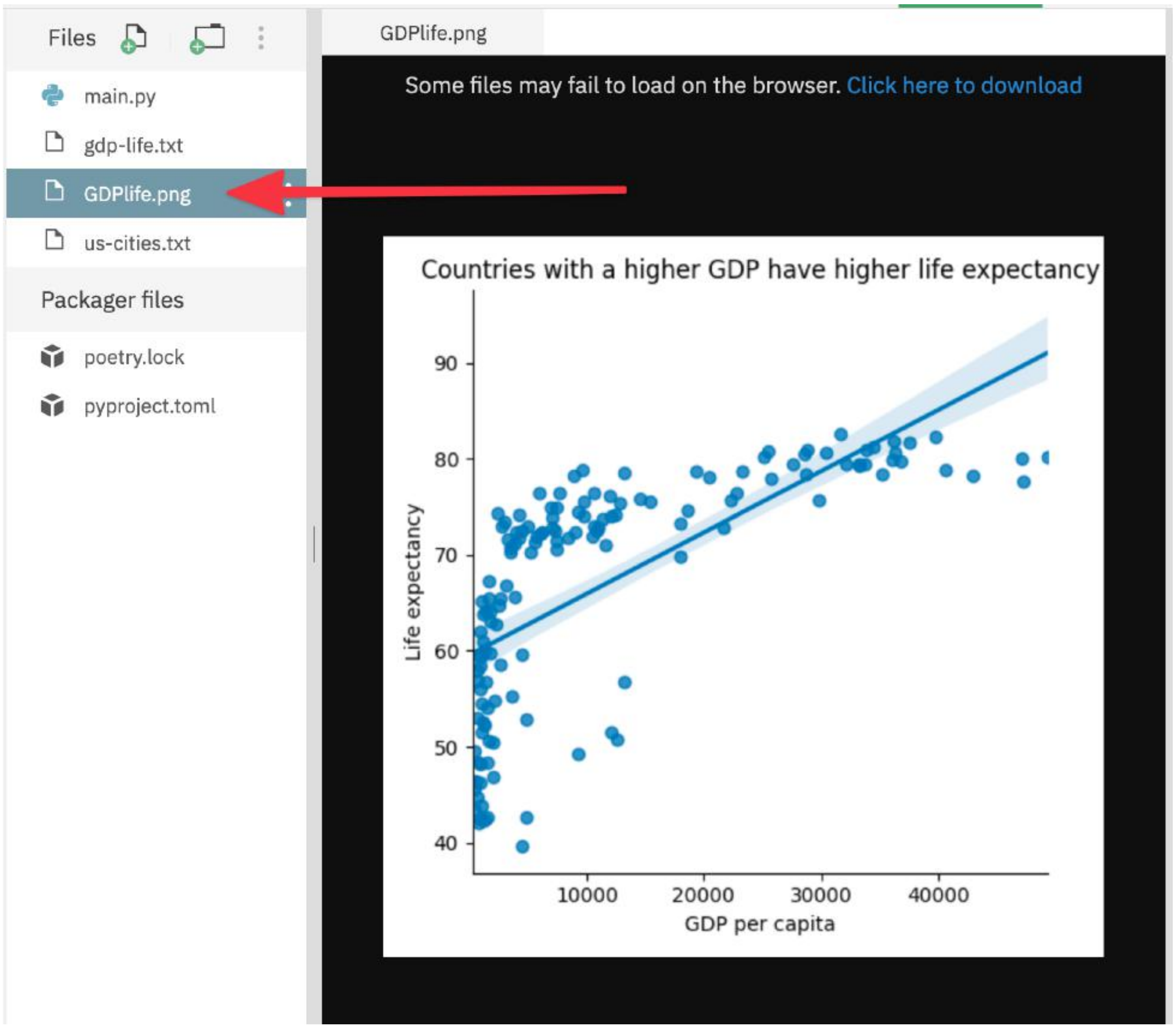


Image 4: ***Saving a PNG file for later use***

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it https://repl.it/@GarethDwyer1/cwr-04-matplotlib-plotting[37].

Where next?

You've learned how to make some basic plots using Python and Repl.it. There are millions of freely available data sets on the internet, waiting for people to explore them. You can find many of these

[37]https://repl.it/@GarethDwyer1/cwr-04-matplotlib-plotting

using Google's Dataset Search[38] service. Pick a topic that you're interested in and try to find out more about it through data visualisations.

Next up, we'll explore the mutiplayer functionality of Repl.it in more detail so that you can code collaboratively with friends or colleagues.

[38]https://datasetsearch.research.google.com/

Multiplayer: Pair programming with Repl.it

Software developers have a reputation for being loners, but they don't always code by themselves. Pair programming[39] is used by many programmers to

- Write bug-free code more efficiently (for example, one person might watch for mistakes while the other codes).
- Share knowledge (a less-experienced programmer might 'follow along' while a more experienced programmer develops something, learning from each step of the process).
- Assess expertise (if you're considering a new hire, watching them code first can be helpful to assess how good a coder they are, but coding with them allows you to also see their experience in teamwork and communication).

Pair programming intuitively sounds like it would be inefficient: after all, the two developers could instead be working on different projects simultaneously. But on top of catching more bugs, two people working together often display more creativity as well. You might think of an idea based on something your buddy said that wouldn't have come to you alone.

If you have a friend handy, work through this tutorial together to gain real pair programming experience. If you're alone, fire up two browsers (or use incognito mode) to sign into two Repl.it accounts simultaneously.

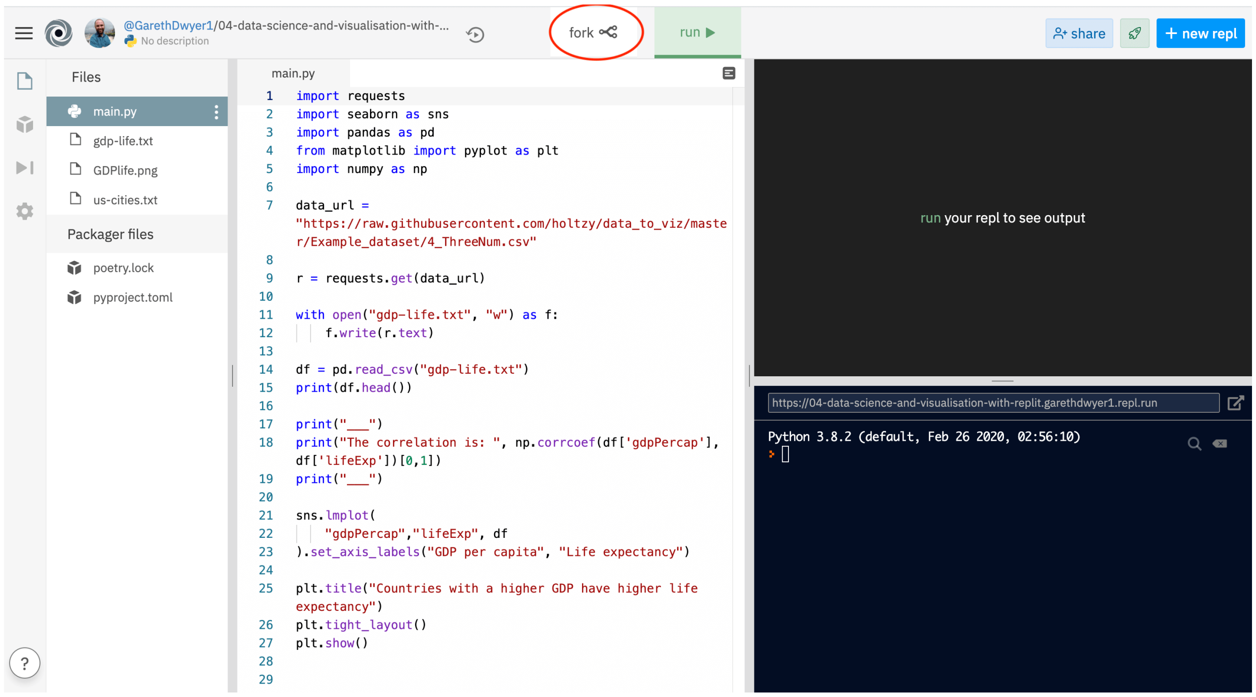
Extending our data science article using pair programming: Getting help

Imagine that you are a developer who has come across the previous tutorial on plotting and graphing[40]. You want to adapt the graphs shown a bit, but you haven't used Python much, so you decide to ask your friend for help.

In this scenario, you are "@Lean3Viljoen94" and the friend that you're asking for help is "@GarethDwyer1".

Start by forking the data science repl[41] and making sure that you can run it.

[39]https://en.wikipedia.org/wiki/Pair_programming
[40]http://www.codewithrepl.it/04-data-science-and-visualisation-with-repl-it.html
[41]https://repl.it/@GarethDwyer1/04-data-science-and-visualisation-with-replit



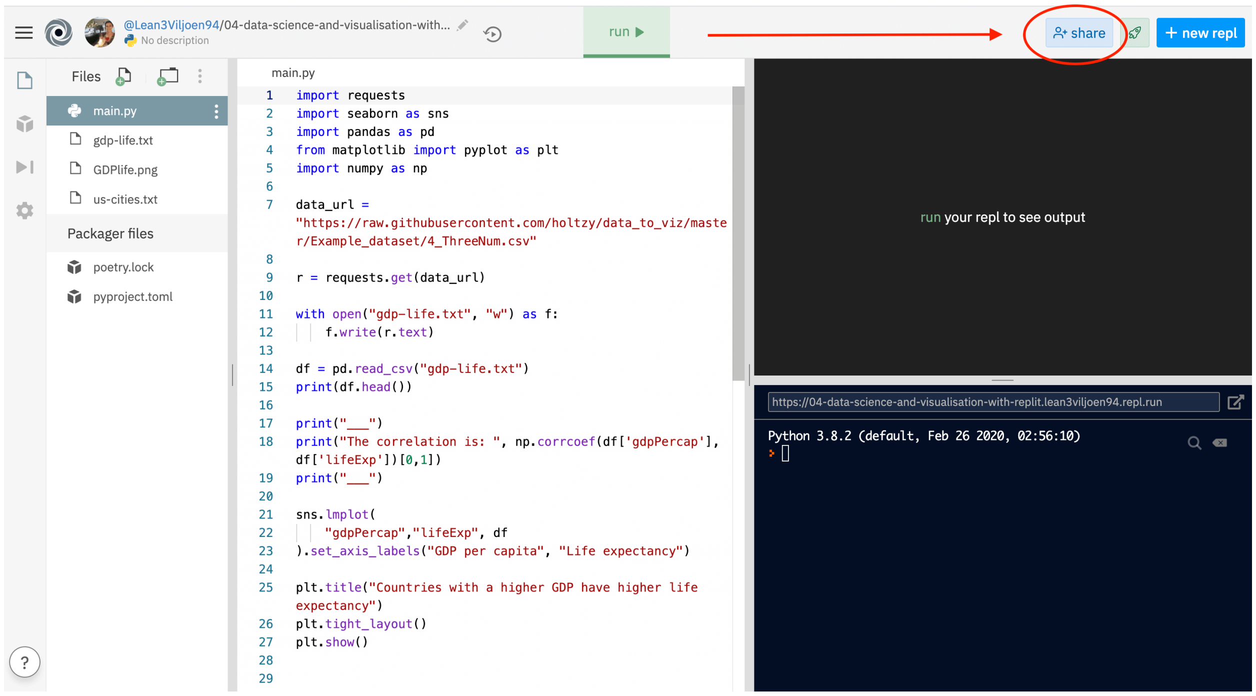
```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url = "https://raw.githubusercontent.com/holtzy/data_to_viz/master/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'], df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df
).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life expectancy")
plt.tight_layout()
plt.show()
```

```
https://04-data-science-and-visualisation-with-replit.garethdwyer1.repl.run
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Image 1: *Forking another user's project*

Now from your own fork, press the `share` button, as shown below.



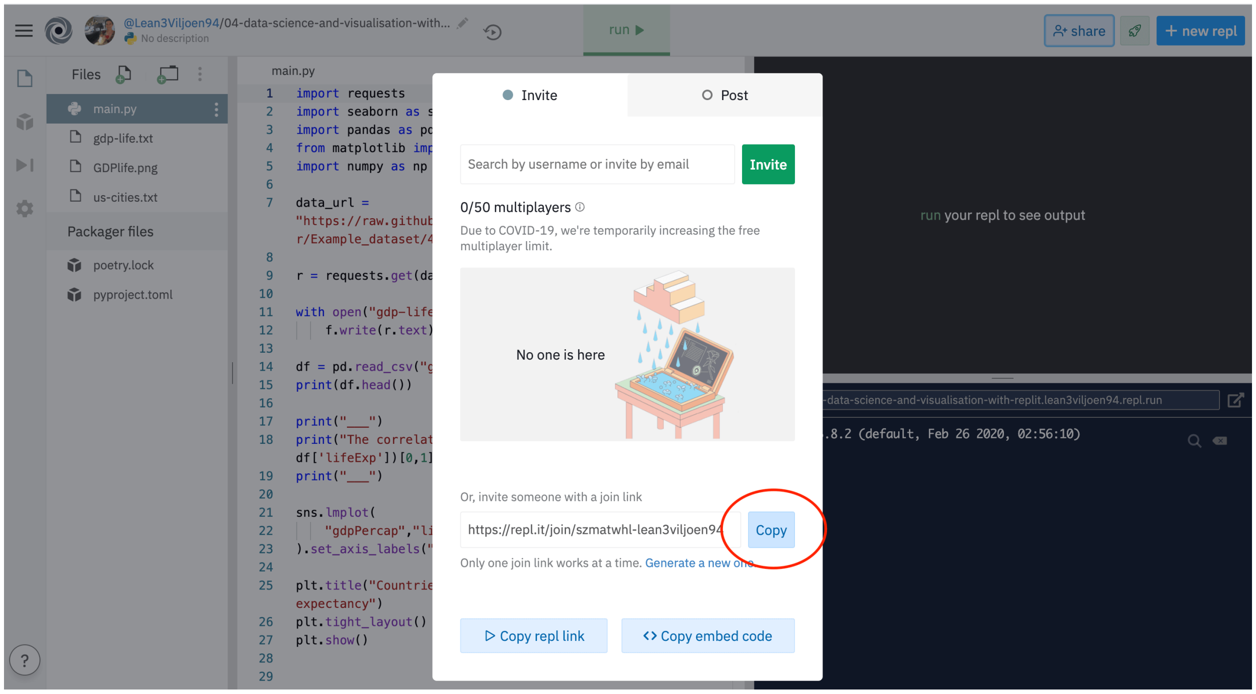
```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url = "https://raw.githubusercontent.com/holtzy/data_to_viz/master/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'], df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df
).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life expectancy")
plt.tight_layout()
plt.show()
```

```
https://04-data-science-and-visualisation-with-replit.lean3viljoen94.repl.run
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Image 2: *Sharing your project with another user*

Copy the invite link, and note that this is different from the normal link to your repl. If you copy the link from your URL bar, you can give people read access to your repl, but by copying the invite link you'll give them <u>write</u> access.



```
import requests
import seaborn as s
import pandas as pc
from matplotlib imp
import numpy as np

data_url =
"https://raw.githut
r/Example_dataset/4

r = requests.get(da

with open("gdp-life
    f.write(r.text)

df = pd.read_csv("g
print(df.head())

print("___")
print("The correlat
df['lifeExp'])[0,1]
print("___")

sns.lmplot(
    "gdpPercap","li
).set_axis_labels("

plt.title("Countrie
expectancy")
plt.tight_layout()
plt.show()
```

Due to COVID-19, we're temporarily increasing the free multiplayer limit.

Or, invite someone with a join link

https://repl.it/join/szmatwhl-lean3viljoen94

Only one join link works at a time. Generate a new one

```
.8.2 (default, Feb 26 2020, 02:56:10)
```

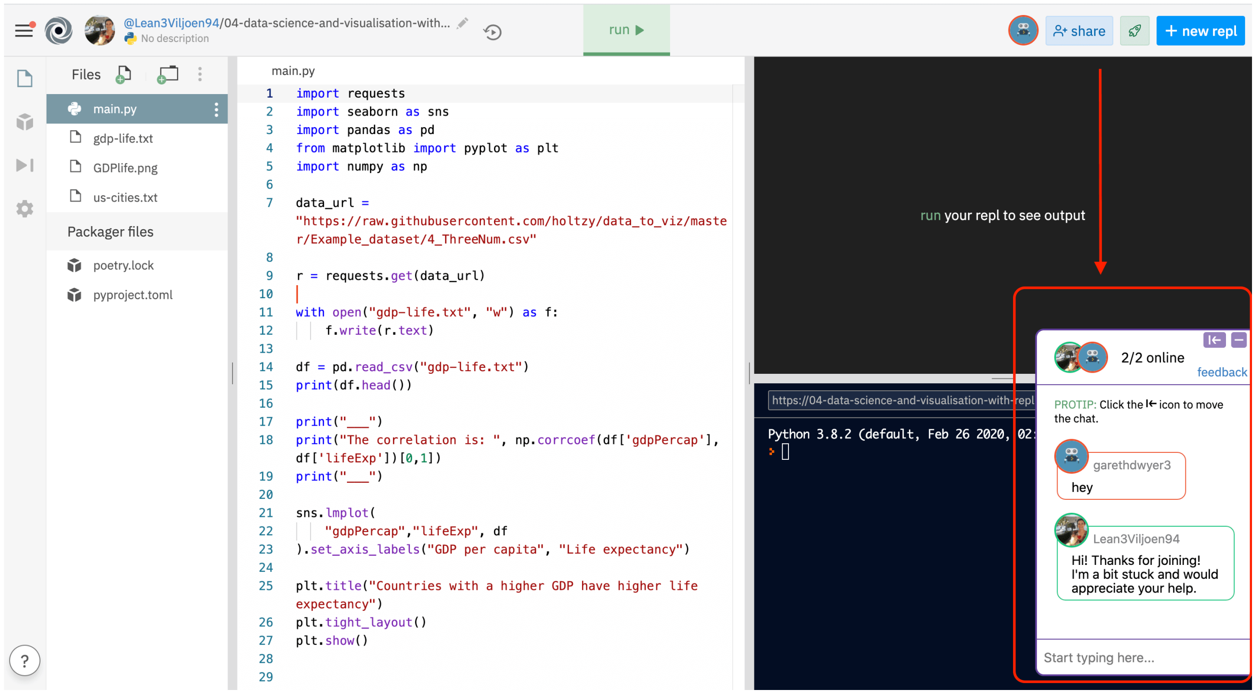
Image 3: ***Sharing options modal***

If you knew your friend's Repl.it username or the email associated with the Repl.it account, you could instead use the `Invite` box at the top. Share the link with your friend and wait for them to join.

As soon as they do, you will see that a chat box pops up in the bottom right corner. Their profile picture or letter will be at the top of the chat box, so you can always know who is currently active.

Remember, you forked the repl in a previous step, so you are the owner of this fork and the "host" of this multiplayer session. If you invite multiple people and then leave, they can continue collaborating without you, but they won't be able to rejoin if the host is no longer in the session.

You can use the team chat feature, as shown below.



```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url =
"https://raw.githubusercontent.com/holtzy/data_to_viz/master/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'],
df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df
).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life
expectancy")
plt.tight_layout()
plt.show()
```

```
https://04-data-science-and-visualisation-with-repl
Python 3.8.2 (default, Feb 26 2020, 02:
>
```

PROTIP: Click the ⇤ icon to move the chat.

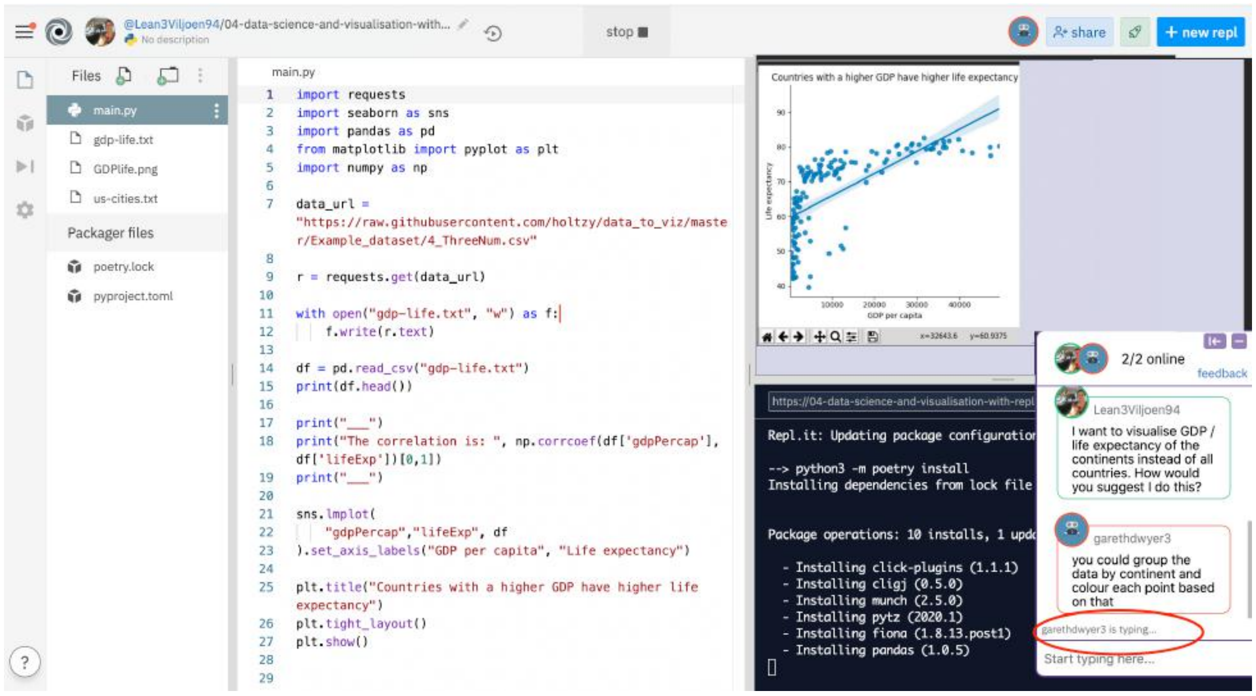
garethdwyer3: hey

Lean3Viljoen94: Hi! Thanks for joining! I'm a bit stuck and would appreciate your help.

Image 4: ***Starting chat with another user***

In the previous tutorial, we looked at GDP by country. Imagine that you are now interested in how this is broken down by continent too. You still want to plot each country as a separate data point, but you want them in different colours, one for each continent. You're not sure how to do this, so you ask for help.

You can see a typing indicator to help decide if you should wait around for a reply or go make coffee.



main.py

```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url =
"https://raw.githubusercontent.com/holtzy/data_to_viz/maste
r/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'],
df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df
).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life
expectancy")
plt.tight_layout()
plt.show()
```

```
https://04-data-science-and-visualisation-with-repl

Repl.it: Updating package configuratio

--> python3 -m poetry install
Installing dependencies from lock file


Package operations: 10 installs, 1 upd

  - Installing click-plugins (1.1.1)
  - Installing cligj (0.5.0)
  - Installing munch (2.5.0)
  - Installing pytz (2020.1)
  - Installing fiona (1.8.13.post1)
  - Installing pandas (1.0.5)
```

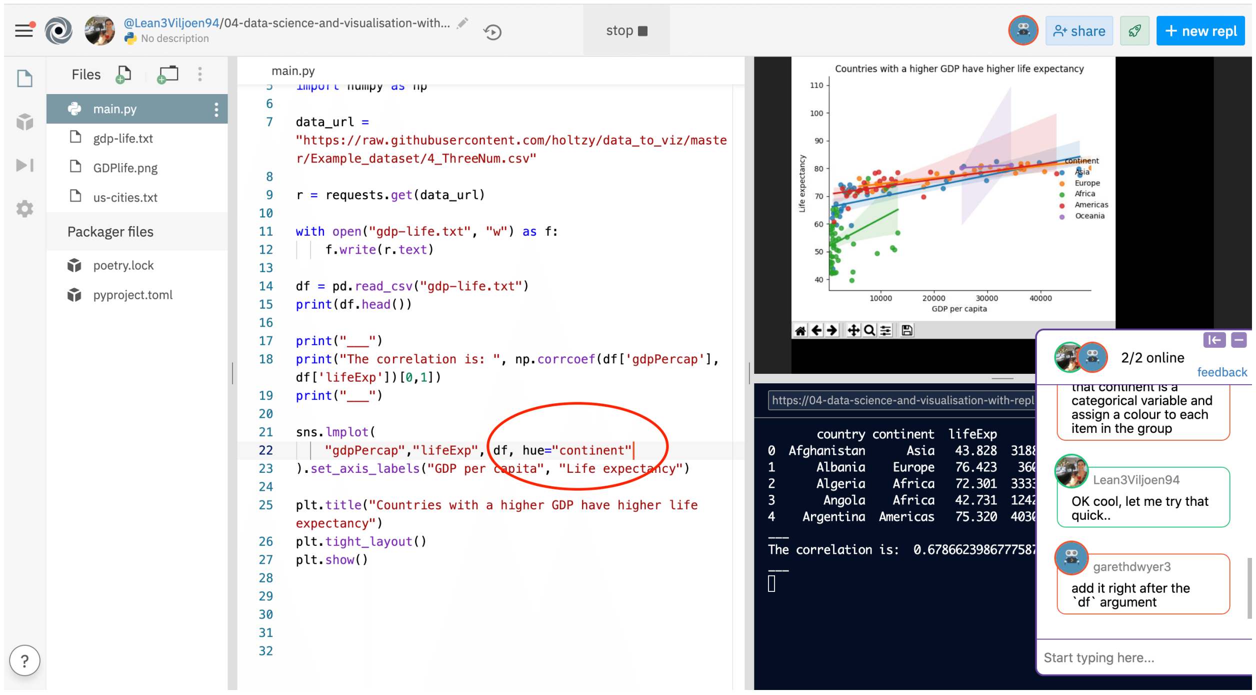
Lean3Viljoen94: I want to visualise GDP / life expectancy of the continents instead of all countries. How would you suggest I do this?

garethdwyer3: you could group the data by continent and colour each point based on that

garethdwyer3 is typing...

Image 5: ***Chat box showing that user is typing***

Your friend tells you about the `hue` argument and points out that you already have this data in the `continent` column in your data frame. You add `hue="continent"` to the graph and re-run it, but it doesn't quite work out how you expect.



```
import numpy as np

data_url =
"https://raw.githubusercontent.com/holtzy/data_to_viz/maste
r/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'],
df['lifeExp'])[0,1])
print("___")

sns.lmplot(
    "gdpPercap","lifeExp", df, hue="continent"
).set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life
expectancy")
plt.tight_layout()
plt.show()
```

```
https://04-data-science-and-visualisation-with-repl

      country continent  lifeExp
0  Afghanistan      Asia   43.828  3188
1      Albania    Europe   76.423   360
2      Algeria    Africa   72.301  3333
3       Angola    Africa   42.731  1242
4    Argentina  Americas   75.320  4030
___
The correlation is:  0.6786623986777587
___
```

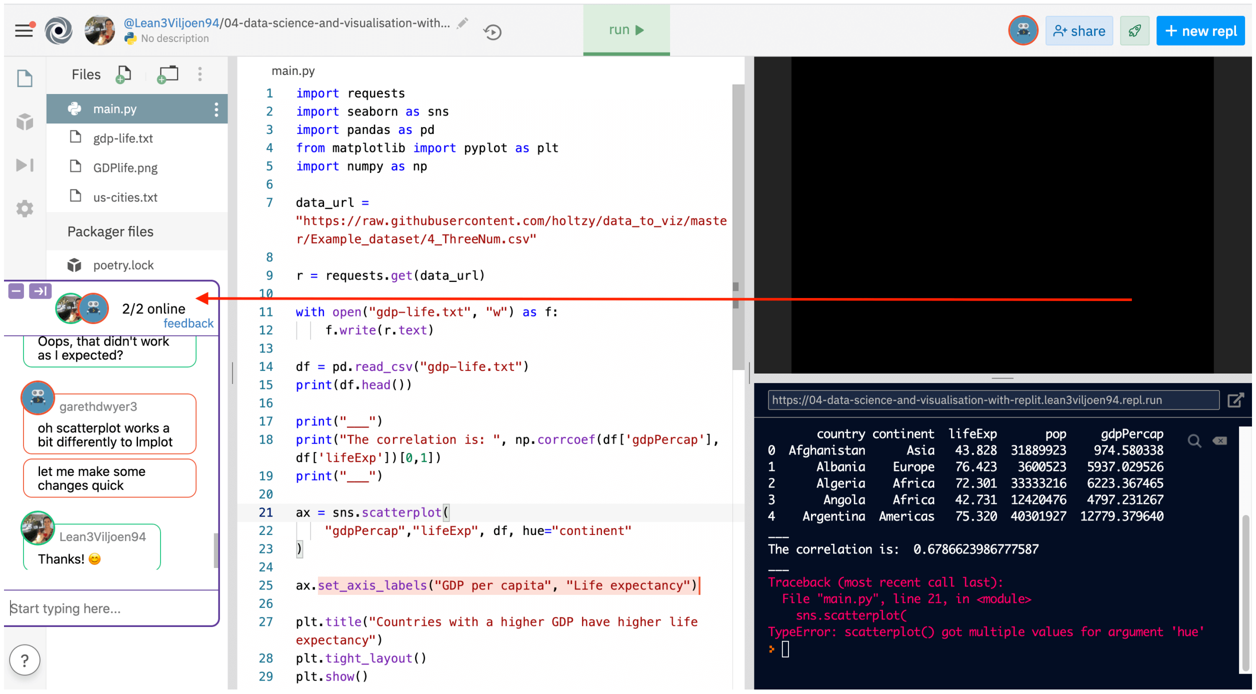
that continent is a categorical variable and assign a colour to each item in the group

Lean3Viljoen94
OK cool, let me try that quick..

garethdwyer3
add it right after the `df` argument

Image 6: *Changing the plot from grouping data by country to grouping by continent*

Your friend suggests maybe a scatter plot without the correlation line might look better, but when you try that it results in an error. The error message is hidden by the chat box, so you move it to the other side of the screen.



Oops, that didn't work as I expected?

garethdwyer3

oh scatterplot works a bit differently to lmplot

let me make some changes quick

Lean3Viljoen94

Thanks! 😊

main.py

```python
import requests
import seaborn as sns
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np

data_url =
"https://raw.githubusercontent.com/holtzy/data_to_viz/maste
r/Example_dataset/4_ThreeNum.csv"

r = requests.get(data_url)

with open("gdp-life.txt", "w") as f:
    f.write(r.text)

df = pd.read_csv("gdp-life.txt")
print(df.head())

print("___")
print("The correlation is: ", np.corrcoef(df['gdpPercap'],
df['lifeExp'])[0,1])
print("___")

ax = sns.scatterplot(
    "gdpPercap","lifeExp", df, hue="continent"
)

ax.set_axis_labels("GDP per capita", "Life expectancy")

plt.title("Countries with a higher GDP have higher life
expectancy")
plt.tight_layout()
plt.show()
```

https://04-data-science-and-visualisation-with-replit.lean3viljoen94.repl.run

```
       country continent  lifeExp       pop     gdpPercap
0  Afghanistan      Asia   43.828  31889923    974.580338
1      Albania    Europe   76.423   3600523   5937.029526
2      Algeria    Africa   72.301  33333216   6223.367465
3       Angola    Africa   42.731  12420476   4797.231267
4    Argentina  Americas   75.320  40301927  12779.379640
___
The correlation is:  0.6786623986777587
___
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    sns.scatterplot(
TypeError: scatterplot() got multiple values for argument 'hue'
```

Image 7: *You can move the chat box to the left of the IDE to see errors better*

This is getting a bit more complicated than you bargained for. Sometimes showing is easier than telling, so your friend starts editing the code directly instead of telling you how to do so using chat. The code

```python
sns.scatterplot(
    "gdpPercap", "lifeExp", df, hue="continent"
).set_axis_labels("GDP per capita", "Life expectancy")
```

changes to

```python
ax = sns.scatterplot(`
    "gdpPercap", "lifeExp", df, hue="continent"
)
ax.set(xlabel="GDP per capita", ylabel="Life expectancy")
```

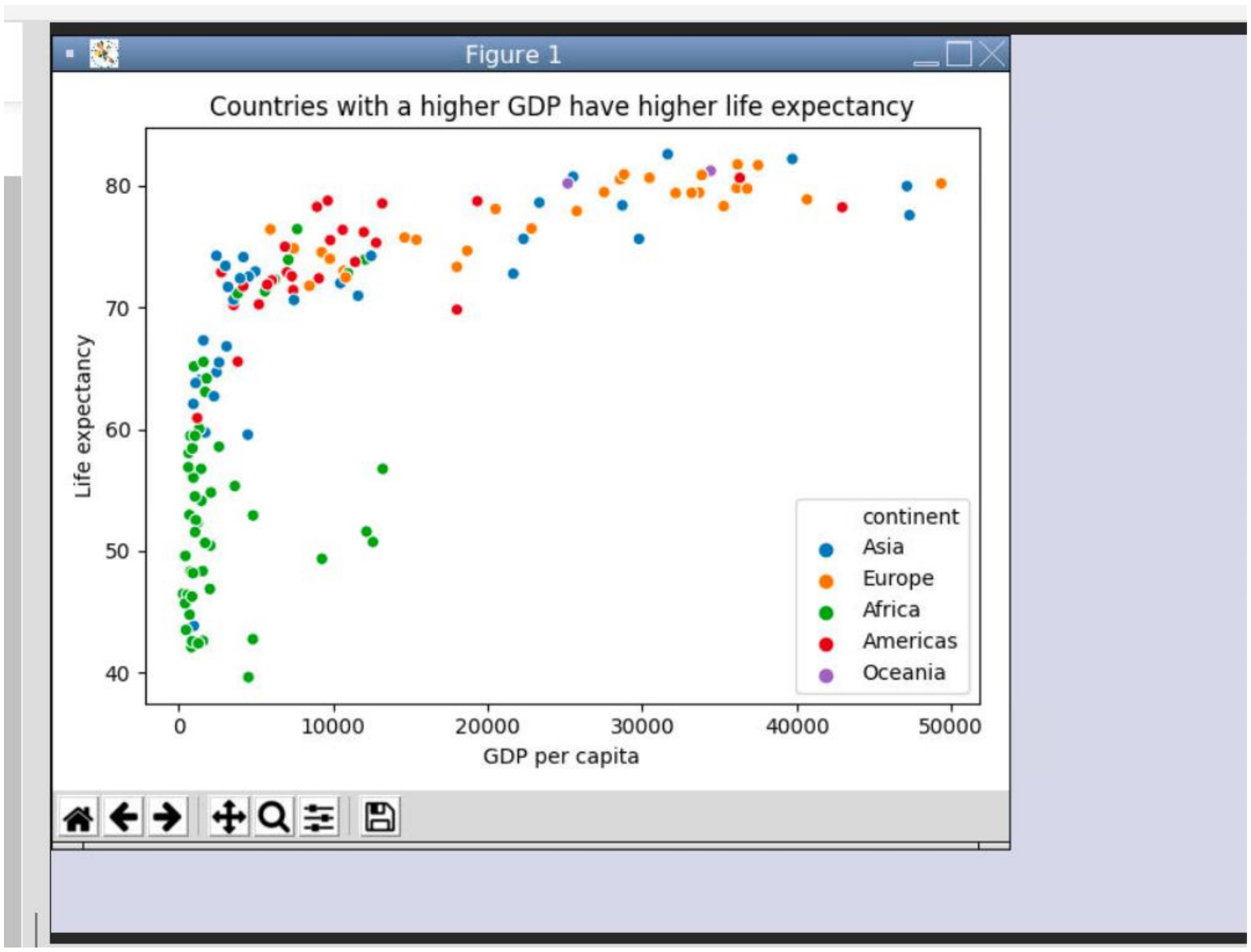


Image 8: ***In our new plot, we can see that African countries tend to have low life expectency and low GDP, but the correlation looks weaker for the other continents***

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it from https://repl.it/@GarethDwyer1/cwr-05-multiplayer[42].

Where next?

Getting help on a single file in a program is only one use for multiplayer, but there are many scenarios where it can be useful to see your teammates' changes in real time. For example, if you're a back-end developer you could work closely with a front-end developer, ironing out any issues with data communication between the back- and front-end in real time, instead of waiting for multiple iterations of several days.

[42]https://repl.it/@GarethDwyer1/cwr-05-multiplayer

That brings us to the end of part 1 of this series and you should now be familiar with all of the basic features of Repl.it.

In part 2, we'll cover more advanced features, such as running projects from GitHub, storing secrets securely, and productivity hacks.

Repl.it and GitHub: Using and contributing to open-source projects

You've probably heard of GitHub[43], which hosts millions of coding projects that you can use or learn from.

In this tutorial, we'll see how to:

- Import open-source projects from GitHub to Repl.it so that we can run them or modify them
- Integrate your own GitHub account with your Repl.it account so that you can work on your private projects
- Push changes back to open-source projects as pull requests.

We'll use a basic Flask `hello world` app for the demonstrations. You can use this same project to follow along, or pick any other project on GitHub that interests you.

Importing a project from GitHub and running it on Repl.it

We'll use the Flask[44] application available at https://github.com/ritza-co/flask-hello-world[45] for demonstration purposes. To import it into Repl.it, press the `+ new repl` button, switch to the "Import From GitHub" tab, and paste in the GitHub URL, as shown below.

[43]https://github.com
[44]https://flask.palletsprojects.com/en/1.1.x/
[45]https://github.com/ritza-co/flask-hello-world

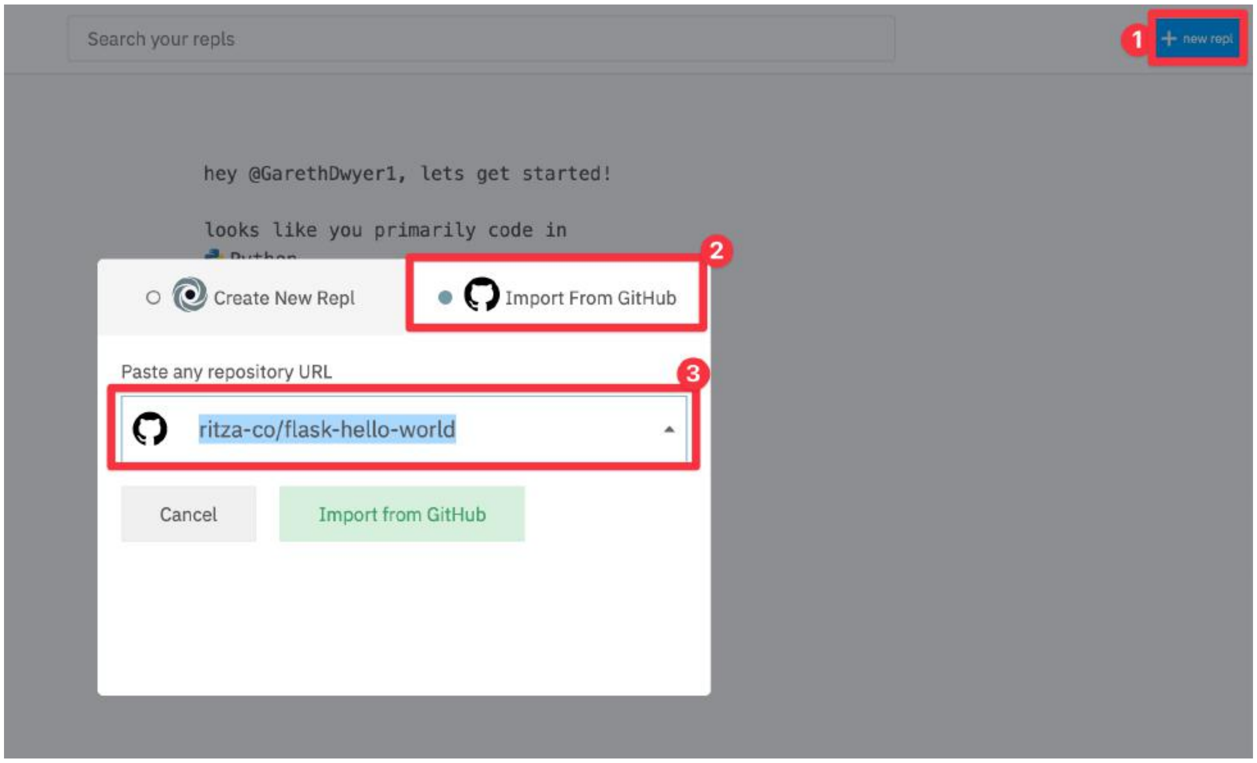
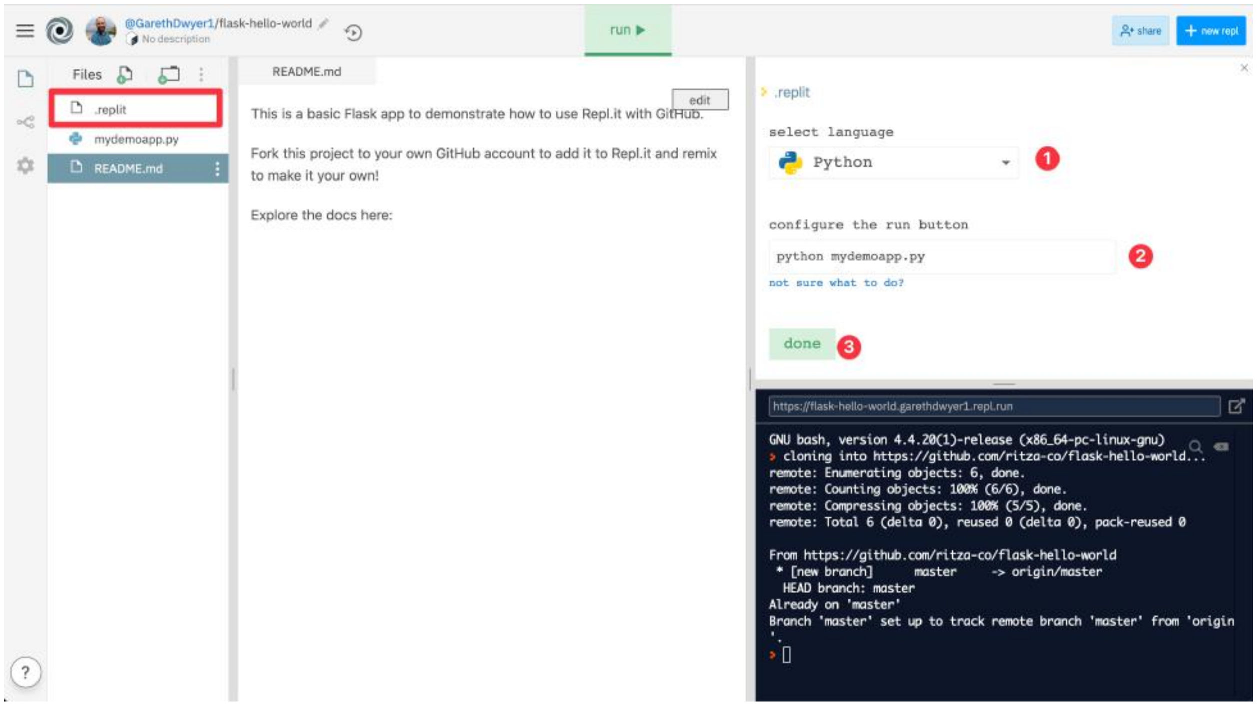


Image 1: ***Importing a repository from GitHub to Repl.it.***

Press the green `Import from GitHub` button and you'll see Repl.it clone the repository and turn it into a repl. In all of our previous projects, we used the `main.py` file that Repl.it automatically creates for all new Python projects, and which it runs automatically when you press the `run` button. Note how in this GitHub project, we have no `main.py` file, and our code is instead in `mydemoapp.py`. Therefore, Repl.it will need some help from you to define how to run the project. This is configured through another special file named `.replit`. Because there was no `main.py` file, Repl.it automatically created this file and will prompt you to configure it.



README.md

This is a basic Flask app to demonstrate how to use Repl.it with GitHub.

Fork this project to your own GitHub account to add it to Repl.it and remix to make it your own!

Explore the docs here:

.replit

select language: Python

configure the run button: python mydemoapp.py

not sure what to do?

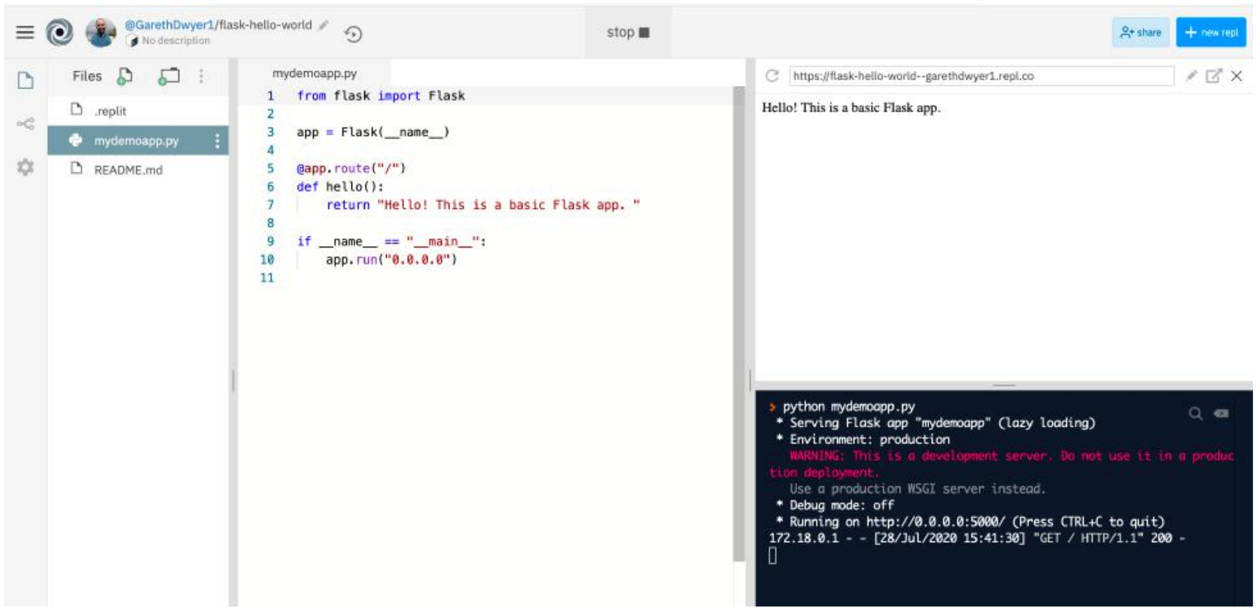
```
https://flask-hello-world.garethdwyer1.repl.run

GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
> cloning into https://github.com/ritza-co/flask-hello-world...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0

From https://github.com/ritza-co/flask-hello-world
 * [new branch]      master     -> origin/master
  HEAD branch: master
Already on 'master'
Branch 'master' set up to track remote branch 'master' from 'origin
'.
> 
```

Image 2: ***Adding a `.replit` file to indicate how the project should be run.***

Select the language (Python) from the first dropdown and type `python mydemoapp.py` in the "configure the run button" input. Every time you press the `run` button, Repl.it will execute the command given here. If you prefer, you can also edit the `.replit` file directly. If you click on it, you'll see it now contains the following configuration, which matches what we provided through the GUI panel.

```
language = "python3"
run = "python mydemoapp.py"
```

If you hit the `run` button, you should see the app start. As you can see, the web application is very basic: all it can do is display a welcome message. If the configuration panel doesn't pop up automatically, you can manually create a file called `.replit` and add the configuration above to get the same result.


@GarethDwyer1/flask-hello-world
No description
stop
share
new repl
Files
.replit
mydemoapp.py
README.md
https://flask-hello-world--garethdwyer1.repl.co

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello! This is a basic Flask app. "

if __name__ == "__main__":
    app.run("0.0.0.0")
```

Hello! This is a basic Flask app.

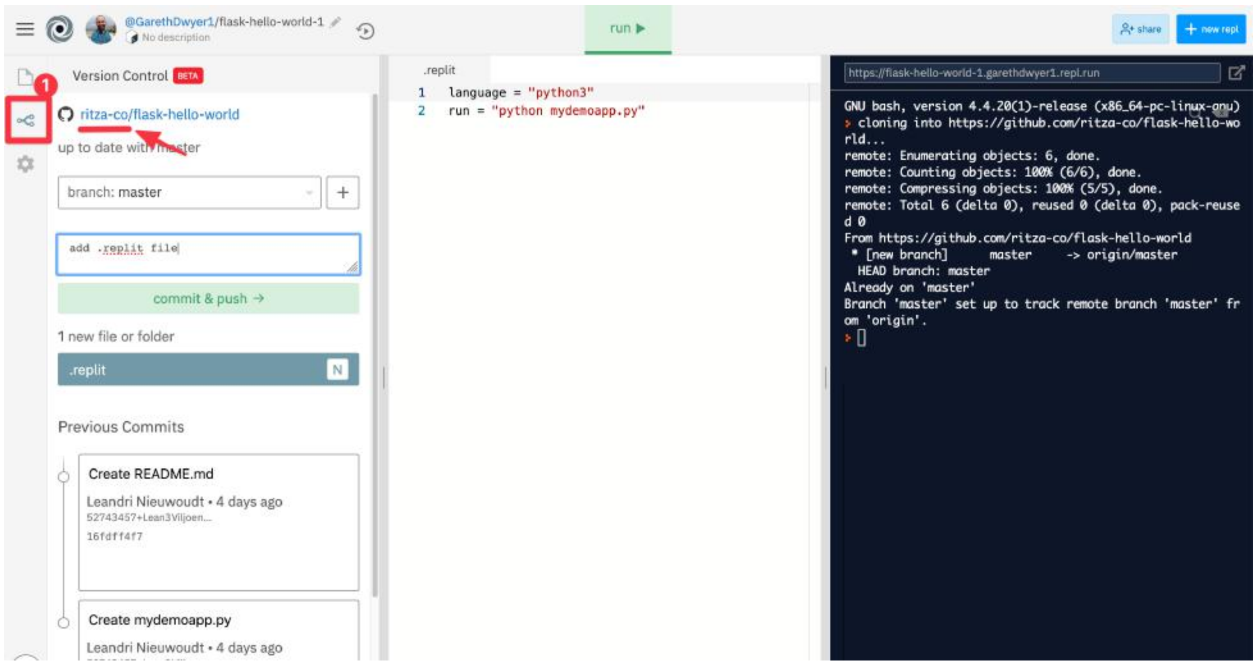
```
> python mydemoapp.py
 * Serving Flask app "mydemoapp" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a produc
tion deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
172.18.0.1 - - [28/Jul/2020 15:41:30] "GET / HTTP/1.1" 200 -
```

Image 3: ***Running the Flask application on Repl.it.***

Some GitHub projects are very large and complicated, and you might not be able to run everything you need directly on Repl.it, but in many cases it just works. Open-source projects can be read and run by anyone, but still have restrictions on who can push changes to them. Next we'll improve this project and request that the owner merges our changes into the original.

Looking at the version control panel in Repl.it

Repl.it includes a version control tab which shows you information about the GitHub repository and in some cases allows you to push your changes made in Repl.it back to GitHub.



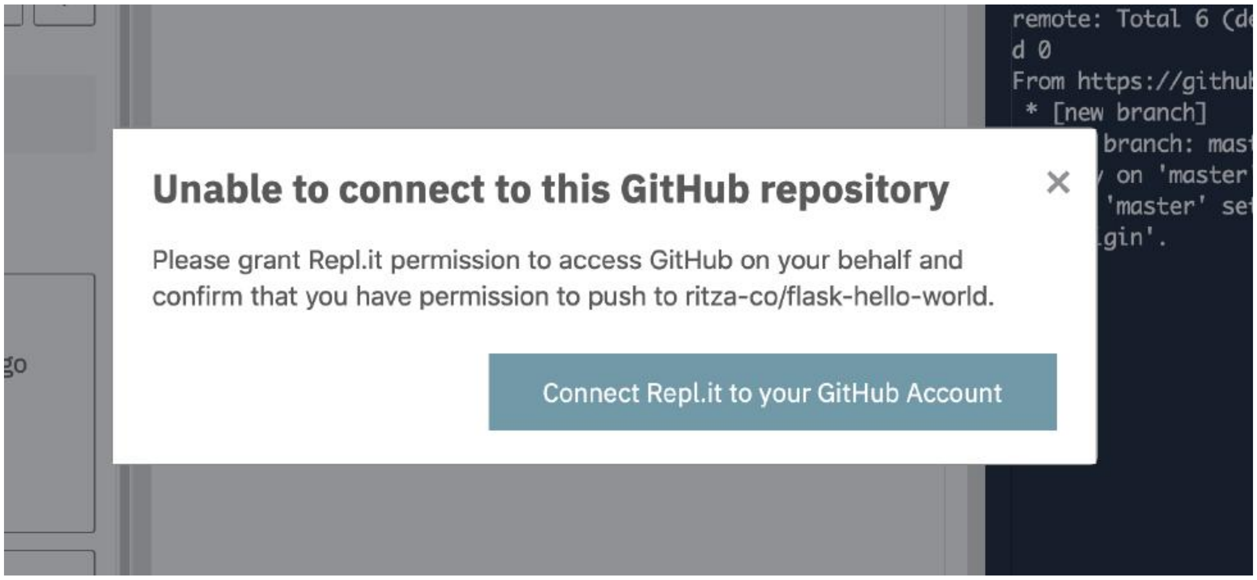
```
language = "python3"
run = "python mydemoapp.py"
```

```
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
> cloning into https://github.com/ritza-co/flask-hello-wo
rld...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reuse
d 0
From https://github.com/ritza-co/flask-hello-world
 * [new branch]      master     -> origin/master
  HEAD branch: master
Already on 'master'
Branch 'master' set up to track remote branch 'master' fr
om 'origin'.
> 
```

Image 4: ***Viewing details about the repository in the version control tab.***

If you select this version control tab from the menu on the left, you'll see a summary of the linked repository. Note that it's already figured out what changes we've made, and it shows that the `.replit` file is new. It would be nice for other people who use this repository with GitHub to have the file automatically, so we might want to push the changes we made back to GitHub.

Note that the owner of the repository is `ritza-co` though, so you won't have write permissions for this repository. If you press the `commit & push` button, you'll see an error as shown below.


Unable to connect to this GitHub repository

Please grant Repl.it permission to access GitHub on your behalf and confirm that you have permission to push to ritza-co/flask-hello-world.

Image 5: ***You need permission to push to repositories on GitHub.***

Forking the project to your own GitHub account

Usually when contributing to open-source projects, you'll first create a "fork" of the original project. This means that you make your own version of the project and, as it's yours, you can make any changes to it that you want. If you think these changes would be useful to others too and are an obvious improvement over the original project, you can make a "pull request", which asks the owner of the original project to merge your changes into the main canonical project.

Create an account on github.com[46] or log in to your existing one and navigate back to the original project (https://github.com/ritza-co/flask-hello-world). In the top right corner, press the `Fork` button to create a copy of the project in your own GitHub account.

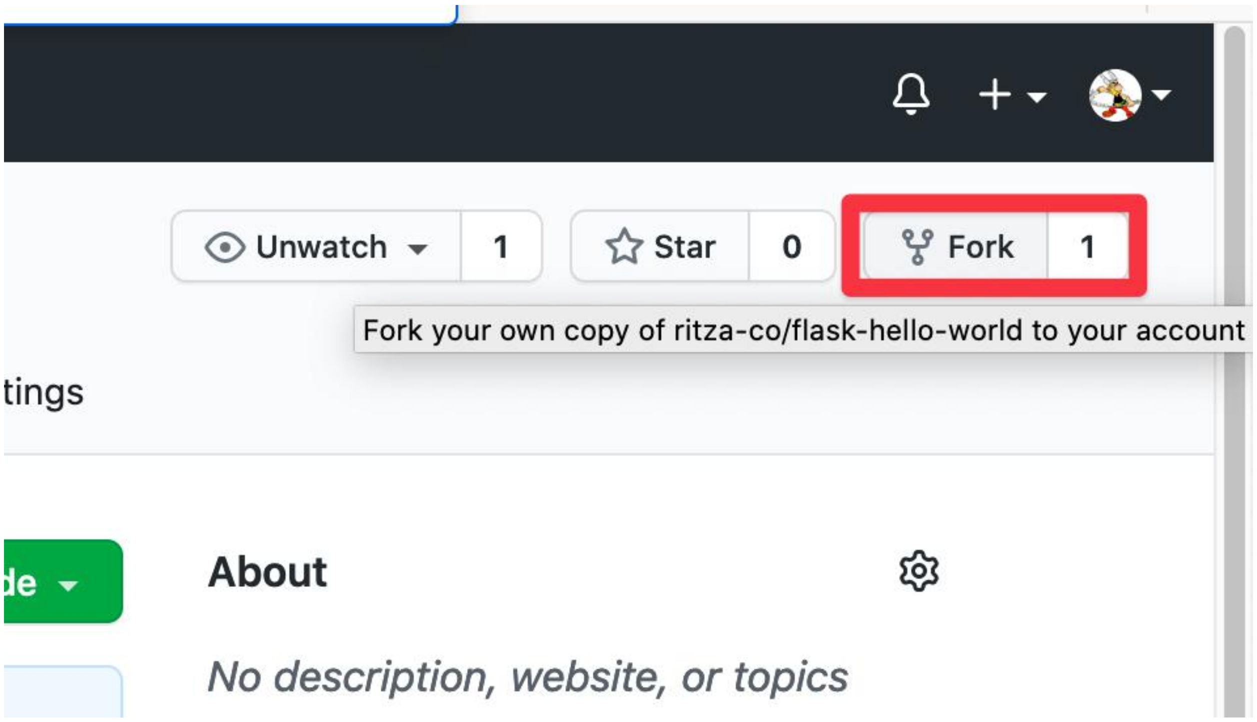

Unwatch
1
Star
0
Fork
1
Fork your own copy of ritza-co/flask-hello-world to your account
tings
About
No description, website, or topics

Image 6: ***Forking a repository in GitHub.***

You should be taken to a new page in GitHub that looks very similar to the old one but which is owned by your own GitHub username. My GitHub username is `sixhobbits` so the new URL for me is https://github.com/sixhobbits/flask-hello-world (but yours will be different).

Now, instead of cloning the original project into Repl.it, create a new repl and import this fork of the project instead. Instead of going through the import UI again, you can also just create and load the relevant import URL. These URLs are in the format `https://repl.it/github/<githubproject>` so in my case I open https://repl.it/github/sixhobbits/flask-hello-world in my browser (you need to substitute your own GitHub username for this to work).

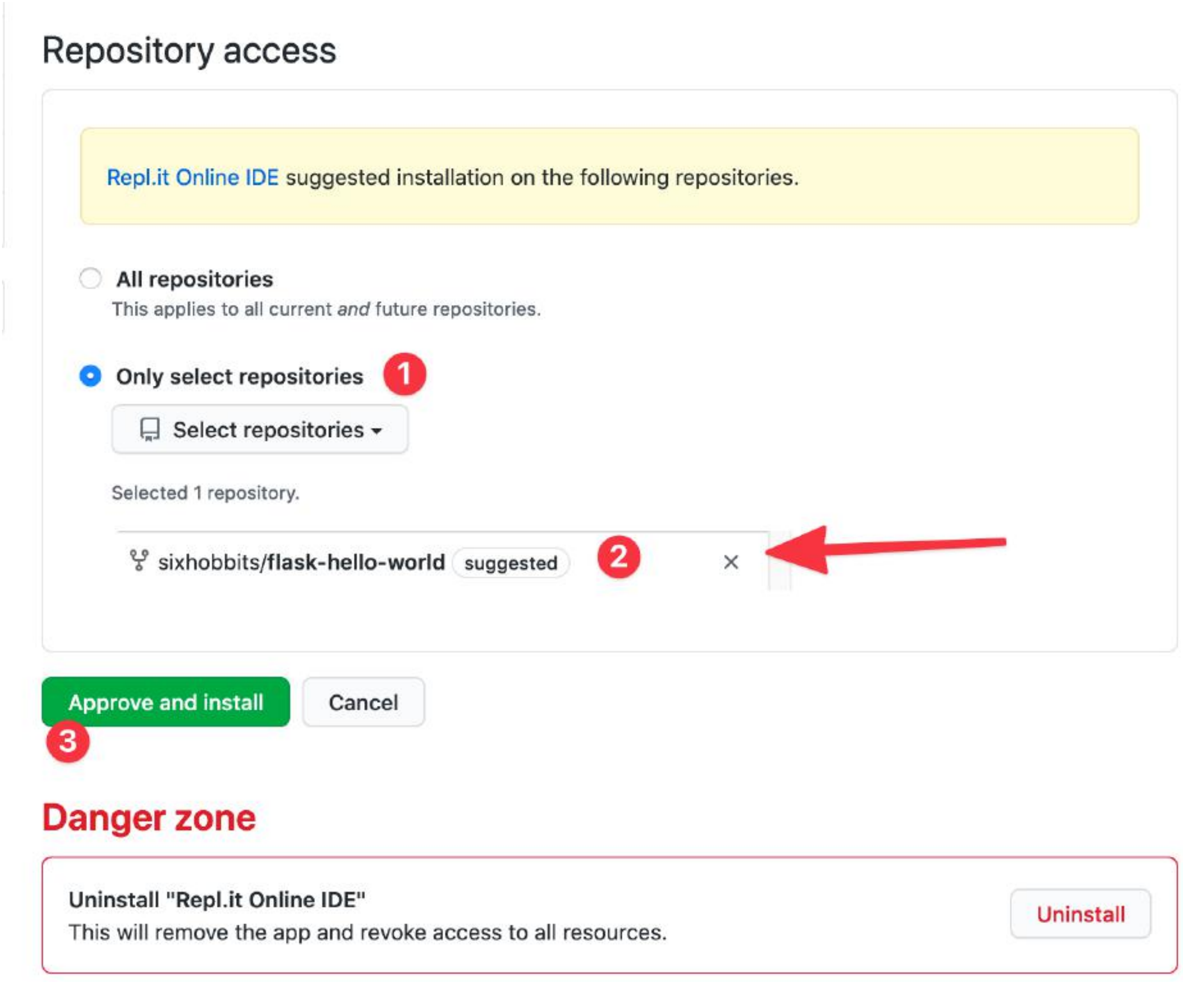
Configure the `.replit` file again and open the version control tab, as before. Under "what did you

[46]https://github.com

change" enter "add .repl file" or a similar message to describe what contributions you're making, and press `commit and push`.

You'll see the error again and be presented with the option to connect your Repl.it account to GitHub to prove that you authorise Repl.it to make these changes to GitHub on your behalf.

You can give Repl.it access to all of your repositories (useful if you want to use this integration a lot), but by default it will only get permission for the specific repository that we're working with.


Repository access

Repl.it Online IDE suggested installation on the following repositories.

All repositories
This applies to all current *and* future repositories.

Only select repositories

Selected 1 repository.

sixhobbits/**flask-hello-world** suggested

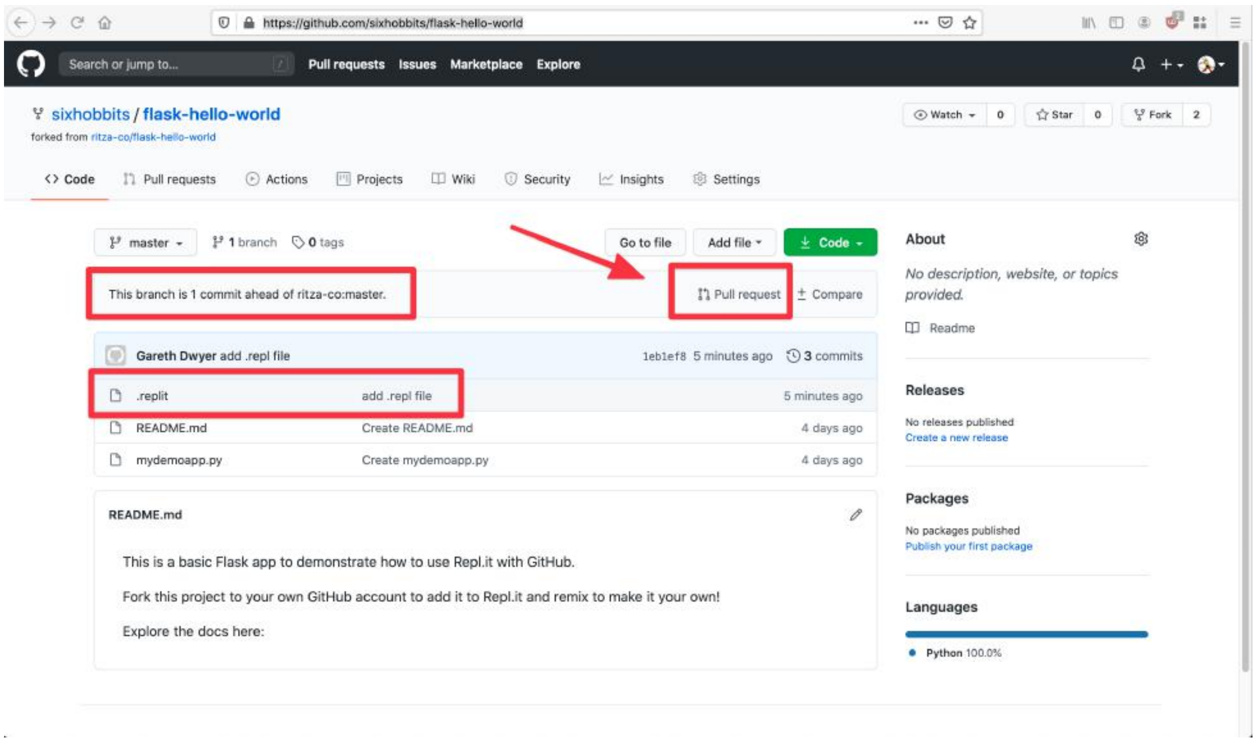
Danger zone

Uninstall "Repl.it Online IDE"
This will remove the app and revoke access to all resources.

Image 7: ***Giving Repl.it permission to access your GitHub data.***

Press the green `approve` button and you'll be directed back to Repl.it. Press the `commit & push` button again on Repl.it and this time everything should work without any errors.

Navigate back to your fork of the GitHub project, and you should see that the changes are reflected in GitHub too.



This branch is 1 commit ahead of ritza-co:master.

Gareth Dwyer add .repl file 1eb1ef8 5 minutes ago 3 commits

.replit	add .repl file	5 minutes ago
README.md	Create README.md	4 days ago
mydemoapp.py	Create mydemoapp.py	4 days ago

No description, website, or topics provided.

No releases published

No packages published

README.md

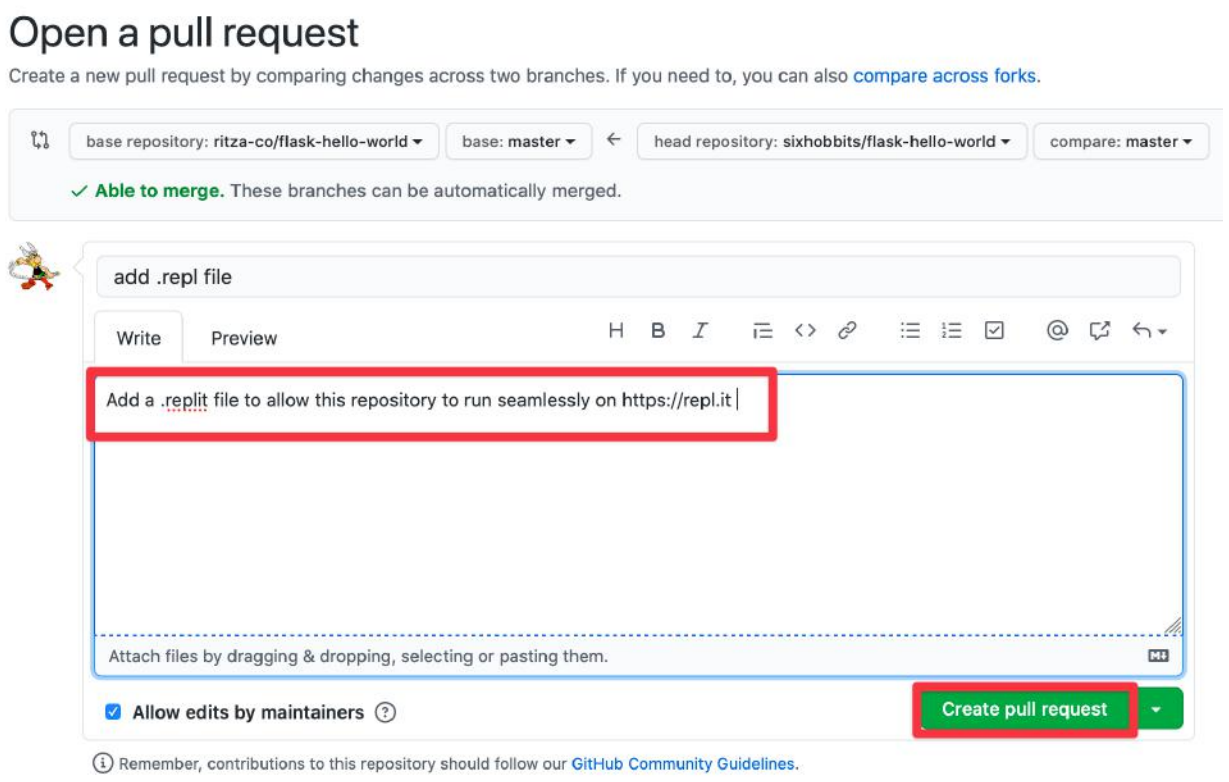
This is a basic Flask app to demonstrate how to use Repl.it with GitHub.

Fork this project to your own GitHub account to add it to Repl.it and remix to make it your own!

Explore the docs here:

Image 8: ***Seeing our changes reflected in our GitHub fork***.

As you can see, the new `.replit` file is visible and GitHub prompts you to make a pull request back into the original repository. Press `Pull request`, `create pull request`, add a comment explaining why your changes should be merged into the original repository, and click `Create pull request` again.


Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base repository: ritza-co/flask-hello-world base: master ← head repository: sixhobbits/flask-hello-world compare: master

✓ Able to merge. These branches can be automatically merged.

add .repl file

Add a .replit file to allow this repository to run seamlessly on https://repl.it

Attach files by dragging & dropping, selecting or pasting them.

Allow edits by maintainers

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Image 9: ***Creating a pull request from GitHub to merge our changes back into the original repository.***

The owner of the repository will get a notification about your proposal and can choose to add your changes or reject them (in this case, don't be too hopeful about your changes being accepted as the `.replit` file being missing is important to follow along the earlier steps of this tutorial ⍰.)

Make it your own

For this tutorial, it's important that you do the steps yourself so that everything is correctly linked to your own GitHub account, but if you want an example to play with, use the one below. It's the same as the Flask app but it can greet individual users dynamically instead of having a static welcome message.

Visit https://cwr-06-github–garethdwyer1.repl.co/Gareth[47] to see it in action (replace the last part of the URL with your own name to receive a personalised greeting).

You can for our repl from https://repl.it/@GarethDwyer1/cwr-06-github[48]

Can you figure out how it works?

[47]https://cwr-06-github--garethdwyer1.repl.co/Gareth
[48]https://repl.it/@GarethDwyer1/cwr-06-github

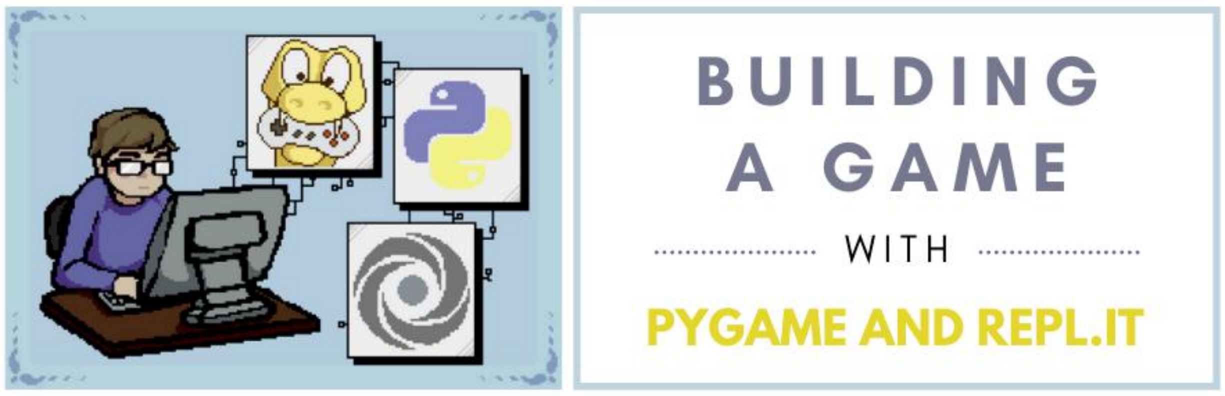
Where next?

Cloning repositories and being able to immediately run them is useful in many scenarios, from just wanting to try out a cool project that you found to running production workflows.

Open-source software only exists because of people who build, maintain, and improve it. There's no global committee that decides who gets to be an open-source software developer and you can be one too. Find a project that you like, look at the "Issues" tab on GitHub to see what problems exist, and try to fix one. Many issues on GitHub are tagged with "Good first issue" to help direct newer developers to places where they can get started.

In the next tutorial, we'll do something a bit different and build a 2D game using PyGame.

Building a game with PyGame and Repl.it



So far, we've mainly seen how to write text-based programs, or those with a basic web front end. In this tutorial, we'll instead build a 2D game using PyGame. You'll use animated sprites and learn how to:

- Make these sprites move
- Recognise when a sprite is clicked with the mouse.

The basic premise of the game is as follows. You're a juggler, learning to juggle. Balls will fall down from the top of the screen, and you'll need to click them to 'throw' them up again. After several successful throws without dropping any balls, more balls will be added to make the game harder.

Creating a PyGame repl

Although PyGame[49] is a standard Python library, Repl.it provides it installed as a separate language. Create a new repl and select PyGame from the language dropdown.

[49]https://en.wikipedia.org/wiki/Pygame

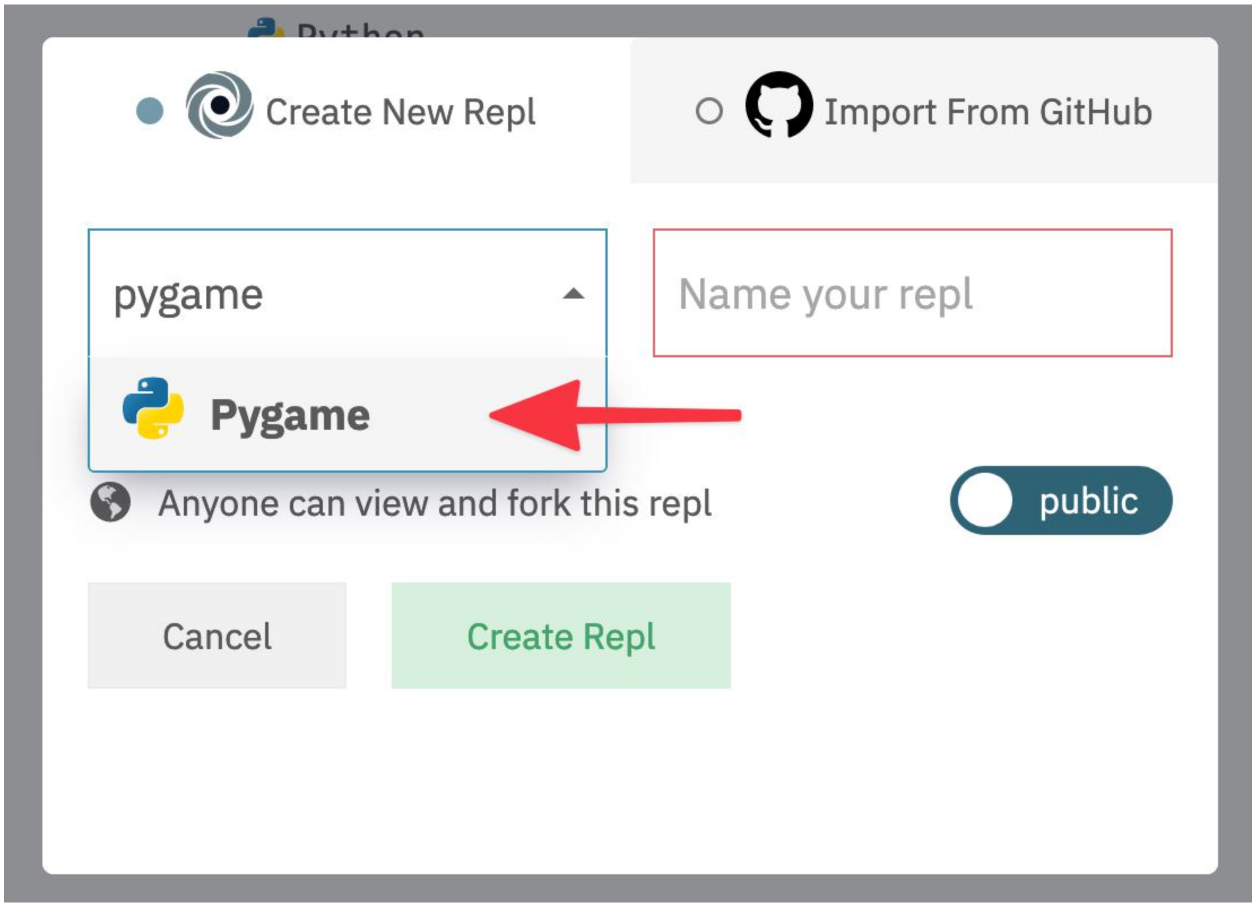


Image 2: *Choosing PyGame from the `Create New Repl` screen.*

You'll see "Python3 with PyGame" displayed in the default console and a separate pane in the Repl.it IDE where you will be able to see and interact with the game you will create.

The first thing we need is a so-called "sprite", which is a basic image file that we will use in our game. Download the tennis ball file available here[50] and save it to your local machine.

Now upload it to your repl using the `upload file` button and you should be able to see a preview of the image by clicking on it in the files pane.

[50]https://raw.githubusercontent.com/ritza-co/public-images/master/small_tennis.png

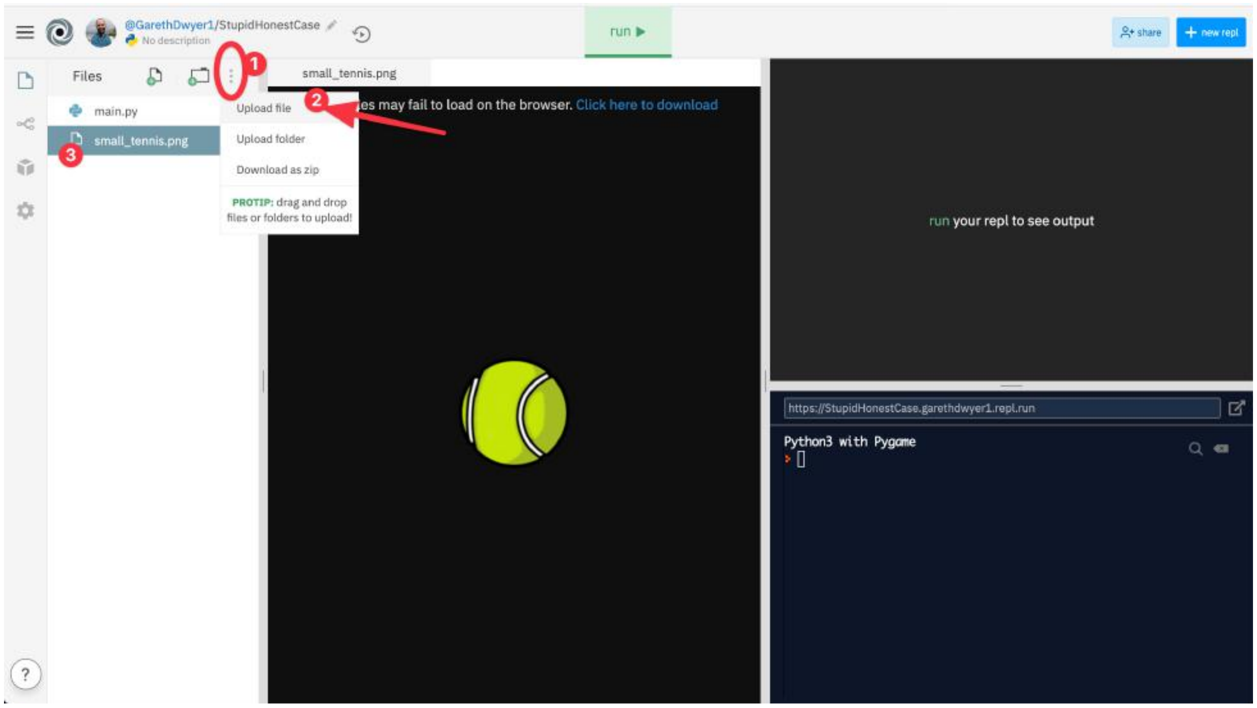

@GarethDwyer1/StupidHonestCase
No description
run ▶
share
new repl
Files
main.py
small_tennis.png
Upload file
Upload folder
Download as zip
PROTIP: drag and drop files or folders to upload!
1
2
3
small_tennis.png
...es may fail to load on the browser. Click here to download
run your repl to see output
https://StupidHonestCase.garethdwyer1.repl.run
Python3 with Pygame

Image 3: ***Viewing our sprite after uploading it***.

Displaying the sprite using PyGame

Our first goal is to display the tennis ball in a game environment using PyGame. To do this, go back to the `main.py` file and add the following code.

```python
import pygame

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.rect = self.image.get_rect()

def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    clock = pygame.time.Clock()
    ball = Ball()

    while True:
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        pygame.display.flip()
        clock.tick(60)

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

This code looks a bit more complicated than it needs to be because in addition to drawing the ball to the screen, it also sets up a game loop. While basic 2D games appear to move objects around the screen, they usually actually simulate this effect by redrawing the entire screen many times per second. To account for this we need to run our logic in a `while True:` loop.

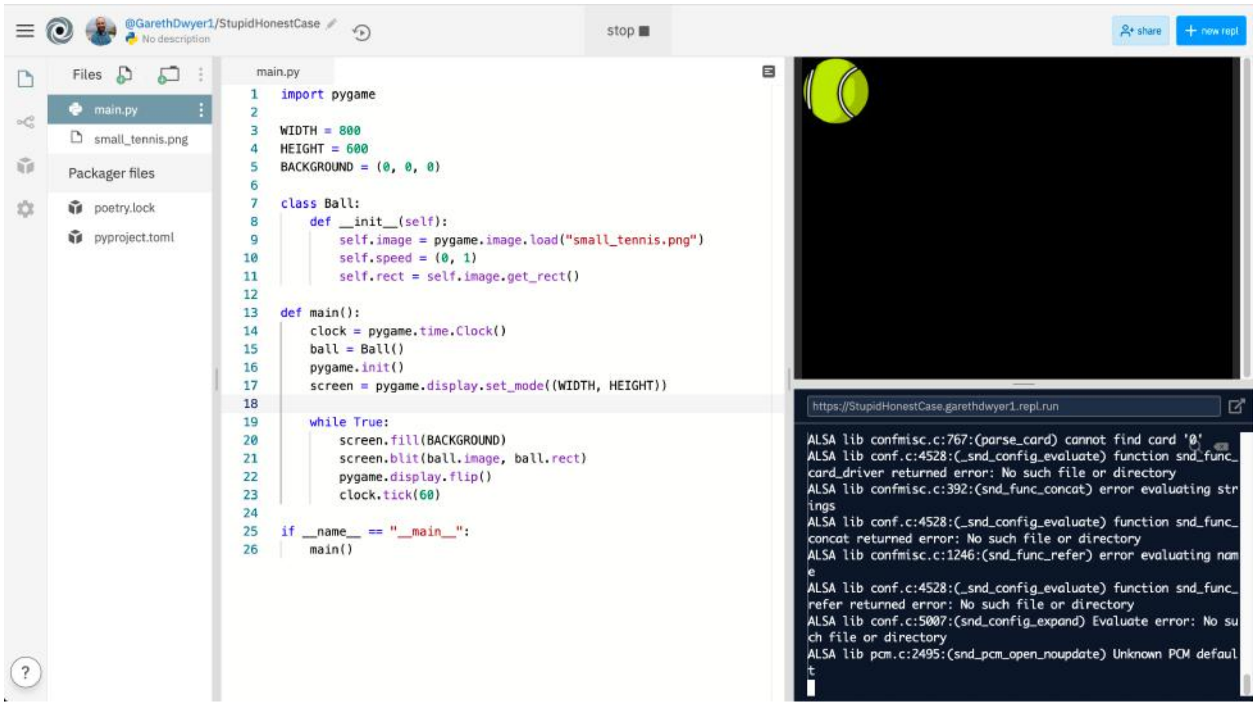
We start by importing PyGame and setting up some global variables: the size of our screen and the background color (black). We then define our `Ball`, setting up an object that knows where to find the image for the ball and how to get the default coordinates of where the image should be drawn.

We then set up PyGame by calling `init()` and starting the screen as well as a clock. The clock is necessary because each loop might take a different amount of time, based on how much logic needs to run to calculate the new screen. PyGame has built-in logic to calculate how much time elapses between calls to `clock.tick()` to draw frames faster or slower as necessary to keep the game experience smooth.

We start the game loop and call `blit` on our ball. Blitting[51] refers to moving all of the pixels from our sprite file (the tennis ball) to our game environment. The `flip()` function updates our screen and the `tick(60)` call means that our game will redraw the screen around 60 times per second.

If you run this code, you should see the ball pop up in the top right pane, as shown below.

[51]https://en.wikipedia.org/wiki/Bit_blit



```
import pygame

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = (0, 1)
        self.rect = self.image.get_rect()

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        pygame.display.flip()
        clock.tick(60)

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

```
ALSA lib confmisc.c:767:(parse_card) cannot find card '0'
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
card_driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating str
ings
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
concat returned error: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating nam
e
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
refer returned error: No such file or directory
ALSA lib conf.c:5007:(snd_config_expand) Evaluate error: No su
ch file or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM defaul
t
```

Image 4: *Drawing the tennis ball in our PyGame environment.*

Making our tennis ball move with each frame

Although PyGame has a lot of built-in logic for handling common game operations, you still need to get your hands dirty with calculating some of the basic movements. For every loop, we need to tell our game the new X and Y coordinates to draw the ball. As we want our ball to move at a constant speed, we'll move the X and Y coordinates each loop.

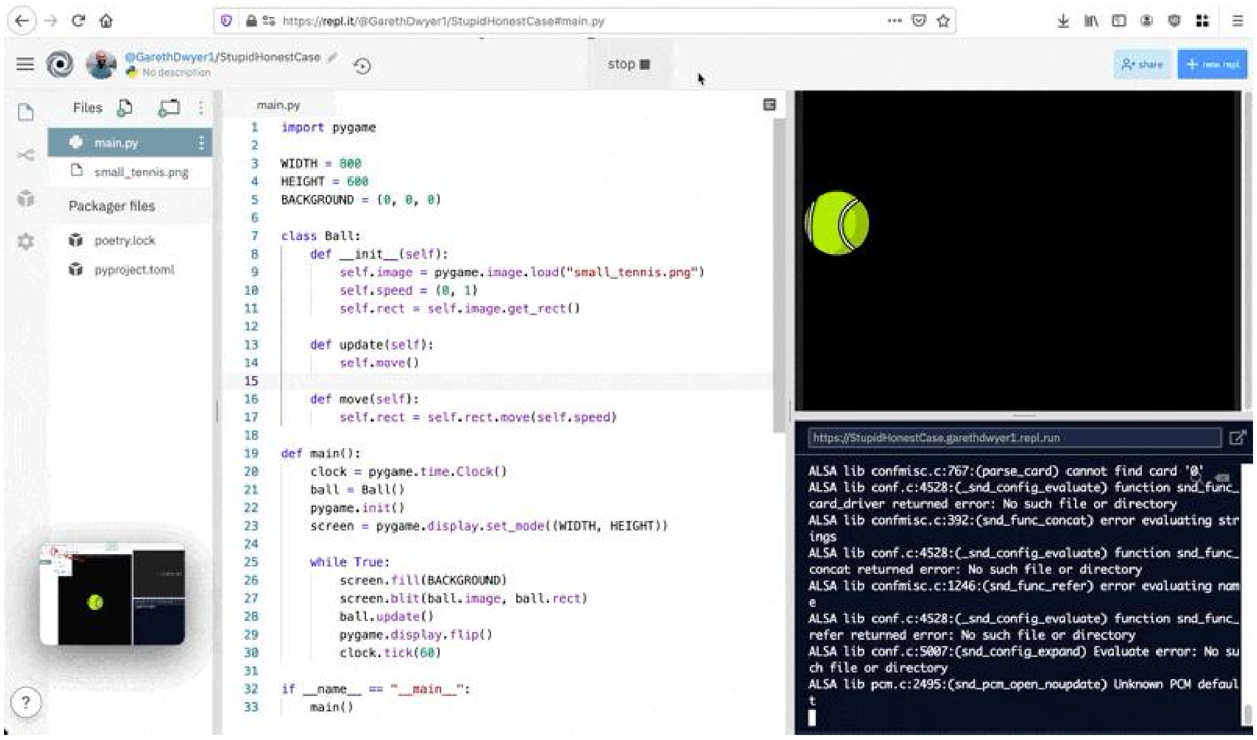
Add two methods to your `Ball` class: `update` and `move`, and add a speed attribute. The new code for your `Ball` class should look as follows.

```python
class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [0, 1]
        self.rect = self.image.get_rect()

    def update(self):
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)
```

Now modify your game loop to include a call to the new `update()` method. The loop should look as follows.

```
    while True:
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        ball.update()
        pygame.display.flip()
        clock.tick(60)
```

The `(0, 1)` tuple causes the ball to move its Y coordinate by `1` each loop and keep a constant X coordinate. This has the effect of making the ball drop slowly down the screen. Run your code again to check that this works.


https://repl.it/@GarethDwyer1/StupidHonestCase#main.py
@GarethDwyer1/StupidHonestCase
No description
stop
share
new repl
Files
main.py
small_tennis.png
Packager files
poetry.lock
pyproject.toml
main.py
https://StupidHonestCase.garethdwyer1.repl.run

```
import pygame

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = (0, 1)
        self.rect = self.image.get_rect()

    def update(self):
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        ball.update()
        pygame.display.flip()
        clock.tick(60)

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

```
ALSA lib confmisc.c:767:(parse_card) cannot find card '0'
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
card_driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating str
ings
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
concat returned error: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating nam
e
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
refer returned error: No such file or directory
ALSA lib conf.c:5007:(snd_config_expand) Evaluate error: No su
ch file or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM defaul
t
```

Image 5: ***The ball falling at a constant rate.***

click to open gif[52]

When the ball gets to the bottom of the screen, it'll just keep falling but that's OK for now. Let's see how we can add click detection.

[52]https://i.ritzastatic.com/repl/codewithrepl/07-pygame/07-05-GIF-falling-ball.gif

Processing events: Detecting mouse clicks

PyGame records all "events", including mouse clicks, and makes these available through `pygame.event.get:()`. We need to check what events happened in each game loop and see if any of them were important.

If the user clicks on an empty space, that will still be recorded but we will simply ignore it. If the user clicks on a falling ball, we want it to change direction.

To achieve this, add a `for` loop inside the existing `while` loop. The entire game loop should look as follows:

```
    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if ball.rect.collidepoint(pygame.mouse.get_pos()):
                    ball.speed = [0,-1]
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        ball.update()
        pygame.display.flip()
        clock.tick(60)
```

With this code, we loop through all events and check for left click (`MOUSEBUTTONDOWN`) events. If we find one, we check if the click happened on top of the ball (using `collidepoint()` which checks for overlapping coordinates), and in this case we reverse the direction of the ball (still no x-axis or horizontal movement, but we make the ball move negatively on the y-axis, which is up.)

If you run this code again, you should now be able to click on the ball (let it fall for a while first) and see it change direction until it goes off the top of the screen.

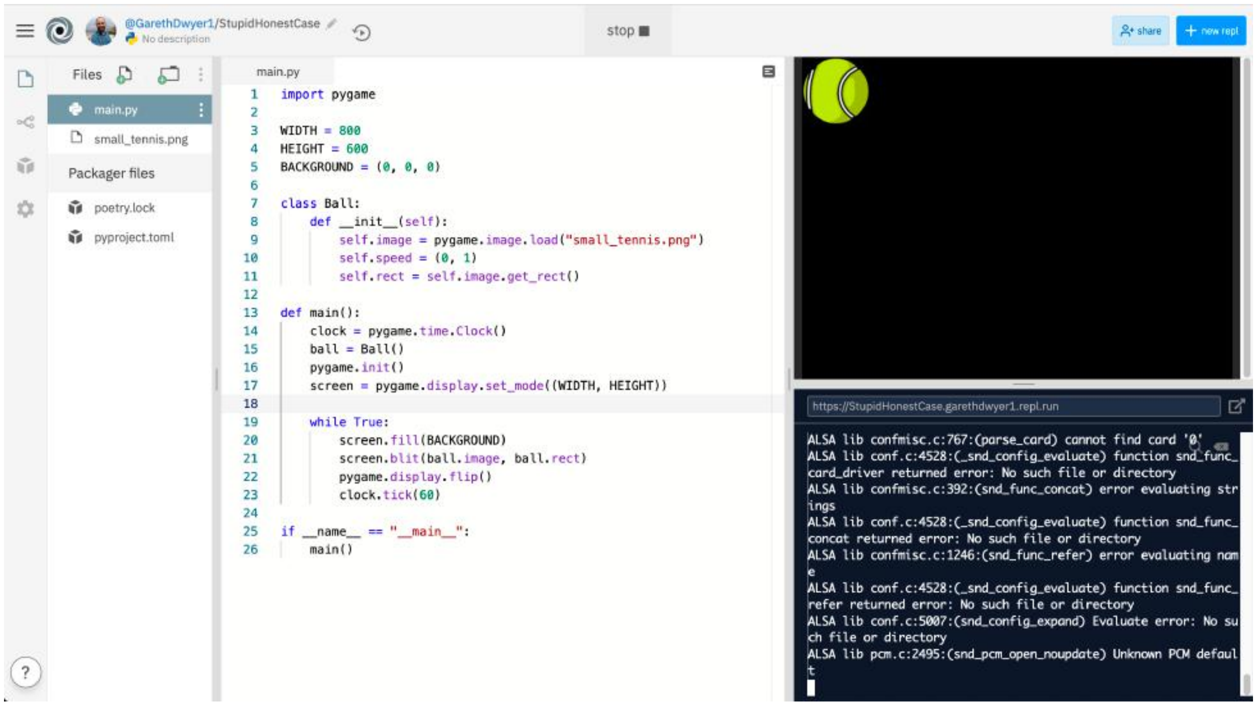
Making the ball bounce off the edges and move randomly

To simulate juggling, we want the ball to bounce off the "roof" (top edge of the screen) and "walls" (left and right edge). If the ball touches the "floor" (bottom edge) we want to kill it and remove it from the game as a dropped ball.

To achieve this, we'll add logic to our `update()` method (this is why we kept it separate from our `move()` method before). Add two lines of code to `update()` to make it look as follows.

```
    def update(self):
        if self.rect.top < 0:
            self.speed = [0, 1]
        self.move()
```

This checks to see if the top of the ball is above the top of the screen. If it is, we set the speed back to `(0, 1)` (moving down).



```
import pygame

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = (0, 1)
        self.rect = self.image.get_rect()

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        pygame.display.flip()
        clock.tick(60)

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

```
ALSA lib confmisc.c:767:(parse_card) cannot find card '0'
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
card_driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating str
ings
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
concat returned error: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating nam
e
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_
refer returned error: No such file or directory
ALSA lib conf.c:5007:(snd_config_expand) Evaluate error: No su
ch file or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM defaul
t
```

Image 6: ***Now we can bounce the ball off the ceiling.***

click to open gif[53]

So far, we have restricted the ball to moving vertically, but we can apply the same principles and move it horizontally or diagonally too. Let's also add some randomness into the mix so that it's less predictable (and harder for the player to press). The ball will randomly change its horizontal movement when it bounces off the ceiling and each time we throw it.

Import the `random` module at the top of your file and use the `random.randrange()` function to specify the range of acceptable horizontal movement. Also modify the `update()` function to detect if the ball is falling off the left or right edges and reverse its horizontal movement in this case.

Finally, modify the collision detection section to add randomness there too.

Your full code should now look as follows.

[53]https://i.ritzastatic.com/repl/codewithrepl/07-pygame/07-06-GIF-bounce-off-roof.gif

```
import pygame
import random

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.uniform(-4,4), 2]
        self.rect = self.image.get_rect()

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.uniform(-4, 4)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if ball.rect.collidepoint(pygame.mouse.get_pos()):
                    ball.speed[0] = random.uniform(-4, 4)
                    ball.speed[1] = -2
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        ball.update()
        pygame.display.flip()
        clock.tick(60)

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

If you run your code again, you should be able to juggle the ball around by clicking on it and watch it randomly bounce off the ceiling and walls.

Adding more balls

Juggling with one ball is no fun, so let's add some more. Because we used Object Oriented Programming (OOP), we can create more balls by instantiating more `Ball()` objects. We'll need to keep track of these so we'll add them to an array. For each iteration of the game loop, we'll need to update the position of each ball, so we'll need one more loop to account for this.

We also want to start keeping track of which of our balls is "alive" (that is, hasn't hit the ground), so add an attribute for this to the `Ball` class too, in the `__init__` function.

```
        self.alive = True
```

In the `main()` function, directly before the `while True:` line, add the following code.

```
    ball1 = Ball()
    ball2 = Ball()
    ball3 = Ball()

    balls = [ball1, ball2, ball3]
```

Now remove the `ball=Ball()`, `ball.update()` and `screen.blit(...)` lines and replace them with a loop that updates all of the balls and removes the dead ones (even though we haven't written the logic yet to stop the balls from ever being alive.)

```
        for i, ball in enumerate(balls):
            if ball.alive:
                screen.blit(ball.image, ball.rect)
                ball.update()
                if not ball.alive:
                    del balls[i]
```

You'll also need to account for multiple balls in the the event detection loop. For each event, loop through all of the balls and check if the mouse click collided with any of them.

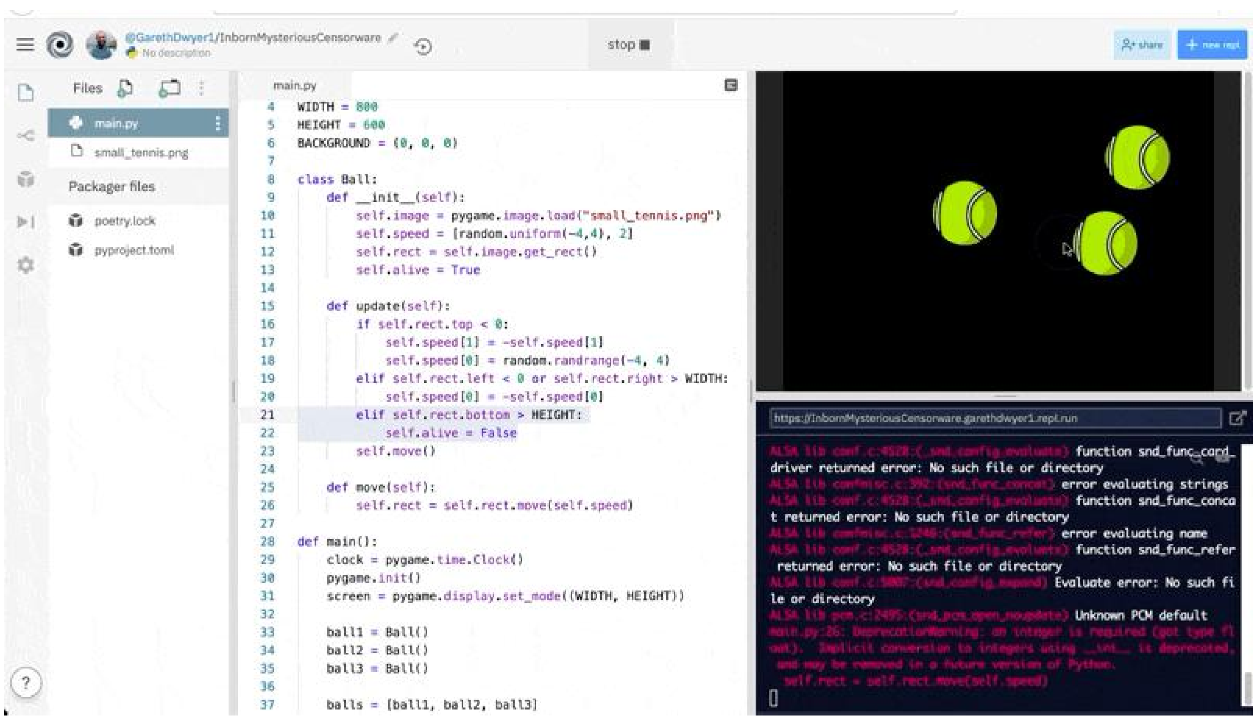
At this point, the full `main()` function should look as follows.

```
def main():
    clock = pygame.time.Clock()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    ball1 = Ball()
    ball2 = Ball()
    ball3 = Ball()

    balls = [ball1, ball2, ball3]

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                for ball in balls:
                    if ball.rect.collidepoint(pygame.mouse.get_pos()):
                        ball.speed[0] = random.randrange(-4, 4)
                        ball.speed[1] = -2
                        break
        screen.fill(BACKGROUND)
        for i, ball in enumerate(balls):
            if ball.alive:
                screen.blit(ball.image, ball.rect)
                ball.update()
                if not ball.alive:
                    del balls[i]
        pygame.display.flip()
        clock.tick(60)
```

To kill balls when they fall through the floor, we can add another check to the `update()` function as follows.

```
        elif self.rect.bottom > HEIGHT:
            self.alive = False
```

Run the code again and you should be able to juggle three balls. See how long you can keep them in the air.



```
WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.uniform(-4,4), 2]
        self.rect = self.image.get_rect()
        self.alive = True

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.randrange(-4, 4)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        elif self.rect.bottom > HEIGHT:
            self.alive = False
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    ball1 = Ball()
    ball2 = Ball()
    ball3 = Ball()

    balls = [ball1, ball2, ball3]
```

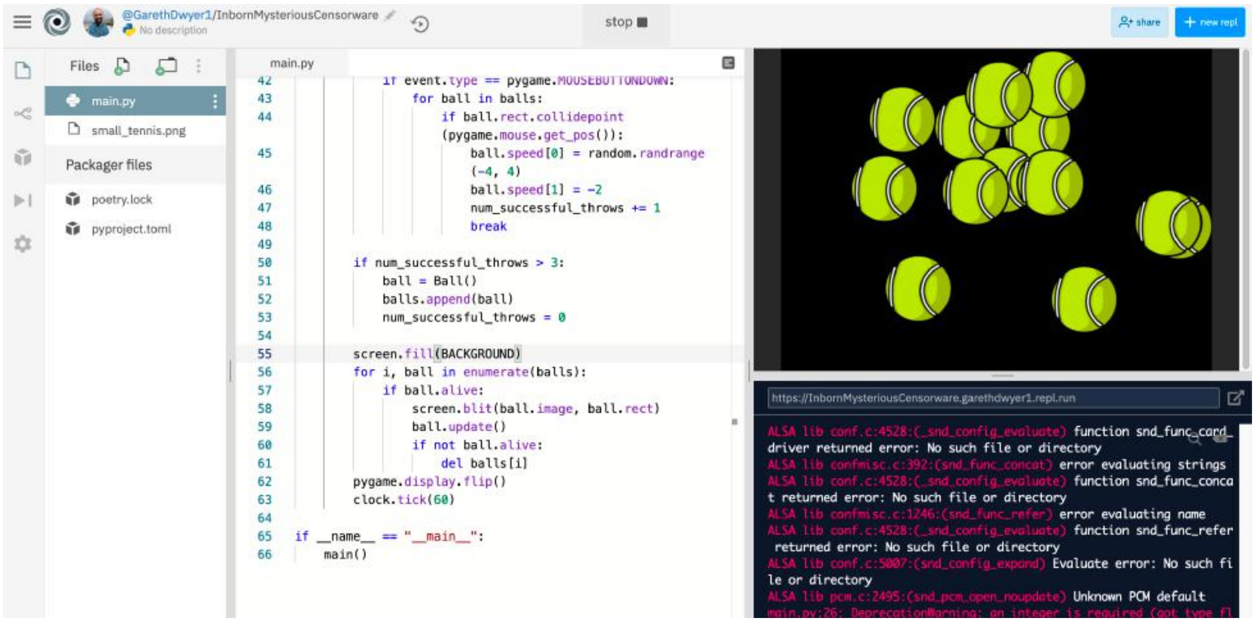
```
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_card_
driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating strings
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_conca
t returned error: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating name
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_refer
 returned error: No such file or directory
ALSA lib conf.c:5007:(_snd_config_expand) Evaluate error: No such fi
le or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM default
main.py:26: DeprecationWarning: an integer is required (got type fl
oat).  Implicit conversion to integers using __int__ is deprecated,
 and may be removed in a future version of Python.
  self.rect = self.rect.move(self.speed)
▯
```

Image 7: ***Juggling three balls.***

click to open gif[54]

If you want a harder version, add a counter to keep track of how many successful throws the player has achieved and add a new ball for every three successful throws.

[54]https://i.ritzastatic.com/repl/codewithrepl/07-pygame/07-07-GIF-three-balls.gif



```
                if event.type == pygame.MOUSEBUTTONDOWN:
                    for ball in balls:
                        if ball.rect.collidepoint
                        (pygame.mouse.get_pos()):
                            ball.speed[0] = random.randrange
                            (-4, 4)
                            ball.speed[1] = -2
                            num_successful_throws += 1
                            break

            if num_successful_throws > 3:
                ball = Ball()
                balls.append(ball)
                num_successful_throws = 0

            screen.fill(BACKGROUND)
            for i, ball in enumerate(balls):
                if ball.alive:
                    screen.blit(ball.image, ball.rect)
                    ball.update()
                    if not ball.alive:
                        del balls[i]
            pygame.display.flip()
            clock.tick(60)

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

https://InbornMysteriousCensorware.garethdwyer1.repl.run

```
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_card_
driver returned error: No such file or directory
ALSA lib confmisc.c:392:(snd_func_concat) error evaluating strings
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_conca
t returned error: No such file or directory
ALSA lib confmisc.c:1246:(snd_func_refer) error evaluating name
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_refer
 returned error: No such file or directory
ALSA lib conf.c:5007:(snd_config_expand) Evaluate error: No such fi
le or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM default
main.py:26: DeprecationWarning: an integer is required (got type fl
```

Image 8: ***Adding more balls***.

Now the game is to see how many balls you can juggle with. If it's too easy, modify the speeds and angles of the balls.

Make it your own

If you followed along, you'll already have your own version of the repl to extend. If not, start from ours. Fork it from https://repl.it/@GarethDwyer1/cwr-07-juggling-with-pygame[55].

Where next?

You've learned how to make 2D games using PyGame. If you want to make more games but are stuck for ideas, check out PyGame's extensive collection of examples[56].

You could also extend the juggling game more. For example, make the balls accelerate as they fall, or increase the speed of all balls over time.

[55]https://repl.it/@GarethDwyer1/cwr-07-juggling-with-pygame
[56]https://www.pygame.org/docs/ref/examples.html

Staying safe: Keeping your passwords and other secrets secure

While developing software fully in public has many benefits, it also means that we need to be extra careful about leaking sensitive information. Because all of our repls are public by default, we shouldn't store passwords, access keys, personal information, or anything else sensitive in them.

Even if you're coding offline or only in private repls, it's good practice to keep your code separate from any private information in any case.

In this tutorial, we'll look at how to use the special `.env` file that Repl.it provides to set environment variables[57]. We can use these to store sensitive information and Repl.it will make sure that this file isn't included when others fork our repl.

Understanding `.env` files

Similarly to the `.replit` file that we saw in a previous tutorial, the `.env` file is a special Repl.it file. If you call a file exactly `.env`, Repl.it will

- Not include this file in any forks of the repl
- Attempt to parse key-value pairs out of this file and make them available to the underlying operating system.

This can be used to store all kinds of configuration, but it's commonly used for passwords, API keys, and database credentials.

The structure of a `.env` file

A `.env` file consists of keys-value pairs, one per line, separated by an = sign. Environment variables are traditionally in ALL_CAPS and separated by underscores. For example, you might have a `.env` file with the following.

```
SECRET_PASSWORD=ThisIsMySuperSecretP@ssword!!
```

With this file present, your scripts can load the variable `SECRET_PASSWORD` from the operating system environment directly.

Unlike in Python, where `x = 1` and `x= 1` are the same, in `.env` files, spaces matter and you should be careful to not add any extra ones.

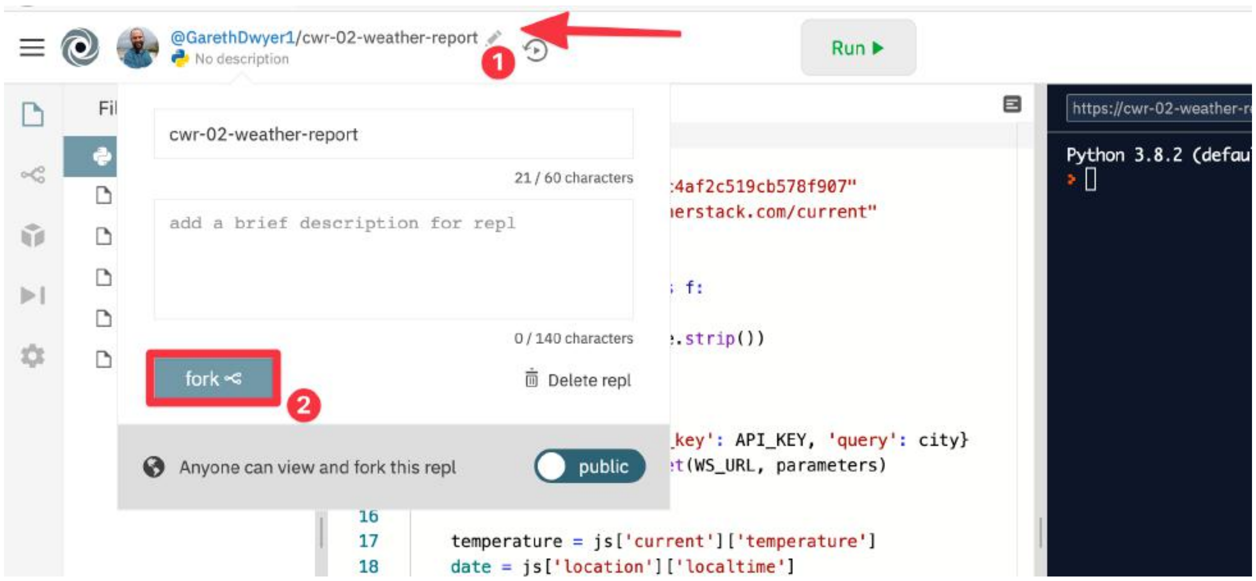
[57]https://en.wikipedia.org/wiki/Environment_variable

Refactoring our weather project to keep our API key secure

In the working with files tutorial[58] we used the WeatherStack API to fetch weather data and save it to disk. Part of this involved getting an API key from WeatherStack. Because WeatherStack limits the number of calls each key can make per day, it would be bad if someone else used up the quota for our key and broke our app as a result.

Let's refactor the WeatherStack project to prevent our key from being made public.

Visit https://repl.it/@GarethDwyer1/cwr-02-weather-report[59] (or your own version of this if you followed along previously) and create a new fork by pressing the pencil icon and then `fork`.



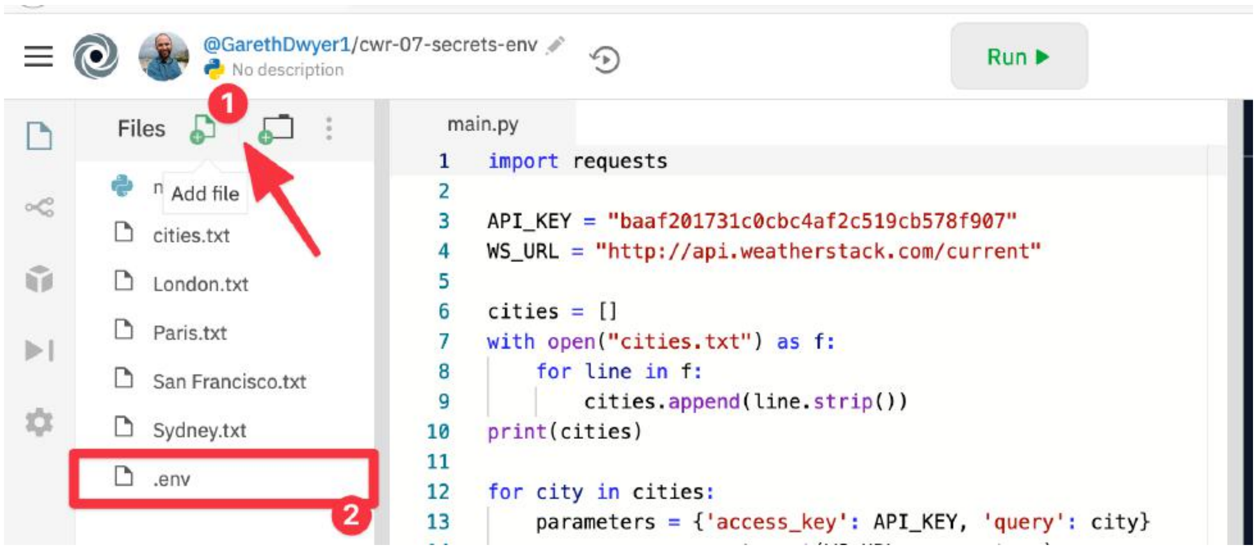
```
:4af2c519cb578f907"
ierstack.com/current"

; f:

:.strip())

_key': API_KEY, 'query': city}
:t(WS_URL, parameters)

    temperature = js['current']['temperature']
    date = js['location']['localtime']
```

Image 1: ***Forking our repl before refactoring it***.

We have `API_KEY` defined near the top of `main.py`, and this is the value that we want to keep secret. Let's move it to a `.env` file instead.

Click on the add file icon and call your new file `.env`. Be sure to add the initial full stop and don't add any spaces.

[58]http://www.codewithrepl.it/02-managing-files-using-repl-it.html
[59]https://repl.it/@GarethDwyer1/cwr-02-weather-report



```python
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "http://api.weatherstack.com/current"

cities = []
with open("cities.txt") as f:
    for line in f:
        cities.append(line.strip())
print(cities)

for city in cities:
    parameters = {'access_key': API_KEY, 'query': city}
```

Image 2: ***Creating the*** `.env` ***file to store sensitive information***.

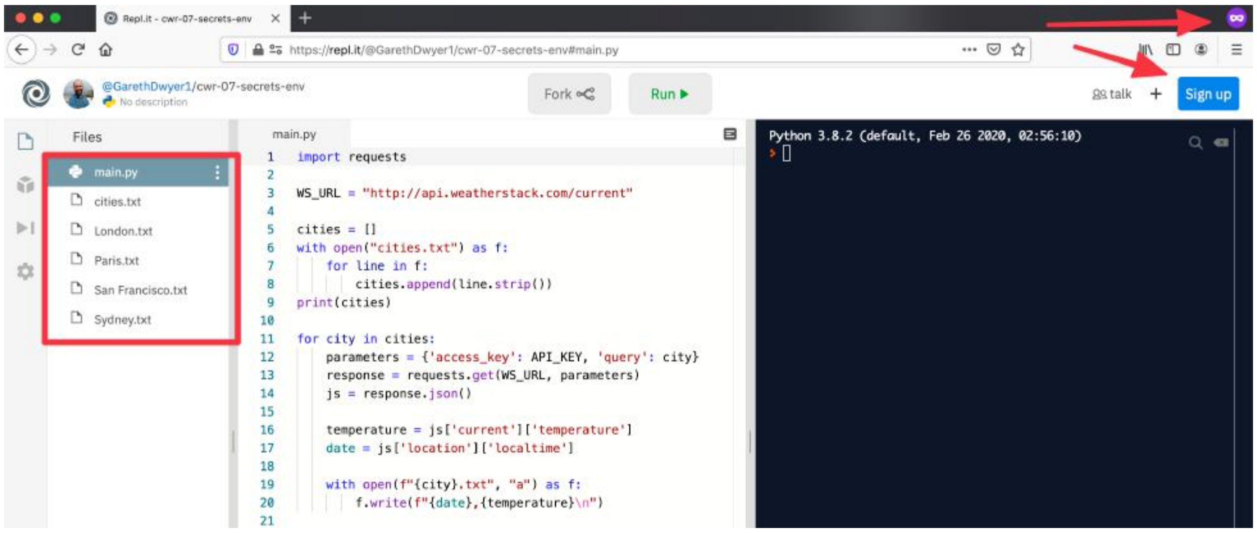
Now remove the API_KEY variable from the `main.py` file and add it to the `.env` file, removing all quotation marks (`"`) and spaces.

Your `.env` file should look as follows (but use your own WeatherStack API instead of the example given here).

```
API_KEY=baaf201731c0cbc4af2c519cb578f907
```

Testing that the file is not copied into others' forks

If you copy your project's URL into an incognito window (or use a separate browser), you'll see all of the other files as usual, but the `.env` file will not be there. Your API key, and the entire `.env` file, can only be seen when you're logged into the Repl.it accout that created it.


Repl.it - cwr-07-secrets-env
https://repl.it/@GarethDwyer1/cwr-07-secrets-env#main.py
@GarethDwyer1/cwr-07-secrets-env
No description
Fork
Run
talk
Sign up
Files
main.py
cities.txt
London.txt
Paris.txt
San Francisco.txt
Sydney.txt

```python
import requests

WS_URL = "http://api.weatherstack.com/current"

cities = []
with open("cities.txt") as f:
    for line in f:
        cities.append(line.strip())
print(cities)

for city in cities:
    parameters = {'access_key': API_KEY, 'query': city}
    response = requests.get(WS_URL, parameters)
    js = response.json()

    temperature = js['current']['temperature']
    date = js['location']['localtime']

    with open(f"{city}.txt", "a") as f:
        f.write(f"{date},{temperature}\n")
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> 
```

Image 3: *Checking that the `.env` file isn't included in public versions of our repl.*

Using environment variables in our script

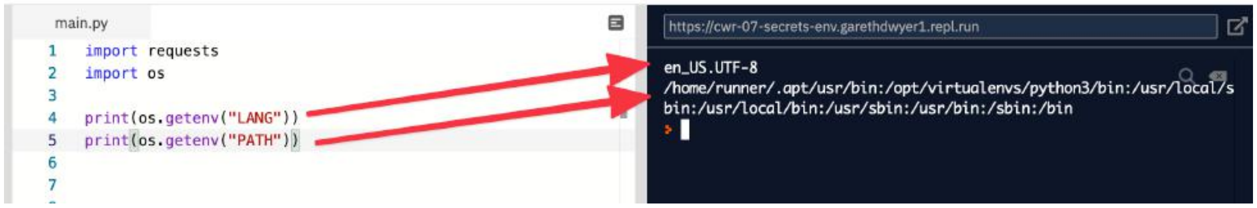
Our API key is now securely defined and available to the project, but we still need to tell our code where to find it. Because this data is stored in environment variables, we need to use the operating system (`os`) module to access it.

At the top of your `main.py` file, add an import for `os` and load the API_KEY into a variable as follows.

```python
import requests
import os

API_KEY = os.getenv("API_KEY")
```

The `getenv` function looks for an environment variable of a specific name. Now our code (and anyone who sees it) only needs to know the name of the key that stores our private API key, instead of the API key itself. You should be able to run your code again at this point to verify that new weather entries are correctly added to the relevant files.

There are many other environment variables that make various parts of an operating system work correctly. For example, you could also take a look at the `LANG` and `PATH` environment variables, which will show you that Repl.it has their servers configured to use US English and 8-bit unicode character encoding, and have some default places where the system looks for executable programs.



```
import requests
import os

print(os.getenv("LANG"))
print(os.getenv("PATH"))
```

```
https://cwr-07-secrets-env.garethdwyer1.repl.run
en_US.UTF-8
/home/runner/.apt/usr/bin:/opt/virtualenvs/python3/bin:/usr/local/s
bin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
>
```

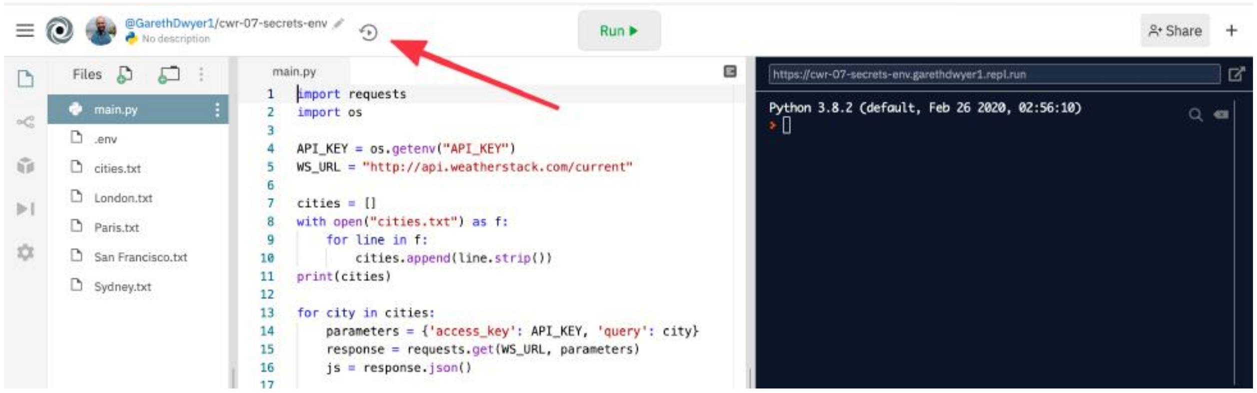
Image 4: ***Looking at other environment variables.***

Time travelling to find secrets

We removed the sensitive information from our project, but it's not actually completely gone. It's securely placed in our `.env` file, but it's also still saved in the repl's history.

Repl.it saves every change you make to a project so that you can always go back to previous versions if you make a mistake or need to check what has changed.

Click on the history button in the top bar, as shown below.

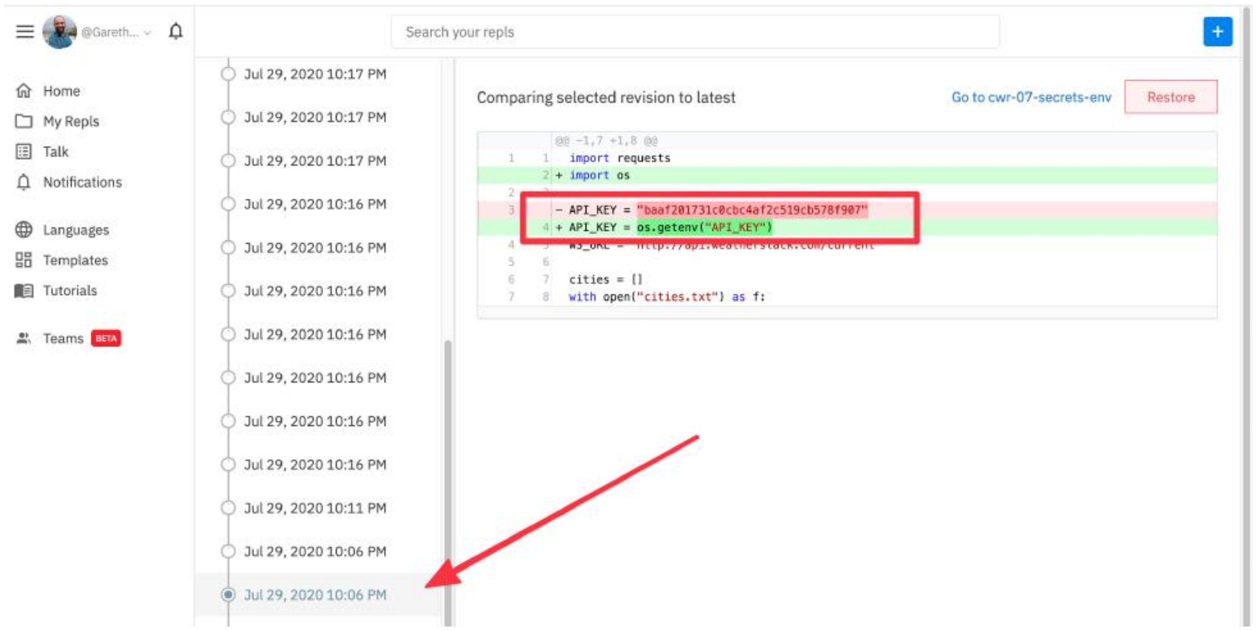


```
import requests
import os

API_KEY = os.getenv("API_KEY")
WS_URL = "http://api.weatherstack.com/current"

cities = []
with open("cities.txt") as f:
    for line in f:
        cities.append(line.strip())
print(cities)

for city in cities:
    parameters = {'access_key': API_KEY, 'query': city}
    response = requests.get(WS_URL, parameters)
    js = response.json()
```

```
https://cwr-07-secrets-env.garethdwyer1.repl.run
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Image 5: ***Diving into the history of our repl.***

You should see a bunch of entries from each change you've made to this project. Click through them and find the one where you deleted your API key. As you can see, the history viewer shows not only which lines have been changed, but also what was there before.



```
@@ -1,7 +1,8 @@
  import requests
+ import os

- API_KEY = "baaf201731c0cbc4af2c519cb578f907"
+ API_KEY = os.getenv("API_KEY")
  WS_URL = "http://api.weatherstack.com/current"

  cities = []
  with open("cities.txt") as f:
```

Image 6: ***Finding the credentials in the change logs.***

Luckily history is not included when other people fork your repl so this is not a huge problem, but it's important to keep in mind where people might find your credentials.

In our case, the worst case scenario is that someone finds our WeatherStack API key and uses up the quota, which is not the end of the world. A far more painful (and very common) scenario involves real money. For example, a developer signs up for a free trial on an expensive service like AWS, links a credit card, and then accidentally pushes the credentials for the service to GitHub or similar. Even if they realise their mistake and delete these within seconds, the credentials themselves are still available in the history of their repository. Hackers have bots that regularly look out for mistakes like this and use the credentials to spin up thousands of servers (often to mine cryptocurrency or join a botnet attack), potentially costing the poor developer thousands of dollars before they notice.

Rotating credentials

Even if there's a small chance that your API key has been exposed, it's important to rotate it. This involves creating a new key, ensuring the new key works with your service, and then disabling the old one.

In the case of WeatherStack, there is no option to create a new key while keeping the old one active, so we need to reset it and then copy the new key to our `.env` file (meaning that our app can't function between the time that we disable the old key and replace it with the new one).

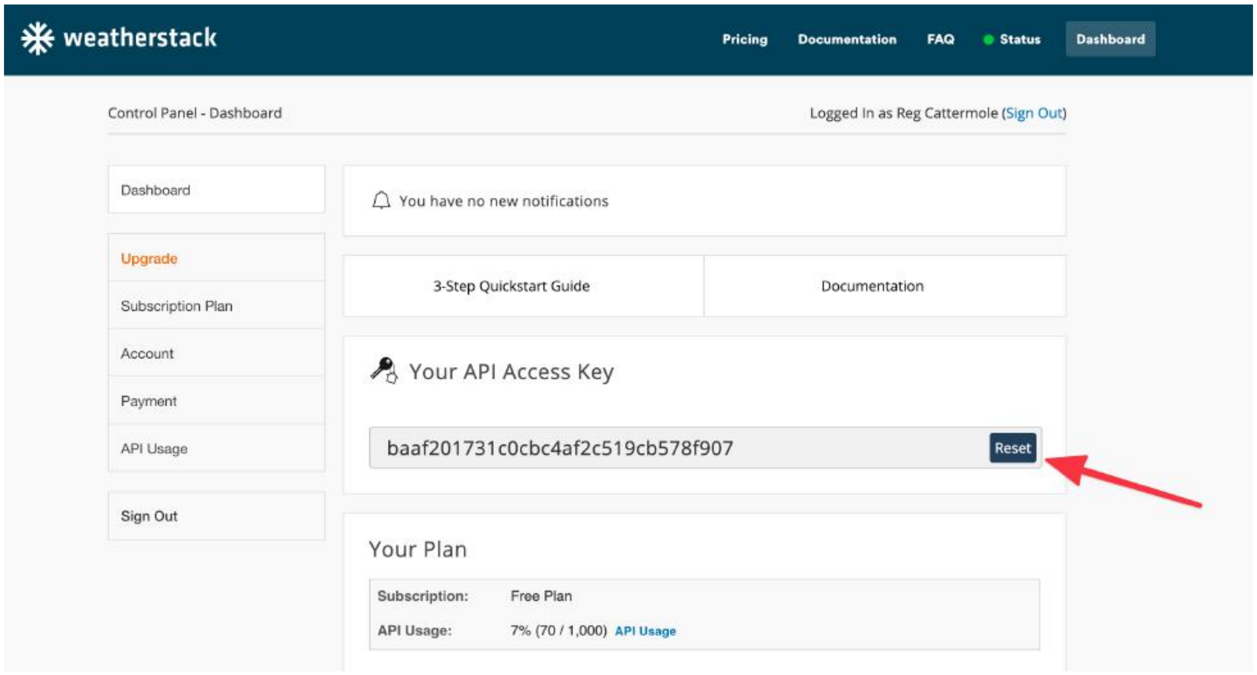


Image 7: *Rotating our WeatherStack API key*.

Visit your WeatherStack account and press the `reset` button to get your new API key.

Make it your own

You can make a copy of the new repl below. If you fork it, the `.env` file will be missing, so you'll need to create it and add your WeatherStack API key before it will run.

Fork this repl from https://repl.it/@GarethDwyer1/cwr-08-secrets-env[60]

Where next?

There's a lot more that you can do with `.env` files. In a later tutorial, we'll use it to store database credentials (which are more important to keep safe than a free API key).

You could also keep other private information in environment variables. For example, if you code a hangman game, you could keep the word that people need to guess in there so that you can share your code without spoiling the game.

[60]https://repl.it/@GarethDwyer1/cwr-08-secrets-env

An introduction to `pytest` and doing test-driven development with Repl.it

In this tutorial we'll introduce test-driven development and you'll see how to use `pytest`[61] to ensure that your code is working as expected.

`pytest` lets you specify inputs and expected outputs for your functions. It runs each input through your function and validates that the output is correct. `pytest` is a Python library and works just like any other Python library: you install it through your package manager and you can import it into your Python code. Tests are written in Python too, so you'll have code testing other code.

Test-driven development or TDD is the practice of writing tests before you write code. You can read more about TDD and why it's popular on Wikipedia[62].

Specifically you'll:

- See how to structure your project to keep your tests separate but still have them refer to your main code files
- Figure out the requirements for a function that can split a full name into first and last name components
- Write tests for this function
- Write the actual function.

Creating a project structure for `pytest`

For large projects, it's useful to keep your testing code separate from your application code. In order for this to work, you'll need your files set up in specific places, and you'll need to create individual Python **modules** so that you can refer to different parts of the project easily.

Create a new Python repl called `namesplitter`. As always, it'll already have a `main.py` file, but we're going to put our name splitting function into a different module called `utils`, which can house any helper code that our main application relies on. We also want a dedicated place for our tests.

Create two new folders: one called `utils` and one called `tests`, using the `add folder` button. Note that when you press this button it will by default create a folder in your currently active folder, so select the `main.py` file after creating the first folder or the second folder will be created inside the first folder.

[61]https://docs.pytest.org/en/stable/
[62]https://en.wikipedia.org/wiki/Test-driven_development

You want both the folders to be at the root level of your project.

Now add a file at the root level of the project called `__init__.py`. This is a special file that indicates to Python that we want our project to be treated as a "module": something that other files can refer to by name and import pieces from. Also add an `__init__.py` file inside the `utils` folder and the `tests` folder. These files will remain empty, but it's important that they exist for our tests to run. Their presence specifies that our main project should be treated as a module and that any code in our `utils` and `tests` folders should be treated as submodules of the main one.

Finally, create the files where we'll actually write code. Inside the `utils` folder create a file called `name_helper.py` and inside the `tests` folder create one called `test_name_helper.py`. Your project should now look as follows. Make sure that you have all the files and folders with exactly these names, in the correct places.

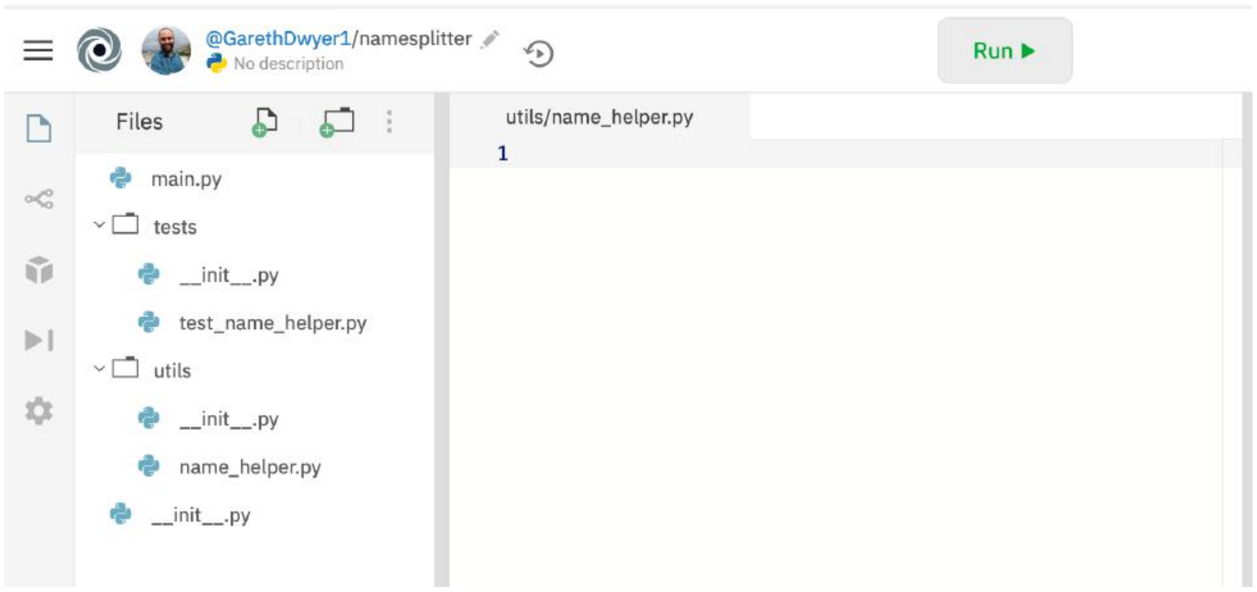


Image 1: ***Setting up our project structure for*** `pytest`.

Defining examples for the name split function

Splitting names is useful in many contexts. For example, it is a common requirement when users sign up on websites with their full names and then companies want to send personalised emails addressing users by their first name only. You might think that this is as simple as splitting a name based on spaces as in the following example.

```python
def split_name(name):
    first_name, last_name = name.split()
    return [first_name, last_name]

print(split_name("John Smith"))
# >>> ["John", "Smith"]
```

While this does indeed work in many cases, names are surprisingly complicated and it's very common to make mistakes when dealing with them as programmers, as discussed in this classic article[63]. It would be a huge project to try and deal with *any* name, but let's imagine that you have requirements to deal with the following kinds of names:

- First Last, e.g. John Smith
- First Middle Last, e.g John Patrick Smith (John Patrick taken as first name)
- First Middle Middle Last, e.g. John Patrick Thomson Smith (John Patrick Thomson taken as first name)
- First last last Last, e.g. Johan van der Berg (note the lowercase letters, Johan taken as first name, the rest as last)
- First Middle last last Last, e.g. Johan Patrick van der Berg (note the lowercase letters, Johan taken as first name, the rest as last)
- Last, e.g. Smith (we can assume that if we are given only one name, it is the last name)

Specifically, you can assume that once you find a name starting with a lowercase letter, it signifies the start of a last name, and that all other names starting with a capital letter are part of the first and middle names. Middle names can be combined with first names.

Of course, this does not cover all possibilities, but it is a good starting point in terms of requirements.

Using TDD, we always write *failing tests* first. The idea is that we should write a test about how some code *should* behave, check to make sure that it breaks in the way we expect (as the code isn't there). Only then do we write the actual code and check that the tests now pass.

Writing the test cases for our names function

Now that we understand what our function should do, we can write tests to check that it does. In the `tests/test_name_helper.py` file, add the following code.

[63]https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/

```python
from namesplitter.utils import name_helper

def test_two_names():
    assert name_helper.split_name("John Smith") == ["John", "Smith"]
```

Note that the `namesplitter` in the first line is taken from the name of your Repl.it project, which defines the names of the parent module. If you called your project something else, you'll need to use that name in the import line. It's important to not include special characters in the project name (including a hyphen, so names like `my-tdd-demo` are out) or the import won't work.

The `assert` keyword simply checks that a specific statement evaluates to `True`. In this case, we call our function on the left-hand side and give the expected value on the right-hand side, and ask `assert` to check if they are the same.

This is our most basic case: we have two names and we simply split them on the single space. Of course, we haven't written the `split_name` function anywhere yet, so we expect this test to fail. Let's check.

Usually you would run your tests by typing `py.test` into your terminal, but using Repl.it things work better if we import `pytest` into our code base and run it from there. This is because a) our terminal is always already activated into a Python environment and b) caching gets updated when we press the `Run` button, so invoking our tests from outside of this means that they could run on old versions of our code, causing confusion.

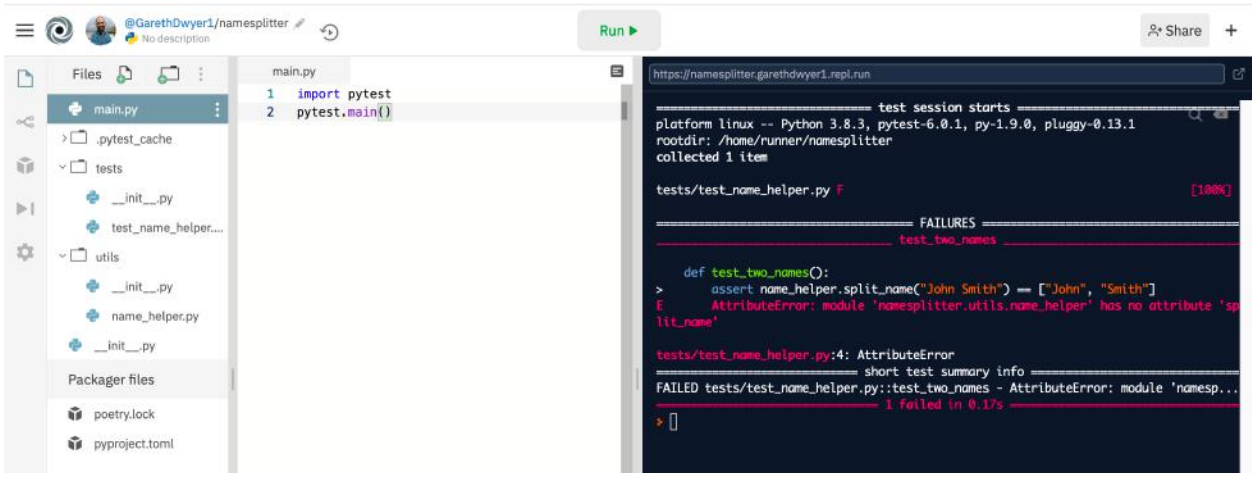
Let's run them from our `main.py` file for now as we aren't using it for anything else yet. Add the following to this file.

```python
import pytest
pytest.main()
```

Press the `Run` button. `pytest` does automatic test discovery so you don't need to tell it which tests to run. It will look for files that start with `test` and for functions that start with `test_` and assume these are tests. (You can read more about exactly how test discovery works and can be configured here[64].)

You should see some scary looking red failures, as shown below. (`pytest` uses dividors such as `======` and `------` to format sections and these can get messy if your output pane is too narrow. If things look a bit wonky try making it wider and rerunning.)

[64]https://docs.pytest.org/en/reorganize-docs/new-docs/user/naming_conventions.html



```
import pytest
pytest.main()
```

```
================================ test session starts ================================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/runner/namesplitter
collected 1 item

tests/test_name_helper.py F                                                   [100%]

===================================== FAILURES ======================================
__________________________________ test_two_names ___________________________________

    def test_two_names():
>       assert name_helper.split_name("John Smith") == ["John", "Smith"]
E       AttributeError: module 'namesplitter.utils.name_helper' has no attribute 'sp
lit_name'

tests/test_name_helper.py:4: AttributeError
============================== short test summary info ==============================
FAILED tests/test_name_helper.py::test_two_names - AttributeError: module 'namesp...
================================= 1 failed in 0.17s =================================
> 
```

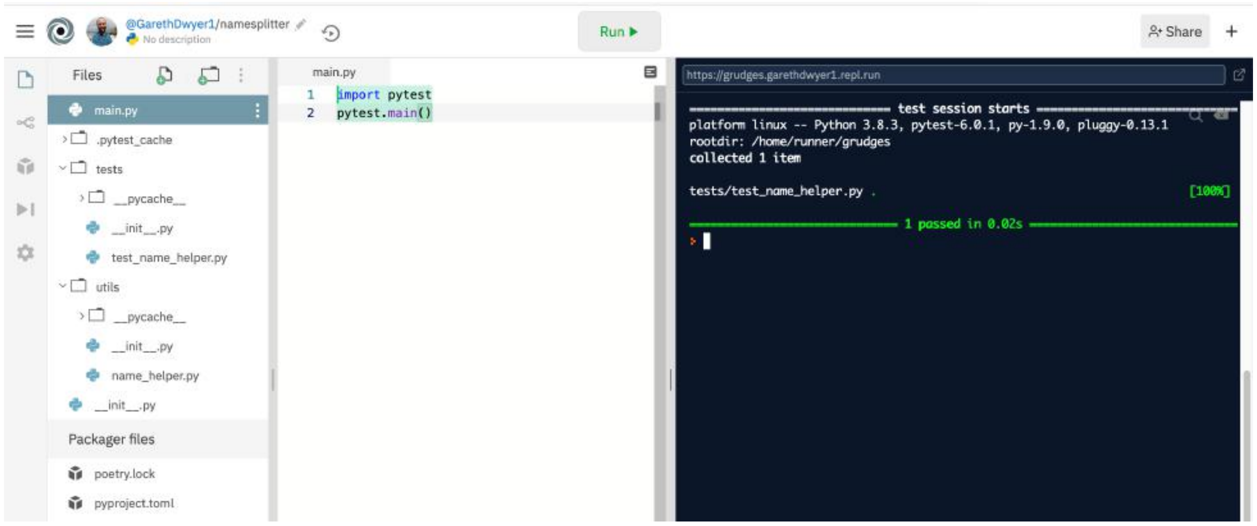
Image 2: ***Reading the* `pytest` *error messages*.**

If you read the output from the top down you'll see a bunch of different things happened. First, `pytest` ran test discovery and found one test. It ran this and it failed so you see the first red `F` above the `FAILURES` section. That tells us exactly which line of the test failed and how. In this case, it was an `AttributeError` as we tried to use `split_name` which was not defined. Let's go fix that.

Head over to the `utils/name_helper.py` file and add the following code.

```python
def split_name(name):
    first_name, last_name = name.split()
    return [first_name, last_name]
```

This is the very simple version we discussed earlier that can only handle two names, but it will solve the name error and TDD is all about small increments. Press `Run` to re-run the tests and you should see a far more friendly green output now, as below, indicating that all of our tests passed.



```
import pytest
pytest.main()
```

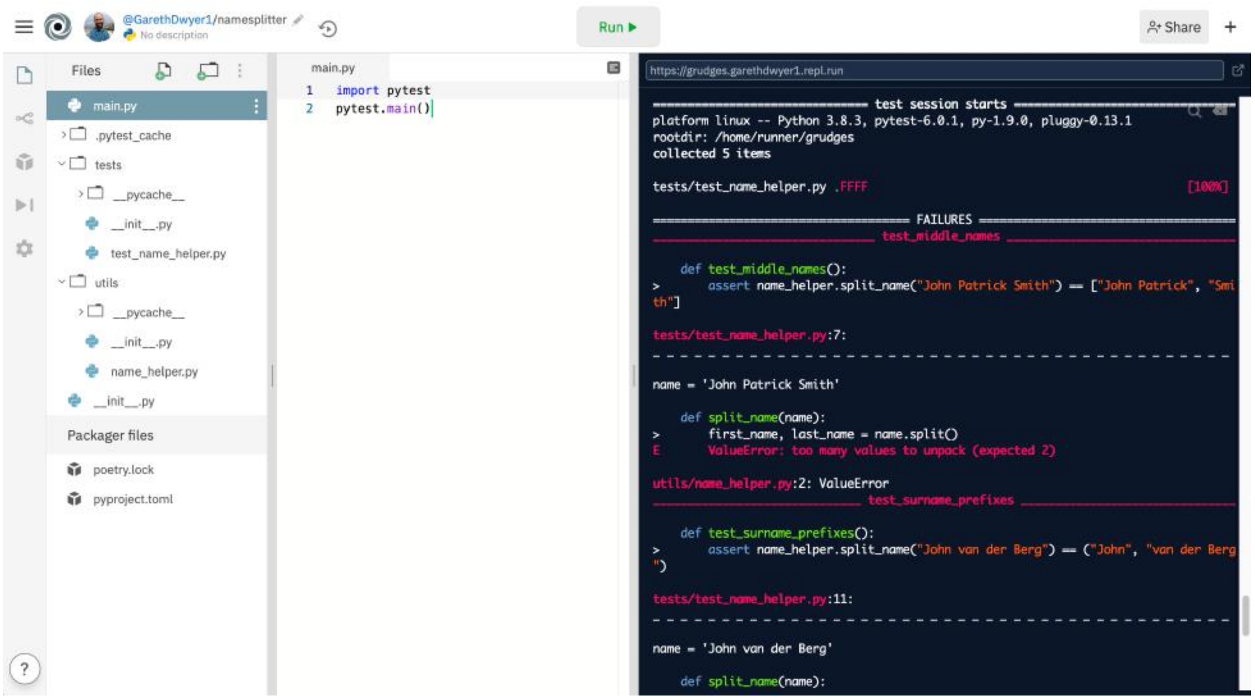
```
============================= test session starts ==============================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/runner/grudges
collected 1 item

tests/test_name_helper.py .                                              [100%]

============================== 1 passed in 0.02s ===============================
>
```

Image 3: *Seeing our tests pass after updating the code.*

Before fixing our function to handle more complex cases, let's first write the tests and check that they fail. Go back to `tests/test_name_helper.py` and add the following four test functions beneath the existing one.

```
from namesplitter.utils import name_helper

def test_two_names():
    assert name_helper.split_name("John Smith") == ["John", "Smith"]

def test_middle_names():
    assert name_helper.split_name("John Patrick Smith") == ["John Patrick", "Smith"]
    assert name_helper.split_name("John Patrick Thomson Smith") == ["John Patrick Th\
omson", "Smith"]

def test_surname_prefixes():
    assert name_helper.split_name("John van der Berg") == ["John", "van der Berg"]
    assert name_helper.split_name("John Patrick van der Berg") == ["John Patrick", "\
van der Berg"]

def test_split_name_onename():
    assert name_helper.split_name("Smith") == ["", "Smith"]

def test_split_name_nonames():
    assert name_helper.split_name("") == ["", ""]
```

Rerun the tests and you should see a lot more output now. If you scroll back up to the most recent `===== test session starts =====` section, it should look as follows.



```
import pytest
pytest.main()
```

```
============================== test session starts ==============================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/runner/grudges
collected 5 items

tests/test_name_helper.py .FFFF                                          [100%]

=================================== FAILURES ====================================
______________________________ test_middle_names _______________________________

    def test_middle_names():
>       assert name_helper.split_name("John Patrick Smith") == ["John Patrick", "Smi
th"]

tests/test_name_helper.py:7:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

name = 'John Patrick Smith'

    def split_name(name):
>       first_name, last_name = name.split()
E       ValueError: too many values to unpack (expected 2)

utils/name_helper.py:2: ValueError
____________________________ test_surname_prefixes _____________________________

    def test_surname_prefixes():
>       assert name_helper.split_name("John van der Berg") == ("John", "van der Berg
")

tests/test_name_helper.py:11:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

name = 'John van der Berg'

    def split_name(name):
```

Image 4: ***Seeing more failures after adding more tests.***

In the top section, the `.FFFF` is shorthand for "five tests were run, the first one passed and the next four failed" (a green dot indicates a pass and a red F indicates a failure). If you had more files with tests in them, you would see a line like this per file, with one character of output per test.

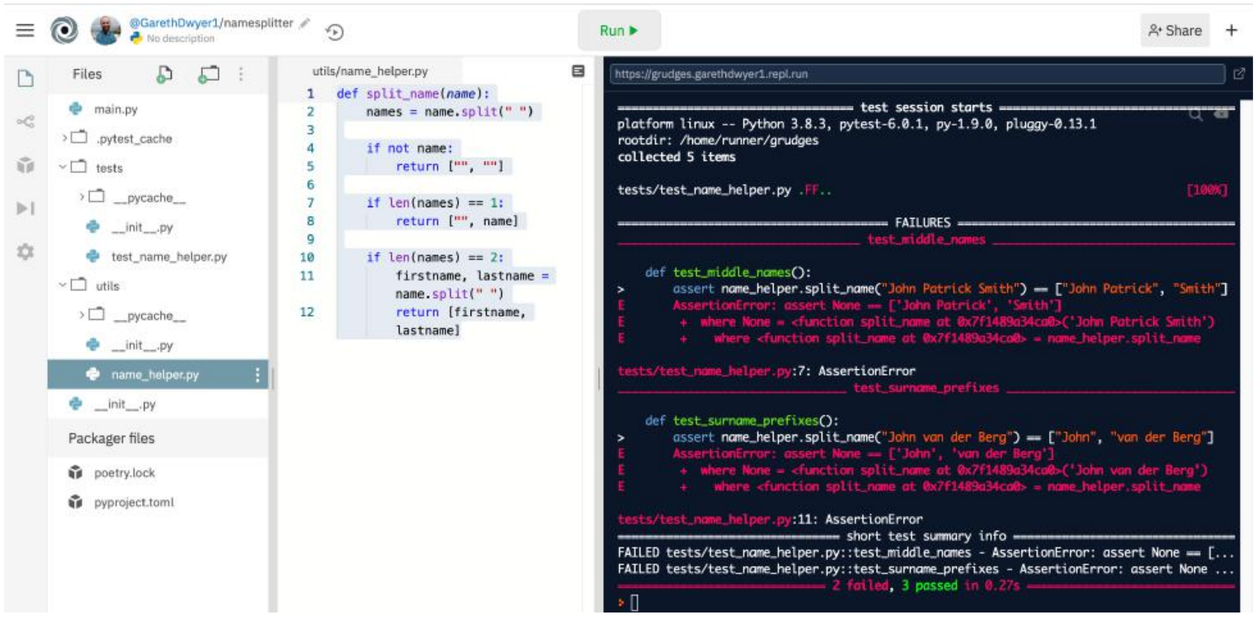
The failures are described in detail after this, but they all amount to variations of the same problem. Our code currently assumes that we will always get exactly two names, so it either has too many or too few values after running `split()` on the test examples.

Fixing our `split_name` function

Go back to `name_helper.py` and modify it to look as follows.

```python
def split_name(name):
    names = name.split(" ")

    if not name:
        return ["", ""]

    if len(names) == 1:
        return ["", name]

    if len(names) == 2:
        firstname, lastname = name.split(" ")
        return [firstname, lastname]
```

This should handle the case of zero, one, or two names. Let's run our tests again to see if we've made progress before we handle the more difficult cases. You should get a lot less output now and three green dots, as shown below.



```
def split_name(name):
    names = name.split(" ")

    if not name:
        return ["", ""]

    if len(names) == 1:
        return ["", name]

    if len(names) == 2:
        firstname, lastname =
        name.split(" ")
        return [firstname,
        lastname]
```

```
================================ test session starts =================================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/runner/grudges
collected 5 items

tests/test_name_helper.py .FF..                                                 [100%]

======================================= FAILURES =======================================
_______________________________ test_middle_names _____________________________________

    def test_middle_names():
>       assert name_helper.split_name("John Patrick Smith") == ["John Patrick", "Smith"]
E       AssertionError: assert None == ['John Patrick', 'Smith']
E        +  where None = <function split_name at 0x7f1489a34ca0>('John Patrick Smith')
E        +    where <function split_name at 0x7f1489a34ca0> = name_helper.split_name

tests/test_name_helper.py:7: AssertionError
_____________________________ test_surname_prefixes ___________________________________

    def test_surname_prefixes():
>       assert name_helper.split_name("John van der Berg") == ["John", "van der Berg"]
E       AssertionError: assert None == ['John', 'van der Berg']
E        +  where None = <function split_name at 0x7f1489a34ca0>('John van der Berg')
E        +    where <function split_name at 0x7f1489a34ca0> = name_helper.split_name

tests/test_name_helper.py:11: AssertionError
=============================== short test summary info ===============================
FAILED tests/test_name_helper.py::test_middle_names - AssertionError: assert None == [...
FAILED tests/test_name_helper.py::test_surname_prefixes - AssertionError: assert None ...
============================== 2 failed, 3 passed in 0.27s ============================
> 
```

Image 5: ***Progress: some of our tests pass now.***

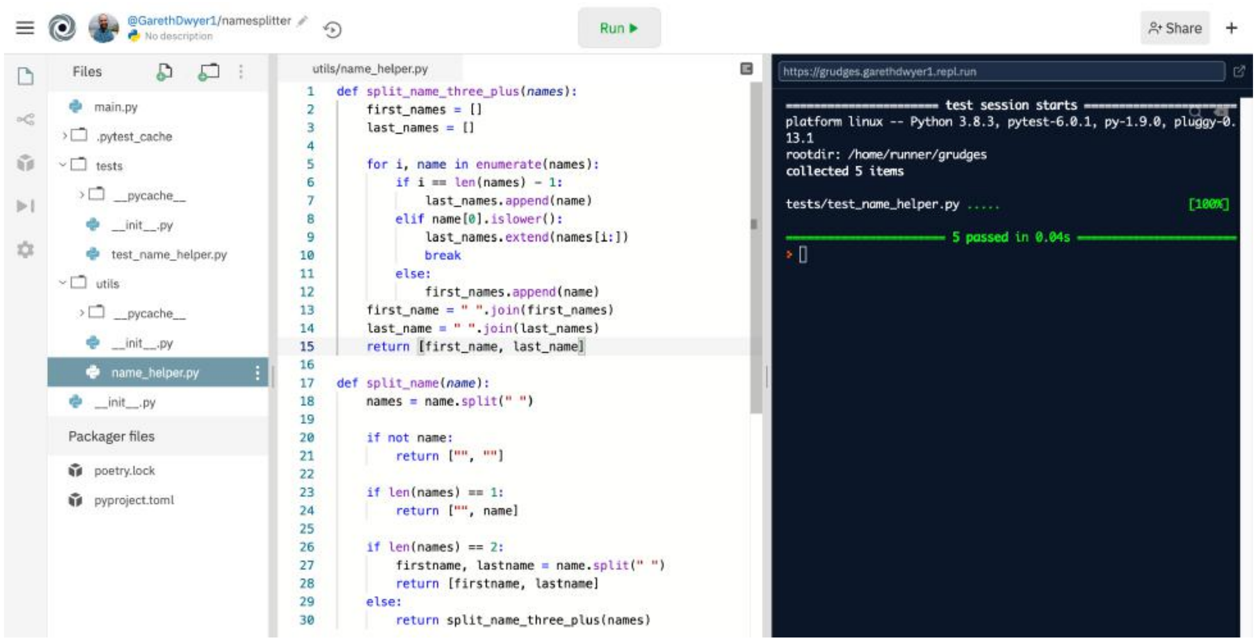
The rest of the output indicates that it's the middle names and surname prefix examples that are still tripping up our function, so let's add the code we need to fix those. Another important aspect of TDD is keeping your functions as small as possible so that they are easier to understand, test, and reuse, so let's write a second function to handle the three or more names cases.

Add the new function called `split_name_three_plus()` and add an `else` clause to the existing `split_name` function where you call this new function. The entire file should now look as follows.

```
def split_name_three_plus(names):
    first_names = []
    last_names = []

    for i, name in enumerate(names):
        if i == len(names) - 1:
            last_names.append(name)
        elif name[0].islower():
            last_names.extend(names[i:])
            break
        else:
```

```python
            first_names.append(name)
    first_name = " ".join(first_names)
    last_name = " ".join(last_names)
    return [first_name, last_name]

def split_name(name):
    names = name.split(" ")

    if not name:
        return ["", ""]

    if len(names) == 1:
        return ["", name]

    if len(names) == 2:
        firstname, lastname = name.split(" ")
        return [firstname, lastname]
    else:
        return split_name_three_plus(names)
```

The new function works by always appending names to the `first_names` list until it gets to the last name, or until it encounters a name that starts with a lowercase letter, at which point it adds all of the remaining names to `last_names` list. If you run the tests again, they should all pass now.



```python
def split_name_three_plus(names):
    first_names = []
    last_names = []

    for i, name in enumerate(names):
        if i == len(names) - 1:
            last_names.append(name)
        elif name[0].islower():
            last_names.extend(names[i:])
            break
        else:
            first_names.append(name)
    first_name = " ".join(first_names)
    last_name = " ".join(last_names)
    return [first_name, last_name]

def split_name(name):
    names = name.split(" ")

    if not name:
        return ["", ""]

    if len(names) == 1:
        return ["", name]

    if len(names) == 2:
        firstname, lastname = name.split(" ")
        return [firstname, lastname]
    else:
        return split_name_three_plus(names)
```

```
======================== test session starts =========================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.
13.1
rootdir: /home/runner/grudges
collected 5 items

tests/test_name_helper.py .....                                  [100%]

========================= 5 passed in 0.04s ==========================
> 
```

Image 6: ***All of the tests pass after adding a new function.***

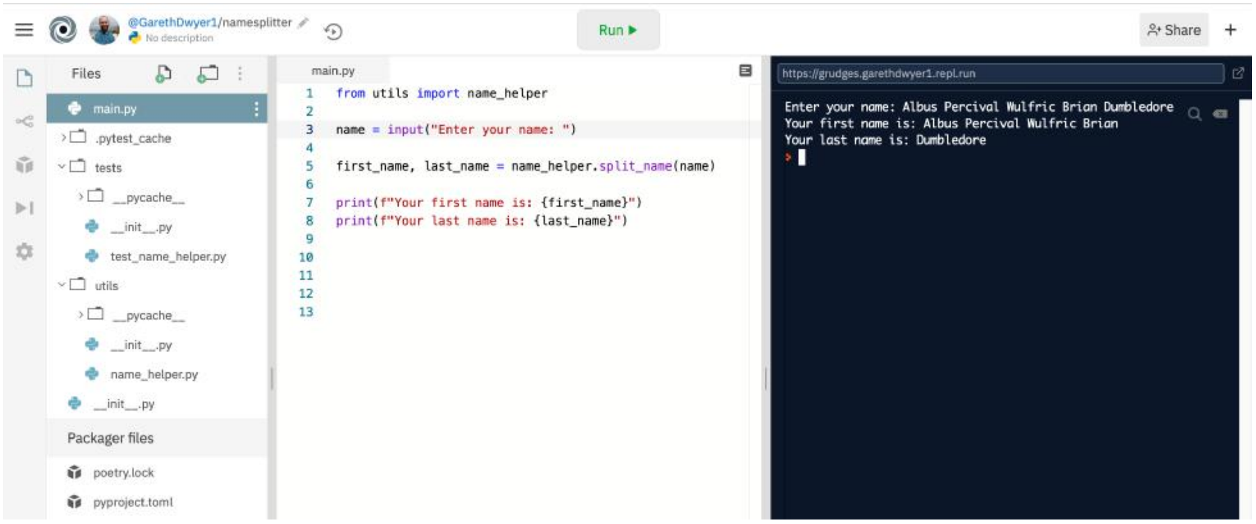
The tests were already helpful in making sure that we understood the problem and that our function worked for specific examples. If we had made any off-by-one mistakes in our code that deals with three or more names, our tests would have caught them. If we need to refactor or change our code in future, we can also use our tests to make sure that our new code doesn't introduce any regressions (where fixing problems causes code to break on other examples that worked before the fix.)

Using our function

Let's build a very basic application to use our function. Replace the testing code in `main.py` with the following.

```python
from utils import name_helper

name = input("Please enter your full name: ")

first_name, last_name = name_helper.split_name(name)

print(f"Your first name is: {first_name}")
print(f"Your last name is: {last_name}")
```

If you run this, it will prompt the user for their name and then display their first and last name.

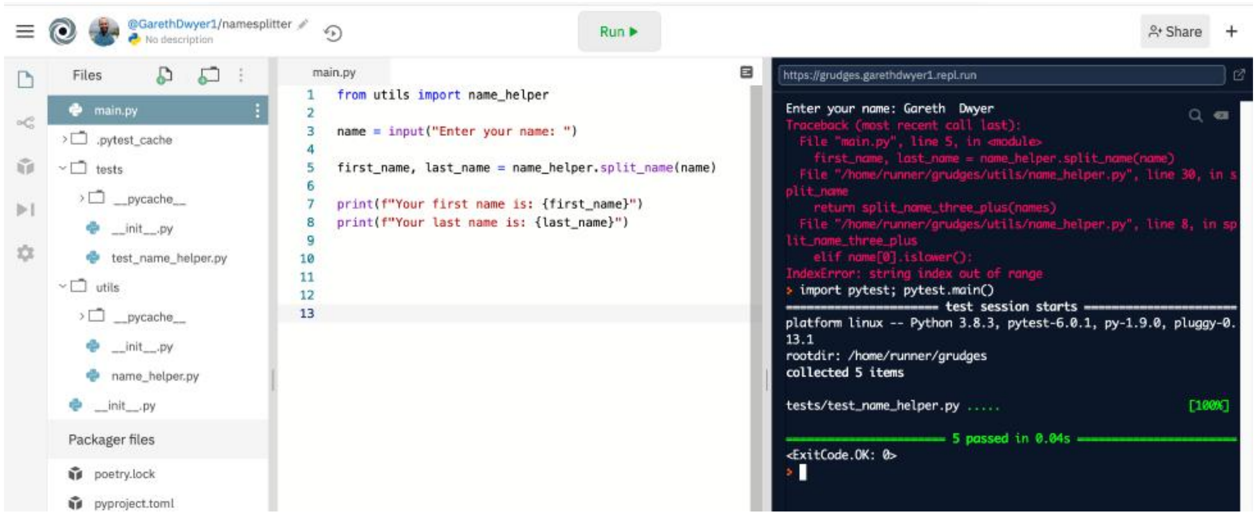


```python
from utils import name_helper

name = input("Enter your name: ")

first_name, last_name = name_helper.split_name(name)

print(f"Your first name is: {first_name}")
print(f"Your last name is: {last_name}")
```

```
Enter your name: Albus Percival Wulfric Brian Dumbledore
Your first name is: Albus Percival Wulfric Brian
Your last name is: Dumbledore
> 
```

Image 7: ***Using our function in a basic console application***.

Because you're using the `main.py` file now, you can also invoke `pytest` directly from the output console on the right by typing `import pytest; pytest.main()`. Note that updates to your code are only properly applied when you press the `Run` button though, so make sure to run your code between changes before running the tests.



```python
from utils import name_helper

name = input("Enter your name: ")

first_name, last_name = name_helper.split_name(name)

print(f"Your first name is: {first_name}")
print(f"Your last name is: {last_name}")
```

```
Enter your name: Gareth  Dwyer
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    first_name, last_name = name_helper.split_name(name)
  File "/home/runner/grudges/utils/name_helper.py", line 30, in s
plit_name
    return split_name_three_plus(names)
  File "/home/runner/grudges/utils/name_helper.py", line 8, in sp
lit_name_three_plus
    elif name[0].islower():
IndexError: string index out of range
> import pytest; pytest.main()
==================== test session starts =====================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.
13.1
rootdir: /home/runner/grudges
collected 5 items

tests/test_name_helper.py .....                            [100%]

===================== 5 passed in 0.04s ======================
<ExitCode.OK: 0>
>
```

Image 8: ***Triggering a new error and invoking `pytest` from the output pane.***

Make it your own

We've written a name splitter that can handle some names more complicated than just "John Smith". It's not perfect though: for example, if you put in a name with two consecutive spaces it will crash our program. You could fork the project and fix this by first writing a test with consecutive spaces and then modifying the code to handle this (and any other edge cases you can think of).

You can find the name splitter repl at https://repl.it/@GarethDwyer1/namesplitter[65]

Where next

You've learned to do TDD in this project. It's a popular style of programming, but it's not for everyone. Even if you decide not to use TDD, having tests is still very useful and it's not uncommon for large projects to have thousands or millions of tests.

Take a look at the big list of naughty strings[66] for a project that collects inputs that often cause software to break. You could also read How SQLite Is Tested[67] which explains how SQLite, a popular lightweight database, has 150 thousand lines of code and nearly 100 million(!) lines of tests.

In the next tutorial, we'll show you how to become a Repl.it poweruser by taking advantage of the productivity features it offers.

[65]https://repl.it/@GarethDwyer1/namesplitter
[66]https://github.com/minimaxir/big-list-of-naughty-strings
[67]https://www.sqlite.org/testing.html

Productivity hacks

The images in this chapter are mostly .gif files, click here[68] *to access the web version of this chapter*

After coding for a while, you may find that there are some repetitive things that take up unnecessary time. For example, searching for and updating a variable name can seem laborious. Luckily, Repl.it has some built-in productivity tools that we'll take a look at in this tutorial.

Specifically, you'll see how to:

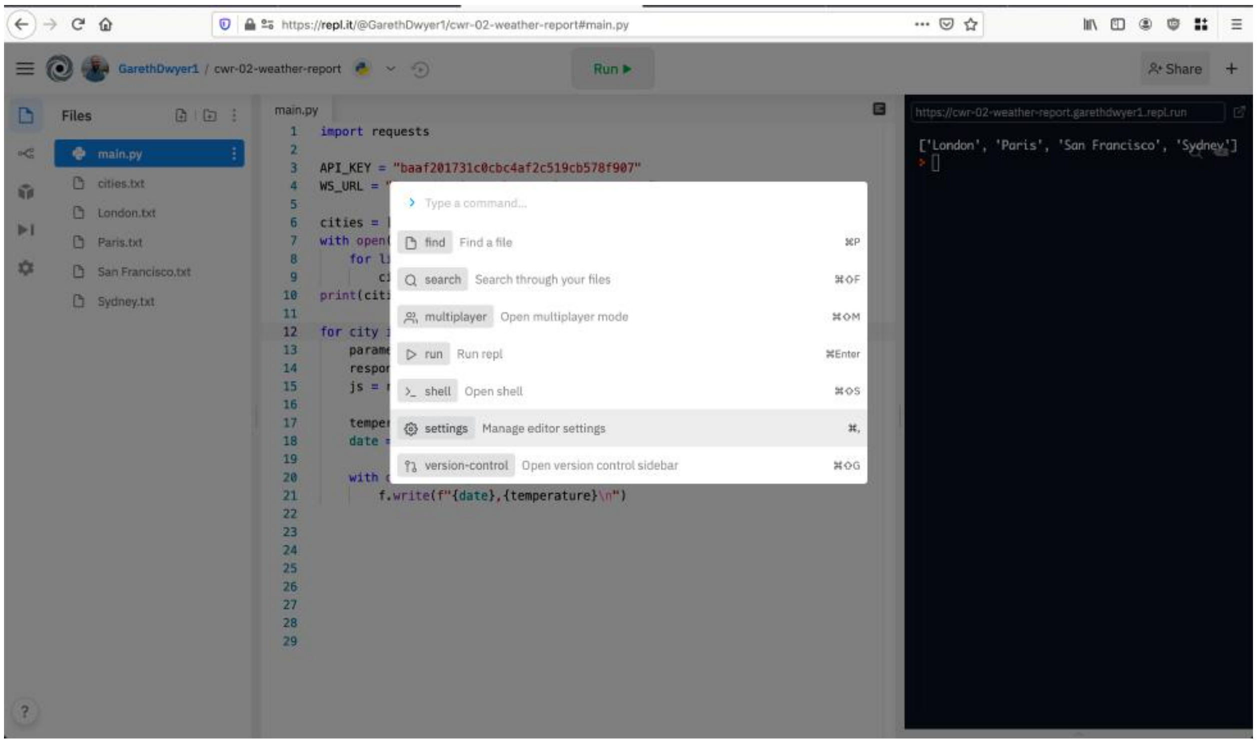
- Make simultaneous changes in several parts of your file using multiple cursors
- Use keyboard shortcuts to quickly carry out tasks without the delay of reaching for your mouse
- Switch to Vim or Emacs keybindings for full mouseless control.

Similarly to learning to touch type, there is often a steep *learning curve* when you start to use advanced code editing features. They might even substantially slow you down at first, but once you master them you'll soar past the limits of what you could achieve without these aids.

Using the global command palette

If you hit `Ctrl+K` (`Cmd+K` on MacOS) you'll see the following modal pop up, which lets you navigate through different parts of Repl.it at lightning speed using only your keyboard. If you have a lot of files, it's often useful to open them like this rather than scrolling through the directory structure in the files pane (the `find` option searches through files by their *name* while the `search` option searches through files by their *contents*.)

[68] https://docs.repl.it/tutorials/10-productivity-hacks



```
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "
cities = [
with open(
    for li
        ci
print(citi

for city i
    parame
    respor
    js = r

    temper
    date =

    with c
        f.write(f"{date},{temperature}\n")
```

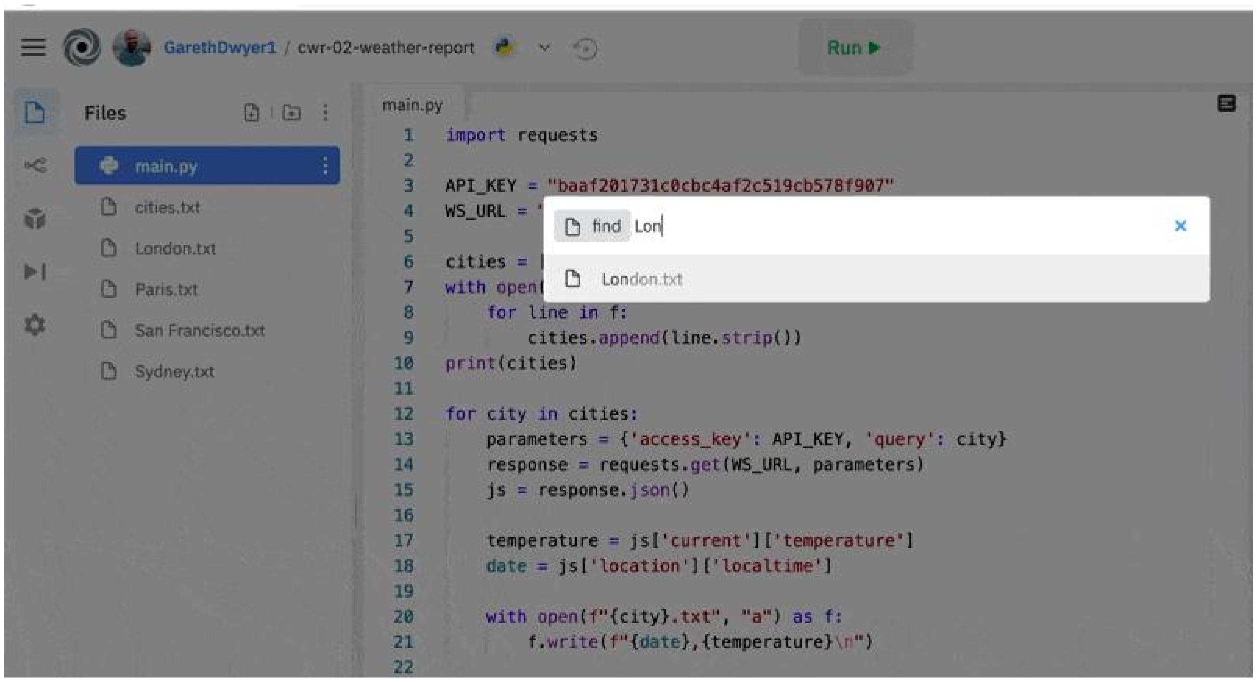
```
> Type a command...
find             Find a file                   ⌘P
search           Search through your files     ⌘⇧F
multiplayer      Open multiplayer mode         ⌘⇧M
run              Run repl                      ⌘Enter
shell            Open shell                    ⌘⇧S
settings         Manage editor settings        ⌘,
version-control  Open version control sidebar  ⌘⇧G
```

```
['London', 'Paris', 'San Francisco', 'Sydney']
> 
```

https://repl.it/@GarethDwyer1/cwr-02-weather-report#main.py
GarethDwyer1 / cwr-02-weather-report
Run ▶
Share
Files
main.py
cities.txt
London.txt
Paris.txt
San Francisco.txt
Sydney.txt
https://cwr-02-weather-report.garethdwyer1.repl.run

Image 1: ***Using the global command palette.***

The keyboard shortcut indicated to the right of each option shows how to activate that option directly without opening up the global command palette, but once it's open you can type in a part of any of the options to activate that option. For example, in our weather project app, I can type `Cmd+K` and then type `fi` (start of `find`) and press `Enter` and then type `Lo` (start of `London.txt`) and press `Enter` again to quickly open the weather log for London.



```
import requests

API_KEY = "baaf201731c0cbc4af2c519cb578f907"
WS_URL = "

cities = [
with open(
    for line in f:
        cities.append(line.strip())
print(cities)

for city in cities:
    parameters = {'access_key': API_KEY, 'query': city}
    response = requests.get(WS_URL, parameters)
    js = response.json()

    temperature = js['current']['temperature']
    date = js['location']['localtime']

    with open(f"{city}.txt", "a") as f:
        f.write(f"{date},{temperature}\n")
```

Image 2: ***Opening a file with the global command palette.***

Of course, with only six files it might be faster to reach for my mouse, but as the find searches through all files in all directories this method can be significantly faster for larger projects.

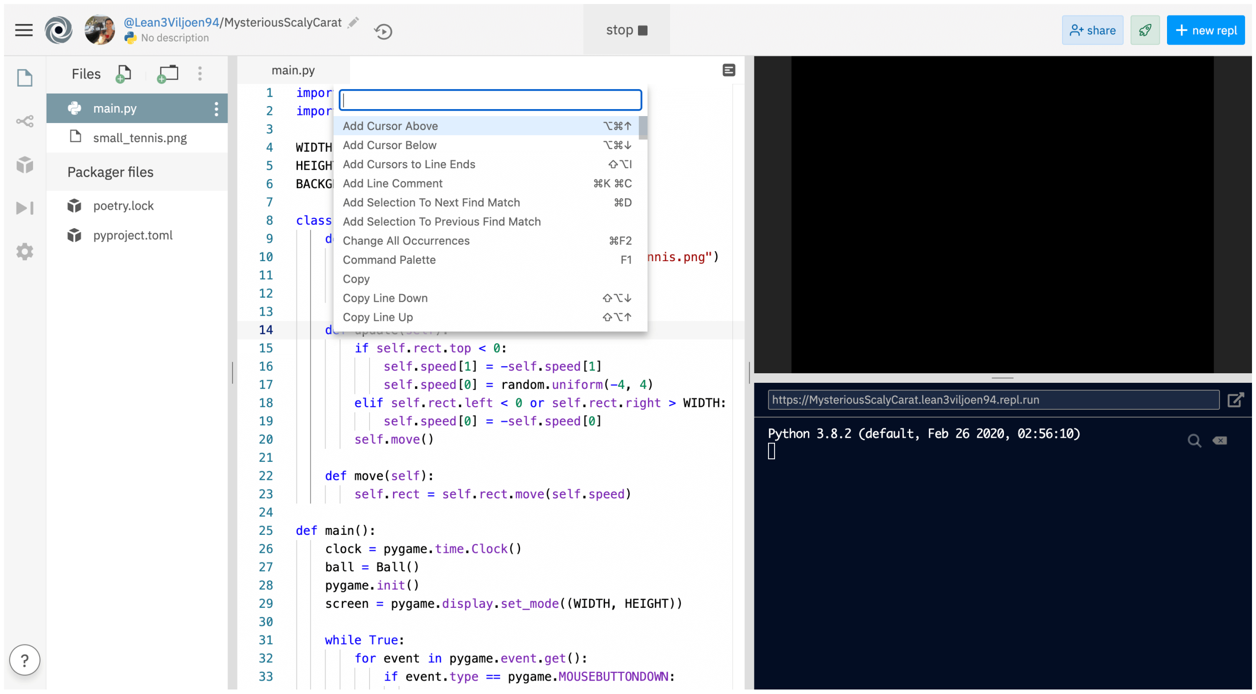
Opening up the multiplayer, version control, and settings panes using this method is also faster once the habit is ingrained compared to moving the mouse to the small icons on the left bar. And while pressing `Ctrl+Enter` or `Cmd+Enter` to run your code is faster than choosing "Run" from this global command ette, `Ctrl+K` is only a single shortcut to remember and it will remind you of any other shortcuts you can't recall.

Using the code editing command palette

The code command palette is similar to the global command palette, but it's specific to editing and navigating your code, allowing you to do advance find+replace procedures, jump to specific functions and more.

To access the code command palette press `F1` or `Ctrl+Shift+P` (`cmd+shift+P` on MacOS). Note that if you're using Firefox the second option will open an incognito window instead.

You can use the shortcuts directly from the command palette by selecting the code you wish to edit and clicking on the command in the drop-down menu, or use it to refresh your memory on the keybindings associated with the shortcuts you use often.



```
impor
impor

WIDTH
HEIGH
BACKG

class
    d
                                    nnis.png")



    d
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.uniform(-4, 4)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

Image 3: *Opening the command palette.*

Let's take a look at how these work by editing the PyGame juggling project[69] that we covered in a previous tutorial[70].

Instead of carrying out the suggested operations as you usually would, use Repl.it's productivity features instead.

Duplicating entire lines

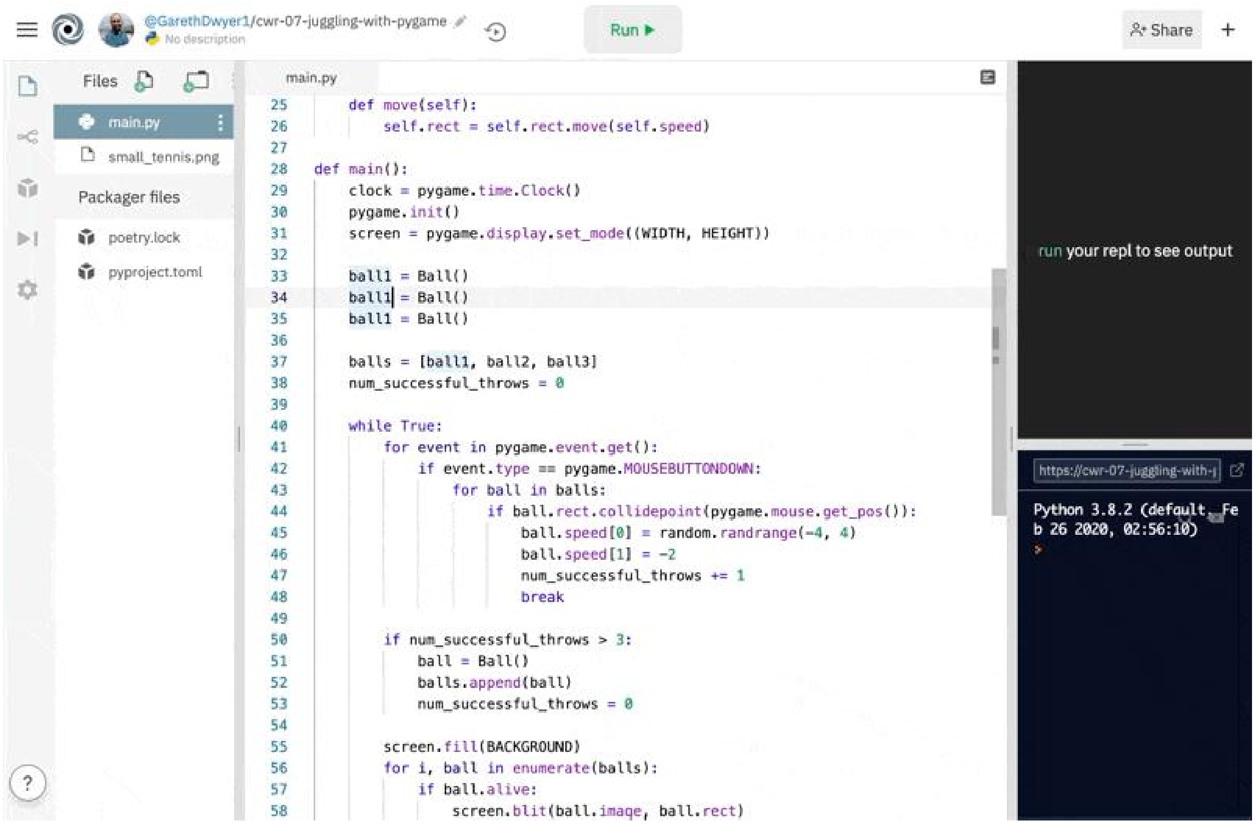
Sometimes you need two very similar lines of code directly after each other. Instead of copying and pasting the line or typing it out again, you can use the *duplicate row* feature, which will replicate the current line either above or below.

For example, our juggling project includes the following code to instantiate the initial three balls.

```
ball1 = Ball()
ball2 = Ball()
ball3 = Ball()
```

Instead of typing out all three lines, you can type out the first one, leave your cursor position on that line, and press `Shift+Alt+down` (`shift+option+down` on MacOS) twice. This will create two copies of the line, directly below the original one, and then you can simply change the number in the variable to account for the second two balls.

[69]https://repl.it/@GarethDwyer1/cwr-07-juggling-with-pygame
[70]http://www.codewithrepl.it/07-building-a-game-with-pygame.html



```
    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    ball1 = Ball()
    ball1 = Ball()
    ball1 = Ball()

    balls = [ball1, ball2, ball3]
    num_successful_throws = 0

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                for ball in balls:
                    if ball.rect.collidepoint(pygame.mouse.get_pos()):
                        ball.speed[0] = random.randrange(-4, 4)
                        ball.speed[1] = -2
                        num_successful_throws += 1
                        break

        if num_successful_throws > 3:
            ball = Ball()
            balls.append(ball)
            num_successful_throws = 0

        screen.fill(BACKGROUND)
        for i, ball in enumerate(balls):
            if ball.alive:
                screen.blit(ball.image, ball.rect)
```

run your repl to see output

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Image 4: ***Copying the current selected line.***

Deleting entire lines

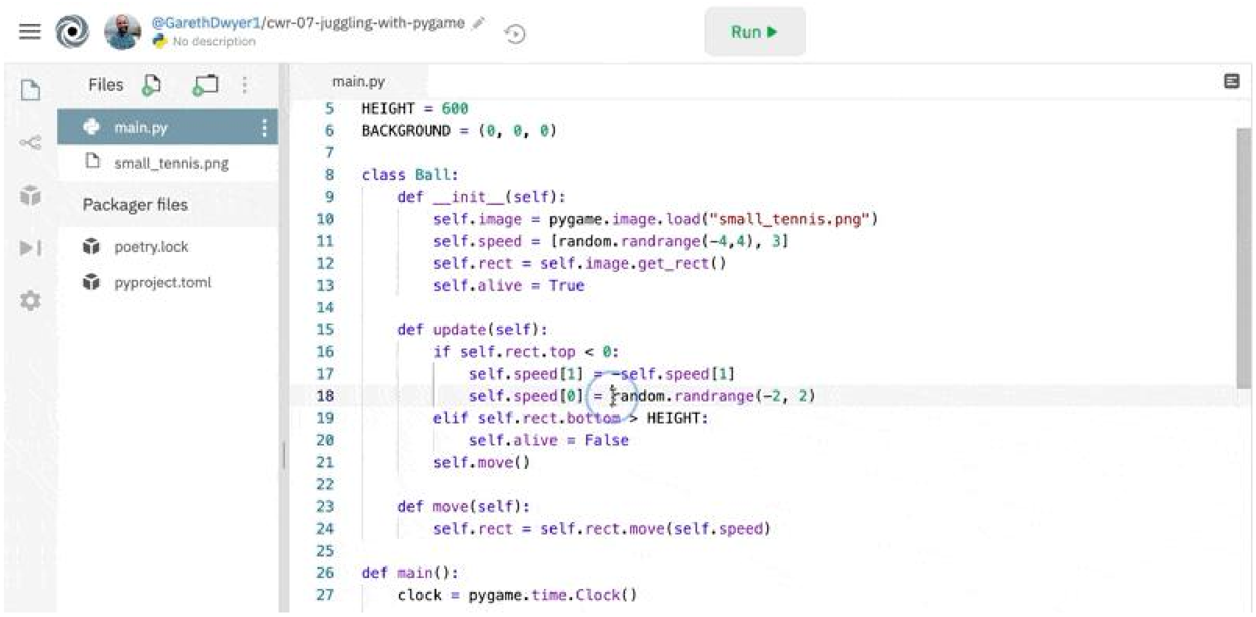
There may be instances where you'd want to delete large chunks of code at a time (it happens to the best of us!).

Pressing `Ctrl+Shift+K` (`cmd+shift+K` on MacOS) deletes the line underneath your cursor (or if you have multiple lines selected it will delete all of them.)

Instead of deleting the entire line, you can also delete from your cursor up to the end of the line or from your cursor to the beginning of the line. The shortcuts for these are

- `Ctrl+Backspace` (`cmd+backspace` on MacOS) to delete backwards
- `Ctrl+K` (same on MacOS) to delete forwards

As an example, below you can see how we might use this to first delete one of our `elif` blocks by doing two "delete line" operations. We then change our random speed to be constant by using a "delete to end of line" operation from the = sign and then typing our constant.



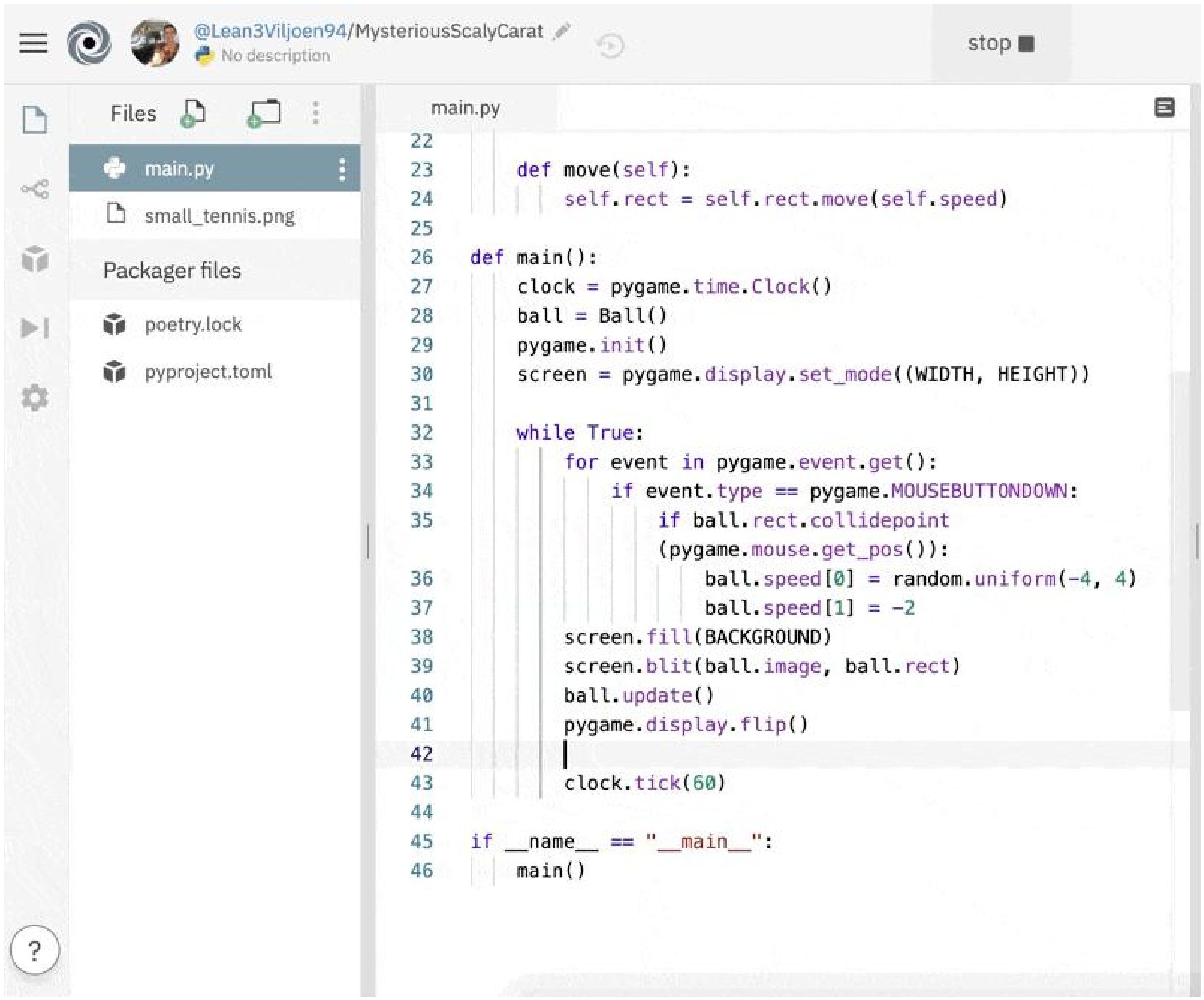
```python
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.randrange(-4,4), 3]
        self.rect = self.image.get_rect()
        self.alive = True

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.randrange(-2, 2)
        elif self.rect.bottom > HEIGHT:
            self.alive = False
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
```

Image 5: ***Deleting selected lines of code***.

Inserting blank lines

It’s also common to need to add a new line of code above or below the current one. Instead of using your mouse or arrow keys to get to the right place and then pressing `Enter`, you can instead use an “insert line” operation.

Press `Ctrl+Shift+Enter` (`cmd+shift+enter` on MacOS) to insert a blank line directly above the current one and move the cursor to the start of it (Repl.it will even maintain the current level of indentatin for you).



```python

    def move(self):
        self.rect = self.rect.move(self.speed)

def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if ball.rect.collidepoint
                (pygame.mouse.get_pos()):
                    ball.speed[0] = random.uniform(-4, 4)
                    ball.speed[1] = -2
        screen.fill(BACKGROUND)
        screen.blit(ball.image, ball.rect)
        ball.update()
        pygame.display.flip()

        clock.tick(60)

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

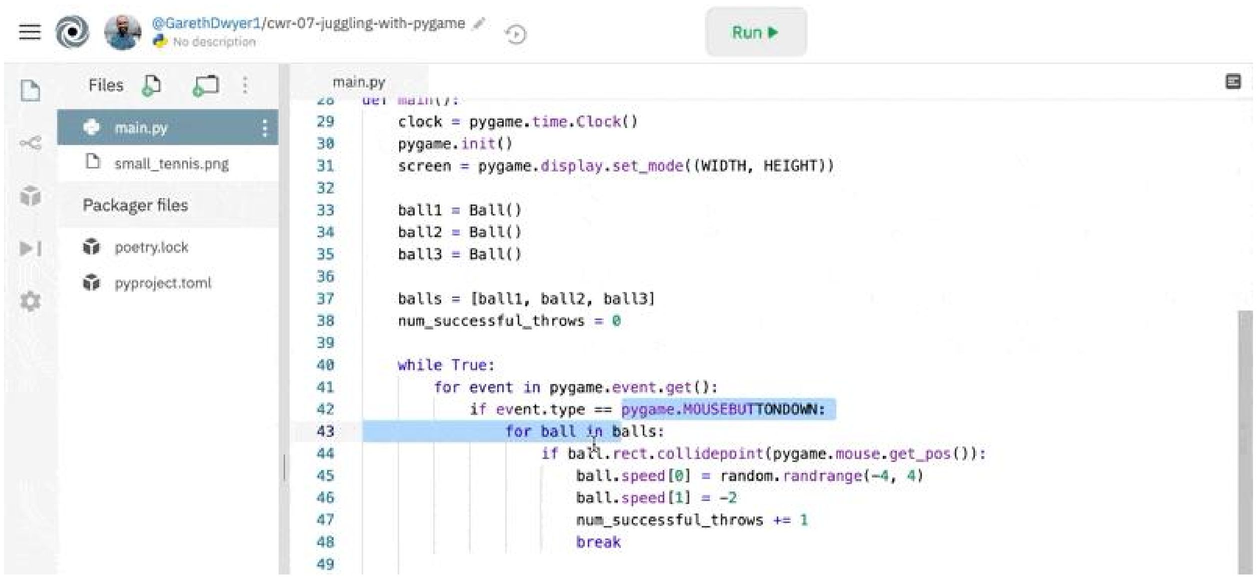
Image 6: ***Inserting lines.***

Indenting and dedenting lines

When writing Python, you probably pay more attention to whitespace (spaces or tabs) than in other langauges, which use braces to handle logic. You're probably used to indenting and dedenting using `Tab` and `Shift+Tab`, which requires you to first place the cursor at the start of the line.

Instead you can use `Ctrl+]` (`cmd+]` on MacOS) to indent and dedent the line no matter where your cursor is. For example, if you need to fix the indentation in the following code, you can
* put your cursor on the `for` line
* press `Ctrl+]`
* press `down`
* press `Shift+down`
* press `Ctrl+]` again twice.

Now your code's indentation will be fixed.



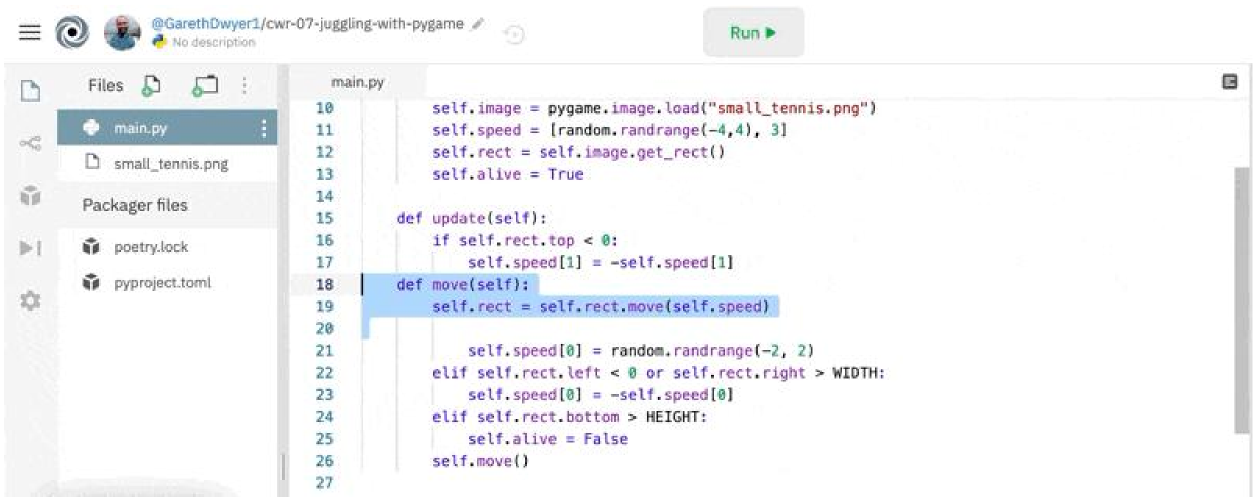
```python
def main():
    clock = pygame.time.Clock()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    ball1 = Ball()
    ball2 = Ball()
    ball3 = Ball()

    balls = [ball1, ball2, ball3]
    num_successful_throws = 0

    while True:
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                for ball in balls:
                    if ball.rect.collidepoint(pygame.mouse.get_pos()):
                        ball.speed[0] = random.randrange(-4, 4)
                        ball.speed[1] = -2
                        num_successful_throws += 1
                        break
```

Image 7: ***Indenting a line***.

Moving blocks of code within a file

Sometimes you need to move a block of code up or down in the file. For example, our `update()` function uses our `move()` function, but `move()` is only defined later. For readability, it's good to try ensure that your functions only call functions that have already been defined further up (assuming that someone else is reading the code top down, they will remember the `move()` function's definition before seeing it used).

Instead of cutting and pasting a block, you can shunt it by pressing `Alt+up` or `Alt+down` (`option+up` and `option+down` on MacOS). As with the others, this works on the line under your cursor or a larger selection.

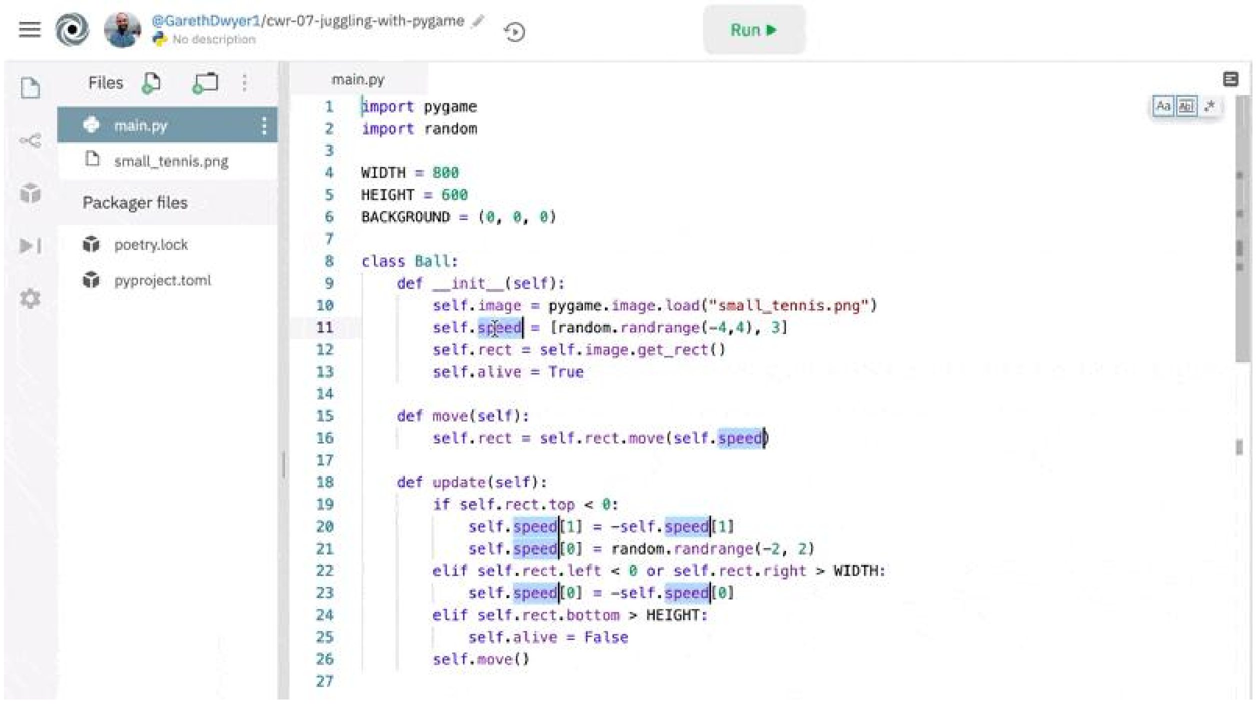


```
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.randrange(-4,4), 3]
        self.rect = self.image.get_rect()
        self.alive = True

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
    def move(self):
        self.rect = self.rect.move(self.speed)

            self.speed[0] = random.randrange(-2, 2)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        elif self.rect.bottom > HEIGHT:
            self.alive = False
        self.move()
```

Image 8: ***Moving the current line selection***.

Adding cursors

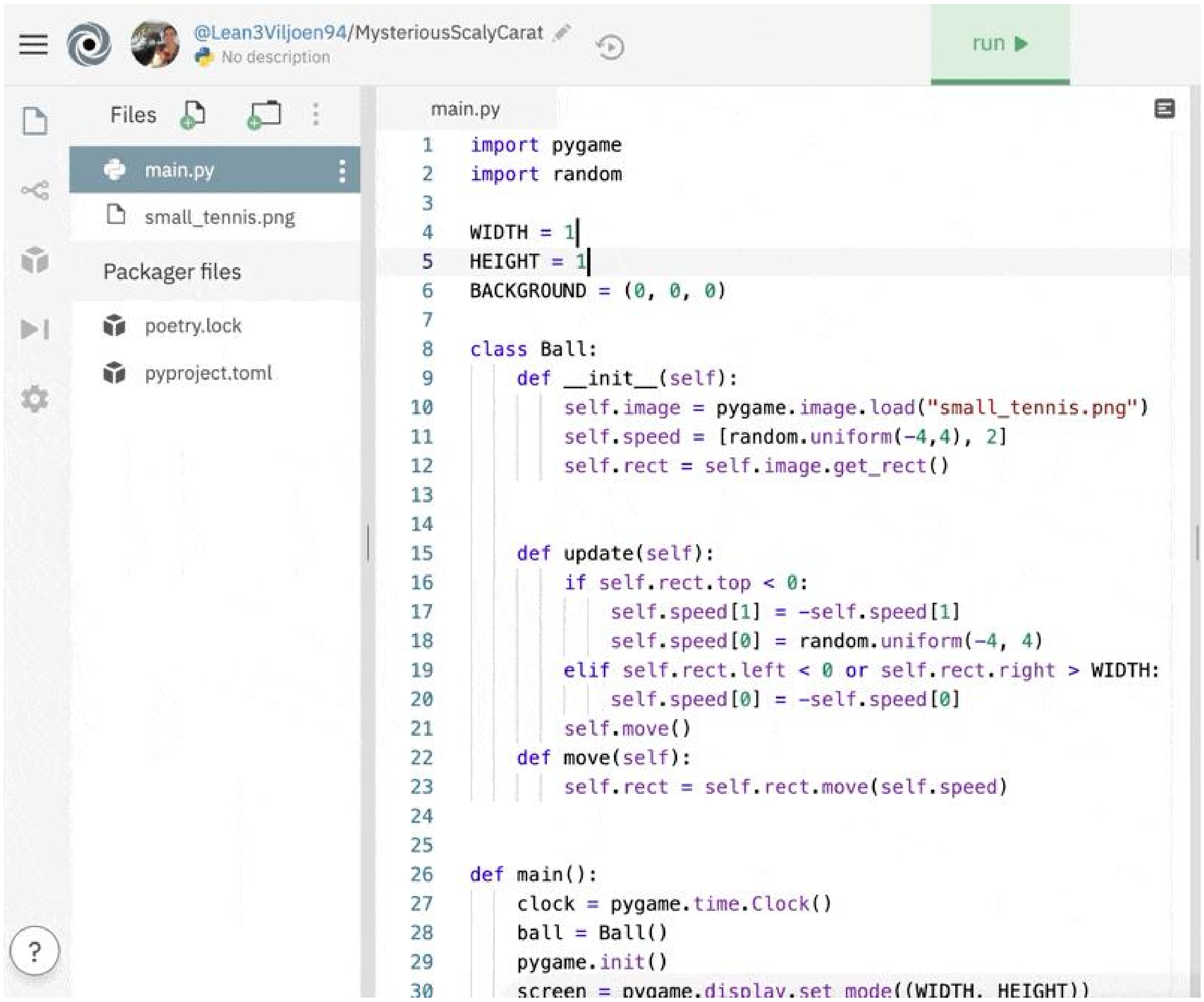
Sometimes it's useful to make exactly the same changes in multiple places at once. For example, we might want to rename our `speed` attribute to `velocity`. Put your cursor anywhere on the word that you want to change and press `Ctrl+D` (`cmd+D` on MacOS). Repeatedly press `Ctrl+D` to select matching words individually, each with their own cursor. Now you can apply edits and they will appear at each selection, as below.



```python
import pygame
import random

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.randrange(-4,4), 3]
        self.rect = self.image.get_rect()
        self.alive = True

    def move(self):
        self.rect = self.rect.move(self.speed)

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.randrange(-2, 2)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        elif self.rect.bottom > HEIGHT:
            self.alive = False
        self.move()

```

Image 9: ***Adding cursors to multiple instances of the same selection***.

If you want multiple cursors on consecutive lines, press `Ctrl+Alt+up` or `Ctrl+Alt+down` (`cmd+option+up` and `cmd+option+down` on MacOs). For example, if we want a square game we could change both width and height to be `1000` simultaneously as follows.



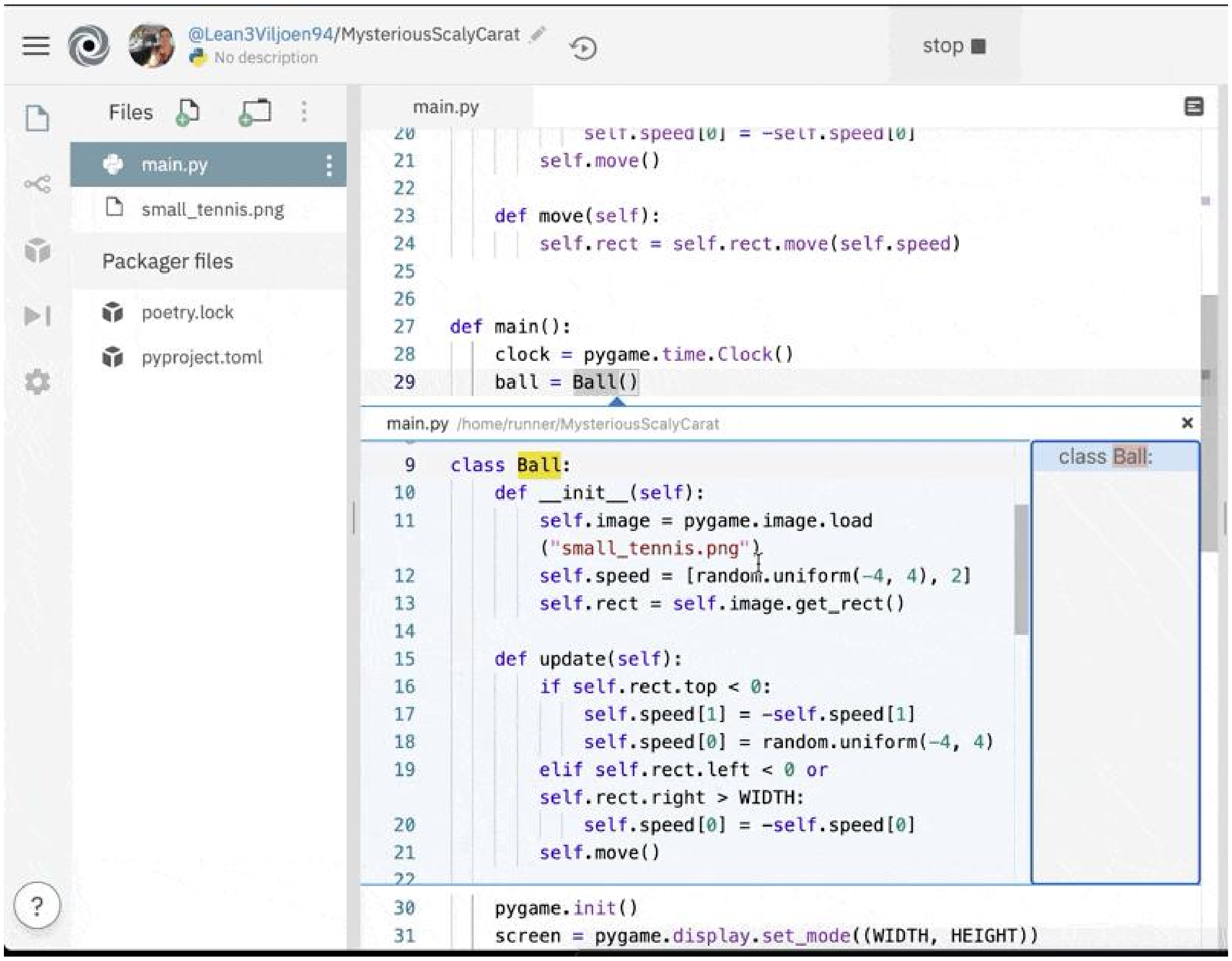
```
import pygame
import random

WIDTH = 1
HEIGHT = 1
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.uniform(-4,4), 2]
        self.rect = self.image.get_rect()


    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.uniform(-4, 4)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        self.move()
    def move(self):
        self.rect = self.rect.move(self.speed)


def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

Image 10: ***Adding cursors to multiple lines***.

Navigating to specific pieces of code

Sometimes, especially in larger projects, you'll call a function or instantiate an object far from where that function or object is defined (either thousands of lines away in the same file, or in a different file altogether).

If you're reading a piece of code that calls a function and you want to quickly see what that function actually does, you can use the **go to definition** keybinding (`F12` or `cmd+F12` on MacOS). This will jump to the definition of the function or class selected. The **peek definition** has a similar functionality, but instead of jumping to the definition, it opens in a separate modal. For example, below, the cursor is on the instantiation of `Ball()` and we can quickly see how this class is defined.



```
            self.speed[0] = -self.speed[0]
        self.move()

    def move(self):
        self.rect = self.rect.move(self.speed)


def main():
    clock = pygame.time.Clock()
    ball = Ball()
```

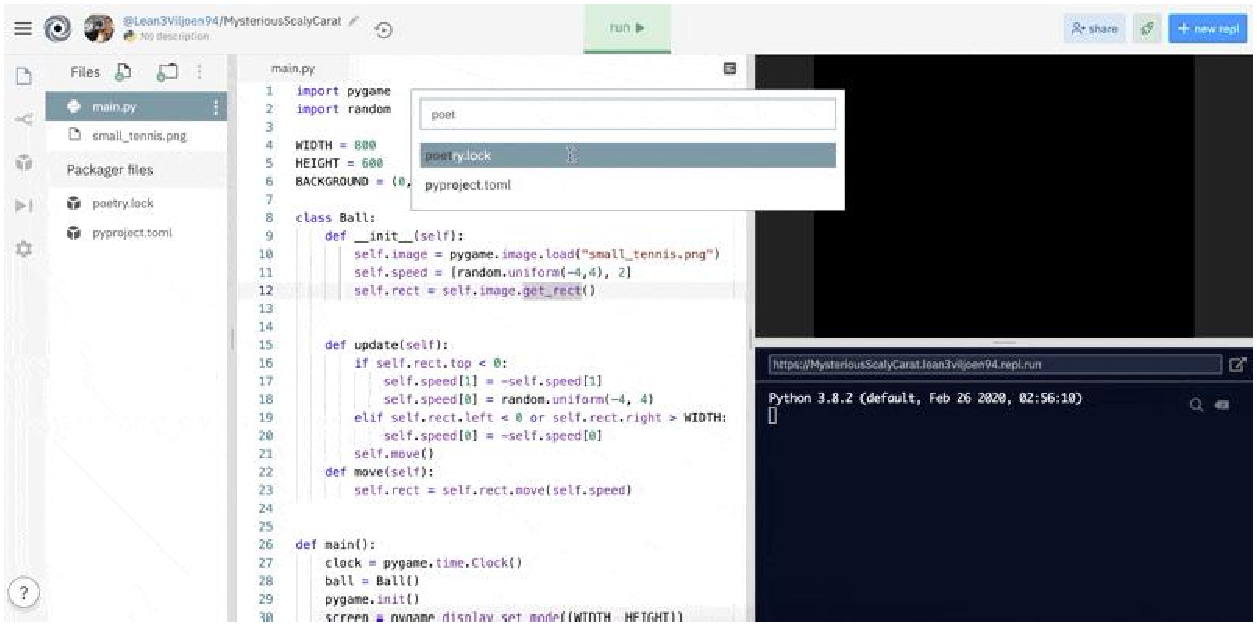
```
class Ball:
    def __init__(self):
        self.image = pygame.image.load
        ("small_tennis.png")
        self.speed = [random.uniform(-4, 4), 2]
        self.rect = self.image.get_rect()

    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.uniform(-4, 4)
        elif self.rect.left < 0 or
        self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        self.move()
```

```
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

Image 11: ***Peeking the definition***.

The **go to line** operation (`Ctrl+G`) allows you to navigate to a line by giving its line number. This is useful to track down the source of those error messages that tell you what line had an issue, or if you're on a call with someone who says "I'm looking at line 23" and you can quickly jump to the same place.

Finally, you can open a specific file by searching for a part of the name by pressing `Ctrl+P` (`cmd+P` on MacOS), which can be quicker than scrolling through the files pane if you have a lot of files.



```
import pygame
import random

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0,

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.speed = [random.uniform(-4,4), 2]
        self.rect = self.image.get_rect()


    def update(self):
        if self.rect.top < 0:
            self.speed[1] = -self.speed[1]
            self.speed[0] = random.uniform(-4, 4)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.speed[0] = -self.speed[0]
        self.move()
    def move(self):
        self.rect = self.rect.move(self.speed)


def main():
    clock = pygame.time.Clock()
    ball = Ball()
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

```
https://MysteriousScalyCarat.lean3viljoen94.repl.run
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

Image 12: ***Opening existing files.***

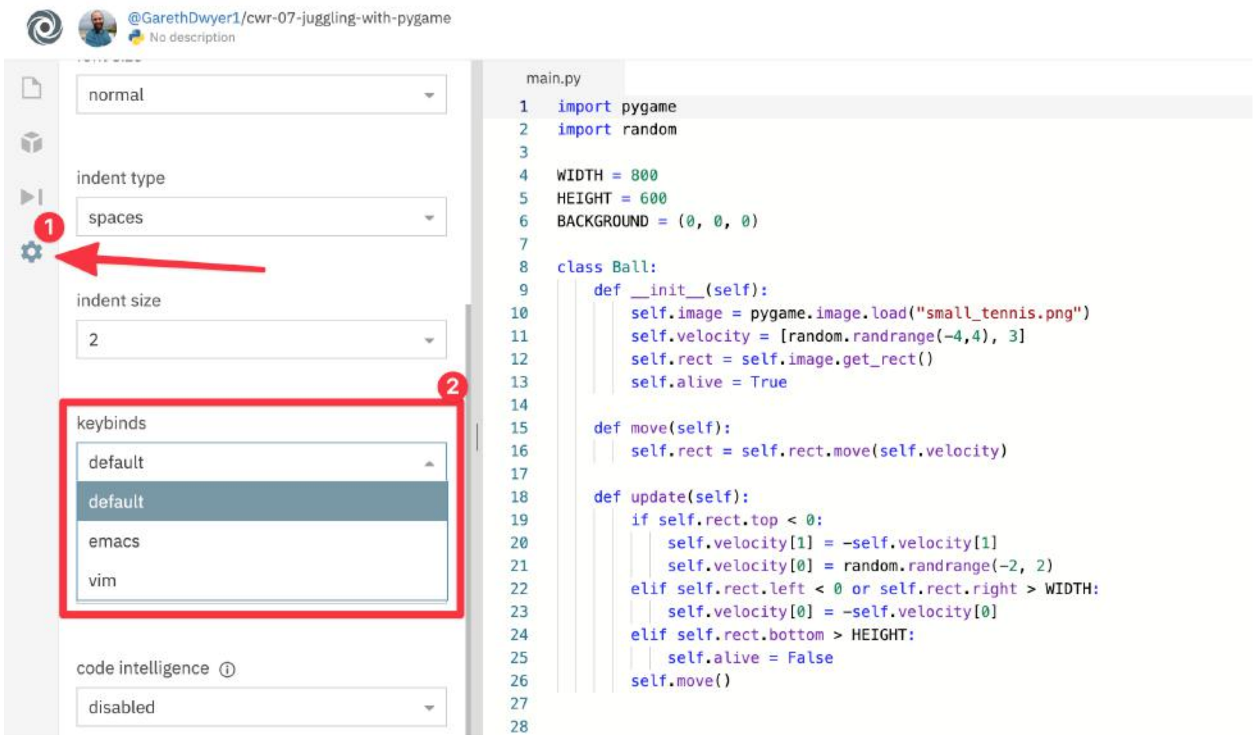
Vim and Emacs key bindings

Once you get hooked on keyboard shortcuts, you might wonder if you ever need to use your mouse again. Most of the time it only slows you down. Luckily, people thought of this decades ago. Before mice existed, all text editing was done using only a keyboard, and many developers still prefer editors that were created in this setting over more modern ones.

The two main keyboard-focused text editors are called Vim[71] and Emacs[72]. They both have steep learning curves (and there's a long-standing tradition that users of either fiercely argue about which is superior), but once you've put in the time to master them you can get rid of your mouse for good.

If you've gotten used to either, you can emulate the experience in Repl.it by switching your keybinds. Go to the "Settings" tab and scroll down to where you can toggle between "default", "emacs" and "vim".

[71]https://www.vim.org/
[72]https://www.gnu.org/software/emacs/



normal

indent type
spaces

indent size
2

keybinds
default
default
emacs
vim

code intelligence
disabled

main.py

```python
import pygame
import random

WIDTH = 800
HEIGHT = 600
BACKGROUND = (0, 0, 0)

class Ball:
    def __init__(self):
        self.image = pygame.image.load("small_tennis.png")
        self.velocity = [random.randrange(-4,4), 3]
        self.rect = self.image.get_rect()
        self.alive = True

    def move(self):
        self.rect = self.rect.move(self.velocity)

    def update(self):
        if self.rect.top < 0:
            self.velocity[1] = -self.velocity[1]
            self.velocity[0] = random.randrange(-2, 2)
        elif self.rect.left < 0 or self.rect.right > WIDTH:
            self.velocity[0] = -self.velocity[0]
        elif self.rect.bottom > HEIGHT:
            self.alive = False
        self.move()
```

Image 13: ***Setting your keybinds to vim or emacs.***

Make it your own

If you want to keep hacking on the PyGame project using your new keyboard prowess, you can continue from where we left off at https://repl.it/@GarethDwyer1/cwr-10-productivity[73].

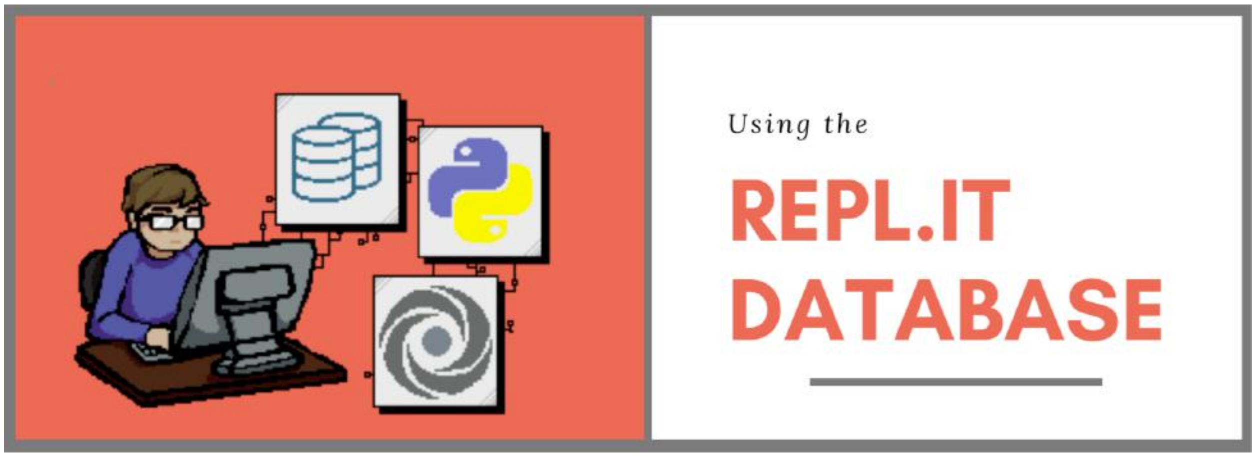
Where next?

Now that you have mastered the productivity features of Repl.it, you can build proof of concept applications in no time.

In the next tutorial, we'll show you how to store data directly in the Repl.it key-value store, one of the simplest verieties of database. This will cover the so-called "CRUD" (Create, Read, Update, Delete) operations that are fundamental to any database-backed software.

[73]https://repl.it/@GarethDwyer1/cwr-10-productivity

Using the Repl.it database



In previous tutorials we used the file system to store data persistently. This works fine for smaller projects, but there are some limitations to storing data directly in a file system. A more advanced way to store data which is used by nearly any production application is a database.

Another advantage of storing data in a database instead of in files is that it separates our code and data cleanly. If we build an application on Repl.it that processes any kind of data, it's likely that we'll want to share the code with other people but **not** the data. Having our data cleanly separated into a private database allows us to do exactly this.

In this tutorial, you'll see how to store data from a Repl.it project directly in the Repl.it key-value store, one of the simplest varieties of database, similar to a Python dictionary and more scalable.

As a demonstration project, we'll build a basic phone book application, storing contact information about friends and family and a command line application to allow users to:

- add new contacts
- search for existing contacts
- update existing contacts
- remove contacts.

This will cover the so-called "CRUD" (Create, Read, Update, Delete) operations that are fundamental to any database-backed software.

Now create a new Python repl called "phonebook".

Adding and reading data using the Repl.it database

In the `main.py` file import the database driver with this code:

```python
from replit import db
```

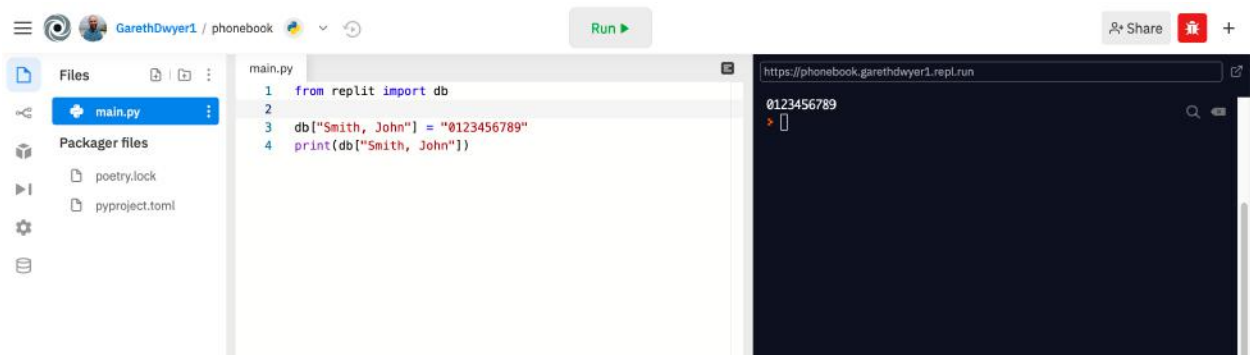
Databases usually store data on a separate physical server from where your code is running, so your code needs to know how to find the database and how to authenticate (to prove that you are authorised to access a specific database to stop other people reading your data).

Usually we would have to supply some kind of credentials for this (e.g. a username and password), as well as an endpoint to indicate where the database can be found. In this case, Repl.it handles everything automatically (as long as you are signed in), so you can start storing data straight away.

The `db` object works very similarly to a global Python dictionary but any data is persistently stored. You can associate a specific value with a given key in the same way. Add the following to your `main.py` file.

```python
db["Smith, John"] = "0123456789"
print(db["Smith, John"])
```

You should see the phone number printed to the console, as shown below.



```python
from replit import db

db["Smith, John"] = "0123456789"
print(db["Smith, John"])
```

```
0123456789
>
```

Image 2: ***Viewing a phone number from the database.***

How is this different from a dictionary?

The main difference between using the database and a Python dictionary is that, with the database, the data is:

- persisted between runs
- kept separate from the code.

For a concrete example, consider storing the same "John Smith" contact in both a dictionary and the database. Replace the code in your `main.py` file with the following and run it.

```python
from replit import db

# database
db["Smith, John"] = "0123456789"
print(db["Smith, John"])

# dictionary
d = {}
d["Smith, John"] = "0123456789"
print(d["Smith, John"])
```

Here we store the information first in the database and print it from the database and then in a dictionary and print it from there. In both cases, we see the result printed and the syntax is exactly the same.

However, if we comment out the lines where we create the association between key and value, and run the code again, we'll see a difference.

```python
from replit import db

# database
# db["Smith, John"] = "0123456789"
print(db["Smith, John"])

# dictionary
# d = {}
# d["Smith, John"] = "0123456789"
print(d["Smith, John"])
```

In this case, the first print still works as the data has persisted in the database. However the dictionary has been cleared between runs so we get the error `NameError: name 'd' is not defined`.

Because each Repl.it project has its own unique database which needs a secret key to access, you can add as much data to your database and still share your project without sharing any of your data.

The database also has some functionality that Python dictionaries do not, such as searching keys by prefix, which we will take a closer look at soon.

Building a basic phonebook application that can read and store data

Let's get started with the application. We'll build two separate components in parallel, piece by piece:

1. The database logic to create, read, update, and delete contacts.
2. The command line interface to prompt the user to choose what to do, get input, and show output.

We'll keep the code that interacts with users in our `main.py` file and the database logic in a new module called `contacts.py`

As we don't have any contacts yet, we'll start by allowing our users to add them.

Allowing the user to add contacts to the phonebook

Let's build the user interaction side first. We need to be able to accept input from the user and show them prompts and output. Add the following code to `main.py`:

```python
def prompt_add_contact():
    name = input("Please enter the contact's name: ")
    number = input("Please enter the contact's phone number: ")
    print(f"Adding {name} with {number}")

prompt_add_contact()
```

This doesn't actually store the contact anywhere yet, but you can test it out to see how it prompts the user for input and then displays a confirmation message.

Next we need to add some logic to store this in our database.

Create a new file called contacts.py and add the following code.

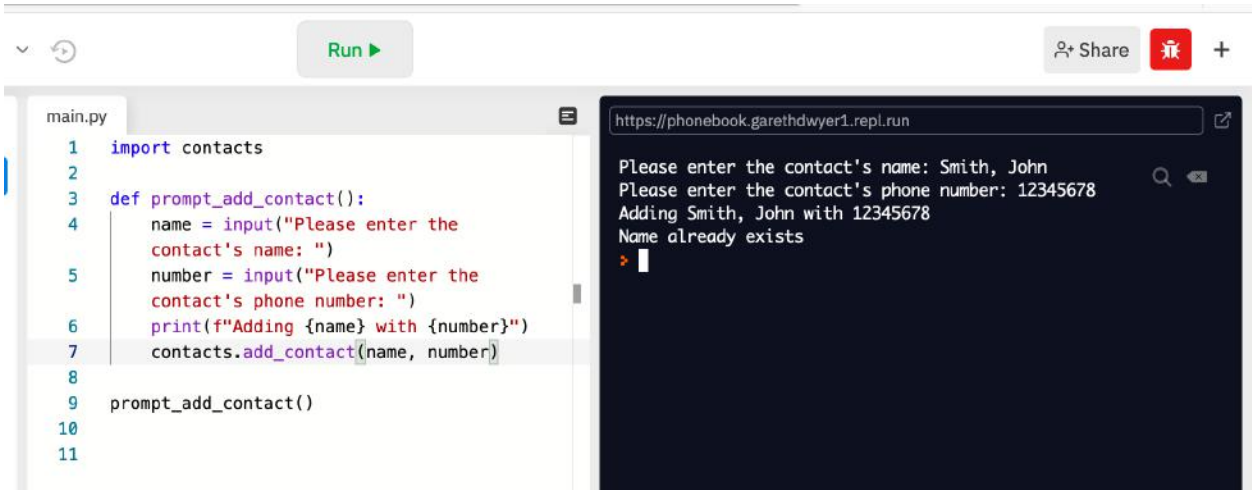
```python
from replit import db

def add_contact(name, phone_number):
    if name in db:
        print("Name already exists")
    else:
        db[name] = phone_number
```

Because we will use people's names as keys in our database and because it's possible that different people share the same name, it's possible that our users could overwrite important phone numbers by adding a new contact with the same name as an existing one. To prevent this, we'll ensure that they use a unique name for each contact and only add information with this method to **new** names.

Back in the `main.py` file add two lines to import our new module and call the add_contact function. The new code should look as follows:

```python
import contacts

def prompt_add_contact():
    name = input("Please enter the contact's name: ")
    number = input("Please enter the contact's phone number: ")
    print(f"Adding {name} with {number}")
    contacts.add_contact(name, number)

prompt_add_contact()
```

Test that this works - run it twice and enter the same name both times, with a different phone number. You should see the confirmation the first time, but the second time it will inform you that the contact already exists, as shown below.



```python
import contacts

def prompt_add_contact():
    name = input("Please enter the
    contact's name: ")
    number = input("Please enter the
    contact's phone number: ")
    print(f"Adding {name} with {number}")
    contacts.add_contact(name, number)

prompt_add_contact()
```

```
Please enter the contact's name: Smith, John
Please enter the contact's phone number: 12345678
Adding Smith, John with 12345678
Name already exists
>
```

Image 3: *Adding new contacts or showing an error.*

Allowing users to retrieve details of stored contacts

Now that we've added a contact to our database, let's allow users to retrieve this information. We want the user to be able to input a name and get the associated phone number in return. We can follow a similar pattern to before: adding a function to both our `main.py` file to handle user interaction and a separate one to our `contacts.py` file to handle database interaction.

In `main.py` add the following function and change the last line to call our new function instead of the `prompt_add_contact()` one, as follows:

```python
def prompt_get_contact():
    name = input("Please enter the name to find: ")
    number = contacts.get_contact(name)
    if number:
        print(f"{name}'s number is {number}")
    else:
        print(f"It looks like {name} does not exist")

prompt_get_contact()
```

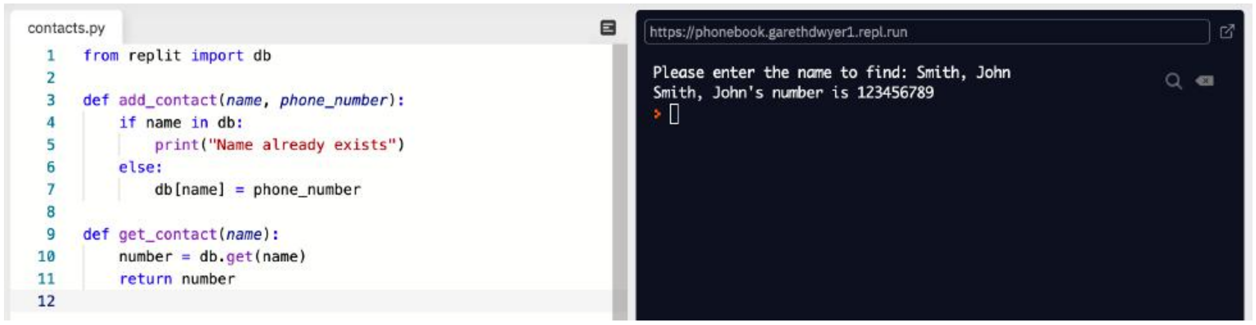
Note that this time we call the `get_contact` function before we write it - we have a blueprint that works now from our previous example so we can skip some back-and-forth steps.

Add the following function to `contacts.py`:

```python
def get_contact(name):
    number = db.get(name)
    return number
```

Our new code to go into `contacts.py` is very simple and it might be tempting to just put this logic directly in the `main.py` file as it's so short. However it's good to stay consistent as each of the files is likely to grow in length and complexity over time, and it will be easier to maintain our codebase if our user interaction code is strictly separate from our database interaction code.

Run the code again and input the same name as before. If all went well, you'll see the number, as in the example below.


contacts.py

```python
from replit import db

def add_contact(name, phone_number):
    if name in db:
        print("Name already exists")
    else:
        db[name] = phone_number

def get_contact(name):
    number = db.get(name)
    return number

```

https://phonebook.garethdwyer1.repl.run

```
Please enter the name to find: Smith, John
Smith, John's number is 123456789
> 
```

Image 4: ***Retrieving contacts from user input.***

Interlude: Creating a main menu

We now have functionality to add and retrieve contacts, and still need to add:

- searching for names with partial matches
- updating existing contacts (name or number)

- removing contacts.

But before we get started on those problems, we need to allow users to choose what kind of functionality they want to activate. With a GUI or web application, we could add some menu items or buttons, but our command line application is driven only by text input and output on a simple console. Let's build a main menu that allows users to specify what they want to do.

To make life easier for our users, we'll let them make choices by inputting a single number that's associated with the relevant menu item.

Change your `main.py` file to look as follows:

```python
import contacts
from os import system

main_message = """WELCOME TO PHONEBOOK
----------------------------------
Please choose:
1 - to add a new contact
2 - to find a contact
----------------------------------
"""

def prompt_add_contact():
    name = input("Please enter the contact's name: ")
    number = input("Please enter the contact's phone number: ")
    print(f"Adding {name} with {number}")
    contacts.add_contact(name, number)

def prompt_get_contact():
    name = input("Please enter the name to find: ")
    number = contacts.get_contact(name)
    if number:
        print(f"{name}'s number is {number}")
    else:
        print(f"It looks like {name} does not exist")

def main():
    print(main_message)
    choice = input("Please make your choice: ").strip()
    if choice == "1":
        prompt_add_contact()
    elif choice == "2":
        prompt_get_contact()
    else:
        print("Invalid input. Please try again.")

while True:
    system("clear")
    main()
    input("Press enter to continue: ")
```

This looks like a lot more code than we had before, but if you ignore the multi-line string at the top and the two functions that we already had, there's not much more. Our new `main()` function asks the users to choose an item from the menu, makes sure that it's a valid choice, and then calls the appropriate function.

Below our `main()` function, we have an infinite loop so that the user can keep using our application without re-running it after the first action. We call `system("clear")` between runs to clean up the old inputs and outputs (and we also added a new import at the top of the file for this).

Extending our search functionality

We already allow users to find contacts by entering their exact name, but it's useful to be able to do partial matches too. If our user inputs "Smith" and we have a "Smith, John" and a "Smith, Mary", we should be able to show the user both of these contacts.

The Repl.it database has a `prefix` function that can find all keys that start with a specific string. Giving "Smi" to this prefix function would match "Smith", "Smith, John" and "Smith, Mary", but **not** "John Smith", as it only matches from the **start** of each key.

You can use this by calling, for example, `db.prefix("Smi")` which will return all of the *keys* that match the "Smi" prefix. Note that this does not return the values (in our case, the phone numbers), so once we have our matches we still need to look up each phone number individually.

We want our application to prefer finding an exact match if one exists, or gracefully fall back to returning a list of matches by prefix only if there is no exact match.

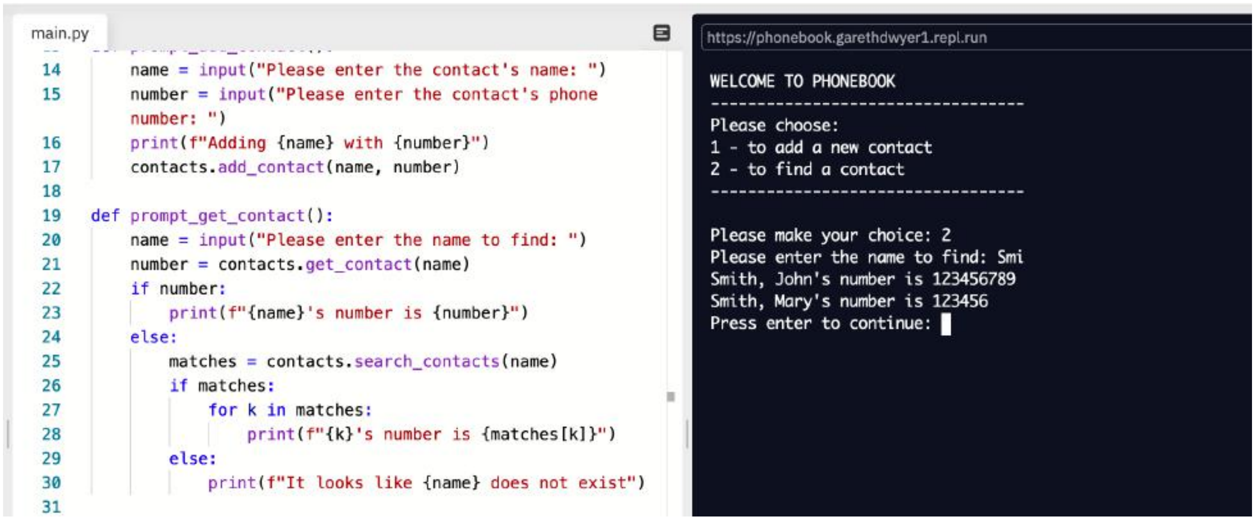
Add a new function to `contacts.py` that can search for contacts and extract each phone number as follows:

```
def search_contacts(search):
    match_keys = db.prefix(search)
    return {k: db[k] for k in match_keys}
```

And over in `main.py` modify the `prompt_get_contacts()` function to call this if necessary (when there is no exact match) as follows:

```python
def prompt_get_contact():
    name = input("Please enter the name to find: ")
    number = contacts.get_contact(name)
    if number:
        print(f"{name}'s number is {number}")
    else:
        matches = contacts.search_contacts(name)
        if matches:
            for k in matches:
                print(f"{k}'s number is {matches[k]}")
        else:
            print(f"It looks like {name} does not exist")
```

Run the code again and choose to add a contact. Enter "Smith, Mary" when prompted and any phone number. When the program starts over, choose to find a contact and input "Smi". It should print out both "Smith" matches that we have, as shown below.



```python
    name = input("Please enter the contact's name: ")
    number = input("Please enter the contact's phone number: ")
    print(f"Adding {name} with {number}")
    contacts.add_contact(name, number)

def prompt_get_contact():
    name = input("Please enter the name to find: ")
    number = contacts.get_contact(name)
    if number:
        print(f"{name}'s number is {number}")
    else:
        matches = contacts.search_contacts(name)
        if matches:
            for k in matches:
                print(f"{k}'s number is {matches[k]}")
        else:
            print(f"It looks like {name} does not exist")
```

```
WELCOME TO PHONEBOOK
---------------------------------
Please choose:
1 - to add a new contact
2 - to find a contact
---------------------------------

Please make your choice: 2
Please enter the name to find: Smi
Smith, John's number is 123456789
Smith, Mary's number is 123456
Press enter to continue:
```

Image 5: ***The user menu: They can now choose what action to do.***

Allowing users to update contacts

There are two ways that users might want to update contacts. They should be able to:

1. Change the name of a contact but keep the same phone number
2. Change the phone number of a contact but keep the same name

Because we are storing contacts as keys and values, to do 1) we need to create a new contact and remove the original one, while for 2) we can simply update the value of the existing key.

We can handle both cases with a single prompt by allowing the user to leave either field blank, in this case preserving the old value. Add the following function to your `main.py` file.

```python
def prompt_update_contact():
    old_name = input("Please enter the name of the contact to update: ")
    old_number = contacts.get_contact(old_name)
    if old_number:
        new_name = input(f"Please enter the new name for this contact (leave blank t\
o keep {old_name}): ").strip()
        new_number = input(f"Please enter the new number for this contact (leave bla\
nk to keep {old_number}): ").strip()

        if not new_number:
            new_number = old_number

        if not new_name:
            contacts.update_number(old_name, new_number)
        else:
            contacts.update_contact(old_name, new_name, new_number)

    else:
        print(f"It looks like {old_name} does not exist")
```

This uses two functions in our `contacts.py` file that don't exist yet. These are:

- `update_number` to keep the contact but change the phone number
- `update_contact` to update the name (and maybe also the number) by removing the old contact and creating a new one.

Create these two functions in `contacts.py` as follows.

```python
def update_number(old_name, new_number):
    db[old_name] = new_number

def update_contact(old_name, new_name, new_number):
    db[new_name] = new_number
    del db[old_name]
```

Note how we can use the `del` Python keyword to remove things from our database. We'll use this again in the next section.

Now we need to allow users to choose "update" as an option from the menu. In the `main.py` file, add a new line to the menu prompt to inform our users about the option and update the `main()` function to call the new update function when appropriate, as follows:

```python
main_message = """WELCOME TO PHONEBOOK
----------------------------------
Please choose:
1 - to add a new contact
2 - to find a contact
3 - to update a contact
----------------------------------
"""
# ...
```

```python
def main():
    print(main_message)
    choice = input("Please make your choice: ").strip()
    if choice == "1":
        prompt_add_contact()
    elif choice == "2":
        prompt_get_contact()
    elif choice == "3":
        prompt_update_contact()
    else:
        print("Invalid input. Please try again.")
```

Test it out! Change someone's name, someone else's number, and then update both the name and the number at once.

Allowing users to remove contacts

Sometimes there are people we just don't want to talk to any more. We've already seen how to remove contacts by updating their key and removing the old one, but let's allow for removals without updates too. By now, you should be familiar with the parts of the code that you need to update. To recap, these are:

- adding a new `prompt_*` function to the `main.py` file
- adding a new `*_contact` function to `contacts.py`
- adding a new line to the menu prompt in `main.py`
- adding a new `elif` block to the `main()` function in `main.py`.

These are each shown in turn below.

```python
def prompt_delete_contact():
    name = input("Please enter the name to delete: ")
    contact = contacts.get_contact(name)
    if contact:
        print(f"Deleting {name}")
        contacts.delete_contact(name)
    else:
        print(f"It looks like {name} does not exist")
```

```python
def delete_contact(name):
    del db[name]
```

```python
main_message = """WELCOME TO PHONEBOOK
----------------------------------
Please choose:
1 - to add a new contact
2 - to find a contact
3 - to update a contact
4 - to delete a contact
----------------------------------
"""
```

```python
def main():
    print(main_message)
    choice = input("Please make your choice: ").strip()
    if choice == "1":
        prompt_add_contact()
    elif choice == "2":
        prompt_get_contact()
    elif choice == "3":
        prompt_update_contact()
    elif choice == "4":
        prompt_delete_contact()
    else:
        print("Invalid input. Please try again.")
```

It may be a bit inconvenient to type out the whole name of a contact that you want to delete, but it's usually acceptable to make "dangerous" operations less user friendly. As there is no way to recover contacts, it's good to make it a bit more difficult to delete them. Maybe our user will change their mind while typing out the name of an old friend to delete the record and reach out instead :).

Make it your own

If you've followed along, you'll have your own version of the repl to extend. Otherwise start from ours at https://repl.it/@GarethDwyer1/cwr-11-phonebook[74].

Where next

You've learned how basic databases work. Databases are a complicated topic on their own and it can take years or decades to master the more advanced aspects of them, but they can also do more than the simple operations that we've covered here. Spend some time reading about PostgreSQL[75] and relational databases[76] in general, or other key-value stores[77] like the Repl.it database.

Even without further research, the basic Create, Read, Update, and Delete (CRUD) operations that we covered here will get you far and you can build nearly any app you can imagine with just these.

Next we'll take a look at playing audio files programmatically so you can use Python to control your music.

[74]https://repl.it/@GarethDwyer1/cwr-11-phonebook
[75]https://www.postgresql.org/
[76]https://en.wikipedia.org/wiki/Relational_database
[77]https://en.wikipedia.org/wiki/Key%E2%80%93value_database

Repl.it Audio

Most people control their music players manually, pressing the pause button to pause a track or hitting a volume up control to raise the volume. With Repl.it, you can automate your media experience using code.

In this tutorial, we'll build a media player that can play audio files programmatically, allowing the user to pause playback, change the track, change the volume, or get looping information by giving text commands.

We'll also outline how this could be integrated into other applications, such as a chatbot, but we'll leave the implementation of that as an exercise for the reader.

Understanding how audio works on Repl.it

In Unix systems, including the ones that Repl.it is built on, everything is a file[78]. You might think of file types like PDFs, text files, image files or audio files, but in fact even things like printers are often "seen" as files by the underlying operating system.

Repl.it uses a special file at `/tmp/audio` to control media output. There are more details on how to manipulate this file directly in the audio docs[79], but Repl.it also provides a higher level Python library that gives us some higher level functions like "play_audio". We'll be using the library in this tutorial.

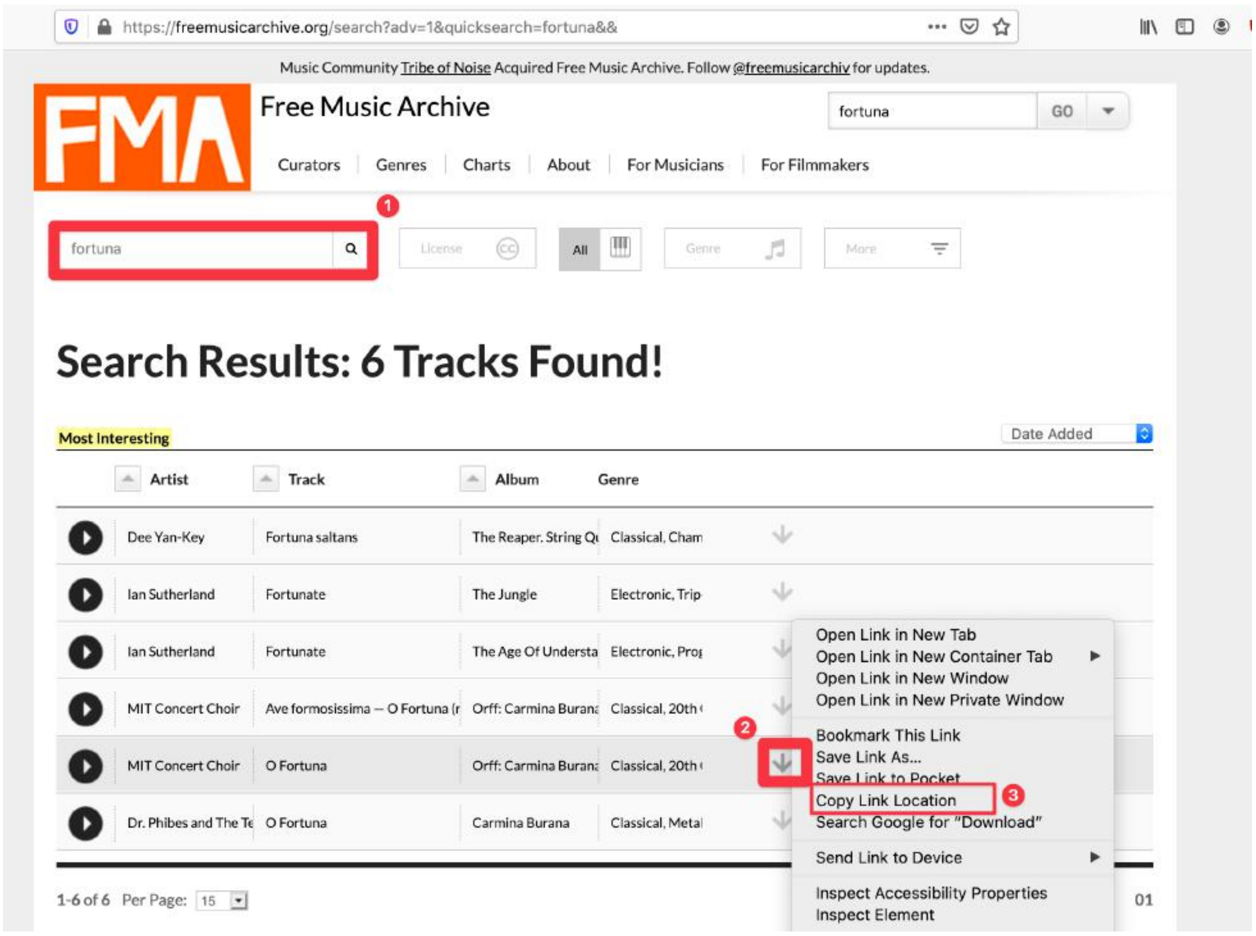
[78]https://en.wikipedia.org/wiki/Everything_is_a_file
[79]https://docs.repl.it/repls/audio

Getting a free audio file from the Free Music Archive

You can use your own mp3 files if you prefer, but as most music is under copy protection, we'll use a file from the Free Music Arhive[80] for demo purposes.

Let's grab the URL of a file we want so that we can use code to download it to our Repl.it project.

Search for a song that you like, right-click on the download link and press "copy link location", as shown below.



Search Results: 6 Tracks Found!

Artist	Track	Album	Genre
Dee Yan-Key	Fortuna saltans	The Reaper. String Qu	Classical, Cham
Ian Sutherland	Fortunate	The Jungle	Electronic, Trip
Ian Sutherland	Fortunate	The Age Of Understa	Electronic, Prog
MIT Concert Choir	Ave formosissima – O Fortuna (r	Orff: Carmina Burana	Classical, 20th (
MIT Concert Choir	O Fortuna	Orff: Carmina Burana	Classical, 20th (
Dr. Phibes and The Te	O Fortuna	Carmina Burana	Classical, Metal

Image 2: ***Downloading an audio track***

Downloading audio files to our project

Our first goal is to download the song and play it.

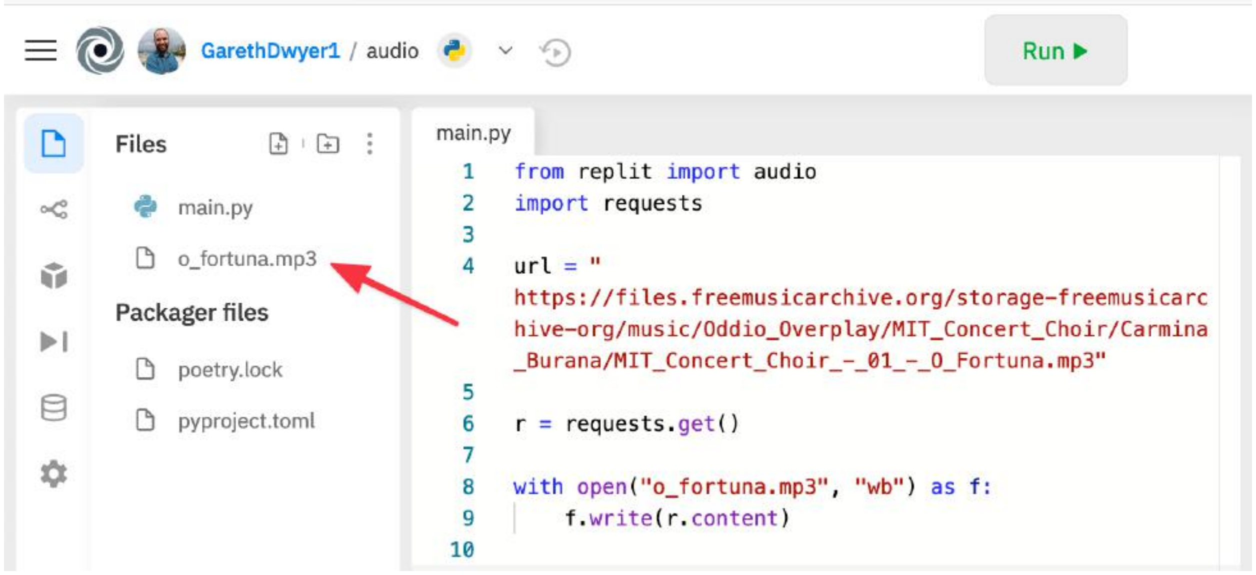
Create a new Python repl called `audio` and add the following code to the `main.py` file.

[80]https://freemusicarchive.org/search

```python
import requests

url = " https://files.freemusicarchive.org/storage-freemusicarchive-org/music/Oddio_\
Overplay/MIT_Concert_Choir/Carmina_Burana/MIT_Concert_Choir_-_01_-_O_Fortuna.mp3"

r = requests.get(url)
with open("o_fortuna.mp3", "wb") as f:
    f.write(r.content)
```

Change the URL to the one you chose and `o_fortuna.mp3` to something more appropriate if you chose a different song.

This downloads the song, opens up a binary file, and writes the contents of the download to the file. You should see the new file pop up in the files tab on the left after you run this code.



```python
from replit import audio
import requests

url = "
https://files.freemusicarchive.org/storage-freemusicarchive-org/music/Oddio_Overplay/MIT_Concert_Choir/Carmina_Burana/MIT_Concert_Choir_-_01_-_O_Fortuna.mp3"

r = requests.get()

with open("o_fortuna.mp3", "wb") as f:
    f.write(r.content)
```

Image 3: ***Viewing the downloaded audio file in your files tab.***

Instead of downloading the audio file using `requests` as shown above, you can also press the `add file` button in your repl and upload an audio file from your local machine.

Playing the audio file using Python

Now that we have the file we can play it by importing the `audio` module and calling the `play_file` method. Replace the code in `main.py` with the following:

```python
from replit import audio
import time

audio.play_file("o_fortuna.mp3")
time.sleep(10)
```

Note that your repl usually dies the moment there is no more code to execute, and playing audio doesn't keep it alive. For now, we are sleeping for 10 seconds which keeps the repl alive and the audio playing. If you run this, you should hear the first 10 seconds of the track before it cuts out.

It's not ideal to keep the execution loop locked up in a `sleep()` call as we can't interact with our program so we can't control the playback in any way.

To keep the music playing until the user presses a key, change the last line to:

```python
choice = input("Press enter to stop the music. ")
```

Now the program is blocked waiting for user input and the music will keep playing until the user enters something.

Let's add some more useful controls.

Allowing the user to pause, change volume, or get information about the currently playing track

The controls we add next are based around:

- `source.volume`: an attribute that we can add to or subtract from to increase or decrease the volume
- `source.paused`: an attribute we can change to True or False to pause or unpause the track
- `source.set_loop()`: a method we can call to specify how many times a track should loop before ending

We can also display useful information about the current status of our media player by looking at:

- `source.loops_remaining`: an attribute to see how many more time a track will loop
- `source.get_remaining()`: a method to see the remaining playtime for the current track.

We'll allow the user to see the current information but for simplicity we'll only update this on each input, so our display will often display 'out of date' information.

Creating the prompt menu

Remove the code in `main.py` and replace it with the following.

```python
import time
from os import system
from replit import audio

main_message = """
+: volume up
-: volume down
k: add loop
j: remove loop
<space>: play/pause
"""
```

Here we add one more import for `system` which we'll use to clear the screen so that the user doesn't see old information. We then define a string that will prompt the user with their options.

Creating the `show_status()` method

Let's add a method that will show the user the current status of our media player. It will take `source` as an input, which is what the `play_media()` method that we already used returns.

```python
def show_status(source):
    time.sleep(0.2)
    system("clear")
    vbar = '|' * int(source.volume * 20)
    vperc = int(source.volume * 100)
    pp = "▯" if source.paused else "▯"

    print(f"Volume: {vbar}  {vperc}% \n")
    print(f"Looping {source.loops_remaining} time(s)")
    print(f"Time remaining: {source.get_remaining()}")
    print(f"Playing: {pp}")
    print(main_message)
```

Note that we add a `time.sleep()` at the top of this function. Because changing the status involves writing to the `/tmp/audio` file we discussed before and reading the status involves reading from this file, we want to wait a short while to ensure we don't read stale information before showing it to the user.

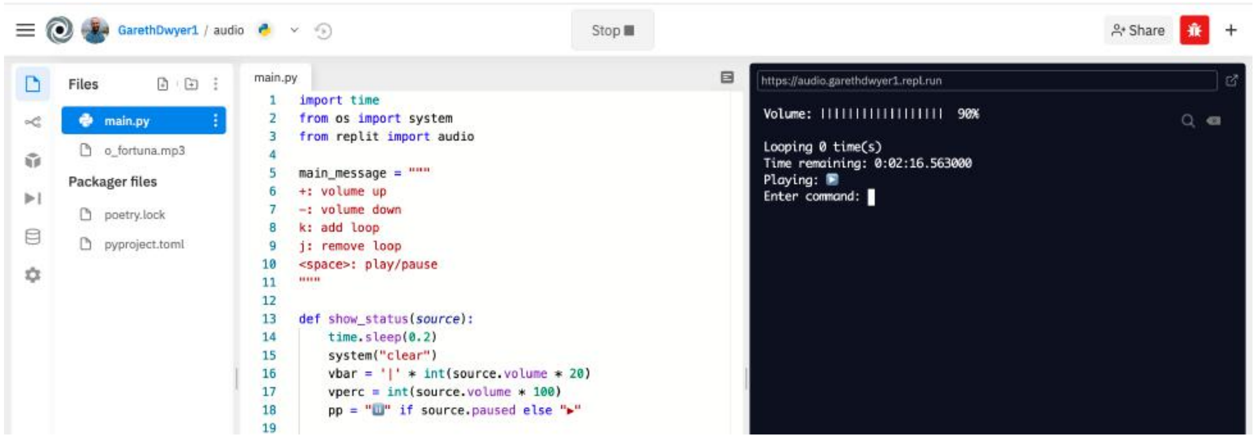
Otherwise our function clears the screen, prints out a text-based volume bar along with the current volume percentage, and shows other information such as whether the track is currently playing or paused, how many loops are left, and how much time is left before the track finishes.

Finally, we need a loop to constantly prompt the user for the next command which will also keep our repl alive and continue playing the track while we are waiting for user input. Add the following `main()` function to `main.py` and call it:

```python
def main():
    source = audio.play_file("o_fortuna.mp3")
    time.sleep(1)
    show_status(source)

    while True:
        choice = input("Enter command: ")
        if choice == '+':
            source.volume += 0.1
        elif choice == '-':
            source.volume -= 0.1
        elif choice == "k":
            source.set_loop(source.loops_remaining + 1)
        elif choice == "j":
            source.set_loop(source.loops_remaining - 1)
        elif choice == " ":
            source.paused = not source.paused
        show_status(source)

main()
```

Once again, you should replace the “o_fortuna” string if you downloaded or uploaded a different audio file.

If you run the repl now you should hear you track play and you can control it by inputting the various commands.



```python
import time
from os import system
from replit import audio

main_message = """
+: volume up
-: volume down
k: add loop
j: remove loop
<space>: play/pause
"""

def show_status(source):
    time.sleep(0.2)
    system("clear")
    vbar = '|' * int(source.volume * 20)
    vperc = int(source.volume * 100)
    pp = "⏸" if source.paused else "▶"
```

```
Volume: ||||||||||||||||||  90%

Looping 0 time(s)
Time remaining: 0:02:16.563000
Playing: ▶
Enter command: █
```

Image 4: *A preview of our audio status dashboard.*

Playing individual tones

Instead of playing audio from files, you can also play specific tones or notes with the `play_tone()` method. This method takes three arguments:

- duration: how long the tone should play for
- pitch: the frequency of the tone (how high or low it sounds)
- wave form: the fundamental wave form[81] that the tone is built on.

If you've ever played a musical instrument, you'll probably have come across notes referred to by the letters A-G. With digital audio, you'll specify the pitch in hertz (Hz). "Middle C" on a piano is usually 262 Hz and the A above this is 440 Hz.

Let's write a program to play "Twinkle Twinkle Little Star". Create a new Python repl and add the following code to `main.py`.

```python
import time
from replit import audio

def play_note(note, duration):
    note_to_freq = {
        "C": 262, "D": 294, "E": 330, "F": 349, "G": 392, "A": 440
    }
    audio.play_tone(duration, note_to_freq[note], 0)
    time.sleep(duration)

play_note("C", 2)
```

Above we set up a convenience function to play specific notes for a specific duration. It includes a dictionary mapping the names of notes to their frequencies. We've only done one octave and no sharps or flats, but you can easily extend this to add the other notes.

It then plays the tone of the note passed in for the specified duration. We sleep for that duration too, as othewise the next note will be played before the previous note is finished. We also pass a `0` to `play_tone` which specifies the default sine waveform. You can change it to `1`, `2`, or `3` for triangle, saw, or square, which you can read about in more detail[82].

Test that you can play a single note as expected. Now you can play the first part of "Twinkle Twinkle Little Star" by defining all of the notes and durations, and then looping through them, calling `play_note` on each in turn.

[81]https://www.perfectcircuit.com/signal/difference-between-waveforms
[82]https://www.perfectcircuit.com/signal/difference-between-waveforms

```python
notes = ["C", "C", "G", "G", "A", "A", "G", "F", "F", "E", "E", "D", "D", "C"]
durations = [2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4]

for i in range(len(notes)):
    play_note(notes[i], durations[i])
```

We can also control the volume of each tone by passing a `volume` argument to `play_tone()`. As for audio files, this is a float where `1` represents 100% volume. If we wanted to implement a *decrescendo* (gradual decrease in volume), we could modify our code to look as follows:

```python
def play_note(note, duration, volume=1):
    note_to_freq = {
        "C": 262, "D": 294, "E": 330, "F": 349, "G": 392, "A": 440
    }
    audio.play_tone(duration, note_to_freq[note], 0, volume=volume)
    time.sleep(duration)


notes = ["C", "C", "G", "G", "A", "A", "G", "F", "F", "E", "E", "D", "D", "C"]
durations = [2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4]

volume = 1
for i in range(len(notes)):
    volume -= 0.05
    play_note(notes[i], durations[i], volume=volume)
```

Here we added a `volume` argument to our `play_note()` function so that we can pass it along to `play_tone()`. Each time around the loop we reduce the volume by 5%. Play it again and you should hear the song slowly fade out (if you add more than 20 notes, the volume will hit 0 so you'll have to reduce the step or increase the volume at some point to stop the song going silent).

Make it your own

If you followed along you'll have your own version to extend, otherwise you can fork our media player repl at https://repl.it/@GarethDwyer1/cwr-12-audio-player.[83]

The "Twinkle Twinkle Little Star" repl can be found at https://repl.it/@GarethDwyer1/cwr-12-audio-twinkle-twinkle[84].

[83]https://repl.it/@GarethDwyer1/cwr-12-audio-player.
[84]https://repl.it/@GarethDwyer1/cwr-12-audio-twinkle-twinkle

Where next

Controlling your audio files through a text-based interface might feel like a downgrade from using a GUI media player, but you can use these concepts to integrate audio controls into your other applications. For example, you could create a Discord chatbot[85] that plays different tracks and automatically pauses or reduces the volume of your music when you join a Discord voice channel. Or you could integrate audio tracks into a web application or game (e.g. playing a victory or defeat sound at a specific volume given certain conditions).

Once you can control something using code, the possibilities are pretty broad, so use your imagination!

You've reached the end of this collection of tutorials that teach you the ins and outs of Repl.it, and you should be able to build any project that you can imagine now.

If you're stuck for ideas, continue on to Part 3 where we'll walk you through eight practical projects, focusing more on coding concepts than Repl.it features.

[85] https://ritza.co/showcase/repl.it/building-a-discord-bot-with-python-and-repl-it.html

Beginner web scraping with Python and Repl.it

In this guide, we'll walk through how to grab data from web sites automatically. Most websites are created with a *human* audience in mind - you use a search engine or type a URL into your web browser, and see information displayed on the page. Sometimes, we might want to *automatically* extract and process this data, and this is where web scraping can save us from boring repetitive labour. We can create a custom computer program to visit web sites, extract specific data and process this data in a particular way.

We'll be extracting news data from the bbc.com[86] news website, but you should be able to adapt it to extract information from any website that you want with a bit of trial and error.

There are many reasons you might wish to use web scraping. For example, you might need to:

- extract numbers from a report that is released weekly and published online
- grab the schedule for your favourite sports team as it's released
- find the release dates for upcoming movies in your favourite genre
- be notified automatically when a website changes

There are many other use cases for web scraping. However, you should also note that copyright law and web scraping laws are complex and differ by country. As long as you aren't blatantly copying their content or doing web scraping for commercial gain, people generally don't mind web scaping. However, there have been some legal cases involving scraping data from LinkedIn[87] and media attention from scraping data from OKCupid[88]. Web scraping can violate the law, go against a particular website's terms of service, or breach ethical guidelines - so take care with where you apply this skill.

With the disclaimer out of the way, let's learn how to scrape!

Overview and requirements

Specifically, in this tutorial we'll cover:

- What a website really is and how HTML works

[86]https://bbc.com/news
[87]https://techcrunch.com/2016/08/15/linkedin-sues-scrapers/
[88]https://www.engadget.com/2016/05/13/scientists-release-personal-data-for-70-000-okcupid-profiles/

- Viewing HTML in your web browser
- Using Python to download web pages
- Using BeautifulSoup[89] to extract parts of scraped data

We'll be using the online programming environment Repl.it[90] so you won't need to install any software locally to follow along step by step. If you want to adapt this guide to your own needs, you should create a free account by going to repl.it[91] and follow their sign up process.

It would help if you have basic familiarity with Python or another high-level programming language, but we'll be explaining each line of code we write in detail so you should be able to keep up, or at least replicate the result, even if you don't.

Webpages: beauty and the beast

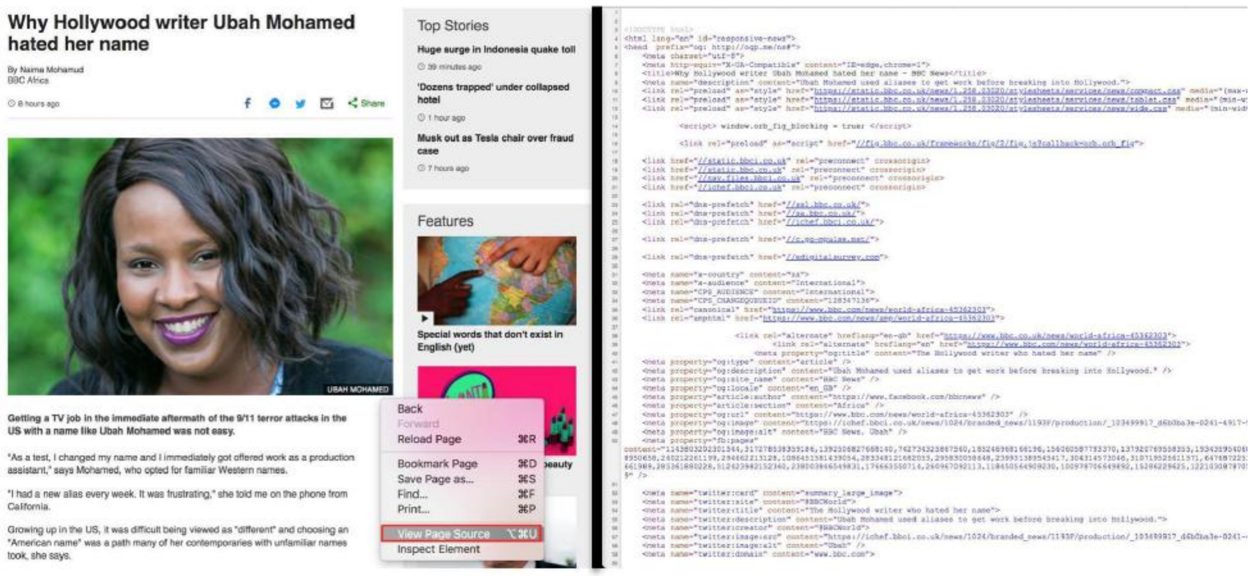
You have no doubt visited web pages using a web browser before. Websites exist in two forms:

1. The one you are used to, where you can see text, images, and other media. Different fonts, sizes, and colours are used to display information in a useful and (usually) aesthetic way.
2. The "source" of the webpage. This is the computer code that tells your web browser (e.g. Mozilla Firefox or Google Chrome) what to display and how to display it.

Websites are created through a combination of three computer languages: HTML, CSS and JavaScript. This in itself is a huge and complicated field with a messy history, but having a basic understanding of how some of it works is necessary to automate web scraping effectively. If you open any website in your browser and right-click somewhere on the page, you'll see a menu which should include an option to "view page source" – to inspect the code form of a website, before your web browser interprets it.

This is shown in the image below: a normal web page on the left, with an open menu (displayed by right-clicking on the page). Clicking "view page source" on this menu produces the result on the right – we can see the code that contains all the data and supporting information that the web browser needs to display the complete page. While the page on the left is easy to read, use, and looks good, the one on the right is a monstrosity. It takes some effort and experience to make any sense of it, but it's possible and necessary if we want to write custom web scrapers.

[89]https://www.crummy.com/software/BeautifulSoup/
[90]https://repl.it
[91]https://repl.it



Why Hollywood writer Ubah Mohamed hated her name

By Naima Mohamud
BBC Africa

8 hours ago

Getting a TV job in the immediate aftermath of the 9/11 terror attacks in the US with a name like Ubah Mohamed was not easy.

"As a test, I changed my name and I immediately got offered work as a production assistant," says Mohamed, who opted for familiar Western names.

"I had a new alias every week. It was frustrating," she told me on the phone from California.

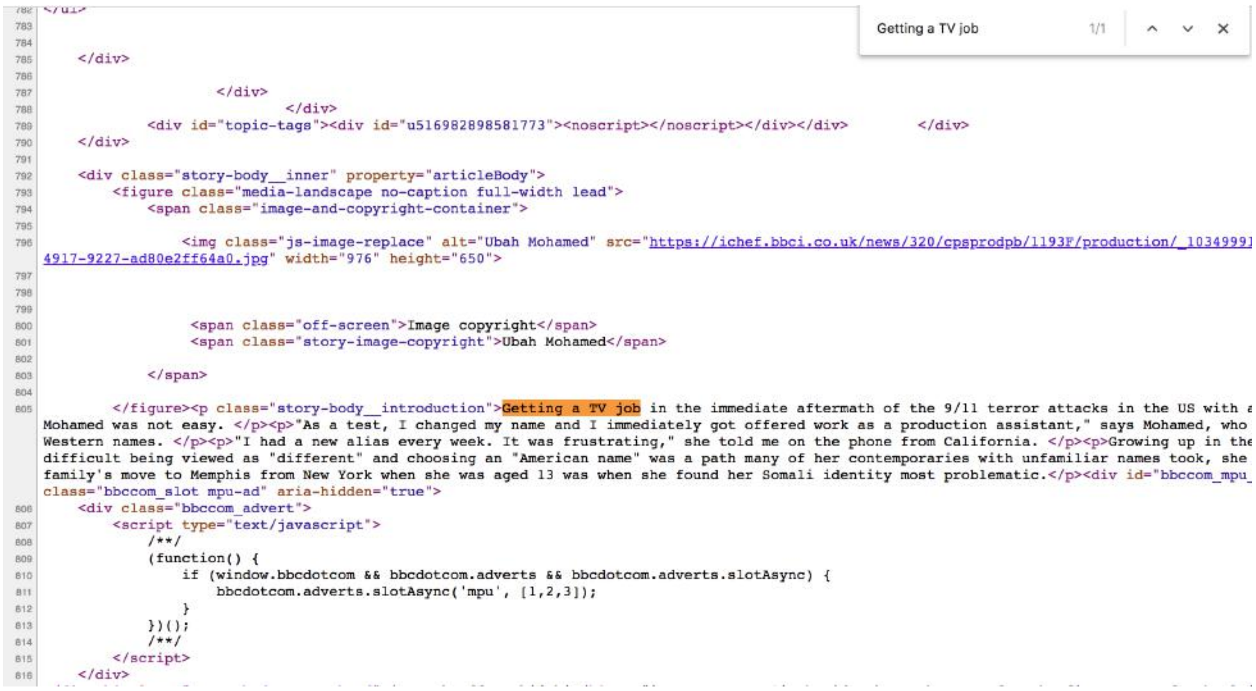
Growing up in the US, it was difficult being viewed as "different" and choosing an "American name" was a path many of her contemporaries with unfamiliar names took, she says.

```
<!DOCTYPE html>
<html lang="en" id="responsive-news">
<head  prefix="og: http://ogp.me/ns#">
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Why Hollywood writer Ubah Mohamed hated her name - BBC News</title>
    <meta name="description" content="Ubah Mohamed used aliases to get work before breaking into Hollywood.">
    <link rel="preload" as="style" href="https://static.bbc.co.uk/news/1.258.03020/stylesheets/services/news/compact.css" media="(max-
    <link rel="preload" as="style" href="https://static.bbc.co.uk/news/1.258.03020/stylesheets/services/news/tablet.css" media="(min-w
    <link rel="preload" as="style" href="https://static.bbc.co.uk/news/1.258.03020/stylesheets/services/news/wide.css" media="(min-wid

            <script> window.orb_fig_blocking = true; </script>

            <link rel="preload" as="script" href="//fig.bbc.co.uk/frameworks/fig/2/fig.js?callback=orb.orb_fig">

    <link href="//static.bbci.co.uk" rel="preconnect" crossorigin>
    <link href="//static.bbc.co.uk" rel="preconnect" crossorigin>
    <link href="//nav.files.bbci.co.uk" rel="preconnect" crossorigin>
    <link href="//ichef.bbci.co.uk" rel="preconnect" crossorigin>

    <link rel="dns-prefetch" href="//ssl.bbc.co.uk/">
    <link rel="dns-prefetch" href="//sa.bbc.co.uk/">
    <link rel="dns-prefetch" href="//ichef.bbci.co.uk/">

    <link rel="dns-prefetch" href="//c.go-mpulse.net/">

    <link rel="dns-prefetch" href="//adigitalsurvey.com">

    <meta name="x-country" content="za">
    <meta name="x-audience" content="International">
    <meta name="CPS_AUDIENCE" content="International">
    <meta name="CPS_CHANGEQUEUEID" content="128347136">
    <link rel="canonical" href="https://www.bbc.com/news/world-africa-45362303">
    <link rel="amphtml" href="https://www.bbc.com/news/amp/world-africa-45362303">

                    <link rel="alternate" hreflang="en-gb" href="https://www.bbc.co.uk/news/world-africa-45362303">
                        <link rel="alternate" hreflang="en" href="https://www.bbc.com/news/world-africa-45362303">
                  <meta property="og:title" content="The Hollywood writer who hated her name" />
    <meta property="og:type" content="article" />
    <meta property="og:description" content="Ubah Mohamed used aliases to get work before breaking into Hollywood." />
    <meta property="og:site_name" content="BBC News" />
    <meta property="og:locale" content="en_GB" />
    <meta property="article:author" content="https://www.facebook.com/bbcnews" />
    <meta property="article:section" content="Africa" />
    <meta property="og:url" content="https://www.bbc.com/news/world-africa-45362303" />
    <meta property="og:image" content="https://ichef.bbci.co.uk/news/1024/branded_news/1193F/production/_103499817_d6b0ba3e-0241-4917-
    <meta property="og:image:alt" content="BBC News, Ubah" />
    <meta property="fb:pages"
content="[illegible]
[illegible]
[illegible]
5" />

    <meta name="twitter:card" content="summary_large_image">
    <meta name="twitter:site" content="@BBCWorld">
    <meta name="twitter:title" content="The Hollywood writer who hated her name">
    <meta name="twitter:description" content="Ubah Mohamed used aliases to get work before breaking into Hollywood.">
    <meta name="twitter:creator" content="@BBCWorld">
    <meta name="twitter:image:src" content="https://ichef.bbci.co.uk/news/1024/branded_news/1193F/production/_103499817_d6b0ba3e-0241-
    <meta name="twitter:image:alt" content="Ubah" />
    <meta name="twitter:domain" content="www.bbc.com">
```

Image 1: ***Normal and source view of the same BBC news article.***

Navigating the source code using Find

The first thing to do is to work out how the two pages correspond: which parts of the normally displayed website match up to which parts of the code. You can use "find" Ctrl + F) in the source code view to find specific pieces of text that are visible in the normal view to help with this. In the web page on the left, we can see that the story starts with the phrase "Getting a TV job". If we search for this phrase in the code view, we can find the corresponding text within the code, on line 805.



Getting a TV job 1/1

```
</ul>


    </div>

                    </div>
                            </div>
            <div id="topic-tags"><div id="u516982898581773"><noscript></noscript></div></div>        </div>
    </div>

    <div class="story-body__inner" property="articleBody">
        <figure class="media-landscape no-caption full-width lead">
            <span class="image-and-copyright-container">

                <img class="js-image-replace" alt="Ubah Mohamed" src="https://ichef.bbci.co.uk/news/320/cpsprodpb/1193F/production/_1034999
4917-9227-ad80e2ff64a0.jpg" width="976" height="650">



                 <span class="off-screen">Image copyright</span>
                 <span class="story-image-copyright">Ubah Mohamed</span>

            </span>

        </figure><p class="story-body__introduction">Getting a TV job in the immediate aftermath of the 9/11 terror attacks in the US with a
Mohamed was not easy. </p><p>"As a test, I changed my name and I immediately got offered work as a production assistant," says Mohamed, who
Western names. </p><p>"I had a new alias every week. It was frustrating," she told me on the phone from California. </p><p>Growing up in the
difficult being viewed as "different" and choosing an "American name" was a path many of her contemporaries with unfamiliar names took, she
family's move to Memphis from New York when she was aged 13 was when she found her Somali identity most problematic.</p><div id="bbccom_mpu_
class="bbccom_slot mpu-ad" aria-hidden="true">
    <div class="bbccom_advert">
        <script type="text/javascript">
            /**/
            (function() {
                if (window.bbcdotcom && bbcdotcom.adverts && bbcdotcom.adverts.slotAsync) {
                    bbcdotcom.adverts.slotAsync('mpu', [1,2,3]);
                }
            })();
            /**/
        </script>
    </div>
```

Image 2: ***Finding text in the source code of a web page***.

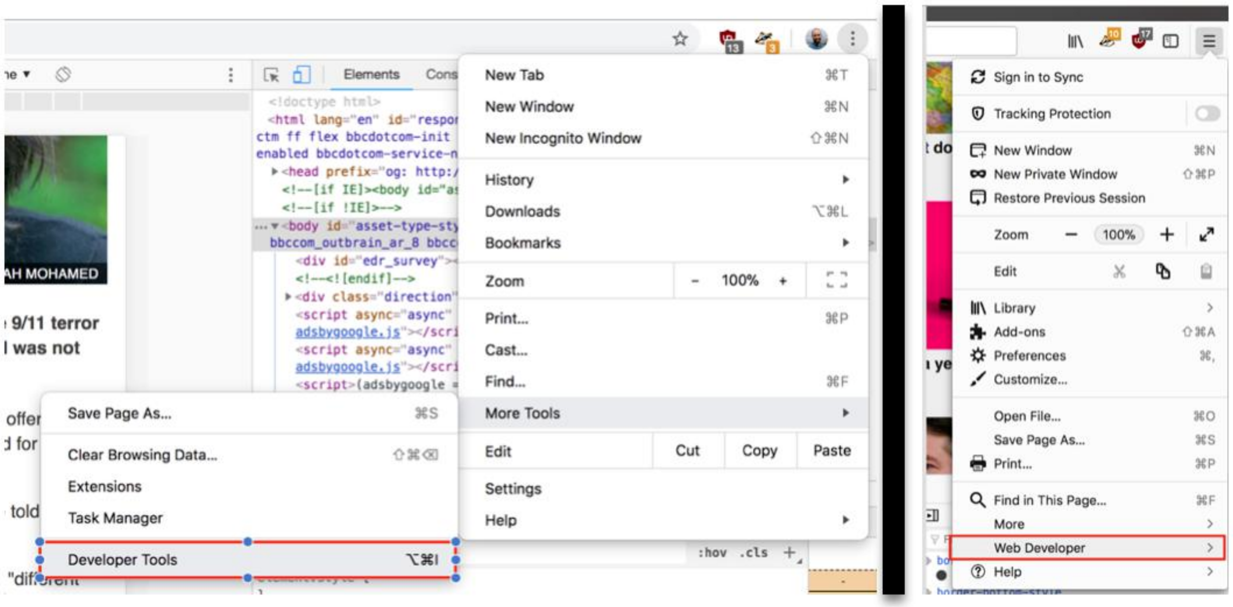
The `<p class="story-body__introduction">` just before the highlighted section is HTML code to specify that a paragraph (`<p>` in HTML) starts here and that this is a special kind of paragraph (an introduction to the story). The paragraph continues until the `</p>` symbol. You don't need to worry about understanding HTML completely, but you should be aware that it contains **both** the text data that makes up the news article and additional data about how to display it.

A large part of web scraping is viewing pages like this to a) identify the data that we are interested in and b) to separate this from the markup and other code that it is mixed with. Even before we start writing our own code, it can still be tricky first to understand other people's.

In most pages, there is a lot of code to define the structure, layout, interactivity, and other functionality of a web page, and relatively little that contains the actual text and images that we usually view. For especially complex pages it can be quite difficult, even with the help of the find function, to locate the code that is responsible for a particular part of the page. For this reason, most web browsers come with so-called "developer tools", which are aimed primarily at programmers to assist in the creation and maintenance of web sites, though these tools are also handy for doing web scraping.

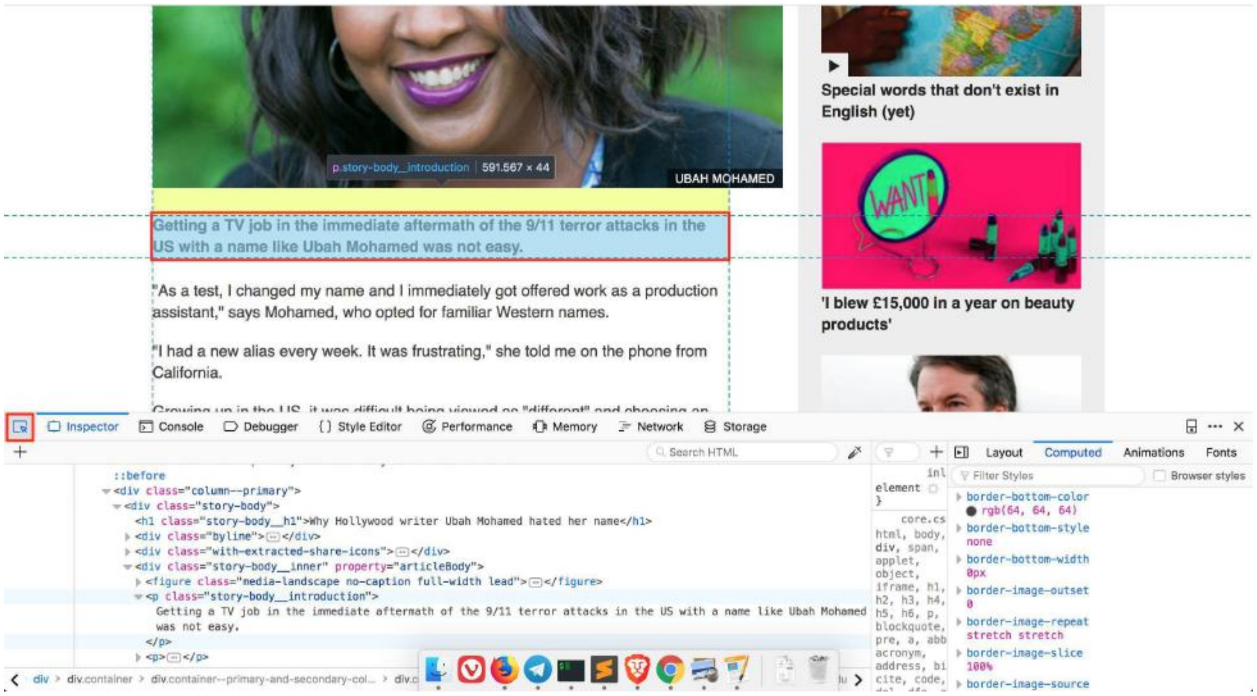
Navigating the source code using developer tools

You can open the developer tools for your browser from the main menu, with Google Chrome shown on the left and Mozilla Firefox on the right below. If you're using a different web browser, you should be able to find a similar setting.



** Image 3:** *Opening Developer Tools in Chrome (left) and Firefox (right)*

Activating the tool brings up a new panel in your web browser, usually at the bottom or on the right-hand side. The tool contains an “Inspector” panel and a selector tool, which can be chosen by pressing the icon highlighted in red below. Once the selector tool is active, you can click on parts of the web page to view the corresponding source code. In the image below, we selected the same first paragraph in the normal view and we can see the `<p class=story-body__introduction">` code again in the panel below.


p.story-body__introduction | 591.567 × 44
UBAH MOHAMED
Special words that don't exist in English (yet)
'I blew £15,000 in a year on beauty products'
Inspector
Console
Debugger
Style Editor
Performance
Memory
Network
Storage
Search HTML
Layout
Computed
Animations
Fonts

Getting a TV job in the immediate aftermath of the 9/11 terror attacks in the US with a name like Ubah Mohamed was not easy.

"As a test, I changed my name and I immediately got offered work as a production assistant," says Mohamed, who opted for familiar Western names.

"I had a new alias every week. It was frustrating," she told me on the phone from California.

```
::before
<div class="column--primary">
 <div class="story-body">
  <h1 class="story-body__h1">Why Hollywood writer Ubah Mohamed hated her name</h1>
  <div class="byline">⋯</div>
  <div class="with-extracted-share-icons">⋯</div>
  <div class="story-body__inner" property="articleBody">
   <figure class="media-landscape no-caption full-width lead">⋯</figure>
   <p class="story-body__introduction">
    Getting a TV job in the immediate aftermath of the 9/11 terror attacks in the US with a name like Ubah Mohamed
    was not easy.
   </p>
   <p>⋯</p>
```

```
border-bottom-color
 rgb(64, 64, 64)
border-bottom-style
none
border-bottom-width
0px
border-image-outset
0
border-image-repeat
stretch stretch
border-image-slice
100%
border-image-source
```

Image 4: ***Viewing the code for a specific element using developer tools***

The Developer Tools are significantly more powerful than using the simple find tool, but they are also more complicated. You should choose a method based on your experience and the complexity of the page that you are trying to analyze.

Downloading a web page with Python

Now that we've seen a bit more of how web pages are built in our browser, we can start retrieving and manipulating them using Python. Since Python is not a web browser, we'll only be able to retrieve and manipulate the HTML source code, rather than viewing the 'normal' representation of a web page.

We'll do this through a Python Repl using the `requests` library. Open repl.it[92] and choose to create a new Python Repl.

[92]https://repl.it

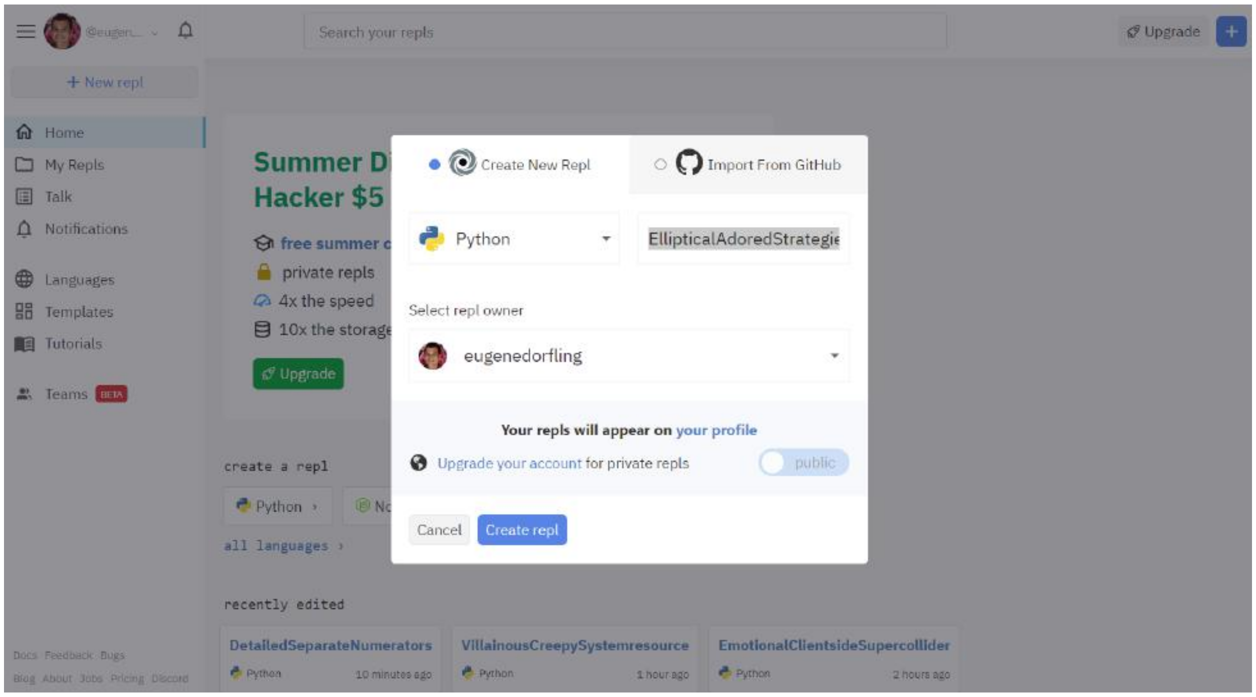

Create New Repl
Import From GitHub
Python
EllipticalAdoredStrategie
Select repl owner
eugenedorfling
Your repls will appear on your profile
Upgrade your account for private repls
public
Cancel
Create repl

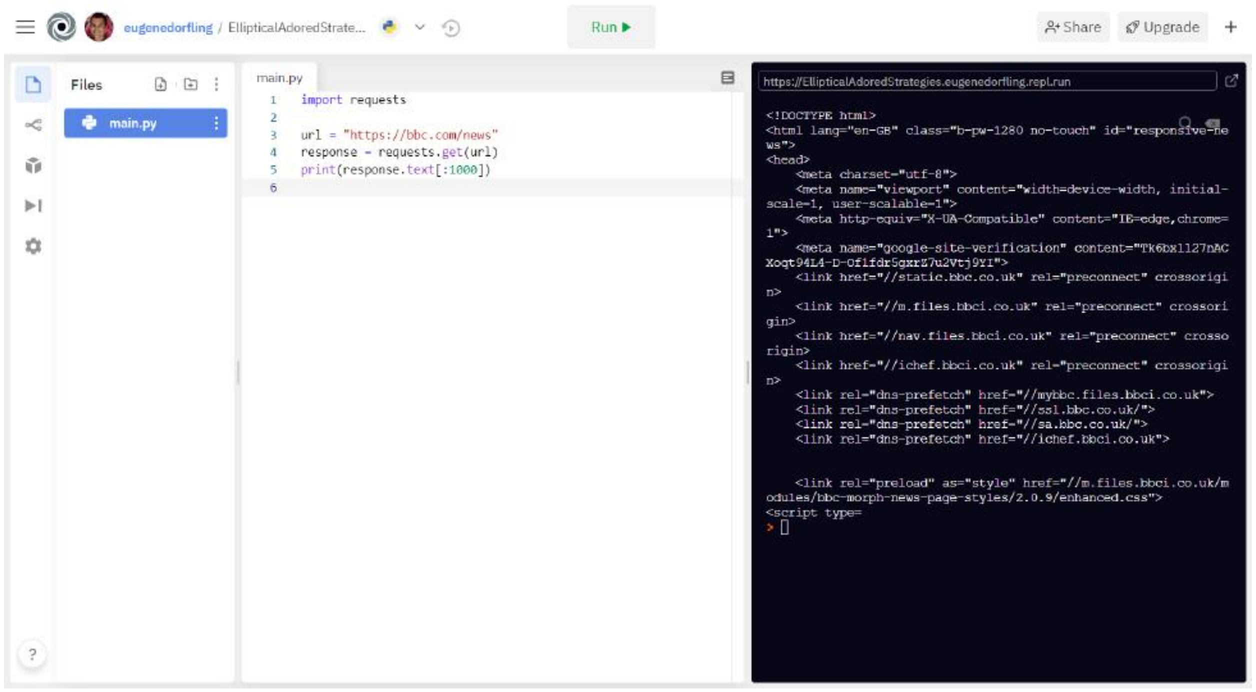
Image 5: ***Create new Repl***

This will take you to a working Python coding environment where you can write and run Python code. To start with, we'll download the content from the BBC News homepage, and print out the first 1000 characters of HTML source code.

You can do this with the following four lines of Python:

```python
import requests

url = "https://bbc.com/news"
response = requests.get(url)
print(response.text[:1000])
```

Put this code in the `main.py` file that Repl automatically creates for you and press the "Run" button. After a short delay, you should see the output in the output pane - the beginning of HTML source code, similar to what we viewed in our web browser above.


eugenedorfling / EllipticalAdoredStrate...
Run ▶
Share
Upgrade
Files
main.py
main.py
https://EllipticalAdoredStrategies.eugenedorfling.repl.run

```
1  import requests
2
3  url = "https://bbc.com/news"
4  response = requests.get(url)
5  print(response.text[:1000])
6
```

```
<!DOCTYPE html>
<html lang="en-GB" class="b-pw-1280 no-touch" id="responsive-ne
ws">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1, user-scalable=1">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=
1">
    <meta name="google-site-verification" content="Tk6bx1127nAC
Xoqt94L4-D-Of1fdr5gxrZ7u2Vtj9YI">
    <link href="//static.bbc.co.uk" rel="preconnect" crossorigi
n>
    <link href="//m.files.bbci.co.uk" rel="preconnect" crossori
gin>
    <link href="//nav.files.bbci.co.uk" rel="preconnect" crosso
rigin>
    <link href="//ichef.bbci.co.uk" rel="preconnect" crossorigi
n>
    <link rel="dns-prefetch" href="//mybbc.files.bbci.co.uk">
    <link rel="dns-prefetch" href="//ssl.bbc.co.uk/">
    <link rel="dns-prefetch" href="//sa.bbc.co.uk/">
    <link rel="dns-prefetch" href="//ichef.bbci.co.uk">


    <link rel="preload" as="style" href="//m.files.bbci.co.uk/m
odules/bbc-morph-news-page-styles/2.0.9/enhanced.css">
<script type=
> 
```

Image 6: ***Downloading a single page using Python***

Let's pull apart each of these lines.

- In line 1, we import the Python `requests` library, which is a library that allows us to make web requests.
- In line 3, we define a variable containing the URL of the main BBC news site. You can visit this URL in your web browser to see the BBC News home page.
- In line 4, we pass the URL we defined to the `requests.get` function, which will visit the web page that the URL points to and fetch the HTML source code. We load this into a new variable called `response`.
- In line 5, we access the `text` attribute of our `response` object, which contains all of the HTML source code. We take only the first 1000 characters of this, and pass them to the `print` function, which simply dumps the resulting text to our output pane.

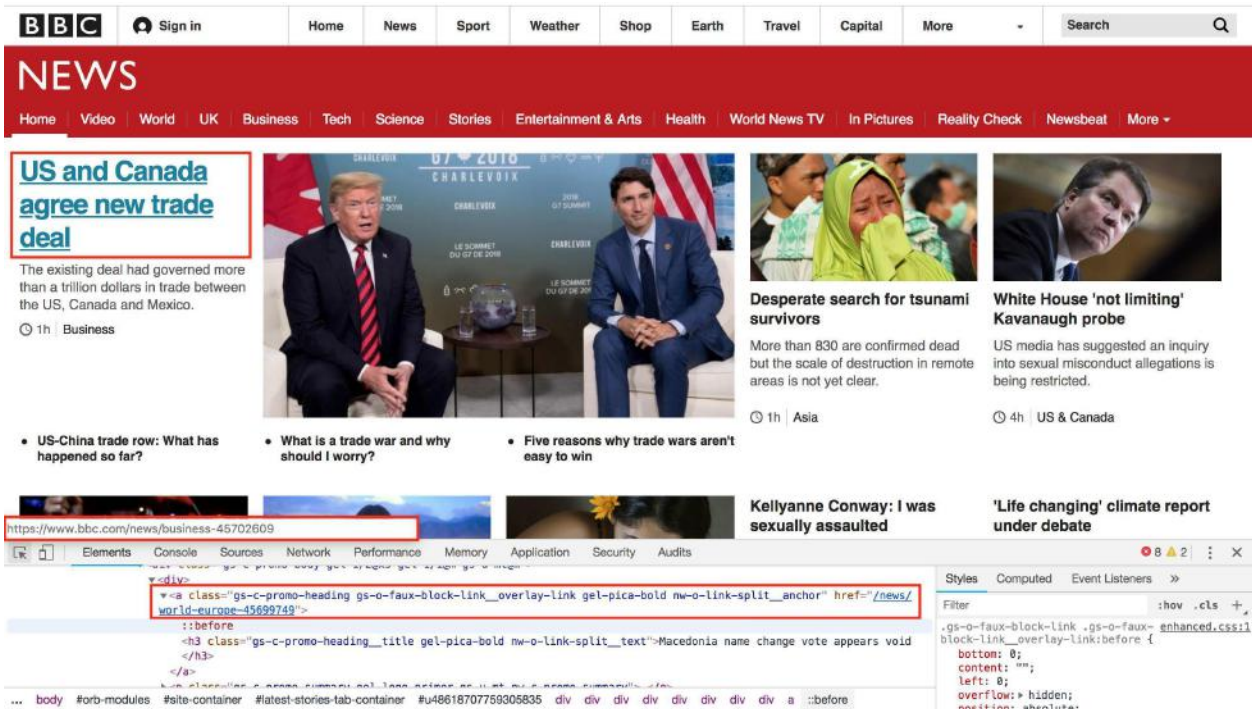
We have now automatically retrieved a web page and we can display parts of the content. We are unlikely to be interested in the full source code dump of a web page (unless we are storing it for archival reasons), so let's extract some interesting parts of the page, instead of only the first 1000 characters.

Using BeautifulSoup to extract all URLs

The world wide web is built from pages that link to each other using hyperlinks, links, or URLs. (These terms are all used more-or-less interchangeably).

Let's assume for now that we want to find all the news articles on the BBC News homepage, and get their URLs. If we look at the main page below, we'll see there are a bunch of stories on the home page. By mousing over any of the headlines with the "inspect" tool, we can see that each has a unique URL which takes us to that news story. For example, mousing over the main "US and Canada agree new trade deal" story in the image below is a link to https://www.bbc.com/news/business-45702609.

If we inspect that element using the browser's developer tools, we can see it is a `<a>` element, which is HTML for a link, with an `<href>` component that points to the URL. Note that the `href` section goes only to the last part of the URL, omitting the https://www.bbc.com part. Because we are already on BBC, the site can use *relative URLs* instead of *absolute URLs*. This means that when you click on the link, your browser will figure out that the URL isn't complete and prepend it with https://www.bbc.com. If you look around the source code of the main BBC page, you'll find both relative and absolute URLs, which already makes scraping all of the URLs on the page more difficult.



US and Canada agree new trade deal

The existing deal had governed more than a trillion dollars in trade between the US, Canada and Mexico.

1h | Business

Desperate search for tsunami survivors

More than 830 are confirmed dead but the scale of destruction in remote areas is not yet clear.

1h | Asia

White House 'not limiting' Kavanaugh probe

US media has suggested an inquiry into sexual misconduct allegations is being restricted.

4h | US & Canada

- US-China trade row: What has happened so far?
- What is a trade war and why should I worry?
- Five reasons why trade wars aren't easy to win

Kellyanne Conway: I was sexually assaulted

'Life changing' climate report under debate

https://www.bbc.com/news/business-45702609

```
▾<div>
  ▾<a class="gs-c-promo-heading gs-o-faux-block-link__overlay-link gel-pica-bold nw-o-link-split__anchor" href="/news/world-europe-45699749">
     ::before
     <h3 class="gs-c-promo-heading__title gel-pica-bold nw-o-link-split__text">Macedonia name change vote appears void
     </h3>
   </a>
```

```
.gs-o-faux-block-link .gs-o-faux-   enhanced.css:1
block-link__overlay-link:before {
  bottom: 0;
  content: "";
  left: 0;
  overflow: ▸ hidden;
```

... body #orb-modules #site-container #latest-stories-tab-container #u48618707759305835 div div div div div div div div a ::before

Image 7: ***Viewing headline links using Developer Tools.***

We could try to use Python's built-in text search functions like `find()` or regular expressions to extract all of the URLs from the BBC page, yet it is not actually possible to do this reliably. HTML is a complex language which allows web developers to do many unusual things. For an amusing take on why we should avoid a "naive" method of looking for links, see this very famous[93] StackOverflow question and the first answer.

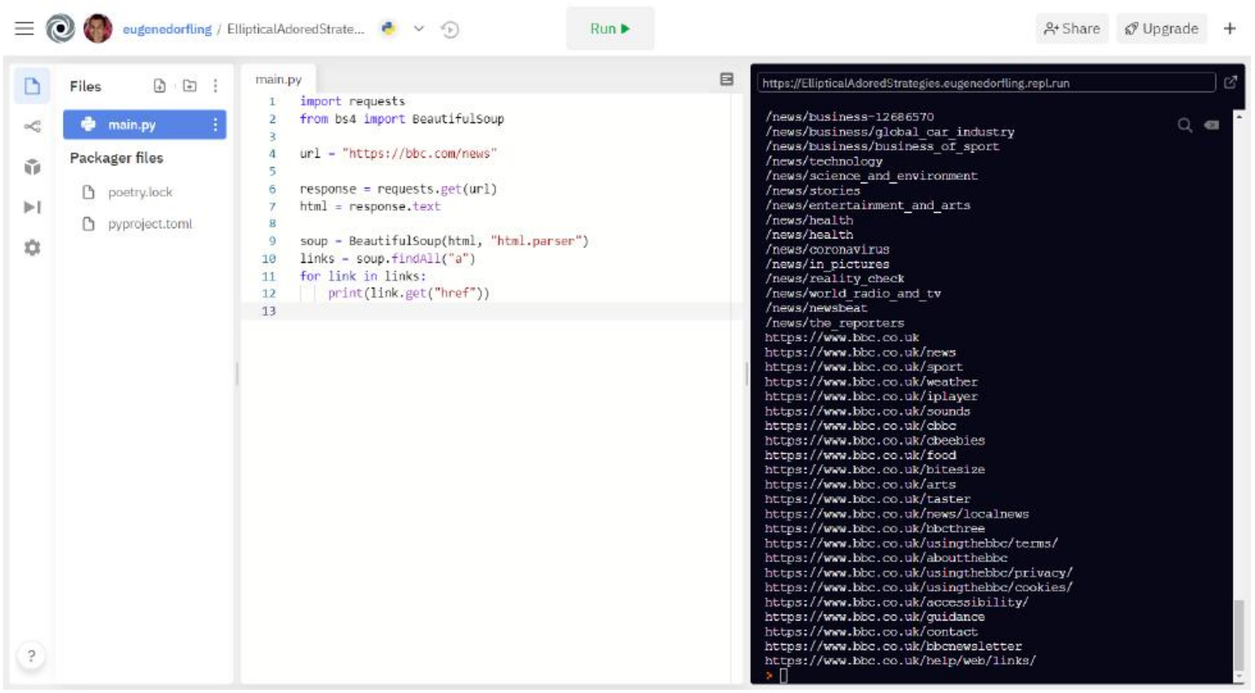
Luckily, there is a powerful and simple-to-use HTML parsing library called BeautifulSoup[94], which will help us extract all the links from a given piece of HTML. We can use it by modifying the code

[93]https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags
[94]https://www.crummy.com/software/BeautifulSoup/

in our Repl to look as follows.

```python
1  import requests
2  from bs4 import BeautifulSoup
3
4  url = "https://bbc.com/news"
5
6  response = requests.get(url)
7  html = response.text
8
9  soup = BeautifulSoup(html, "html.parser")
10 links = soup.findAll("a")
11 for link in links:
12     print(link.get("href"))
```

If you run this code, you'll see that it outputs dozens of URLs, one per line. You'll probably notice that the code now takes quite a bit longer to run than before – BeautifulSoup is not built into Python, it is a third-party module. This means that before running the code, Repl has to go and fetch this library and install it for you. Subsequent runs will be faster.


eugenedorfling / EllipticalAdoredStrate...
Run ▶
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
main.py
https://EllipticalAdoredStrategies.eugenedorfling.repl.run

```python
import requests
from bs4 import BeautifulSoup

url = "https://bbc.com/news"

response = requests.get(url)
html = response.text

soup = BeautifulSoup(html, "html.parser")
links = soup.findAll("a")
for link in links:
    print(link.get("href"))
```

```
/news/business-12686570
/news/business/global_car_industry
/news/business/business_of_sport
/news/technology
/news/science_and_environment
/news/stories
/news/entertainment_and_arts
/news/health
/news/health
/news/coronavirus
/news/in_pictures
/news/reality_check
/news/world_radio_and_tv
/news/newsbeat
/news/the_reporters
https://www.bbc.co.uk
https://www.bbc.co.uk/news
https://www.bbc.co.uk/sport
https://www.bbc.co.uk/weather
https://www.bbc.co.uk/iplayer
https://www.bbc.co.uk/sounds
https://www.bbc.co.uk/cbbc
https://www.bbc.co.uk/cbeebies
https://www.bbc.co.uk/food
https://www.bbc.co.uk/bitesize
https://www.bbc.co.uk/arts
https://www.bbc.co.uk/taster
https://www.bbc.co.uk/news/localnews
https://www.bbc.co.uk/bbcthree
https://www.bbc.co.uk/usingthebbc/terms/
https://www.bbc.co.uk/aboutthebbc
https://www.bbc.co.uk/usingthebbc/privacy/
https://www.bbc.co.uk/usingthebbc/cookies/
https://www.bbc.co.uk/accessibility/
https://www.bbc.co.uk/guidance
https://www.bbc.co.uk/contact
https://www.bbc.co.uk/bbcnewsletter
https://www.bbc.co.uk/help/web/links/
>
```

Image 8: ***Extracting all links from BBC News.***

The code is similar to what we had before with a few additions.

- In line 2, we import the BeautifulSoup library, which is used for parsing and processing HTML.

- One line 9, we transform our HTML into “soup”. This is BeautifulSoup’s representation of a web page, which contains a bunch of useful programmatic features to search and modify the data in the page. We use the “html.parser” option to parse HTML which is included by default – BeautifulSoup also allows you specify a custom HTML parser here. For example, you could install and specify a faster parser which can be useful if you need to process a lot of HTML data.
- In line 10, we find all the `a` elements in our HTML and extract them to a list. Remember, when we were looking at the URLs using our web browser (Image 7), we noted that the `<a>` element in HTML was used to define links, with the `href` attribute being used to specify where the link should go to. This line finds all of the HTML `<a>` elements.
- In line 11, we loop through all of the links we have, and in line 12 we print out the `href` section.

These last two lines show why BeautifulSoup is useful. To try and find and extract these elements without it would be remarkably difficult, but now we can do it in two lines of readable code!

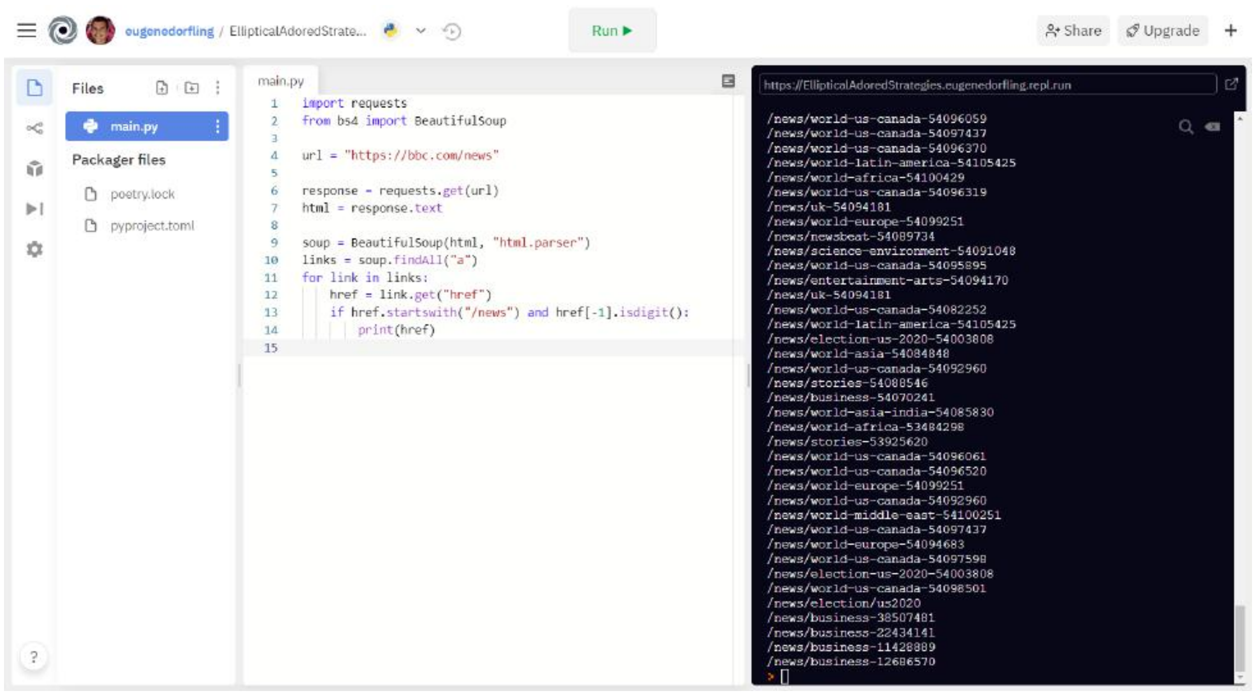
If we look at the URLs in the output pane, we’ll see quite a mixed bag of results. We have absolute URLs (starting with “http”) and relative ones (starting with “/”). Most of them go to general pages rather than specific news articles. We need to find a pattern in the links we’re interested in (that go to news articles) so that we can extract only those.

Again, trial and error is the best way to do this. If we go to the BBC News home page and use developer tools to inspect the links that go to news articles, we’ll find that they all have a similar pattern. They are relative URLs which start with “/news” and end with a long number, e.g. `/news/newsbeat-45705989`

We can make a small change to our code to only output URLs that match this pattern. Replace the last two lines of our Python code with the following four lines:

```python
for link in links:
    href = link.get("href")
    if href.startswith("/news") and href[-1].isdigit():
        print(href)
```

Here we still loop through all of the links that BeautifulSoup found for us, but now we extract the `href` to its own variable immediately after. We then inspect this variable to make sure that it matches our conditions (starts with “/news” and ends with a digit), and only if it does, then we print it out.



```
import requests
from bs4 import BeautifulSoup

url = "https://bbc.com/news"

response = requests.get(url)
html = response.text

soup = BeautifulSoup(html, "html.parser")
links = soup.findAll("a")
for link in links:
    href = link.get("href")
    if href.startswith("/news") and href[-1].isdigit():
        print(href)
```

```
/news/world-us-canada-54096059
/news/world-us-canada-54097437
/news/world-us-canada-54096370
/news/world-latin-america-54105425
/news/world-africa-54100429
/news/world-us-canada-54096319
/news/uk-54094181
/news/world-europe-54099251
/news/newsbeat-54089734
/news/science-environment-54091048
/news/world-us-canada-54095895
/news/entertainment-arts-54094170
/news/uk-54094181
/news/world-us-canada-54082252
/news/world-latin-america-54105425
/news/election-us-2020-54003808
/news/world-asia-54084848
/news/world-us-canada-54092960
/news/stories-54088546
/news/business-54070241
/news/world-asia-india-54085830
/news/world-africa-53484298
/news/stories-53925620
/news/world-us-canada-54096061
/news/world-us-canada-54096520
/news/world-europe-54099251
/news/world-us-canada-54092960
/news/world-middle-east-54100251
/news/world-us-canada-54097437
/news/world-europe-54094683
/news/world-us-canada-54097598
/news/election-us-2020-54003808
/news/world-us-canada-54098501
/news/election/us2020
/news/business-38507481
/news/business-22434141
/news/business-11428889
/news/business-12686570
> 
```

Image 9: ***Printing only links to news articles from BBC.***

Fetching all of the articles from the homepage

Now that we have the link to every article on the BBC News homepage, we can fetch the data for each one of these individual articles. As a toy project, let's extract the proper nouns (people, places, etc) from each article and print out the most common ones to get a sense on what things are being talked about today.

Adapt your code to look as follows:

```
import requests
import string

from collections import Counter

from bs4 import BeautifulSoup


url = "https://bbc.com/news"


response = requests.get(url)
html = response.text
```

```
soup = BeautifulSoup(html, "html.parser")
links = soup.findAll("a")

news_urls = []
for link in links:
    href = link.get("href")
    if href.startswith("/news") and href[-1].isdigit():
        news_url = "https://bbc.com" + href
        news_urls.append(news_url)


all_nouns = []
for url in news_urls[:10]:
    print("Fetching {}".format(url))
    response = requests.get(url)
    html = response.text
    soup = BeautifulSoup(html, "html.parser")

    words = soup.text.split()
    nouns = [word for word in words if word.isalpha() and word[0] in string.ascii_up\
percase]
    all_nouns += nouns

print(Counter(all_nouns).most_common(100))
```

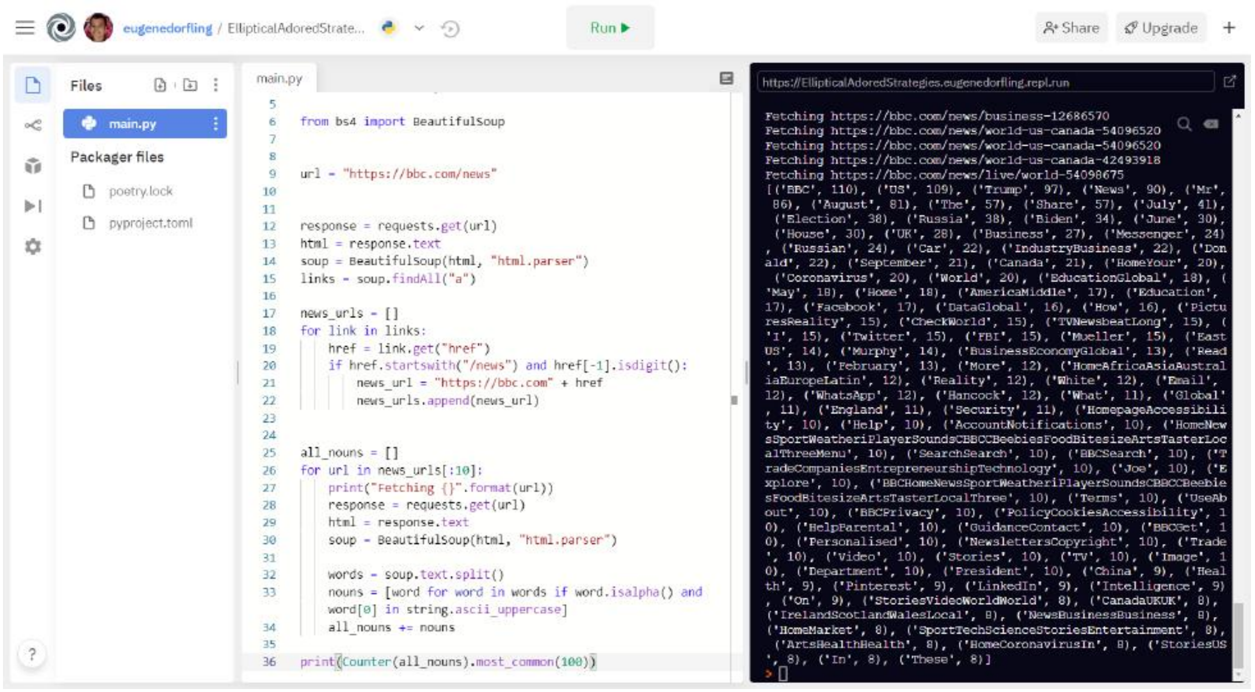
This code is quite a bit more complicated than what we previously wrote, so don't worry if you don't understand all of it. The main changes are:

- At the top, we add two new imports in addition to the `requests` library. The first new module is one for `string`, which is a standard Python module that contains some useful word and letter shortcuts. We'll use it to identify all the capital letters in our alphabet. The second module is a `Counter`, which is part of the built-in `collections` module. This will let us find the most common nouns in a list, once we have built a list of all the nouns.
- We've added `news_urls = []` at the top of the first `for` loop. Instead of printing out each URL once we've identified it as a "news URL", we add it to this list so we can download each page later. Inside the for loop two lines down, we combine the root domain ("http://bbc.com") with each href attribute and then add the complete URL to our `news_urls` list.
- We then go into another for loop, where we loop through the first 10 news URLs (if you have more time, you can remove the `[:10]` part to iterate through all the news pages, but for efficiency, we'll just demonstrate with the first 10).
- We print out the URL that we're fetching (as it takes a second or so to download each page, it's nice to display some feedback so we can see that the program is working).
- We then fetch the page and turn it into soup, as we did before.

- With `words = soup.text.split()` we extract all the text from the page and split this resulting big body of text into individual words. The Python `split()` function splits on white space, which is a crude way to extract words from a piece of text, but it will serve our purpose for now.
- The next line loops through all the words in that given article and keeps only the ones that are made up of numeric characters and which start with a capital letter (`string.ascii_uppercase` is just the uppercase alphabet). This is also an extremely crude way of extracting nouns, and we will get a lot of words (like those at the start of sentences) which are not actually proper nouns, but again it's a good enough approximation for now.
- We then add all the words that look like nouns to our `all_nouns` list and move on to the next article to do the same.
- Finally, once we've downloaded all the pages, we print out the 100 most common nouns along with a count of how often they appeared using Python's convenient `Counter` object.

You should see output similar to that in the image below (though your words will be different, as the news changes every few hours). We have the most common "nouns" followed by a count of how often that noun appeared in all 10 of the articles we looked at.

We can see that our crude extraction and parsing methods are far from perfect – words like "Twitter" and "Facebook" appear in most articles because of the social media links at the bottom of each article, so their presence doesn't mean that Facebook and Twitter themselves are in the news today. Similarly, words like "From" aren't nouns, and other words like "BBC" and "Business" are also included because they appear on each page, outside of the main article text.


eugenedorfling / EllipticalAdoredStrate...
Run ▶
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
https://EllipticalAdoredStrategies.eugenedorfling.repl.run

```python
from bs4 import BeautifulSoup


url = "https://bbc.com/news"


response = requests.get(url)
html = response.text
soup = BeautifulSoup(html, "html.parser")
links = soup.findAll("a")

news_urls = []
for link in links:
    href = link.get("href")
    if href.startswith("/news") and href[-1].isdigit():
        news_url = "https://bbc.com" + href
        news_urls.append(news_url)


all_nouns = []
for url in news_urls[:10]:
    print("Fetching {}".format(url))
    response = requests.get(url)
    html = response.text
    soup = BeautifulSoup(html, "html.parser")

    words = soup.text.split()
    nouns = [word for word in words if word.isalpha() and
    word[0] in string.ascii_uppercase]
    all_nouns += nouns

print(Counter(all_nouns).most_common(100))
```

```
Fetching https://bbc.com/news/business-12686570
Fetching https://bbc.com/news/world-us-canada-54096520
Fetching https://bbc.com/news/world-us-canada-54096520
Fetching https://bbc.com/news/world-us-canada-42493918
Fetching https://bbc.com/news/live/world-54098675
[('BBC', 110), ('US', 109), ('Trump', 97), ('News', 90), ('Mr',
 86), ('August', 81), ('The', 57), ('Share', 57), ('July', 41),
 ('Election', 38), ('Russia', 38), ('Biden', 34), ('June', 30),
 ('House', 30), ('UK', 28), ('Business', 27), ('Messenger', 24)
, ('Russian', 24), ('Car', 22), ('IndustryBusiness', 22), ('Don
ald', 22), ('September', 21), ('Canada', 21), ('HomeYour', 20),
 ('Coronavirus', 20), ('World', 20), ('EducationGlobal', 18), (
'May', 18), ('Home', 18), ('AmericaMiddle', 17), ('Education',
17), ('Facebook', 17), ('DataGlobal', 16), ('How', 16), ('Pictu
resReality', 15), ('CheckWorld', 15), ('TVNewsbeatLong', 15), (
'I', 15), ('Twitter', 15), ('FBI', 15), ('Mueller', 15), ('East
US', 14), ('Murphy', 14), ('BusinessEconomyGlobal', 13), ('Read
', 13), ('February', 13), ('More', 12), ('HomeAfricaAsiaAustral
iaEuropeLatin', 12), ('Reality', 12), ('White', 12), ('Email',
12), ('WhatsApp', 12), ('Hancock', 12), ('What', 11), ('Global'
, 11), ('England', 11), ('Security', 11), ('HomepageAccessibili
ty', 10), ('Help', 10), ('AccountNotifications', 10), ('HomeNew
sSportWeatheriPlayerSoundsCBBCCBeebiesFoodBitesizeArtsTasterLoc
alThreeMenu', 10), ('SearchSearch', 10), ('BBCSearch', 10), ('T
radeCompaniesEntrepreneurshipTechnology', 10), ('Joe', 10), ('E
xplore', 10), ('BBCHomeNewsSportWeatheriPlayerSoundsCBBCCBeebie
sFoodBitesizeArtsTasterLocalThree', 10), ('Terms', 10), ('UseAb
out', 10), ('BBCPrivacy', 10), ('PolicyCookiesAccessibility', 1
0), ('HelpParental', 10), ('GuidanceContact', 10), ('BBCGet', 1
0), ('Personalised', 10), ('NewslettersCopyright', 10), ('Trade
', 10), ('Video', 10), ('Stories', 10), ('TV', 10), ('Image', 1
0), ('Department', 10), ('President', 10), ('China', 9), ('Heal
th', 9), ('Pinterest', 9), ('LinkedIn', 9), ('Intelligence', 9)
, ('On', 9), ('StoriesVideoWorldWorld', 8), ('CanadaUKUK', 8),
('IrelandScotlandWalesLocal', 8), ('NewsBusinessBusiness', 8),
('HomeMarket', 8), ('SportTechScienceStoriesEntertainment', 8),
 ('ArtsHealthHealth', 8), ('HomeCoronavirusIn', 8), ('StoriesUS
', 8), ('In', 8), ('These', 8)]
> 
```

Image: 10 ***The final output of our program, showing the words that appear most often in BBC articles.***

Where next?

We've completed the basics of web scraping and have looked at how the web works, how to extract information from web pages, and how to do some very basic text extraction. You will probably want to do something other than extract words from BBC! You can fork this Repl from https://repl.it/@GarethDwyer1/beginnerwebscraping and modify it to change which site it scrapes and what content it extracts. You can also join the Repl Discord Server[95] to chat with other developers who are working on similar projects and who will happily exchange ideas with you or help if you get stuck.

We have walked through a very flexible method of web scraping, but it's the "quick and dirty" way. If BBC updates their website and some of our assumptions (e.g. that news URLs will end with a number) break, our web scraper will also break.

Once you've done a bit of web scraping, you'll notice that the same patterns and problems come up again and again. Because of this, there are many frameworks and other tools that solve these common problems (finding all the URLs on the page, extracting text from the other code, dealing with changing web sites, etc), and for any big web scraping project, you'll definitely want to use these instead of starting from scratch.

Some of the best Python web scraping tools are:

- **Scrapy**[96]: A framework used by people who want to scrape millions or even billions of web pages. Scrapy lets you build "spiders" – programmatic robots that move around the web at high speed, gathering data based on rules that you specify.
- **Newspaper**[97]: we touched on how it was difficult to separate the main text of an online news article from all the other content on the page (headers, footers, adverts, etc). This problem is an incredibly difficult one to solve. Newspaper uses a combination of manually specified rules and some clever algorithms to remove the "boilerplate" or non-core text from each article.
- **Selenium**[98]: we scraped some basic content without using a web browser, and this works fine for images and text. Many parts of the modern web are dynamic though – e.g. they only load when you scroll down a page far enough or click on a button to reveal more content. These dynamic sites are challenging to scrape, but Selenium allows you to fire up a real web browser and control it just as a human would (but automatically), and this allows you to access this kind of dynamic content.

There is no shortage of other tools, and a lot can be done simply by using them in combination with each other. Web scraping is a vast world that we've only just touched on, but we'll explore some more web scraping use cases in the next chapter, in particular, building news word clouds.

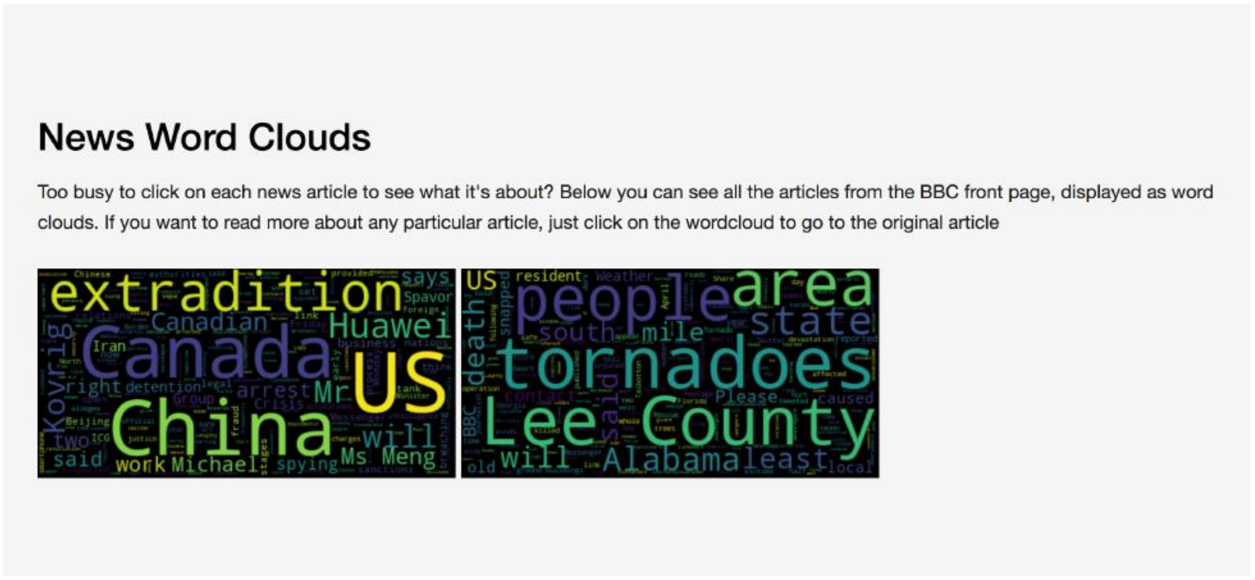
[95]https://discord.com/login?redirect_to=%2Fchannels%2F%40me
[96]https://scrapy.org/
[97]https://github.com/codelucas/newspaper
[98]https://www.seleniumhq.org/

Building news word clouds using Python and Repl.it

Word clouds, which are images showing scattered words in different sizes, are a popular way to visualise large amounts of text. Words that appear more frequently in the given text are larger, and less common words are smaller or not shown at all.

In this tutorial, we'll build a web application using Python and Flask that transforms the latest news stories into word clouds and displays them to our visitors.

At the end of this tutorial, our users will see a page similar to the one shown below, but containing the latest news headlines from BBC news. We'll learn some tricks about web scraping, RSS feeds, and building image files directly in memory along the way.



News Word Clouds

Too busy to click on each news article to see what it's about? Below you can see all the articles from the BBC front page, displayed as word clouds. If you want to read more about any particular article, just click on the wordcloud to go to the original article

Image: 1

Overview and requirements

We'll be building a simple web application step-by-step and explaining each line of code in detail. To follow, you should have some basic knowledge of programming and web concepts, such as what if statements are and how to use URLs. We'll be using Python for this tutorial, but we won't assume that you're a Python expert.

Specifically, we'll:

- Look at RSS feeds and how to use them in Python
- Show how to set up a basic Flask web application
- Use BeautifulSoup to extract text from online news articles
- Use WordCloud to transform the text into images
- Import Bootstrap and add some basic CSS styling

We'll be using the online programming environment Repl.it[99] so you won't need to install any software locally to follow along step by step. If you want to adapt this guide to your own needs, you should create a free account by going to repl.it[100] and follow their sign up process.

Web scraping

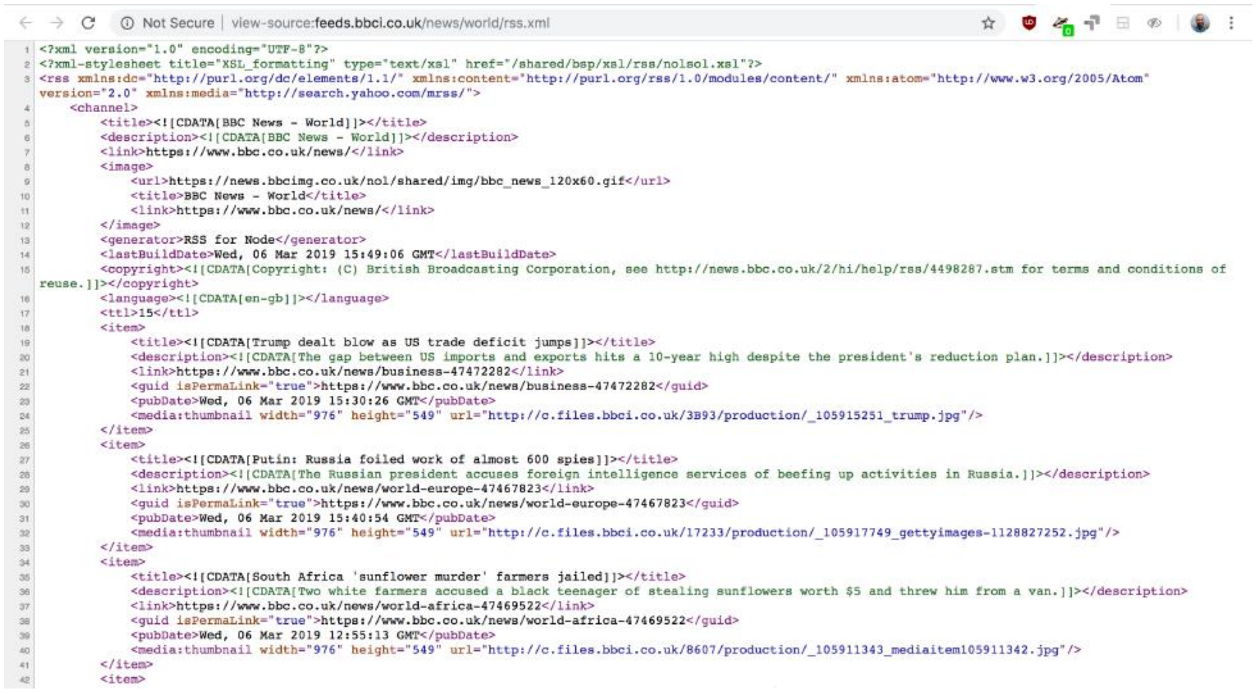
We previously looked at basic web scraping in an introduction to web scraping[101]. If you're completely new to the idea of automatically retrieving content from the internet, have a look at that tutorial first.

In this tutorial, instead of scraping the links to news articles directly from the BBC homepage, we'll be using RSS feeds[102] - an old but popular standardised format that publications use to let readers know when new content is available.

Taking a look at RSS Feeds

RSS feeds are published as XML documents. Every time BBC (and other places) publishes a new article to their home page, they also update an XML machine-readable document at http://feeds.bbci.co.uk/news/world/r This is a fairly simple feed consisting of a `<channel>` element, which has some metadata and then a list of `<item>` elements, each of which represents a new article. The articles are arranged chronologically, with the newest ones at the top, so it's easy to retrieve new content.

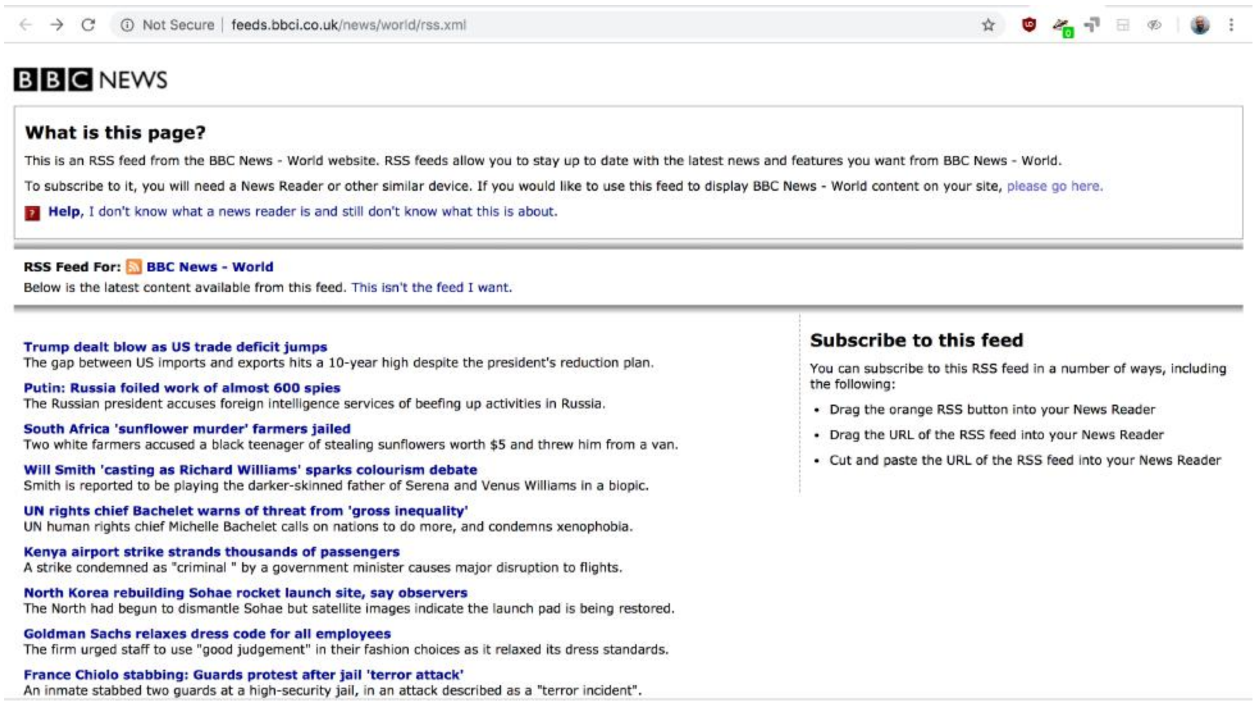
[99]https://repl.it
[100]https://repl.it
[101]https://www.codementor.io/garethdwyer/beginner-web-scraping-with-python-and-repl-it-nzr27jvnq
[102]https://en.wikipedia.org/wiki/RSS


Not Secure | view-source:feeds.bbci.co.uk/news/world/rss.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet title="XSL_formatting" type="text/xsl" href="/shared/bsp/xsl/rss/nolsol.xsl"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:content="http://purl.org/rss/1.0/modules/content/" xmlns:atom="http://www.w3.org/2005/Atom"
version="2.0" xmlns:media="http://search.yahoo.com/mrss/">
    <channel>
        <title><![CDATA[BBC News - World]]></title>
        <description><![CDATA[BBC News - World]]></description>
        <link>https://www.bbc.co.uk/news/</link>
        <image>
            <url>https://news.bbcimg.co.uk/nol/shared/img/bbc_news_120x60.gif</url>
            <title>BBC News - World</title>
            <link>https://www.bbc.co.uk/news/</link>
        </image>
        <generator>RSS for Node</generator>
        <lastBuildDate>Wed, 06 Mar 2019 15:49:06 GMT</lastBuildDate>
        <copyright><![CDATA[Copyright: (C) British Broadcasting Corporation, see http://news.bbc.co.uk/2/hi/help/rss/4498287.stm for terms and conditions of
reuse.]]></copyright>
        <language><![CDATA[en-gb]]></language>
        <ttl>15</ttl>
        <item>
            <title><![CDATA[Trump dealt blow as US trade deficit jumps]]></title>
            <description><![CDATA[The gap between US imports and exports hits a 10-year high despite the president's reduction plan.]]></description>
            <link>https://www.bbc.co.uk/news/business-47472282</link>
            <guid isPermaLink="true">https://www.bbc.co.uk/news/business-47472282</guid>
            <pubDate>Wed, 06 Mar 2019 15:30:26 GMT</pubDate>
            <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/3B93/production/_105915251_trump.jpg"/>
        </item>
        <item>
            <title><![CDATA[Putin: Russia foiled work of almost 600 spies]]></title>
            <description><![CDATA[The Russian president accuses foreign intelligence services of beefing up activities in Russia.]]></description>
            <link>https://www.bbc.co.uk/news/world-europe-47467823</link>
            <guid isPermaLink="true">https://www.bbc.co.uk/news/world-europe-47467823</guid>
            <pubDate>Wed, 06 Mar 2019 15:40:54 GMT</pubDate>
            <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/17233/production/_105917749_gettyimages-1128827252.jpg"/>
        </item>
        <item>
            <title><![CDATA[South Africa 'sunflower murder' farmers jailed]]></title>
            <description><![CDATA[Two white farmers accused a black teenager of stealing sunflowers worth $5 and threw him from a van.]]></description>
            <link>https://www.bbc.co.uk/news/world-africa-47469522</link>
            <guid isPermaLink="true">https://www.bbc.co.uk/news/world-africa-47469522</guid>
            <pubDate>Wed, 06 Mar 2019 12:55:13 GMT</pubDate>
            <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/8607/production/_105911343_mediaitem105911342.jpg"/>
        </item>
        <item>
```

Image: 2

If you click on the link above, you won't see the XML directly. Instead, it has some associated styling information so that most web browsers will display something that's a bit more human friendly. For example, opening the page in Google Chrome shows the page below. In order to view the raw XML directly, you can right-click on the page and click "view source".


Not Secure | feeds.bbci.co.uk/news/world/rss.xml

BBC NEWS

What is this page?

This is an RSS feed from the BBC News - World website. RSS feeds allow you to stay up to date with the latest news and features you want from BBC News - World.

To subscribe to it, you will need a News Reader or other similar device. If you would like to use this feed to display BBC News - World content on your site, please go here.

Help, I don't know what a news reader is and still don't know what this is about.

RSS Feed For: **BBC News - World**

Below is the latest content available from this feed. This isn't the feed I want.

Trump dealt blow as US trade deficit jumps
The gap between US imports and exports hits a 10-year high despite the president's reduction plan.

Putin: Russia foiled work of almost 600 spies
The Russian president accuses foreign intelligence services of beefing up activities in Russia.

South Africa 'sunflower murder' farmers jailed
Two white farmers accused a black teenager of stealing sunflowers worth $5 and threw him from a van.

Will Smith 'casting as Richard Williams' sparks colourism debate
Smith is reported to be playing the darker-skinned father of Serena and Venus Williams in a biopic.

UN rights chief Bachelet warns of threat from 'gross inequality'
UN human rights chief Michelle Bachelet calls on nations to do more, and condemns xenophobia.

Kenya airport strike strands thousands of passengers
A strike condemned as "criminal " by a government minister causes major disruption to flights.

North Korea rebuilding Sohae rocket launch site, say observers
The North had begun to dismantle Sohae but satellite images indicate the launch pad is being restored.

Goldman Sachs relaxes dress code for all employees
The firm urged staff to use "good judgement" in their fashion choices as it relaxed its dress standards.

France Chiolo stabbing: Guards protest after jail 'terror attack'
An inmate stabbed two guards at a high-security jail, in an attack described as a "terror incident".

Subscribe to this feed

You can subscribe to this RSS feed in a number of ways, including the following:

- Drag the orange RSS button into your News Reader
- Drag the URL of the RSS feed into your News Reader
- Cut and paste the URL of the RSS feed into your News Reader

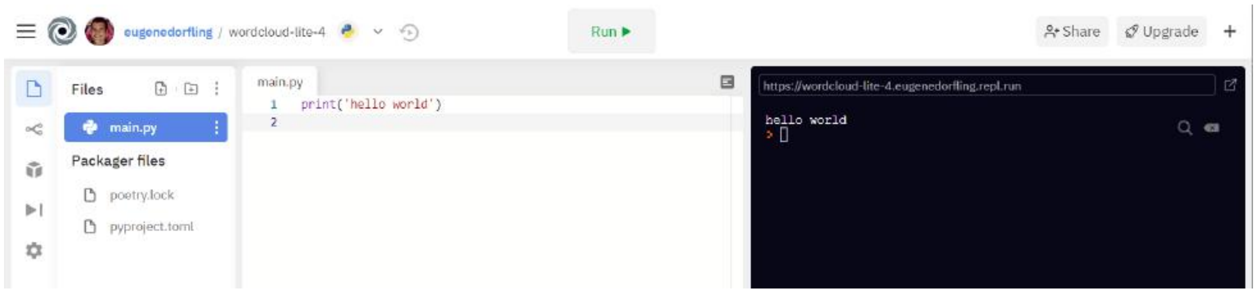
Image: 3

RSS feeds are used internally by software such as the news reader Feedly[103] and various email clients. We'll be consuming these RSS feeds with a Python library to retrieve the latest articles from BBC.

Setting up our online environment (Repl.it)

In this tutorial, we'll be building our web application using Repl.it[104], which will allow us to have a consistent code editor, environment, and deployment framework in a single click. Head over there and create an account. Choose to create a Python Repl, and you should see an editor where you can write and run Python code, similar to the image below. You can write Python code in the middle pane, run it by pressing the green "run" button, and see the output in the right pane. In the left pane, you can see a list of files, with `main.py` added there by default.

[103] feedly.com
[104] repl.it


eugenedorfling / wordcloud-lite-4
Run ▶
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
https://wordcloud-lite-4.eugenedorfling.repl.run

```
print('hello world')
```

```
hello world
> 
```

Image: 4

Pulling data from our feed and extracting URLs

In the previous webscraping tutorial[105] we used BeautifulSoup[106] to look for hyperlinks in a page and extract them. Now that we are using RSS, we can simply parse the feed as described above to find these same URLs. We will be using the Python feedparser[107] library to do this.

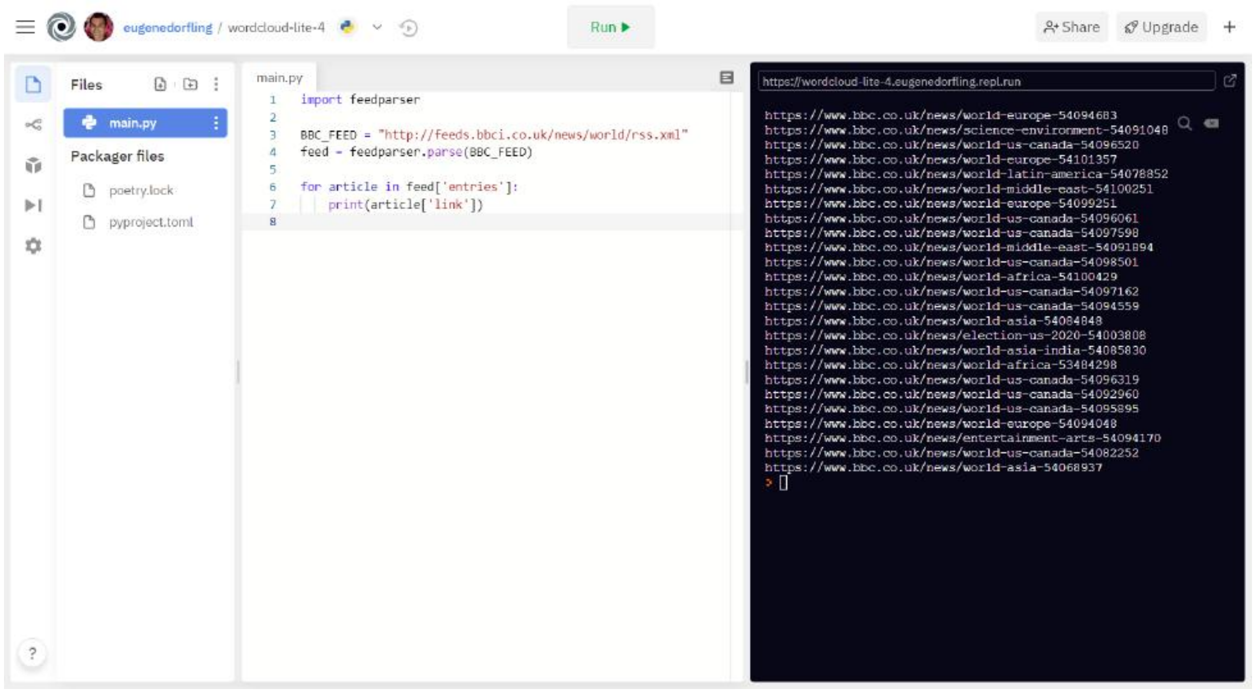
Let's start by simply printing out the URLs for all of the latest articles from BBC. Switch back to the `main.py` file in the Repl.it IDE and add the following code.

```python
1 import feedparser
2 
3 BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
4 feed = feedparser.parse(BBC_FEED)
5 
6 for article in feed['entries']:
7     print(article['link'])
```

Feedparser does most of the heavy lifting for us, so we don't have to get too close to the slightly cumbersome XML format. In the code above, we parse the feed into a nice Python representation (line 4), loop through all of the `entries` (the `<item>` entries from the XML we looked at earlier), and print out the `link` elements.

If you run this code, you should see a few dozen URLs output on the right pane, as in the image below.

[105] https://www.codementor.io/garethdwyer/beginner-web-scraping-with-python-and-repl-it-nzr27jvnq
[106] https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[107] https://pythonhosted.org/feedparser/


eugenedorfling / wordcloud-lite-4
Run ▶
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
https://wordcloud-lite-4.eugenedorfling.repl.run

```python
import feedparser

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
feed = feedparser.parse(BBC_FEED)

for article in feed['entries']:
    print(article['link'])
```

```
https://www.bbc.co.uk/news/world-europe-54094683
https://www.bbc.co.uk/news/science-environment-54091048
https://www.bbc.co.uk/news/world-us-canada-54096520
https://www.bbc.co.uk/news/world-europe-54101357
https://www.bbc.co.uk/news/world-latin-america-54078852
https://www.bbc.co.uk/news/world-middle-east-54100251
https://www.bbc.co.uk/news/world-europe-54099251
https://www.bbc.co.uk/news/world-us-canada-54096061
https://www.bbc.co.uk/news/world-us-canada-54097598
https://www.bbc.co.uk/news/world-middle-east-54091894
https://www.bbc.co.uk/news/world-us-canada-54098501
https://www.bbc.co.uk/news/world-africa-54100429
https://www.bbc.co.uk/news/world-us-canada-54097162
https://www.bbc.co.uk/news/world-us-canada-54094559
https://www.bbc.co.uk/news/world-asia-54084848
https://www.bbc.co.uk/news/election-us-2020-54003808
https://www.bbc.co.uk/news/world-asia-india-54085830
https://www.bbc.co.uk/news/world-africa-53484298
https://www.bbc.co.uk/news/world-us-canada-54096319
https://www.bbc.co.uk/news/world-us-canada-54092960
https://www.bbc.co.uk/news/world-us-canada-54095895
https://www.bbc.co.uk/news/world-europe-54094048
https://www.bbc.co.uk/news/entertainment-arts-54094170
https://www.bbc.co.uk/news/world-us-canada-54082252
https://www.bbc.co.uk/news/world-asia-54068937
> 
```

Image: 5

Setting up a web application with Flask

We don't just want to print this data out in the Repl console. Instead, our application should return information to anyone who uses a web browser to visit our application. We'll, therefore, install the lightweight web framework Flask[108] and use this to serve web content to our visitors.

In the `main.py` file, we need to modify our code to look as follows:

```python
import feedparser
from flask import Flask

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"

@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    urls = []

    for article in feed['entries']:
```

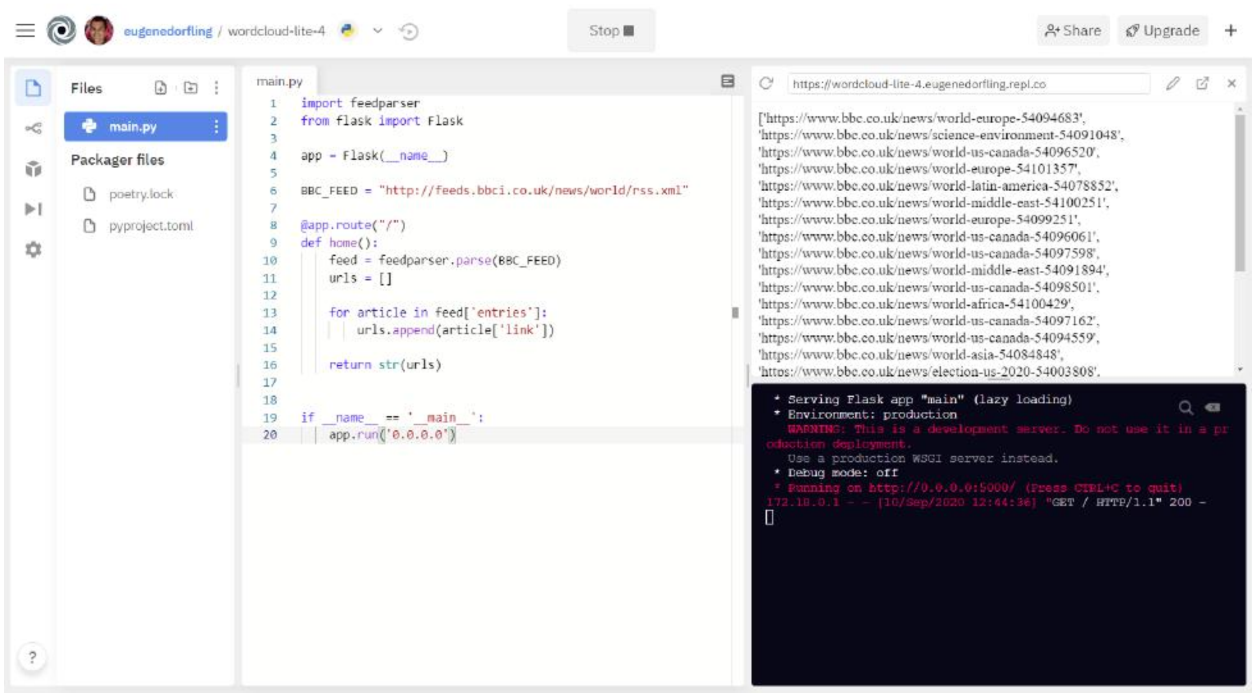
[108] http://flask.pocoo.org/

```
14            urls.append(article['link'])
15
16        return str(urls)
17
18
19    if __name__ == '__main__':
20        app.run('0.0.0.0')
```

Here we still parse the feed and extract all of the latest article URLs, but instead of printing them out, we add them to a list (`urls`), and return them from a function. The interesting parts of this code are:

- **Line 2**: we import Flask
- **Line 4**: we initialise Flask to turn our project into a web application
- **Line 8**: we use a decorator to define the homepage of our application (an empty route, or `/`).
- **Lines 19-20**: We run Flask's built-in webserver to serve our content.

Press "run" again, and you should see a new window appear in the top right pane. Here we can see a basic web page (viewable already to anyone in the world by sharing the URL you see above it), and we see the same output that we previously printed to the console.


eugenedorfling / wordcloud-lite-4
Stop
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
main.py
https://wordcloud-lite-4.eugenedorfling.repl.co

```
import feedparser
from flask import Flask

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"

@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    urls = []

    for article in feed['entries']:
        urls.append(article['link'])

    return str(urls)


if __name__ == '__main__':
    app.run('0.0.0.0')
```

```
['https://www.bbc.co.uk/news/world-europe-54094683',
'https://www.bbc.co.uk/news/science-environment-54091048',
'https://www.bbc.co.uk/news/world-us-canada-54096520',
'https://www.bbc.co.uk/news/world-europe-54101357',
'https://www.bbc.co.uk/news/world-latin-america-54078852',
'https://www.bbc.co.uk/news/world-middle-east-54100251',
'https://www.bbc.co.uk/news/world-europe-54099251',
'https://www.bbc.co.uk/news/world-us-canada-54096061',
'https://www.bbc.co.uk/news/world-us-canada-54097598',
'https://www.bbc.co.uk/news/world-middle-east-54091894',
'https://www.bbc.co.uk/news/world-us-canada-54098501',
'https://www.bbc.co.uk/news/world-africa-54100429',
'https://www.bbc.co.uk/news/world-us-canada-54097162',
'https://www.bbc.co.uk/news/world-us-canada-54094559',
'https://www.bbc.co.uk/news/world-asia-54084848',
'https://www.bbc.co.uk/news/election-us-2020-54003808',
```

```
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a pr
oduction deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
172.18.0.1 - - [10/Sep/2020 12:44:36] "GET / HTTP/1.1" 200 -
```

Image: 6

Downloading articles and extracting the text

The URLs aren't that useful to us, as we eventually want to display a summary of the *content* of each URL. The actual text of each article isn't included in the RSS feed that we have (some RSS feeds contain the full text of each article), so we'll need to do some more work to download each article. First, we'll add the third-party libraries `requests` and `BeautifulSoup` as dependencies, again just using the "magic import". We'll be using these to download the content of each article from the URL and strip out extra CSS and JavaScript to leave us with plain text.

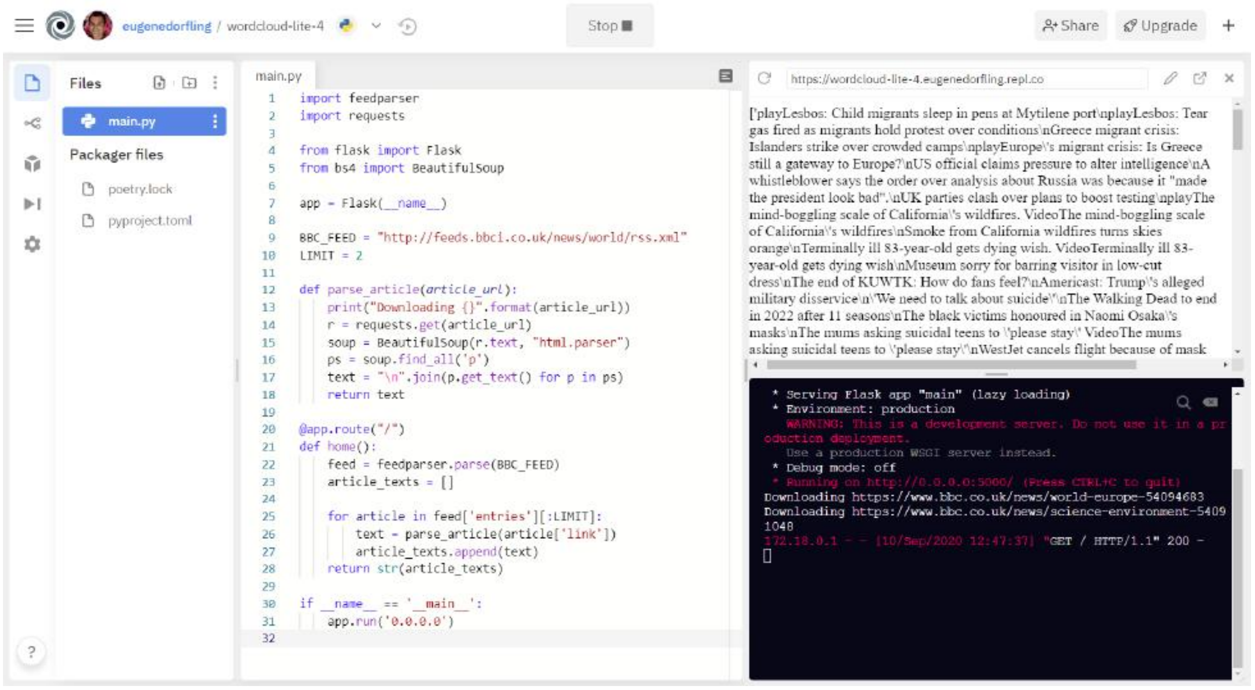
Now we're ready to download the content from each article and serve that up to the user. Modify the code in `main.py` to look as follows.

```python
import feedparser
import requests

from flask import Flask
from bs4 import BeautifulSoup

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
LIMIT = 2

def parse_article(article_url):
    print("Downloading {}".format(article_url))
    r = requests.get(article_url)
    soup = BeautifulSoup(r.text, "html.parser")
    ps = soup.find_all('p')
    text = "\n".join(p.get_text() for p in ps)
    return text

@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    article_texts = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        article_texts.append(text)
    return str(article_texts)

if __name__ == '__main__':
    app.run('0.0.0.0')
```

Let’s take a closer look at what has changed.

- We import our new libraries on **lines 2 and 5**.
- We create a new global variable `LIMIT` on **line 10** to limit how many articles we want to download.
- **Lines 12-18** define a new function that takes a URL, downloads the article, and extracts the text. It does this using a crude algorithm that assumes anything inside HTML `<p>` (paragraph) tags is interesting content.
- We modify **lines 23, 25, 26, and 27** so that we use the new `parse_article` function to get the actual content of the URLs that we found in the RSS feed and return that to the user instead of returning the URL directly. Note that we limit this to two articles by truncating our list to `LIMIT` for now, as the downloads take a while and Repl’s resources on free accounts are limited.

If you run the code now, you should see output similar to that shown in the image below (you may need to hit refresh in the right pane). You can see text from the first article in the top-right pane now, and the text for the second article is further down the page. You’ll notice that out text extraction algorithm isn’t perfect and there’s still some extra text about “Share this” at the top that isn’t actually part of the article, but this is good enough for us to create word clouds from later.


eugenedorfling / wordcloud-lite-4
Stop
Share
Upgrade
Files
main.py
Packager files
poetry.lock
pyproject.toml
https://wordcloud-lite-4.eugenedorfling.repl.co

```
1   import feedparser
2   import requests
3
4   from flask import Flask
5   from bs4 import BeautifulSoup
6
7   app = Flask(__name__)
8
9   BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
10  LIMIT = 2
11
12  def parse_article(article_url):
13      print("Downloading {}".format(article_url))
14      r = requests.get(article_url)
15      soup = BeautifulSoup(r.text, "html.parser")
16      ps = soup.find_all('p')
17      text = "\n".join(p.get_text() for p in ps)
18      return text
19
20  @app.route("/")
21  def home():
22      feed = feedparser.parse(BBC_FEED)
23      article_texts = []
24
25      for article in feed['entries'][:LIMIT]:
26          text = parse_article(article['link'])
27          article_texts.append(text)
28      return str(article_texts)
29
30  if __name__ == '__main__':
31      app.run('0.0.0.0')
32
```

```
['playLesbos: Child migrants sleep in pens at Mytilene port\nplayLesbos: Tear
gas fired as migrants hold protest over conditions\nGreece migrant crisis:
Islanders strike over crowded camps\nplayEurope\'s migrant crisis: Is Greece
still a gateway to Europe?\nUS official claims pressure to alter intelligence\nA
whistleblower says the order over analysis about Russia was because it "made
the president look bad".\nUK parties clash over plans to boost testing\nplayThe
mind-boggling scale of California\'s wildfires. VideoThe mind-boggling scale
of California\'s wildfires\nSmoke from California wildfires turns skies
orange\nTerminally ill 83-year-old gets dying wish. VideoTerminally ill 83-
year-old gets dying wish\nMuseum sorry for barring visitor in low-cut
dress\nThe end of KUWTK: How do fans feel?\nAmericast: Trump\'s alleged
military disservice\n\'We need to talk about suicide\'\nThe Walking Dead to end
in 2022 after 11 seasons\nThe black victims honoured in Naomi Osaka\'s
masks\nThe mums asking suicidal teens to \'please stay\' VideoThe mums
asking suicidal teens to \'please stay\'\nWestJet cancels flight because of mask
```

```
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a pr
oduction deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Downloading https://www.bbc.co.uk/news/world-europe-54094683
Downloading https://www.bbc.co.uk/news/science-environment-5409
1048
172.18.0.1 - - [10/Sep/2020 12:47:37] "GET / HTTP/1.1" 200 -
```

Image: 7

Returning HTML instead of plain text to the user

Although Flask allows us to return Python `str` objects directly to our visitors, the raw result is ugly compared to how people are used to seeing web pages. To take advantage of HTML formatting and

CSS styling, it's better to define HTML *templates*, and use Flask's template engine, `jinja`, to inject dynamic content into these. Before we get to creating image files from our text content, let's set up a basic Flask template.

To use Flask's templates, we need to set up a specific file structure. Press the "new folder" button (next to the "new file" button, on the left pane), and name the resulting new folder `templates`. This is a special name recognised by Flask, so make sure you get the spelling exactly correct.

Select the new folder and press the "new file" button to create a new file inside our `templates` folder. Call the file `home.html`. Note below how the `home.html` file is indented one level, showing that it is inside the folder. If yours is not, drag and drop it into the `templates` folder so that Flask can find it.

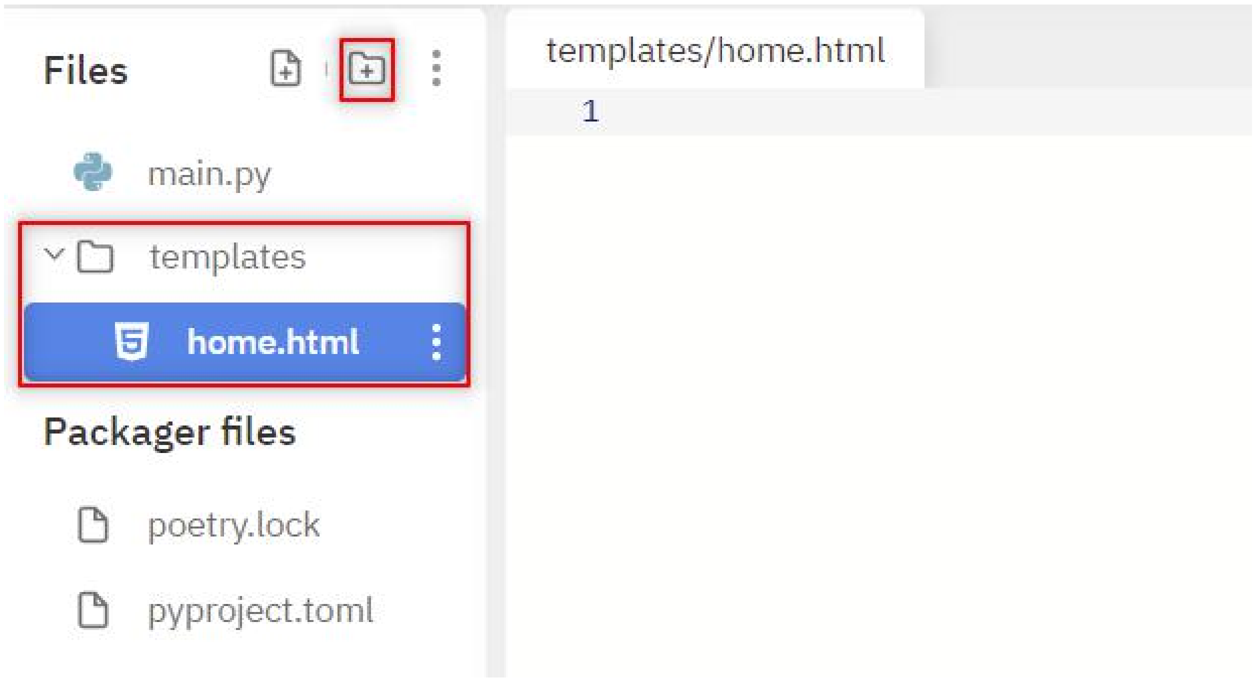


Image: 8

In the `home.html` file, add the following code, which is a mix between standard HTML and Jinja's templating syntax to mix dynamic content into the HTML.

```
<html>
    <body>
        <h1>News Word Clouds</h1>
        <p>Too busy to click on each news article to see what it's about? Below you \
can see all the articles from the BBC front page, displayed as word clouds. If you w\
ant to read more about any particular article, just click on the wordcloud to go to \
the original article</p>
        {% for article in articles %}
            <p>{{article}}</p>
        {% endfor %}
    </body>
</html>
```

Jinja uses the specials characters `{%` and `{{` (in opening and closing pairs) to show where dynamic content (e.g. variables calculated in our Python code) should be added and to define control structures. Here we loop through a list of `articles` and display each one in a set of `<p>` tags.

We'll also need to tweak our Python code a bit to account for the template. In the `main.py` file, make the following changes.

- Add a new import near the top of the file, below the existing Flask import

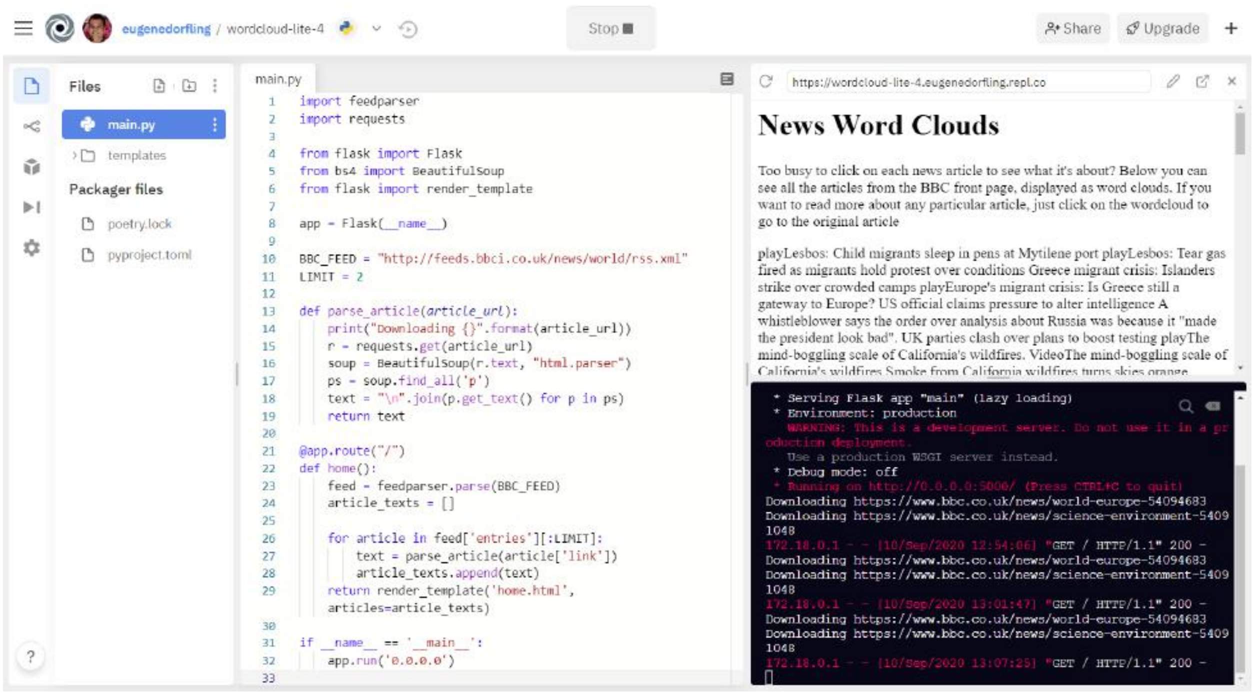
```
from flask import render_template
```

- Update the last line of the `home()` function to make a call to `render_template` instead of returning a `str` directly as follows.

```
@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    article_texts = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        article_texts.append(text)
    return render_template('home.html', articles=article_texts)
```

The `render_template` call tells Flask to prepare some HTML to return to the user by combining data from our Python code and the content in our `home.html` template. Here we pass `article_texts` to the renderer as `articles`, which matches the `articles` variable we loop through in `home.html`.

If everything went well, you should see different output now, which contains our header from the HTML and static first paragraph, followed by two paragraphs showing the same article content that we pulled before. If you don't see the updated webpage, you may need to hit refresh in the right pane again.


eugenedorfling / wordcloud-lite-4
Stop
Share
Upgrade
Files
main.py
templates
Packager files
poetry.lock
pyproject.toml
https://wordcloud-lite-4.eugenedorfling.repl.co

```python
import feedparser
import requests

from flask import Flask
from bs4 import BeautifulSoup
from flask import render_template

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
LIMIT = 2

def parse_article(article_url):
    print("Downloading {}".format(article_url))
    r = requests.get(article_url)
    soup = BeautifulSoup(r.text, "html.parser")
    ps = soup.find_all('p')
    text = "\n".join(p.get_text() for p in ps)
    return text

@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    article_texts = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        article_texts.append(text)
    return render_template('home.html',
    articles=article_texts)

if __name__ == '__main__':
    app.run('0.0.0.0')
```

News Word Clouds

Too busy to click on each news article to see what it's about? Below you can see all the articles from the BBC front page, displayed as word clouds. If you want to read more about any particular article, just click on the wordcloud to go to the original article

playLesbos: Child migrants sleep in pens at Mytilene port playLesbos: Tear gas fired as migrants hold protest over conditions Greece migrant crisis: Islanders strike over crowded camps playEurope's migrant crisis: Is Greece still a gateway to Europe? US official claims pressure to alter intelligence A whistleblower says the order over analysis about Russia was because it "made the president look bad". UK parties clash over plans to boost testing playThe mind-boggling scale of California's wildfires. VideoThe mind-boggling scale of California's wildfires Smoke from California wildfires turns skies orange

```
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a pr
oduction deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Downloading https://www.bbc.co.uk/news/world-europe-54094683
Downloading https://www.bbc.co.uk/news/science-environment-5409
1048
172.18.0.1 - - [10/Sep/2020 12:54:06] "GET / HTTP/1.1" 200 -
Downloading https://www.bbc.co.uk/news/world-europe-54094683
Downloading https://www.bbc.co.uk/news/science-environment-5409
1048
172.18.0.1 - - [10/Sep/2020 13:01:47] "GET / HTTP/1.1" 200 -
Downloading https://www.bbc.co.uk/news/world-europe-54094683
Downloading https://www.bbc.co.uk/news/science-environment-5409
1048
172.18.0.1 - - [10/Sep/2020 13:07:25] "GET / HTTP/1.1" 200 -
```

Image: 9

Now it's time to move on to generating the actual word clouds.

Generating word clouds from text in Python

Once again, there's a nifty Python library that can help us. This one will take in text and return word clouds as images and is called `wordcloud`.

Images are usually served as files living on your server or from an image host like imgur[109]. Because we'll be creating small, short-lived images dynamically from the text, we'll simply keep them in memory instead of saving them anywhere permanently. In order to do this, we'll have to mess around a bit with the Python `io` and `base64` libraries, alongside our newly installed `wordcloud` library.

To import all the new libraries, we'll be using to process images, modify the top of our `main.py` to look as follows.

[109]https://imgur.com/

```python
import base64
import feedparser
import io
import requests

from bs4 import BeautifulSoup
from wordcloud import WordCloud
from flask import Flask
from flask import render_template
```

We'll be converting the text from each article into a separate word cloud, so it'll be useful to have another helper function that can take text as input and produce the word cloud as output. We can use base64[110] to represent the images, which can then be displayed directly in our visitors' web browsers.

Add the following function to the `main.py` file.

```python
def get_wordcloud(text):
    pil_img = WordCloud().generate(text=text).to_image()
    img = io.BytesIO()
    pil_img.save(img, "PNG")
    img.seek(0)
    img_b64 = base64.b64encode(img.getvalue()).decode()
    return img_b64
```

This is probably the hardest part of our project in terms of readability. Normally, we'd generate the word cloud using the `wordcloud` library and then save the resulting image to a file. However, because we don't want to use our file system here, we'll create a `BytesIO` Python object in memory instead and save the image directly to that. We'll convert the resulting bytes to base64 in order to finally return them as part of our HTML response and show the image to our visitors.

In order to use this function, we'll have to make some small tweaks to the rest of our code.

For our template, in the `home.html` file, change the for loop to read as follows.

```
{% for article in articles %}
    <img src="data:image/png;base64,{{article}}">
{% endfor %}
```

Now instead of displaying our article in `<p>` tags, we'll put it inside an `<img/>` tag so that it can be displayed as an image. We also specify that it is formatted as a png and encoded as base64.

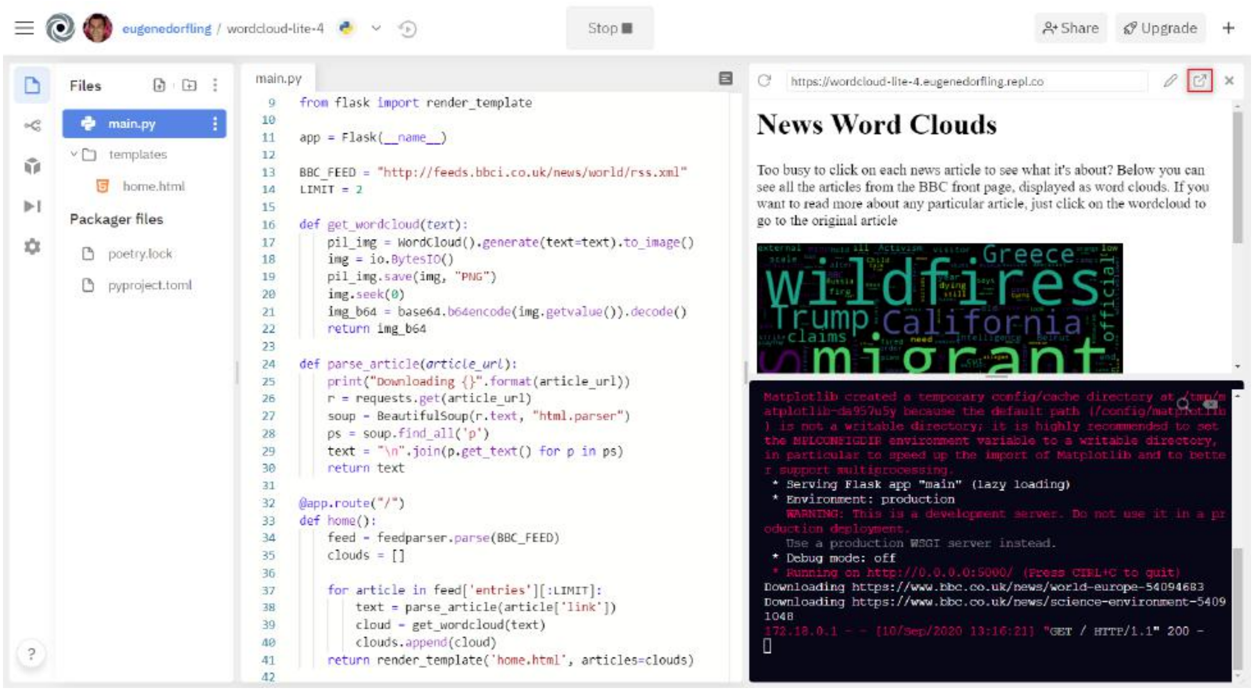
The last thing we need to do is modify our `home()` function to call the new `get_wordcloud()` function and to build and render an array of images instead of an array of text. Change the `home()` function to look as follows.

[110]https://en.wikipedia.org/wiki/Base64

```
 1 @app.route("/")
 2 def home():
 3     feed = feedparser.parse(BBC_FEED)
 4     clouds = []
 5
 6     for article in feed['entries'][:LIMIT]:
 7         text = parse_article(article['link'])
 8         cloud = get_wordcloud(text)
 9         clouds.append(cloud)
10     return render_template('home.html', articles=clouds)
```

We made changes on lines 4, 8, 9, and 10, to change to a `clouds` array, populate that with images from our `get_wordcloud()` function, and return that in our `render_template` call.

If you restart the Repl and refresh the page, you should see something similar to the following. We can see the same content from the articles, however, we can now see the important keywords without having to read the entire article.



```
from flask import render_template

app = Flask(__name__)

BBC_FEED = "http://feeds.bbci.co.uk/news/world/rss.xml"
LIMIT = 2

def get_wordcloud(text):
    pil_img = WordCloud().generate(text=text).to_image()
    img = io.BytesIO()
    pil_img.save(img, "PNG")
    img.seek(0)
    img_b64 = base64.b64encode(img.getvalue()).decode()
    return img_b64

def parse_article(article_url):
    print("Downloading {}".format(article_url))
    r = requests.get(article_url)
    soup = BeautifulSoup(r.text, "html.parser")
    ps = soup.find_all('p')
    text = "\n".join(p.get_text() for p in ps)
    return text

@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    clouds = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        cloud = get_wordcloud(text)
        clouds.append(cloud)
    return render_template('home.html', articles=clouds)
```

News Word Clouds

Too busy to click on each news article to see what it's about? Below you can see all the articles from the BBC front page, displayed as word clouds. If you want to read more about any particular article, just click on the wordcloud to go to the original article

```
Matplotlib created a temporary config/cache directory at /tmp/m
atplotlib-da957u5y because the default path (/config/matplotlib
) is not a writable directory; it is highly recommended to set
the MPLCONFIGDIR environment variable to a writable directory,
in particular to speed up the import of Matplotlib and to bette
r support multiprocessing.
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a pr
oduction deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Downloading https://www.bbc.co.uk/news/world-europe-54094683
Downloading https://www.bbc.co.uk/news/science-environment-5409
1048
172.18.0.1 - - [10/Sep/2020 13:16:21] "GET / HTTP/1.1" 200 -
```

Image:10

For a larger view, you can pop out the website in a new browser tab using the button in the top right of the Repl editor (indicated in red above).

The last thing we need to do is add some styling to make the page look a bit prettier and link the images to the original articles.

Adding some finishing touches

Our text looks a bit stark, and our images touch each other, which makes it hard to see that they are separate images. We'll fix that up by adding a few lines of CSS and importing the Bootstrap framework.

Adding CSS

Edit the `home.html` file to look as follows

```
1  <html>
2    <head>
3      <title>News in WordClouds | Home</title>
4      <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/\
5  css/bootstrap.min.css" integrity="sha384-HSMxcRTRxnN+Bdg0JdbxYKrThecOKuH5zCYotlSAcp1\
6  +c8xmyTe9GYg1l9a69psu" crossorigin="anonymous">
7  
8      <style type="text/css">
9        body {padding: 20px;}
10       img{padding: 5px;}
11     </style>
12   </head>
13 
14   <body>
15     <h1>News Word Clouds</h1>
16       <p>Too busy to click on each news article to see what it's about? Below you ca\
17 n see all the articles from the BBC front page, displayed as word clouds. If you wan\
18 t to read more about any particular article, just click on the wordcloud to go to th\
19 e original article</p>
20       {% for article in articles %}
21         <a href="{{article.url}}"><img src="data:image/png;base64,{{article.image}}"\
22 ></a>
23       {% endfor %}
24   </body>
25 </html>
```

On **line 3** we add a title, which is displayed in the browser tab. On **line 4**, we import Bootstrap[111], which has some nice CSS defaults right out the box (it's probably a bit heavy-weight for our project as we have so little content and won't use most of Bootstrap's features, but it's nice to have if you're planning on extending the project.)

[111] https://getbootstrap.com/

On **lines 6-8**, we add padding to the main body to stop the text going to close to the edges of the screen, and also add padding to our images to stop them touching each other.

On **line 16**, we use an `<a>` tag to add a link to our image. We also change the Jinja templates to `{{article.url}}` and `{{article.image}}` so that we can have images that link back to the original news article.

Now we need to tweak our backend code again to pass through the URL and image for each article, as the template currently doesn't have access to the URL.

Passing through the URLs

To easily keep track of pairs of URLs and images, we'll add a basic Python helper class called `Article`. In the `main.py` file, add the following code before the function definitions.

```python
class Article:
    def __init__(self, url, image):
        self.url = url
        self.image = image
```

This is a simple class with two attributes: url and image. We'll store the original URL from the RSS feed in `url` and the final base64 wordcloud in `image`.

To use this class, modify the `home()` function to look as follows.

```python
@app.route("/")
def home():
    feed = feedparser.parse(BBC_FEED)
    articles = []

    for article in feed['entries'][:LIMIT]:
        text = parse_article(article['link'])
        cloud = get_wordcloud(text)
        articles.append(Article(article['link'], cloud))
    return render_template('home.html', articles=articles)
```

We changed the name of our `clouds` list to `articles`, and populated it by initialising `Article` objects in the for loop and appending them to this list. We then pass across `articles=articles` instead of `articles=clouds` in the return statement so that the template can access our list of `Article` objects, which each contain the image and the URL of each article.

If you refresh the page again and expand the window using the pop out button, you'll be able to click any of the images to go to the original article, allowing readers to view a brief summary of the day's news or to read more details about any stories that catch their eye.

Where next?

We've included several features in our web application, and looked at how to use RSS feeds and process and serve images directly in Python, but there are a lot more features we could add. For example:

- Our application only shows two stories at a time as the download time is slow. We could instead look at implementing a threaded solution to downloading web pages so that we could process several articles in parallel. Alternatively (or in addition), we could also download the articles on a schedule and cache the resulting images so that we don't have to do the resource heavy downloading and parsing each time a visitor visits our site.
- Our web application only shows articles from a single source (BBC), and only from today. We could add some more functionality to show articles from different sources and different time frames. We could also consider allowing the viewer to choose which category of articles to view (news, sport, politics, finance, etc) by using different RSS feeds as sources.
- Our design and layout is very basic. We could make our site look better and be more responsive by adding more CSS. We could lay out the images in a grid of rows and columns to make it look better on smaller screens such as mobile phones.

If you'd like to keep working on the web application, simply head over to the Repl[112] and fork it to continue your own version.

In the next chapter, we'll be looking at how to build our own Discord Chatbot.

[112]https://repl.it/@GarethDwyer1/news-to-wordcloud

Building a Discord Bot with Python and Repl.it

In this tutorial, we'll use Repl.it[113] and Python to build a Discord Chatbot. If you're reading this tutorial, you probably have at least heard of Discord and likely have an existing account. If not, Discord is a VoIP and Chat application that is designed to replace Skype for gamers. The bot we create in this tutorial will be able to join a Discord server and respond to messages sent by people.

If you prefer JavaScript, the next chapter is the same tutorial using NodeJS instead of Python.

You'll find it easier to follow along if you have some Python knowledge and have used Discord or a similar app such as Skype or Telegram before. We won't be covering the very basics of Python, but we will explain each line of code in detail, so if you have any experience with programming, you should be able to follow along.

Overview and requirements

We'll be doing all of our coding through the Repl.it web IDE and hosting our bot with Repl.it as well, so you won't need to install any additional software on your machine. For this tutorial you will need to create a Discord[114] account (if you already have one, you can skip this). There are instructions for how to do this in the next section.

In this tutorial, we will be covering:

- Creating an application and a bot user in your Discord account
- Creating a server on Discord
- Adding our bot to our Discord server

Let's get through these admin steps first and then we can get to the fun part of coding our bot.

Creating a bot in Discord and getting a token

You can sign up for a free account over at the Discord register page[115], and can download one of their desktop or mobile applications from the Discord homepage[116]. You can also use Discord in the browser.

Once you have an account, you'll want to create a Discord application. Visit the Discord developer's page[117] and press the "New application" button, as in the image below.

[113]https://repl.it
[114]https://discordapp.com/
[115]https://discordapp.com/register
[116]https://discordapp.com/
[117]https://discordapp.com/developers/applications/

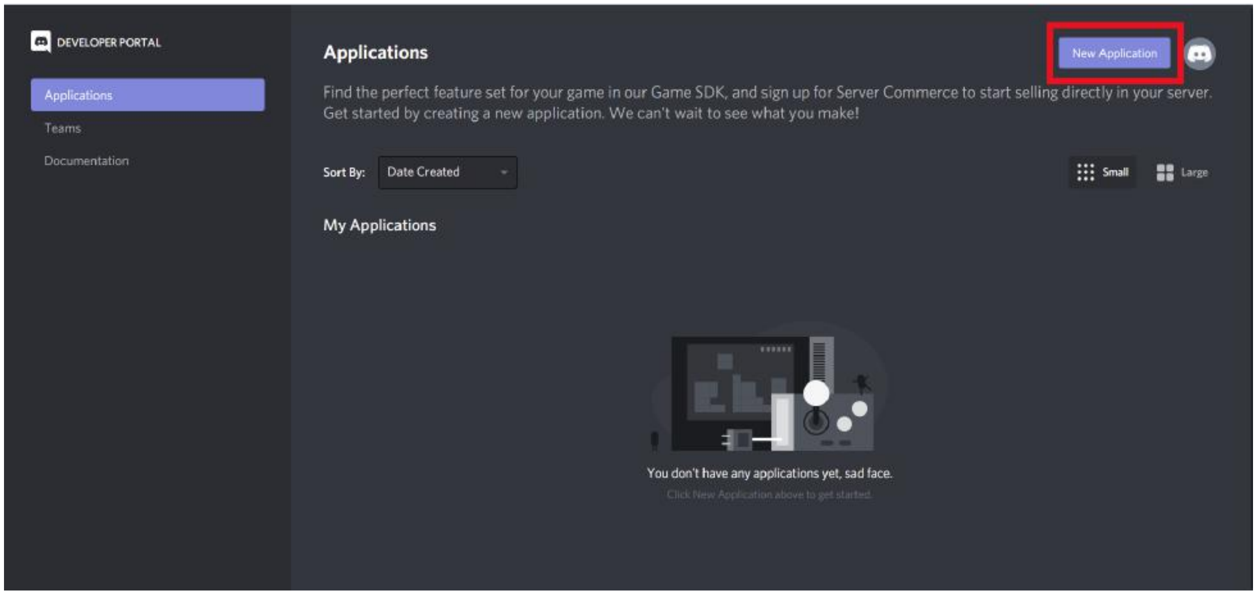

DEVELOPER PORTAL
Applications
Teams
Documentation
Applications
New Application
Find the perfect feature set for your game in our Game SDK, and sign up for Server Commerce to start selling directly in your server. Get started by creating a new application. We can't wait to see what you make!
Sort By: Date Created
Small
Large
My Applications
You don't have any applications yet, sad face.
Click New Application above to get started.

Image: 1 ***Creating a new Discord application***

Fill out a name for your bot and select "Create".

The first thing to do on the next page is to note your Client ID, which you'll need to add the bot to the server. You can come back later and get it from this page, or copy it somewhere where you can easily find it later.

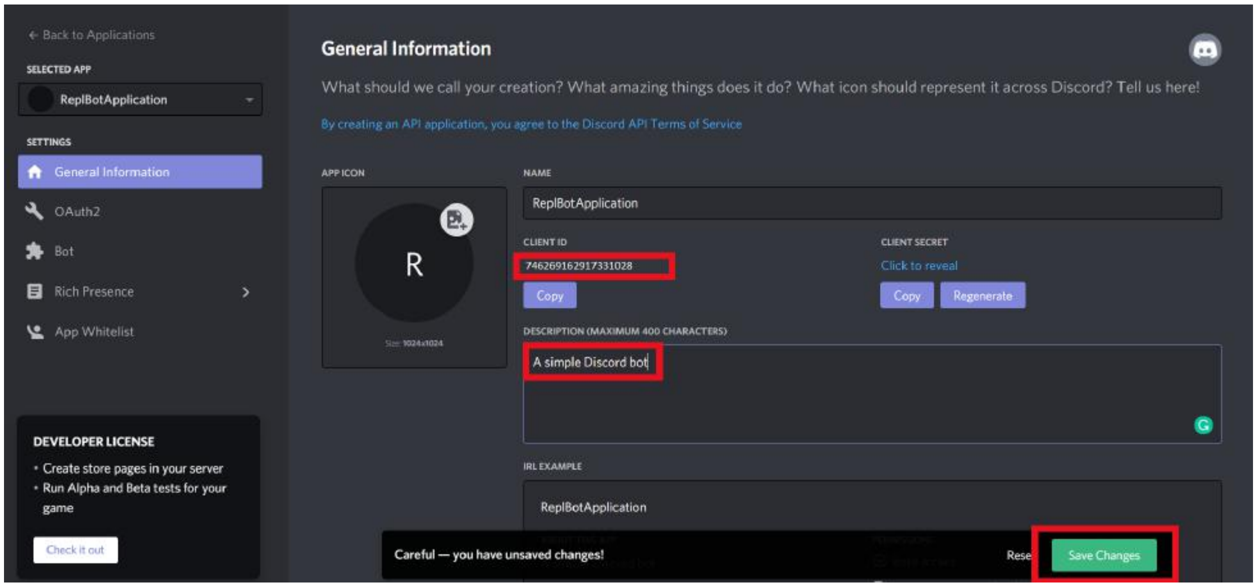
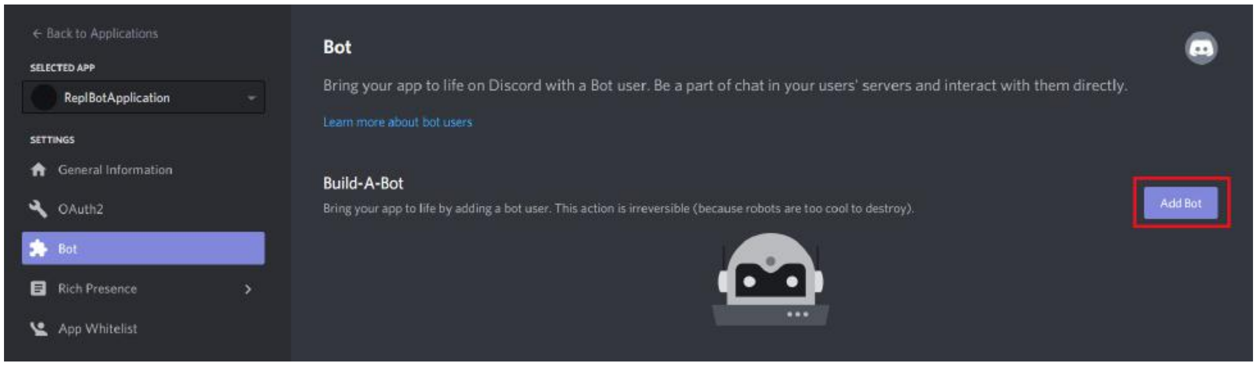

Back to Applications
SELECTED APP
ReplBotApplication
SETTINGS
General Information
OAuth2
Bot
Rich Presence
App Whitelist
DEVELOPER LICENSE
Create store pages in your server
Run Alpha and Beta tests for your game
Check it out
General Information
What should we call your creation? What amazing things does it do? What icon should represent it across Discord? Tell us here!
By creating an API application, you agree to the Discord API Terms of Service
APP ICON
R
Size: 1024x1024
NAME
ReplBotApplication
CLIENT ID
746269162917331028
Copy
CLIENT SECRET
Click to reveal
Copy
Regenerate
DESCRIPTION (MAXIMUM 400 CHARACTERS)
A simple Discord bot
IRL EXAMPLE
ReplBotApplication
Careful — you have unsaved changes!
Reset
Save Changes

Image: 2 ***Record your Client ID***

You can also rename the application and provide a description for your bot at this point and press "Save Changes".

You have now created a Discord application. The next step is to add a bot to this application, so head over to the "Bot" tab using the menu on the left and press the "Add Bot" button, as indicated below. Click "Yes, do it" when Discord asks if you're sure about bringing a new bot to life.

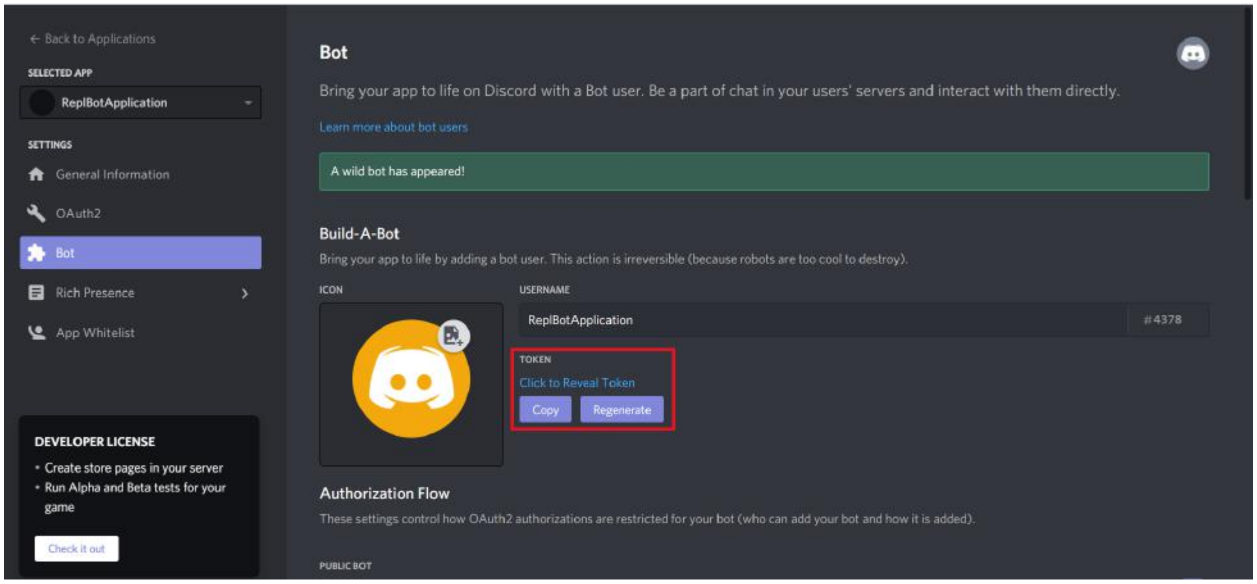


Bring your app to life on Discord with a Bot user. Be a part of chat in your users' servers and interact with them directly.

Bring your app to life by adding a bot user. This action is irreversible (because robots are too cool to destroy).

Imgage: 3 ***Adding a bot to our Discord Application***

The last thing we'll need from our bot is a Token. Anyone who has the bot's token can prove that they own the bot, so you'll need to be careful not to share this with anyone. You can get the token by pressing "Click to Reveal Token", or copy it to your clipboard without seeing it by pressing "Copy".



Bring your app to life on Discord with a Bot user. Be a part of chat in your users' servers and interact with them directly.

A wild bot has appeared!

Bring your app to life by adding a bot user. This action is irreversible (because robots are too cool to destroy).

- Create store pages in your server
- Run Alpha and Beta tests for your game

These settings control how OAuth2 authorizations are restricted for your bot (who can add your bot and how it is added).

Image: 4 ***Generating a token for our Discord bot***

Take note of your token or copy it to your clipboard, as we'll need to add it to our code soon.

Creating a Discord server

If you don't have a Discord server to add your bot to, you can create one by either opening the desktop Discord application that you downloaded earlier or returning to the Discord home page in your browser. Press the "+" icon indicated by the exclamation mark, as shown below, to create a server.

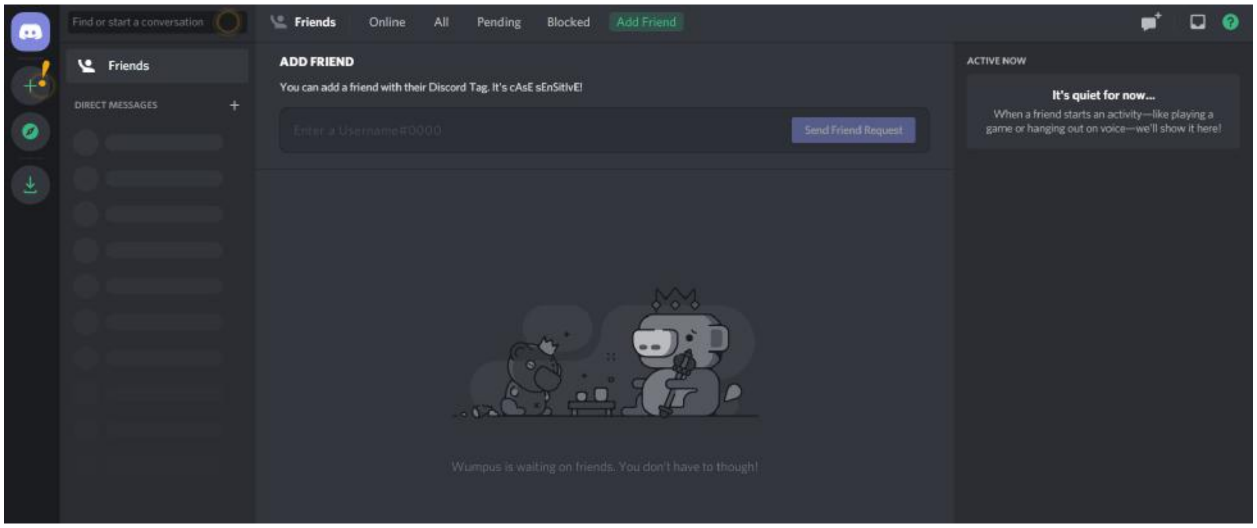

Find or start a conversation
Friends
Online
All
Pending
Blocked
Add Friend
DIRECT MESSAGES
ADD FRIEND
You can add a friend with their Discord Tag. It's cAsE sEnSitIvE!
Enter a Username#0000
Send Friend Request
ACTIVE NOW
It's quiet for now...
When a friend starts an activity—like playing a game or hanging out on voice—we'll show it here!
Wumpus is waiting on friends. You don't have to though!

Image: 5 ***Creating a Discord server***

Press "Create a server" in the screen that follows, and then give your server a name. Once the server is up and running, you can chat with yourself, or invite some friends to chat with you. Soon we'll invite our bot to chat with us as well.

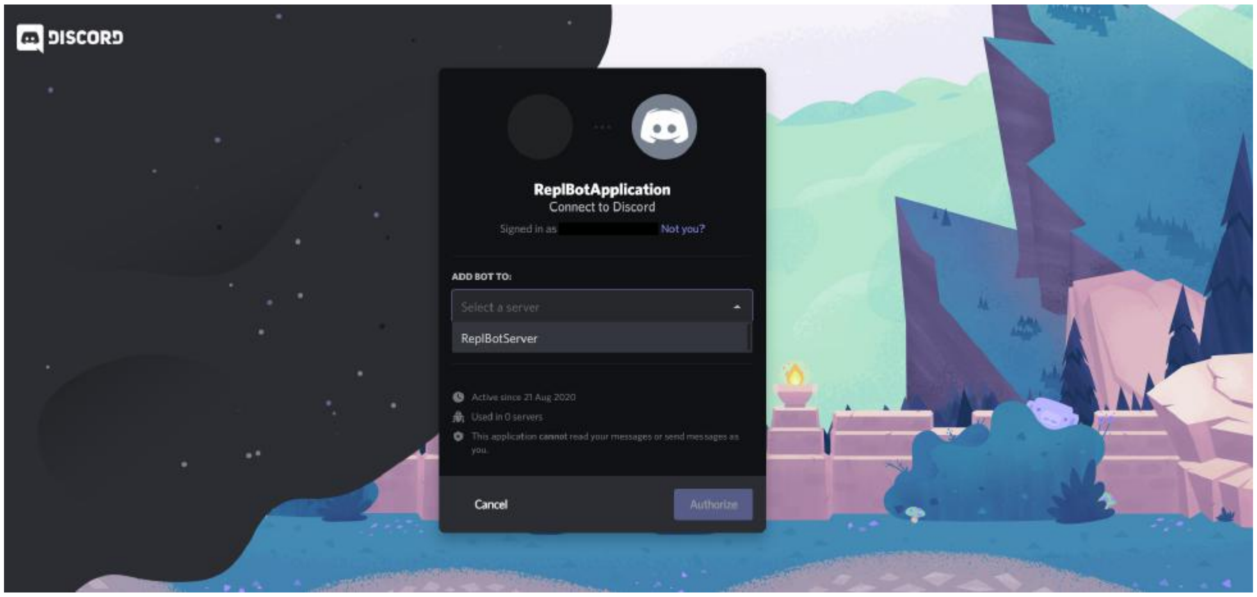
Adding your Discord bot to your Discord server

Our Discord bot is still just a shell at this stage as we haven't written any code to allow him to do anything, but let's go ahead and add him to our Discord server anyway. To add a bot to your server, you'll need the Client ID from the "General Information" page that we looked at before when we created our ReplBotApplication (ie. the client ID, not the secret bot Token).

Create a URL that looks as follows, but using your Client ID instead of mine at the end:

https://discordapp.com/api/oauth2/authorize?scope=bot&client_id=746269162917331028

Visit the URL that you created in your web browser and you'll see a page similar to the following where you can choose which server to add your bot to.



Active since 21 Aug 2020

Used in 0 servers

This application **cannot** read your messages or send messages as you.

Image: 6 ***Authorizing our bot to join our server***

Select the server we created in the step before this and hit the "authorize" button. After completing the captcha, you should get an in-app Discord notification telling you that your bot has joined your server.

Now we can get to the fun part of building a brain for our bot!

Creating a Repl and installing our Discord dependencies

The first thing we need to do is create a Python Repl to write the code for our Discord bot. Over at repl.it[118], create a new Repl, choosing "Python" as your language.

We don't need to reinvent the wheel, as there is already a great Python wrapper for the Discord bot API over on GitHub[119], which makes it a lot faster to get set up with a basic Python discord bot. To use library, we can simply write `import discord` at the top of `main.py`. Repl.it will handle installing this dependency when you press the "run" button.

Our bot is nearly ready to go – but we still need to plug in our secret token. This will authorize our code to control our bot.

Setting up authorization for our bot

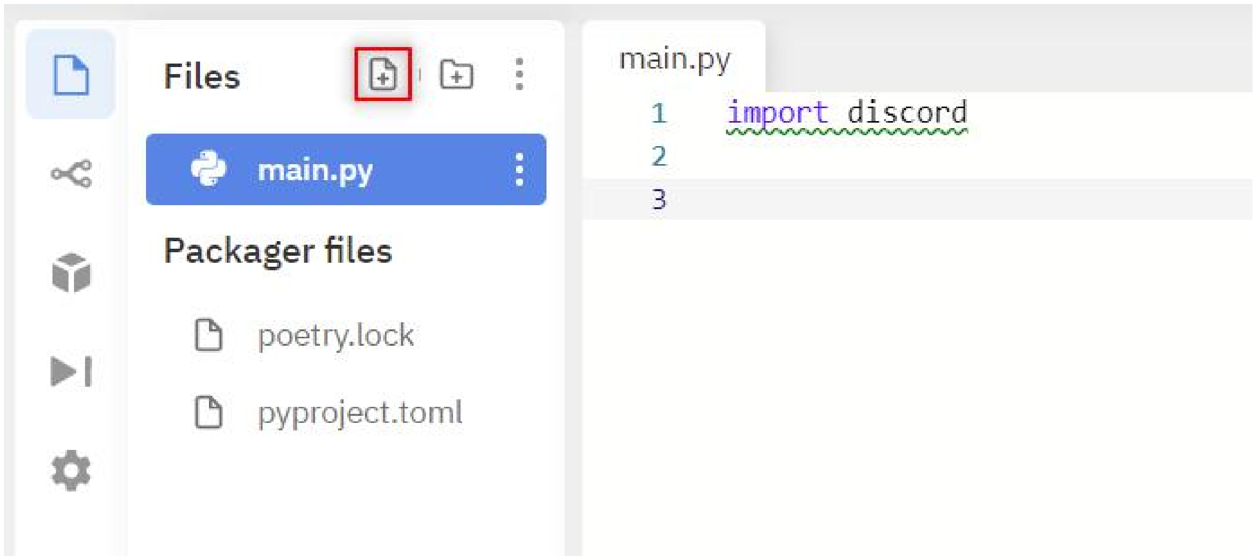
By default, Repl.it code is public. This is great as it encourages collaboration and learning, but we need to be careful not to share our secret bot token (which gives anyone who has access to it full

[118]https://repl.it
[119]https://github.com/Rapptz/discord.py

control of our bot).

To get around the problem of needing to give our *code* access to the token while allowing others to access our code but *not* our token, we'll be using environment variables[120]. On a normal machine, we'd set these directly on our operating system, but using Repl.it we don't have access to this. Repl.it allows us to set secret environment variables through a special `.env` file.

First, we need to create a new file called exactly `.env`. Select "Add file" in the left pane, as shown in the image below, and name this file `.env`. It is important not to leave out the `.` at the beginning.



```
import discord
```

Image: 7 ***Create a new file called .env***

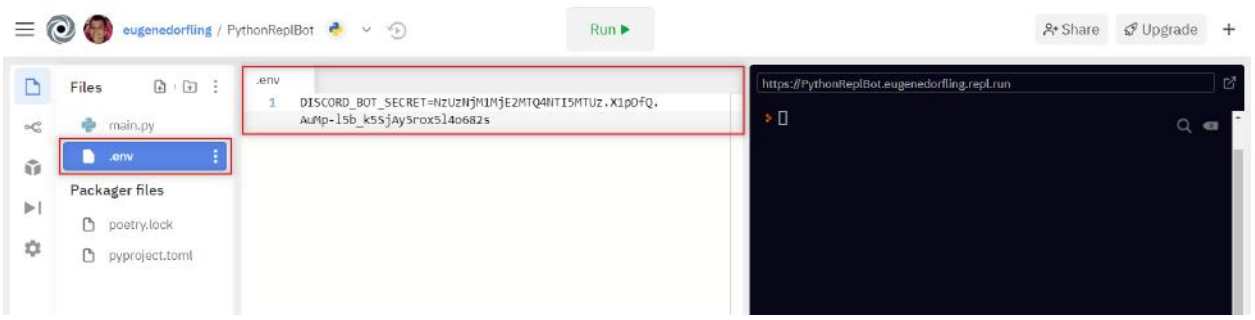
Open this new file and add a variable to define your bot's secret token (note that this is the second token that we got while setting up the bot – different from the Client ID that we used to add our bot to our server). It should look something like:

```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

You'll need to:

- **Replace** the token (after the = sign) with the token that Discord gave you when creating your own bot.
- Be careful about **spacing**. Unlike in Python, if you put a space on either side of the = in your `.env` file, these spaces will be part of the variable name or the value, so make sure you don't have any spaces around the = or at the end of the line.
- Run the code again. Sometimes you'll need to refresh the whole page to make sure that your environment variables are successfully loaded.

[120] https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps


eugenedorfling / PythonReplBot
Run
Share
Upgrade
Files
main.py
.env
Packager files
poetry.lock
pyproject.toml
https://PythonReplBot.eugenedorfling.repl.run

```
.env
1  DISCORD_BOT_SECRET=NzUzNjM1MjE2MTQ4NTI5MTUz.X1pDfQ.
   AuMp-l5b_k5SjAy5rox5l4o682s
```

Image: 8 ***Creating our .env file***

Let's make a Discord bot that repeats everything we say but in reverse. We can do this in only a few lines of code. In your `main.py` file, add the following:

```
 1  import discord
 2  import os
 3  
 4  client = discord.Client()
 5  
 6  @client.event
 7  async def on_ready():
 8      print("I'm in")
 9      print(client.user)
10  
11  @client.event
12  async def on_message(message):
13      if message.author != client.user:
14          await message.channel.send(message.content[::-1])
15  
16  token = os.environ.get("DISCORD_BOT_SECRET")
17  client.run(token)
```

Let's tear this apart line by line to see what it does.

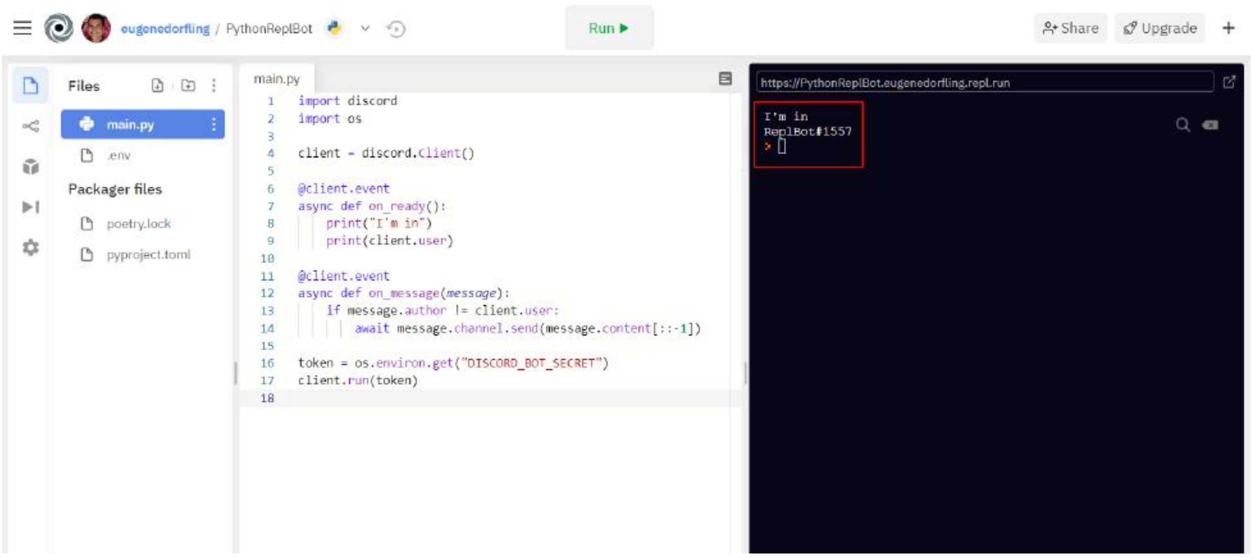
- **Lines 1-2** import the discord library that we installed earlier and the built-in operating system library, which we'll need to access our bot's secret token.
- In **line 4**, we create a Discord `Client`. This is a Python object that we'll use to send various commands to Discord's servers.
- In **line 6**, we say we are defining an `event` for our client. This line is a Python decorator, which will take the function directly below it and modify it in some way. The Discord bot is going to run *asynchronously*, which might be a bit confusing if you're used to running standard Python. We won't go into asynchronous Python in depth here, but if you're interested in what this is and why it's used, there's a good guide over at FreeCodeCamp[121]. In short, instead of running

[121]https://medium.freecodecamp.org/a-guide-to-asynchronous-programming-in-python-with-asyncio-232e2afa44f6

the code in our file from top to bottom, we'll be running pieces of code in response to specific events.
- In **lines 7-9** we define what kind of event we want to respond to, and what the response should be. In this case, we're saying that in response to the `on_ready` event (when our bot joins a server successfully), we should output some information server-side (i.e. this will be displayed in our Repl's output, but not sent as a message through to Discord). We'll print a simple `I'm in` message to see that the bot is there and print our bot's user id (if you're running multiple bots, this will make it easier to work out who's doing what).
- **Lines 11-14** are similar, but instead of responding to an `on_ready` event, we tell our bot how to handle new messages. **Line 13** says we only want to respond to messages that aren't from us (otherwise our bot will keep responding to himself – you can remove this line to see why that's a problem), and **line 14** says we'll send a new message to the same channel where we received a message (`message.channel`) and the content we'll send will be the same message that we received, but backwards (`message.content[::-1] - ::-1` is a slightly odd but useful Python idiom to reverse a string or list).

The last two lines get our secret token from the environment variables that we set up earlier and then tell our bot to start up.

Press the big green "Run" button again and you should see your bot reporting a successful channel join in the Repl output.



```
1  import discord
2  import os
3
4  client = discord.Client()
5
6  @client.event
7  async def on_ready():
8      print("I'm in")
9      print(client.user)
10
11 @client.event
12 async def on_message(message):
13     if message.author != client.user:
14         await message.channel.send(message.content[::-1])
15
16 token = os.environ.get("DISCORD_BOT_SECRET")
17 client.run(token)
18
```

```
I'm in
ReplBot#1557
> 
```

Image: 9 ***Seeing our bot join our server***

Open Discord, and from within the server we created earlier, select your ReplBotApplication from the pane on the right-hand side of the screen.

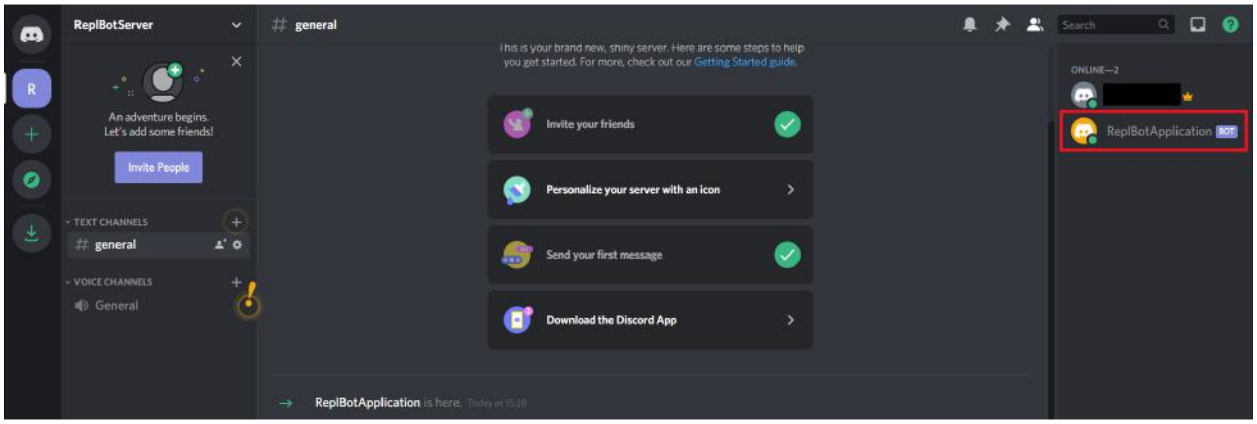


Image:10 ***The Repl bot is active***

.

Once you have selected this, you will be able to send a message (by typing into the box highlighted below) and see your bot respond!

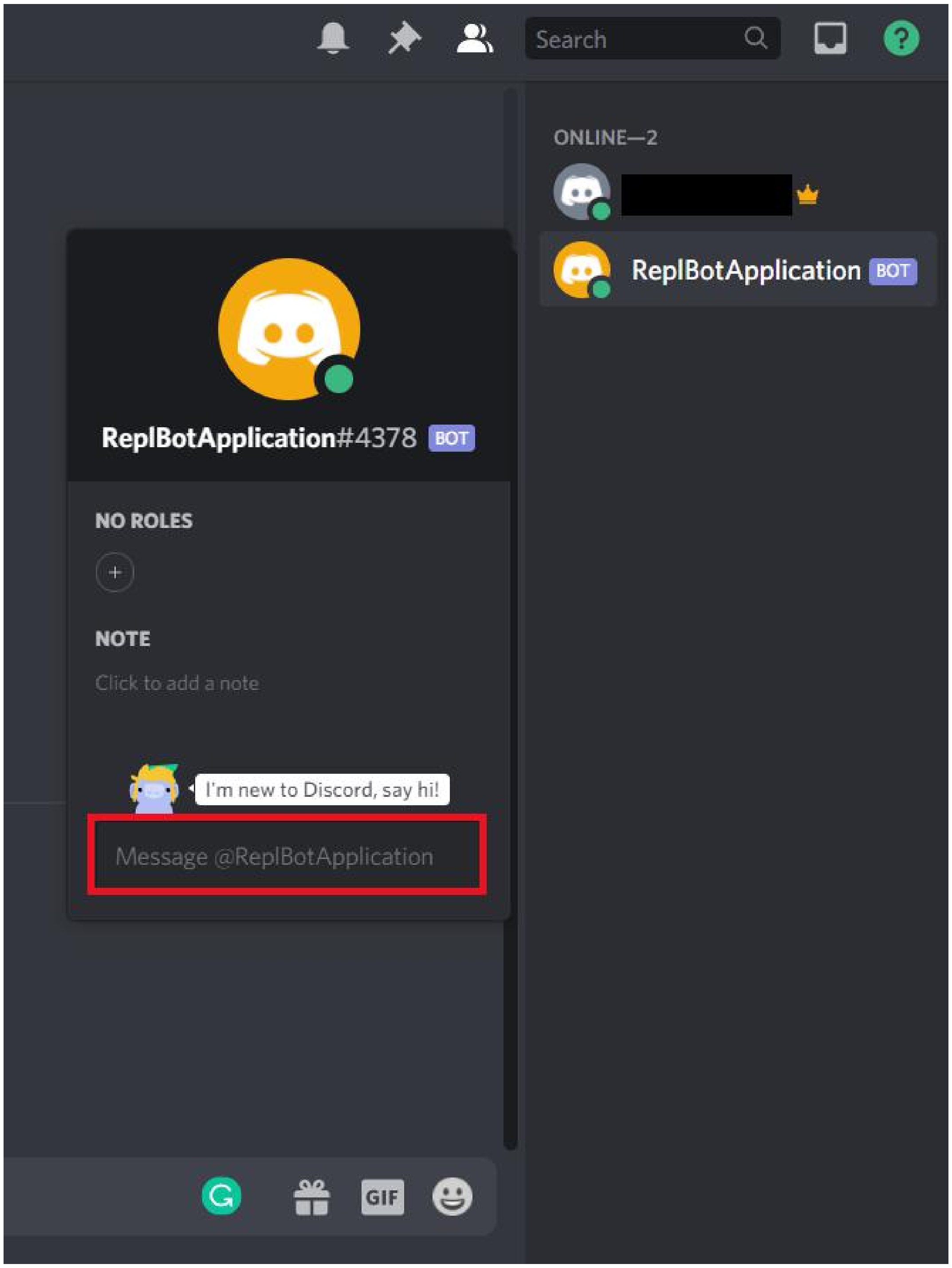
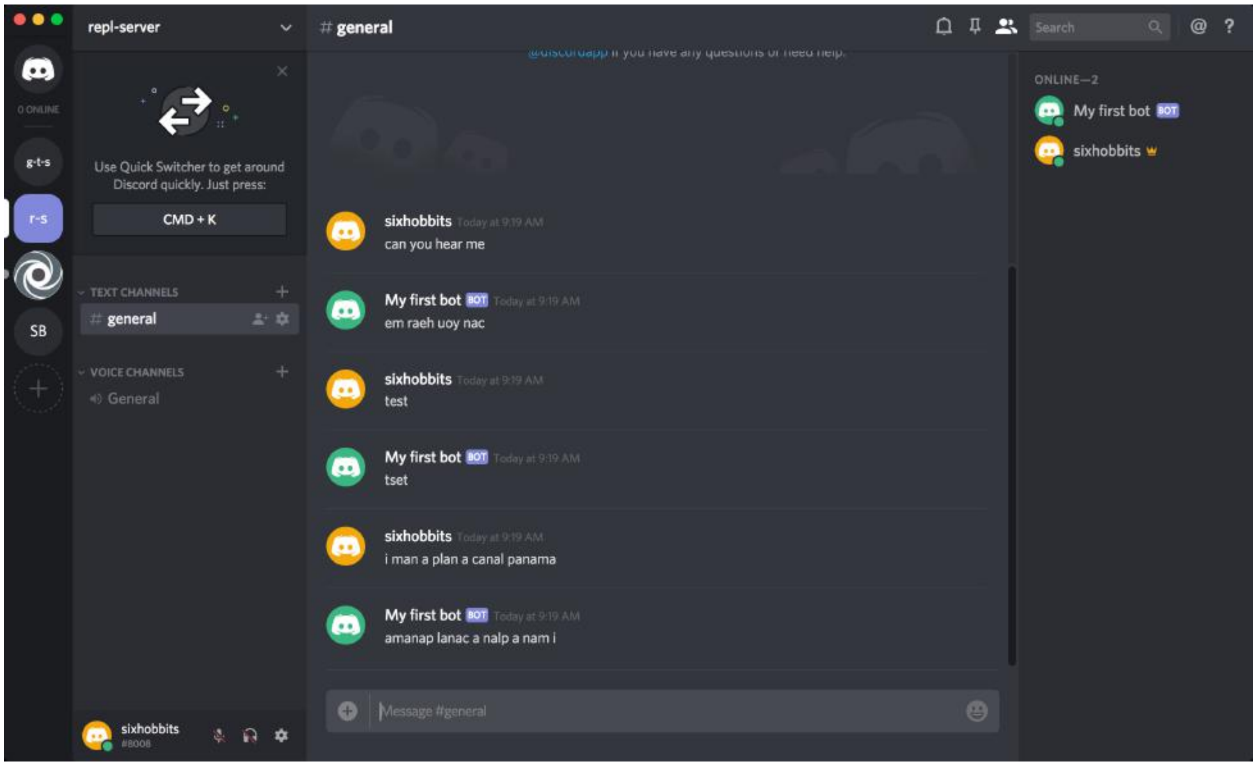

Search
ONLINE—2
[redacted]
ReplBotApplication BOT
ReplBotApplication#4378 BOT
NO ROLES
NOTE
Click to add a note
I'm new to Discord, say hi!
Message @ReplBotApplication
GIF

Image:11 ***Send a message to your bot***

.

The bot responds each time, reversing the text we enter.



sixhobbits Today at 9:19 AM
can you hear me

My first bot BOT Today at 9:19 AM
em raeh uoy nac

sixhobbits Today at 9:19 AM
test

My first bot BOT Today at 9:19 AM
tset

sixhobbits Today at 9:19 AM
i man a plan a canal panama

My first bot BOT Today at 9:19 AM
amanap lanac a nalp a nam i

Image:12 ***Our bot can talk!***

.

Keeping our bot alive

Your bot can now respond to messages, but only for as long as your Repl is running. If you close your browser tab or shut down your computer, your bot will stop and no longer respond to messages on Discord.

Repl.it will keep your code running after you close the browser tab only if you are running a web server. Because we are using the Python discord.py library, our bot doesn't require an explicit web server, but we can create a server and run it in a separate thread just to keep our Repl alive. We'll do this using the Flask[122] framework.

Create a new file in your project called `keep_alive.py` and add the following code:

[122]http://flask.pocoo.org/

```python
from flask import Flask
from threading import Thread

app = Flask('')

@app.route('/')
def home():
    return "I'm alive"

def run():
  app.run(host='0.0.0.0',port=8080)

def keep_alive():
    t = Thread(target=run)
    t.start()
```

We won't go over this in detail as it's not central to our bot, but here we start a web server that will return "I'm alive" if anyone visits it, and we'll provide a method to start this in a new thread (leaving the main thread for our Repl bot).

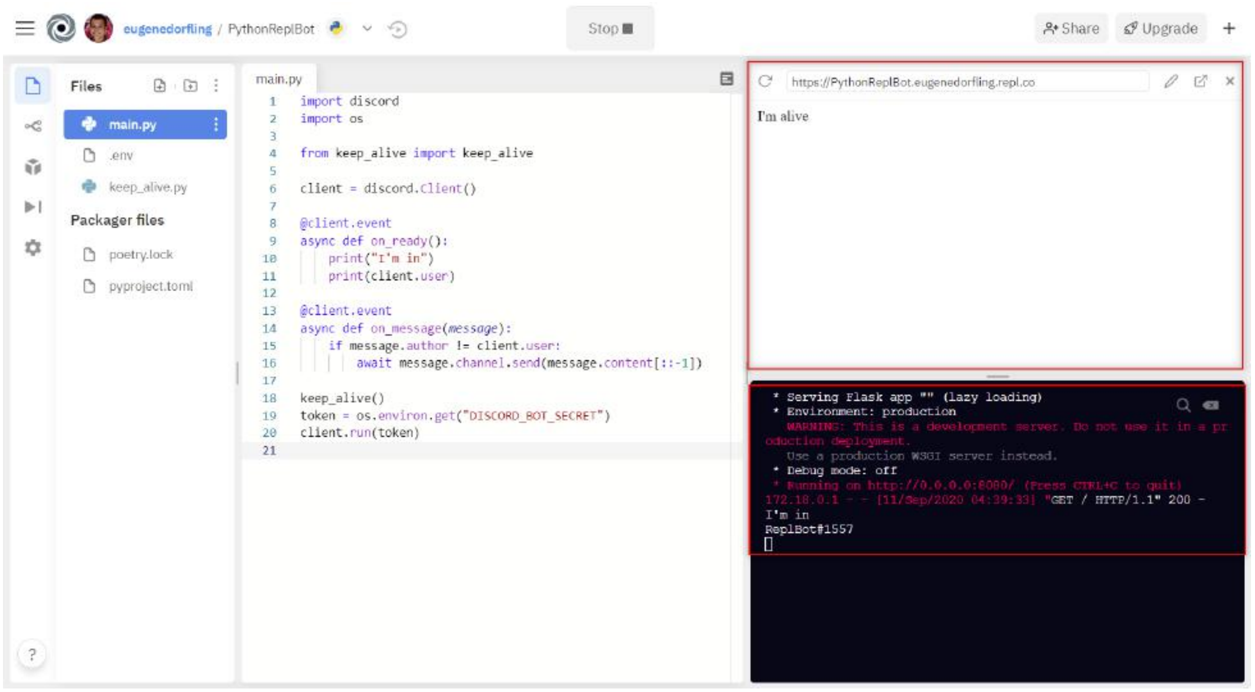
In our `main.py` file, we need to add an import for this server at the top. Add the following line near the top of `main.py`.

```python
from keep_alive import keep_alive
```

In `main.py` we need to start up the web server just before you start up the bot. Add these three lines to `main.py`, just before the line with `token = os.environ.get("DISCORD_BOT_SECRET")`:

```python
keep_alive()
token = os.environ.get("DISCORD_BOT_SECRET")
client.run(token)
```

After doing this and hitting the green "Run" button again, you should see some changes to your Repl. For one, you'll see a new pane in the top right which shows the web output from your server. We can see that visiting our Repl now returns a basic web page showing the "I'm alive" string that we told our web server to return by default. In the bottom-right pane, you can also see some additional output from Flask starting up and running continuously, listening for requests.



```python
import discord
import os

from keep_alive import keep_alive

client = discord.Client()

@client.event
async def on_ready():
    print("I'm in")
    print(client.user)

@client.event
async def on_message(message):
    if message.author != client.user:
        await message.channel.send(message.content[::-1])

keep_alive()
token = os.environ.get("DISCORD_BOT_SECRET")
client.run(token)
```

```
I'm alive
```

```
 * Serving Flask app "" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a pr
oduction deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
172.18.0.1 - - [11/Sep/2020 04:39:33] "GET / HTTP/1.1" 200 -
I'm in
ReplBot#1557
```

Image:13 ***Output from our Flask server***

Now your bot will stay alive even after closing your browser or shutting down your development machine. Repl will still clean up your server and kill your bot after about one hour of inactivity, so if you don't use your bot for a while, you'll have to log into Repl and start the bot up again. Alternatively, you can set up a third-party (free!) service like Uptime Robot[123]. Uptime Robot pings your site every 5 minutes to make sure it's still working – usually to notify you of unexpected downtime, but in this case, the constant pings have the side effect of keeping our Repl alive as it will never go more than an hour without receiving any activity.

Forking and extending our basic bot

This is not a very useful bot as is, but the possibilities are only limited by your creativity now! You can have your bot receive input from a user, process the input, and respond in any way you choose. In fact, with the basic input and output that we've demonstrated, we have most of the components of any modern computer, all of which are based on the Von Neumann architecture[124] (we could easily add the missing memory by having our bot write to a file, or with a bit more effort link in a SQLite database[125] for persistent storage).

If you followed along with this tutorial, you'll have your own basic Repl bot to play around with and extend. If you were simply reading, you can easily fork this bot at https://repl.it/@GarethDwyer1/discord-bot[126] and extend it how you want (you'll need to add your own token and recreate the `.env` file

[123]https://uptimerobot.com/
[124]https://en.wikipedia.org/wiki/Von_Neumann_architecture
[125]https://www.sqlite.org/index.html
[126]https://repl.it/@GarethDwyer1/discord-bot

still). Happy hacking!

If you're stuck for ideas, why not link up your Discord bot to the Twitch API[127] to get notified when your favourite streamers are online, or build a text adventure[128].

In the next chapter, we'll build exactly the same bot again but using NodeJS instead of Python. Even if you prefer Python, it's often a good idea to build the same project in two languages so that you can better appreciate the differences and similarities.

[127]https://dev.twitch.tv/
[128]https://en.wikipedia.org/wiki/Interactive_fiction

Building a Discord bot with Node.js and Repl.it

In this tutorial, we'll use Repl.it[129] and Node.js to build a Discord Chatbot. If you're reading this tutorial, you probably have at least heard of Discord and likely have an existing account. If not, Discord is a VoIP and Chat application that is designed to replace Skype for gamers. The bot we create in this tutorial will be able to join a Discord server and respond to messages sent by people.

If you don't like JavaScript, there's also a Python version of this tutorial in the previous chapter.

You'll find it easier to follow along if you have some JavaScript knowledge and have used Discord or a similar app such as Skype or Telegram before. We won't be covering the very basics of JavaScript, but we will explain each line of code in detail, so if you have any experience with programming, you should be able to follow along.

Overview and requirements

We'll be doing all of our coding through the Repl.it web IDE and hosting our bot with Repl.it as well, so you won't need to install any additional software on your machine. For this tutorial you will need to create a Discord[130] account (if you already have one, you can skip this). There are instructions for how to do this in the next section.

In this tutorial, we will be covering:

- Creating an application and a bot user in your Discord account
- Creating a server on Discord
- Adding our bot to our Discord server

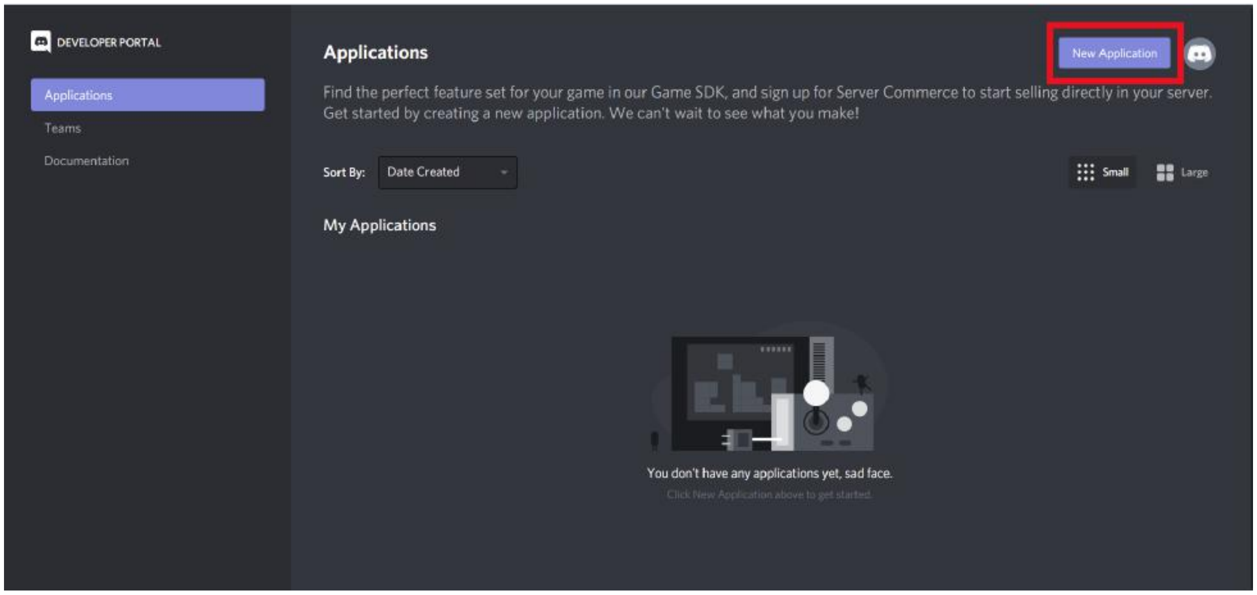
Let's get through these admin steps first and then we can get to the fun part of coding our bot.

Creating a bot in Discord and getting a token

You can sign up for a free account over at the Discord register page[131], and can download one of their desktop or mobile applications from the Discord homepage[132]. You can also use Discord in the browser.

Once you have an account, you'll want to create a Discord application. Visit the Discord developer's page[133] and press the "New application" button, as in the image below.

[129]https://repl.it
[130]https://discordapp.com/
[131]https://discordapp.com/register
[132]https://discordapp.com/
[133]https://discordapp.com/developers/applications/



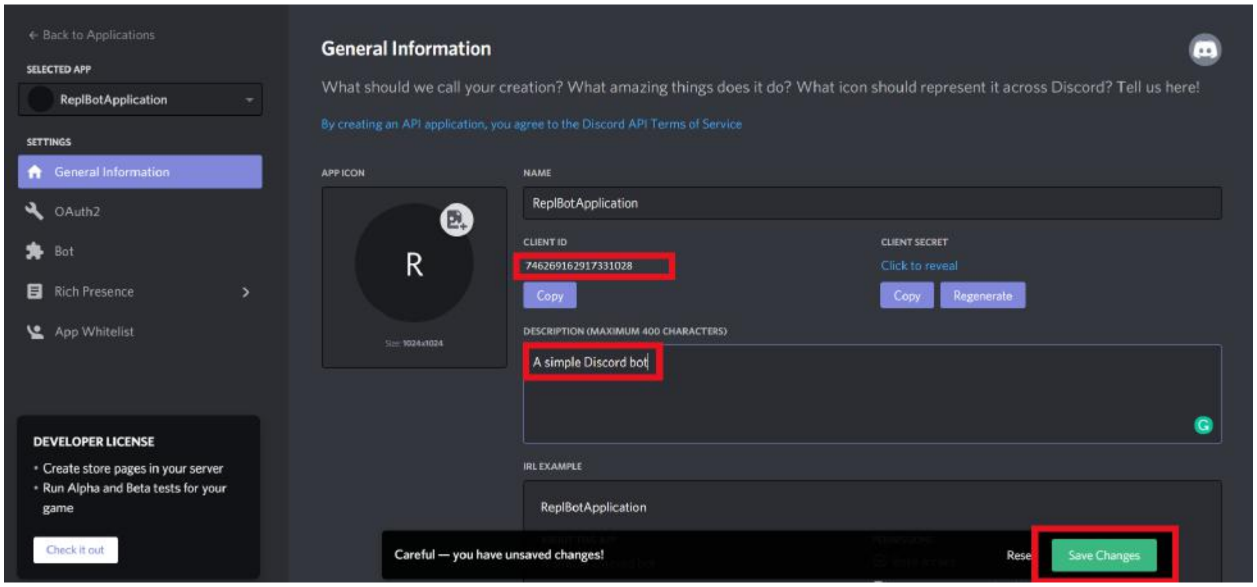
Find the perfect feature set for your game in our Game SDK, and sign up for Server Commerce to start selling directly in your server. Get started by creating a new application. We can't wait to see what you make!

You don't have any applications yet, sad face.
Click New Application above to get started.

Image: 1 ***Creating a new Discord application***

Fill out a name for your bot and select "Create".

The first thing to do on the next page is to note your Client ID, which you'll need to add the bot to the server. You can come back later and get it from this page, or copy it somewhere where you can easily find it later.



What should we call your creation? What amazing things does it do? What icon should represent it across Discord? Tell us here!

By creating an API application, you agree to the Discord API Terms of Service

NAME: RepIBotApplication

CLIENT ID: 746269162917331028

DESCRIPTION (MAXIMUM 400 CHARACTERS): A simple Discord bot

IRL EXAMPLE: RepIBotApplication

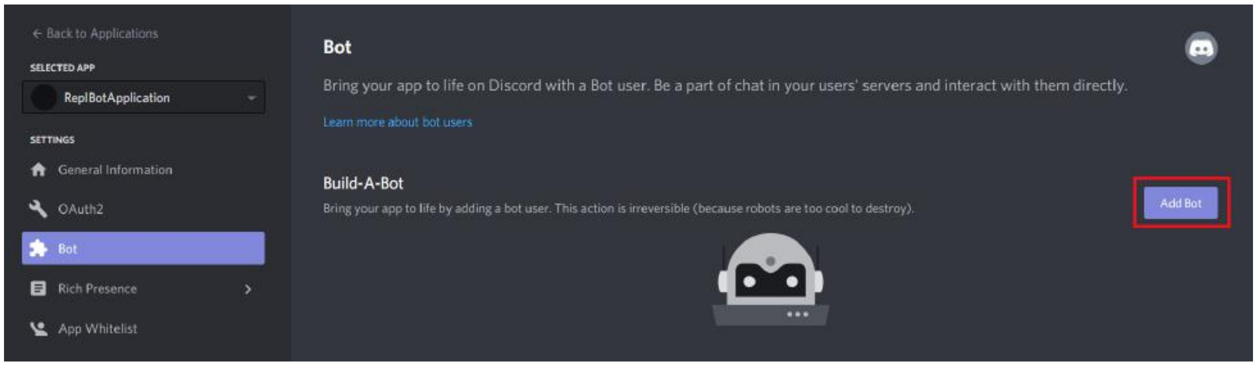
DEVELOPER LICENSE
- Create store pages in your server
- Run Alpha and Beta tests for your game

Careful — you have unsaved changes!

Image: 2 ***Record your Client ID***

You can also rename the application and provide a description for your bot at this point and press "Save Changes".

You have now created a Discord application. The next step is to add a bot to this application, so head over to the "Bot" tab using the menu on the left and press the "Add Bot" button, as indicated below. Click "Yes, do it" when Discord asks if you're sure about bringing a new bot to life.

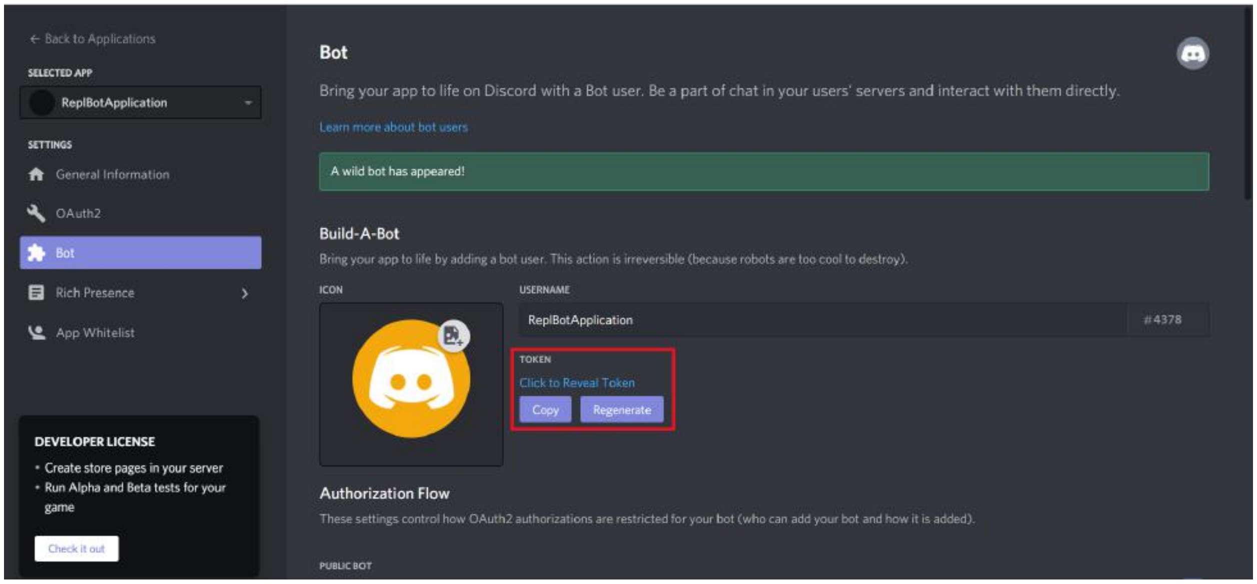


Bring your app to life on Discord with a Bot user. Be a part of chat in your users' servers and interact with them directly.

Bring your app to life by adding a bot user. This action is irreversible (because robots are too cool to destroy).

Image: 3 ***Adding a bot to our Discord Application***

The last thing we'll need from our bot is a Token. Anyone who has the bot's token can prove that they own the bot, so you'll need to be careful not to share this with anyone. You can get the token by pressing "Click to Reveal Token", or copy it to your clipboard without seeing it by pressing "Copy".



Bring your app to life on Discord with a Bot user. Be a part of chat in your users' servers and interact with them directly.

A wild bot has appeared!

Bring your app to life by adding a bot user. This action is irreversible (because robots are too cool to destroy).

ReplBotApplication #4378

- Create store pages in your server
- Run Alpha and Beta tests for your game

These settings control how OAuth2 authorizations are restricted for your bot (who can add your bot and how it is added).

Image: 4 ***Generating a token for our Discord bot***

Take note of your token or copy it to your clipboard, as we'll need to add it to our code soon.

Creating a Discord server

If you don't have a Discord server to add your bot to, you can create one by either opening the desktop Discord application that you downloaded earlier or returning to the Discord home page in your browser. Press the "+" icon indicated by the exclamation mark, as shown below, to create a server.

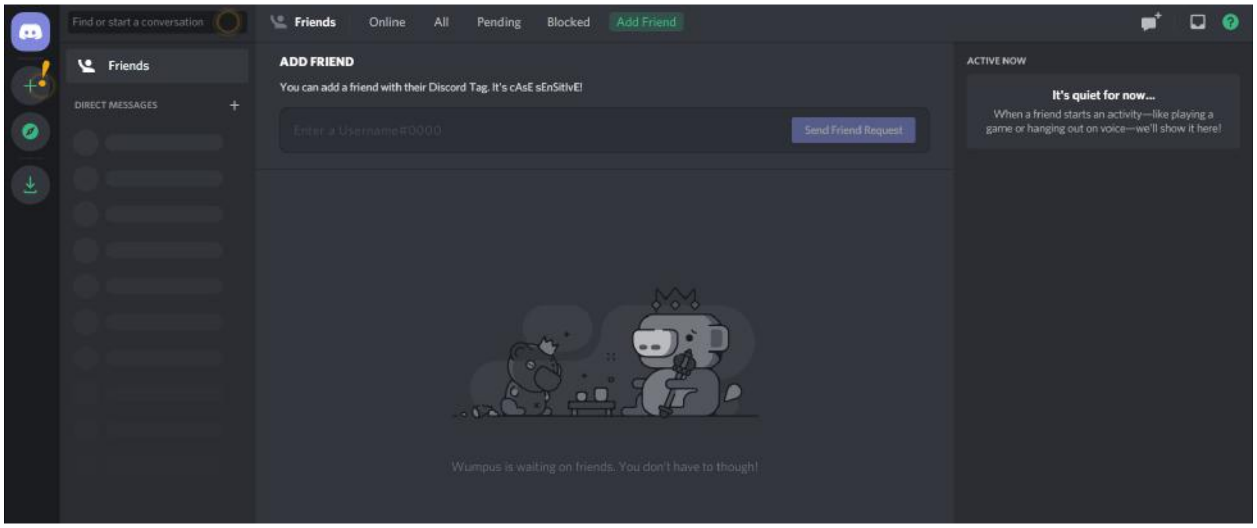

Find or start a conversation
Friends
Online
All
Pending
Blocked
Add Friend
DIRECT MESSAGES
ADD FRIEND
You can add a friend with their Discord Tag. It's cAsE sEnSitIvE!
Enter a Username#0000
Send Friend Request
ACTIVE NOW
It's quiet for now...
When a friend starts an activity—like playing a game or hanging out on voice—we'll show it here!
Wumpus is waiting on friends. You don't have to though!

Image: 5 ***Creating a Discord server***

Press “Create a server” in the screen that follows, and then give your server a name. Once the server is up and running, you can chat with yourself, or invite some friends to chat with you. Soon we’ll invite our bot to chat with us as well.

Adding your Discord bot to your Discord server

Our Discord bot is still just a shell at this stage as we haven’t written any code to allow him to do anything, but let’s go ahead and add him to our Discord server anyway. To add a bot to your server, you’ll need the Client ID from the “General Information” page that we looked at before when we created our ReplBotApplication (ie. the client ID, not the secret bot Token).

Create a URL that looks as follows, but using your Client ID instead of mine at the end:

https://discordapp.com/api/oauth2/authorize?scope=bot&client_id=746269162917331028

Visit the URL that you created in your web browser and you’ll see a page similar to the following where you can choose which server to add your bot to.

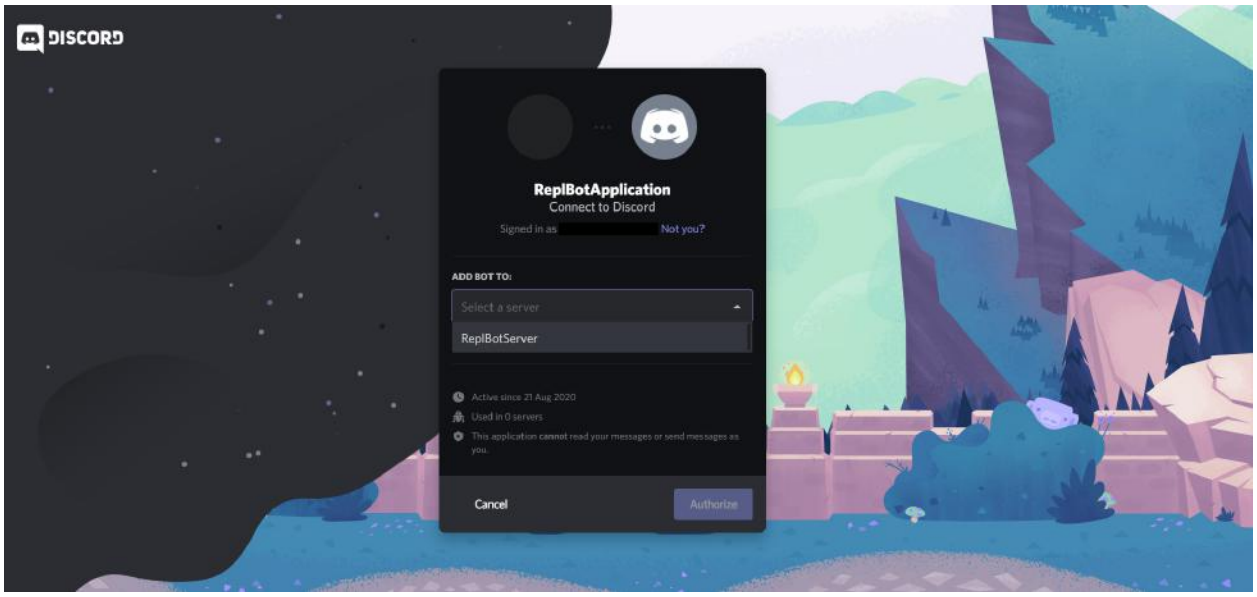


Image: 6 ***Authorizing our bot to join our server***

Select the server we created in the step before this and hit the "authorize" button. After completing the captcha, you should get an in-app Discord notification telling you that your bot has joined your server.

Now we can get to the fun part of building a brain for our bot!

Creating a Repl and installing our Discord dependencies

The first thing we need to do is create a Node.js Repl to write the code for our Discord bot. Over at repl.it[134], create a new Repl, choosing "Node.js" as your language.

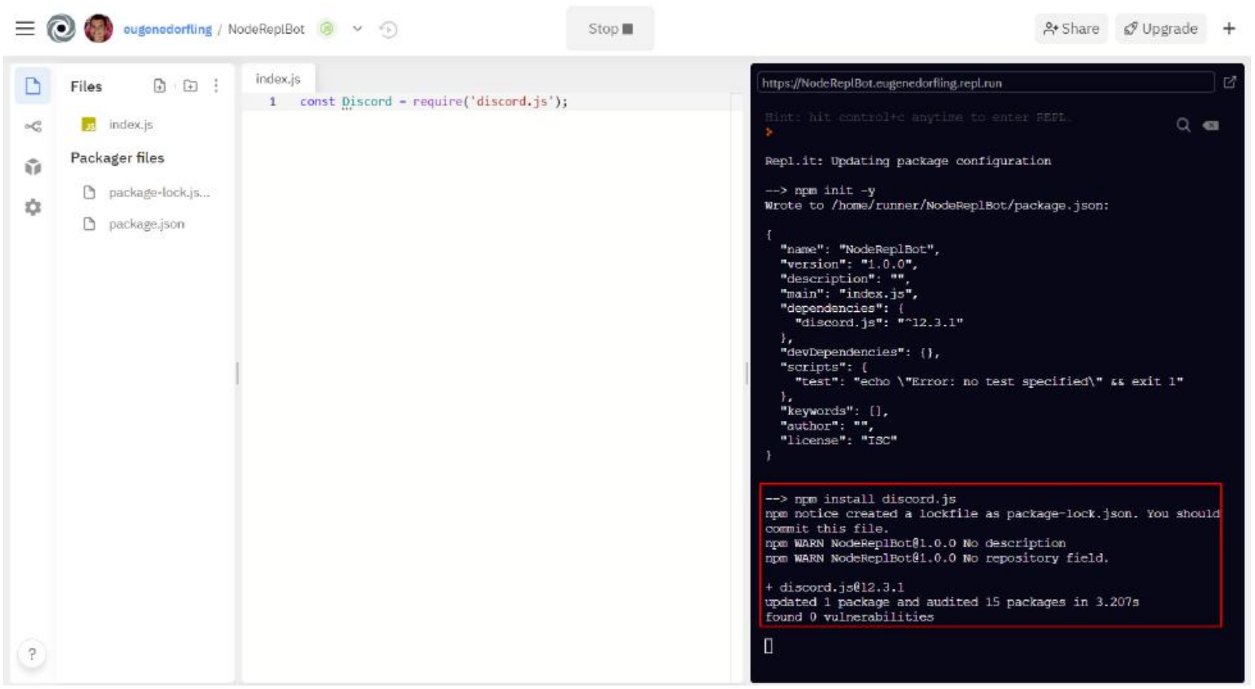
We don't need to reinvent the wheel as there is already a great Node wrapper for the Discord bot API called discord.js[135]. Normally we would install this third-party library through npm[136], but because we're using Repl.it, we can skip the installation. Our Repl will automatically pull in all dependencies.

In the default `index.js` file that is included with your new Repl, add the following line of code.

```
const Discord = require('discord.js');
```

Press the "Run" button and you should see Repl.it installing the Discord library in the output pane on the right, as in the image below.

[134]https://repl.it
[135]https://discord.js.org/
[136]https://www.npmjs.com/



```
const Discord = require('discord.js');
```

```
Hint: hit control+c anytime to enter REPL.
>

Repl.it: Updating package configuration

--> npm init -y
Wrote to /home/runner/NodeReplBot/package.json:

{
  "name": "NodeReplBot",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {
    "discord.js": "^12.3.1"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

--> npm install discord.js
npm notice created a lockfile as package-lock.json. You should
commit this file.
npm WARN NodeReplBot@1.0.0 No description
npm WARN NodeReplBot@1.0.0 No repository field.

+ discord.js@12.3.1
updated 1 package and audited 15 packages in 3.207s
found 0 vulnerabilities
```

Image: 7 ***Installing Discord.js in our Repl***

Our bot is nearly ready to go – but we still need to plug in our secret token. This will authorize our code to control our bot.

Setting up authorization for our bot

By default, Repl code is public. This is great as it encourages collaboration and learning, but we need to be careful not to share our secret bot token (which gives anyone who has access to it full control of our bot).

To get around the problem of needing to give our *code* access to the token while allowing others to access our code but *not* our token, we'll be using environment variables[137]. On a normal machine, we'd set these directly on our operating system, but using Repl.it we don't have access to this. Repl.it allows us to set secrets in environment variables through a special `.env` file.

First, we need to create a new file called exactly `.env`. Select "Add file" and name this file `.env`. It is important not to leave out the `.` at the beginning. Open this new file and add a variable to define your bot's secret token (note that this is the second token that we got while setting up the bot – different from the Client ID that we used to add our bot to our server). It should look something like:

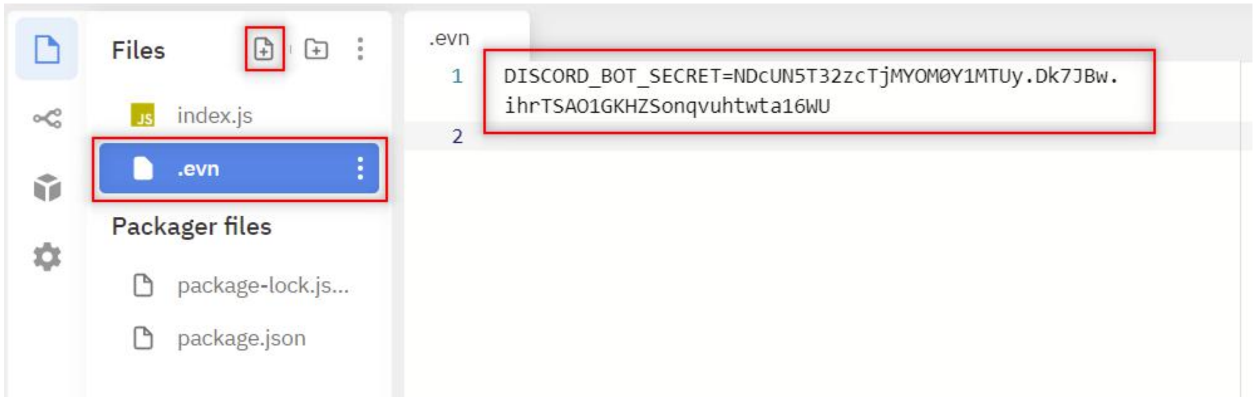
```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.ihrTSAO1GKHZSonqvuhtwta16WU
```

[137]https://www.digitalocean.com/community/tutorials/how-to-read-and-set-environmental-and-shell-variables-on-a-linux-vps

You'll need to:

- **Replace** the token below (after the = sign) with the token that Discord gave you when creating your own bot.
- Be careful about **spacing**. If you put a space on either side of the = in your `.env` file, these spaces will be part of the variable name or the value, so make sure you don't have any spaces around the = or at the end of the line.
- Run the code again. Sometimes you'll need to refresh the whole page to make sure that your environment variables are successfully loaded.

In the image below we've highlighted the "Add file" button, the new file (`.env`) and how to define the secret token for our bot's use.



```
DISCORD_BOT_SECRET=NDcUN5T32zcTjMYOM0Y1MTUy.Dk7JBw.
ihrTSAO1GKHZSonqvuhtwta16WU
```

Image: 8 ***Creating our .env file***

Let's make a Discord bot that repeats everything we say but in reverse. We can do this in only a few lines of code. In your `index.js` file, add the following:

```
const Discord = require('discord.js');
const client = new Discord.Client();
const token = process.env.DISCORD_BOT_SECRET;

client.on('ready', () => {
  console.log("I'm in");
  console.log(client.user.username);
});

client.on('message', msg => {
    if (msg.author.id != client.user.id) {
        msg.channel.send(msg.content.split('').reverse().join(''));
    }
});

client.login(token);
```

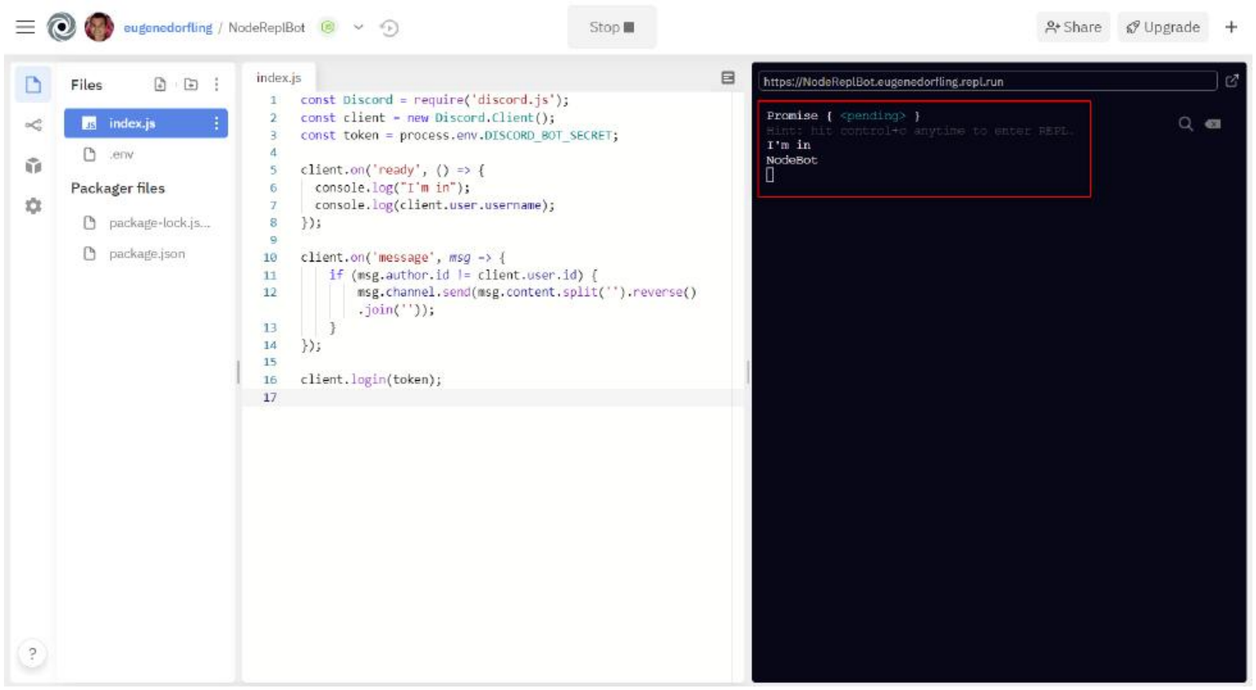
Let's tear this apart line by line to see what it does.

- **Line 1** is what we had earlier. This line both tells Repl.it to install the third party library and brings it into this file so that we can use it.
- In **line 2**, we create a Discord `Client`. We'll use this client to send commands to the Discord *server* to control our bot and send it commands.
- In **line 3** we retrieve our secret token from the environment variables (which Repl.it sets from our `.env` file).
- In **line 5**, we define an `event` for our client, which defines how our bot should react to the "ready" event. The Discord bot is going to run *asynchronously*, which might be a bit confusing if you're used to running standard synchronous code. We won't go into asynchronous coding in depth here, but if you're interested in what this is and why it's used, there's a good guide over at RisingStack[138]. In short, instead of running the code in our file from top to bottom, we'll be running pieces of code in response to specific events.
- In **lines 6-8** we define how our bot should respond to the "ready" event, which is fired when our bot successfully joins a server. We instruct our bot to output some information server side (i.e. this will be displayed in our Repl's output, but not sent as a message through to Discord). We'll print a simple `I'm in` message to see that the bot is there and print our bot's username (if you're running multiple bots, this will make it easier to work out who's doing what).
- **Lines 10-14** are similar, but instead of responding to an "ready" event, we tell our bot how to handle new messages. **Line 11** says we only want to respond to messages that aren't from us (otherwise our bot will keep responding to himself – you can remove this line to see why that's a problem), and **line 12** says we'll send a new message to the same channel where we received a message (`msg.channel`) and the content we'll send will be the same message that we received, but backwards. To reverse a string, we split it into its individual characters, reverse the resulting array, and then join it all back into a string again.

The last line fires up our bot and uses the token we loaded earlier to log into Discord.

Press the big green "Run" button again and you should see your bot reporting a successful channel join in the Repl output.

[138]https://blog.risingstack.com/node-hero-async-programming-in-node-js/

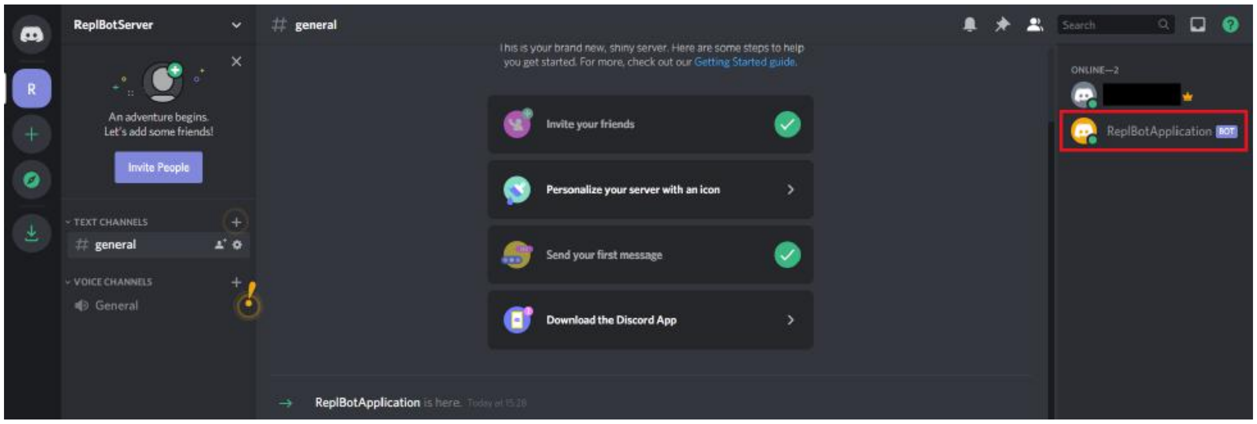


```javascript
const Discord = require('discord.js');
const client = new Discord.Client();
const token = process.env.DISCORD_BOT_SECRET;

client.on('ready', () => {
  console.log("I'm in");
  console.log(client.user.username);
});

client.on('message', msg -> {
    if (msg.author.id != client.user.id) {
      msg.channel.send(msg.content.split('').reverse()
      .join(''));
    }
});

client.login(token);
```

```
Promise { <pending> }
Hint: hit control+c anytime to enter REPL.
I'm in
NodeBot
```

Image: 9 ***Repl output showing channel join***

Open Discord, and from within the server we created earlier, select your ReplBotApplication from the pane on the right-hand side of the screen.



This is your brand new, shiny server. Here are some steps to help you get started. For more, check out our Getting Started guide.

[redacted]

ReplBotApplication is here. Today at 15:28

Image:10 ***The Repl bot is active***

.

Once you have selected this, you will be able to send a message (by typing into the box highlighted below) and see your bot respond!

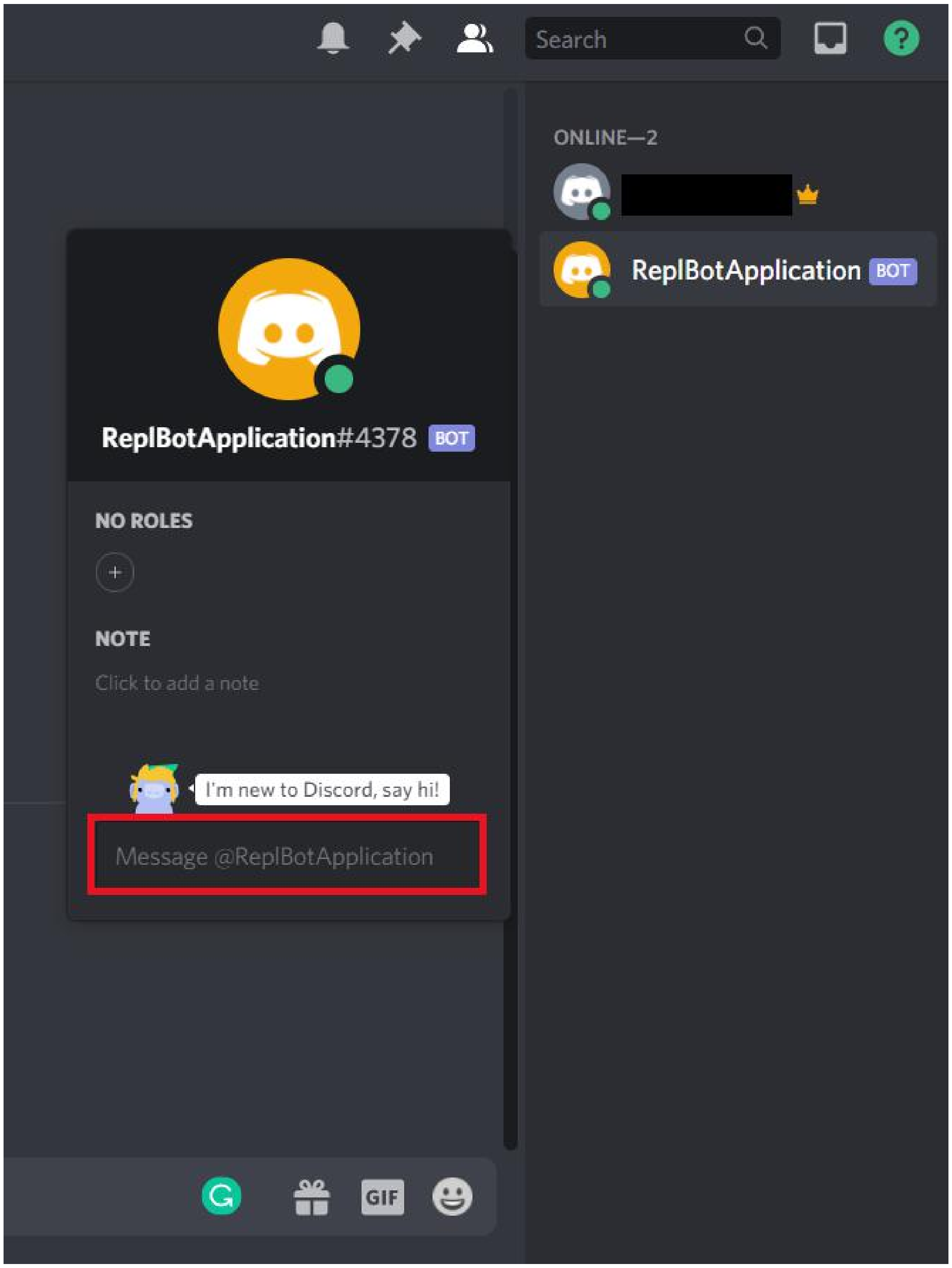
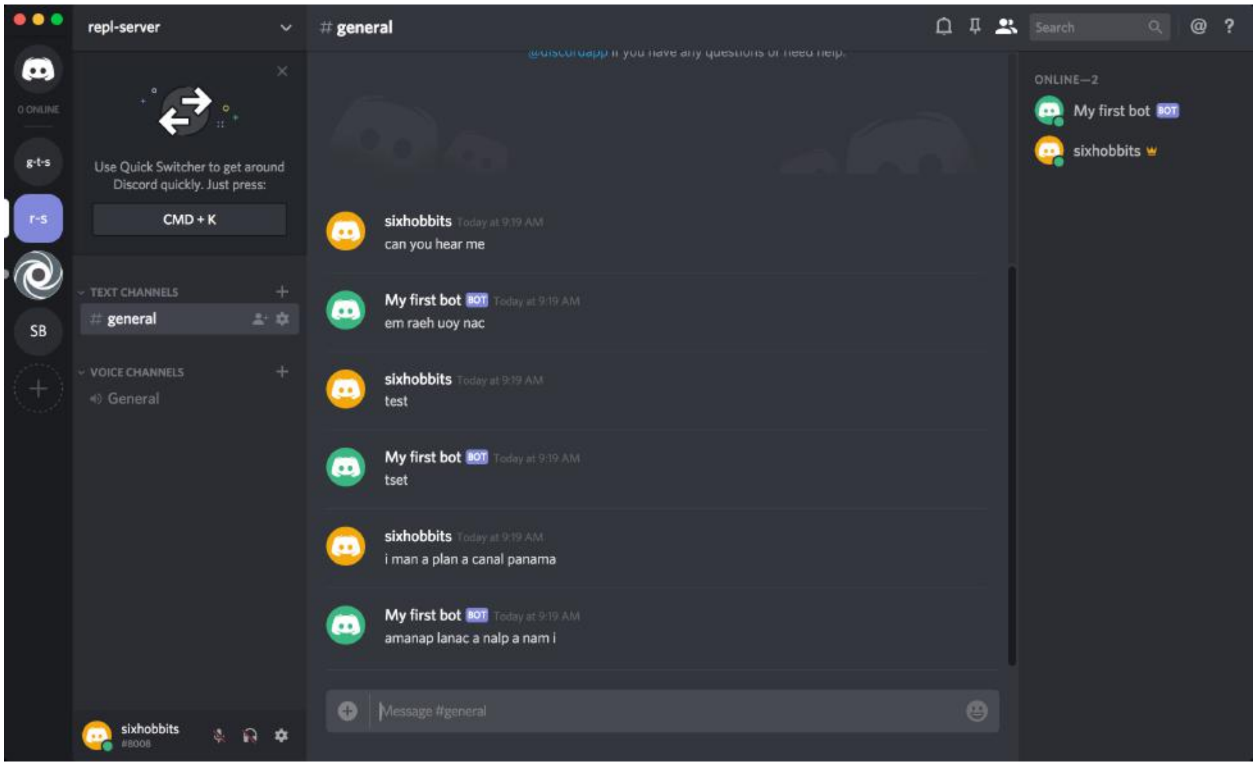

Search
ONLINE—2
[redacted]
ReplBotApplication BOT
ReplBotApplication#4378 BOT
NO ROLES
NOTE
Click to add a note
I'm new to Discord, say hi!
Message @ReplBotApplication
GIF

Image:11 ***Send a message to your bot***

.

The bot responds each time, reversing the text we enter.



sixhobbits Today at 9:19 AM
can you hear me

My first bot BOT Today at 9:19 AM
em raeh uoy nac

sixhobbits Today at 9:19 AM
test

My first bot BOT Today at 9:19 AM
tset

sixhobbits Today at 9:19 AM
i man a plan a canal panama

My first bot BOT Today at 9:19 AM
amanap lanac a nalp a nam i

Image:12 ***Our bot can talk!***

.

Keeping our bot alive

Your bot can now respond to messages, but only for as long as your Repl is running. If you close your browser tab or shut down your computer, your bot will stop and no longer respond to messages on Discord.

Repl will keep your code running after you close the browser tab only if you are running a web server. Our bot doesn't require an explicit web server to run, but we can create a server and run it in the background just to keep our Repl alive.

Create a new file in your project called `keep_alive.js` and add the following code:

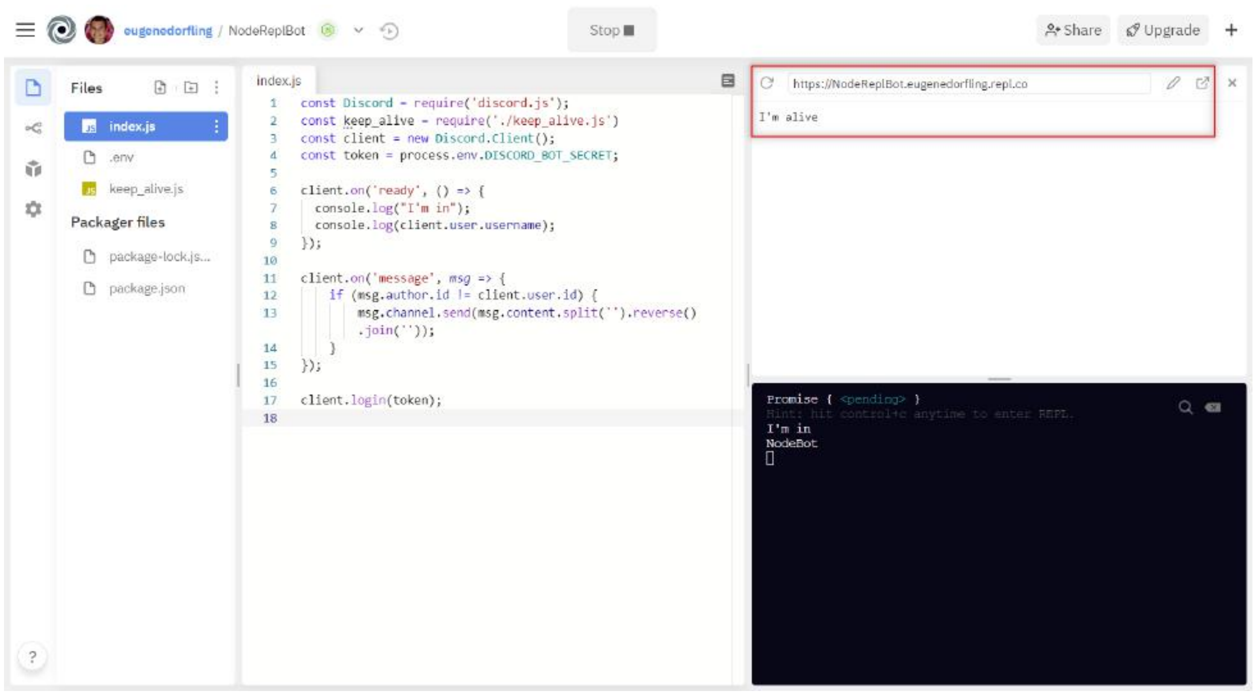
```
var http = require('http');

http.createServer(function (req, res) {
  res.write("I'm alive");
  res.end();
}).listen(8080);
```

We won't go over this in detail as it's not central to our bot, but here we start a web server that will return "I'm alive" if anyone visits it.

In our `index.js` file, we need to add a require statement for this server at the top. Add the following line near the top of `index.js`.

```
const keep_alive = require('./keep_alive.js')
```

After doing this and hitting the green "Run" button again, you should see some changes to your Repl. For one, you'll see a new pane in the top right which shows the web output from your server. We can see that visiting our Repl now returns a basic web page showing the "I'm alive" string that we told our web server to return by default.



```
const Discord = require('discord.js');
const keep_alive = require('./keep_alive.js')
const client = new Discord.Client();
const token = process.env.DISCORD_BOT_SECRET;

client.on('ready', () => {
  console.log("I'm in");
  console.log(client.user.username);
});

client.on('message', msg => {
    if (msg.author.id != client.user.id) {
        msg.channel.send(msg.content.split('').reverse()
        .join(''));
    }
});

client.login(token);
```

```
I'm alive
```

```
Promise { <pending> }
Hint: hit control+c anytime to enter REPL.
I'm in
NodeBot
```

Image:13 ***Running a Node server in the background***

Now your bot will stay alive even after closing your browser or shutting down your development machine. Repl will still clean up your server and kill your bot after about one hour of inactivity, so if you don't use your bot for a while, you'll have to log into Repl and start the bot up again. Alternatively, you can set up a third-party (free!) service like Uptime Robot[139]. Uptime Robot pings your site every 5 minutes to make sure it's still working – usually to notify you of unexpected downtime, but in this case the constant pings have the side effect of keeping our Repl alive as it will never go more than an hour without receiving any activity. Note that you need to select the HTTP option instead of the Ping option when setting up Uptime Robot, as Repl.it requires regular HTTP requests to keep your chatbot alive.

[139]https://uptimerobot.com/

Forking and extending our basic bot

This is not a very useful bot as is, but the possibilities are only limited by your creativity now! You can have your bot receive input from a user, process the input, and respond in any way you choose. In fact, with the basic input and output that we've demonstrated, we have most of the components of any modern computer, all of which are based on the Von Neumann architecture[140] (we could easily add the missing memory by having our bot write to a file, or with a bit more effort link in a SQLite database[141] for persistent storage).

If you followed along this tutorial, you'll have your own basic Repl bot to play around with and extend. If you were simply reading, you can easily fork my bot at https://repl.it/@GarethDwyer1/discord-bot-node[142] and extend it how you want (you'll need to add your own token and recreate the `.env` file still). Happy hacking!

If you're stuck for ideas, why not link up your Discord bot to the Twitch API[143] to get notified when your favourite streamers are online, or build a text adventure[144].

In the next chapter, we'll be looking at building our own basic web application, using Django. This tutorial will also introduce you to HTML, JavaScript, and jQuery and will assist you in getting to the point where you can begin to build your own custom web applications.

[140]https://en.wikipedia.org/wiki/Von_Neumann_architecture
[141]https://www.sqlite.org/index.html
[142]https://repl.it/@GarethDwyer1/discord-bot-node
[143]https://dev.twitch.tv/
[144]https://en.wikipedia.org/wiki/Interactive_fiction

Creating and hosting a basic web application with Django and Repl.it

In this tutorial, we'll be using Django to create an online service that shows visitors their current weather and location. We'll develop the service and host it using repl.it[145].

To work through this tutorial, you should ideally have basic knowledge of Python and some knowledge of web application development. However, we'll explain all of our reasoning and each line of code thoroughly , so if you have any programming experience you should be able to follow along as a complete Python or web app beginner too. We'll also be making use of some HTML, JavaScript, and jQuery, so if you have been exposed to these before you'll be able to work through more quickly. If you haven't, this will be a great place to start.

To display the weather at the user's current location, we'll have to tie together a few pieces. The main components of our system are:

- A Django[146] application, to show the user a webpage with dynamic data
- Ipify[147] to get our visitors' IP address so that we can guess their location
- ip-api[148] to look up our visitors' city and country using their IP address
- Open Weather Map[149] to get the current weather at our visitors' location.

The main goals of this tutorial are to:

[145]https://repl.it
[146]https://www.djangoproject.com/
[147]https://www.ipify.org/
[148]http://ip-api.com/
[149]https://openweathermap.org

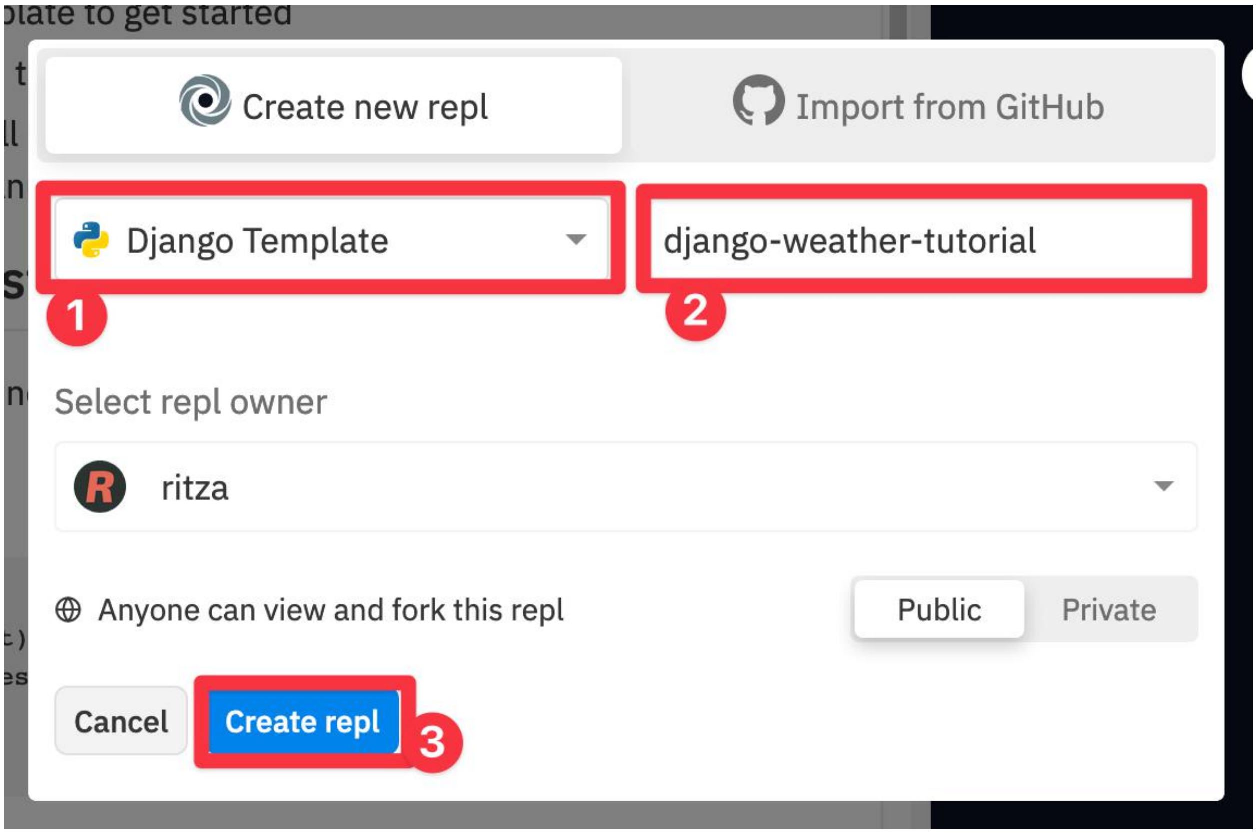
- Show how to set up and host a Django application using repl.it.
- Show how to join existing APIs together to create a new service.

By using this tutorial as a starting point, you can easily create your own bespoke web applications. Instead of showing weather data to your visitors, you could, for example, pull and combine data from any of the hundreds of APIs found at this list of public APIs[150].

Setting up

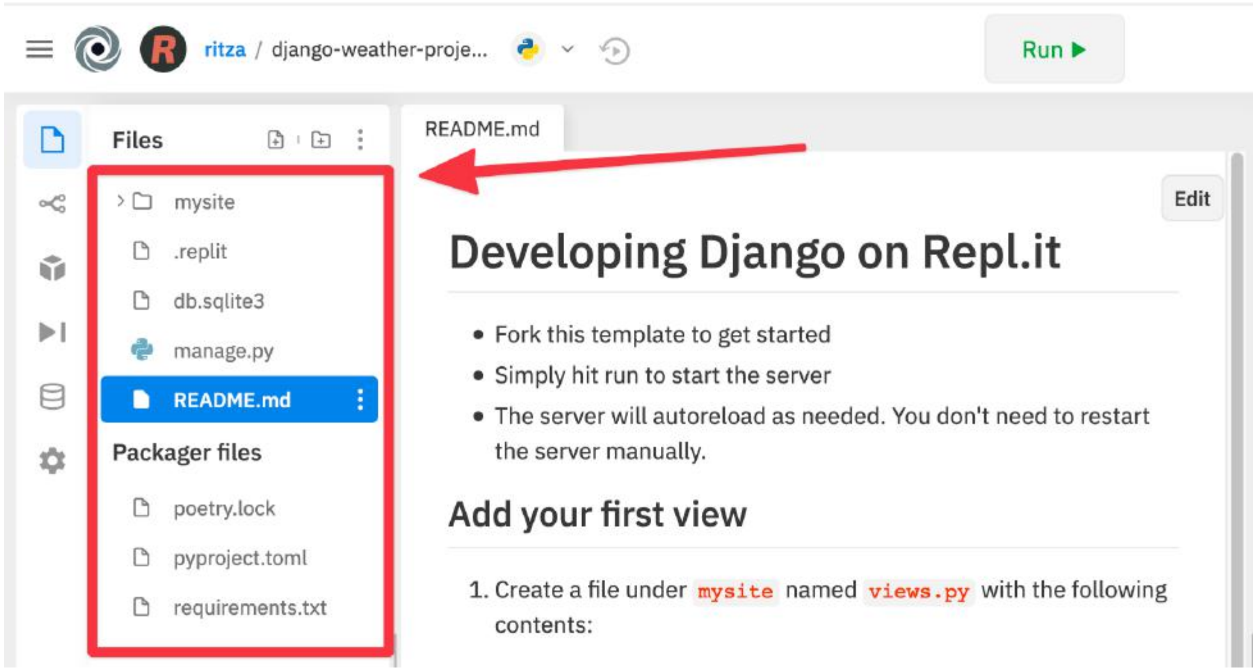
You won't need to install any software or programming languages on your local machine, as we'll be developing our application directly through repl.it[151]. Head over there and create an account.

Press the + button in the top right to create a new project and search for "Django Template". Give your project a name and press "Create repl".



By default, Django comes with a pretty complicated folder structure of existing files and folders. There's also a README.md file that will open by default, giving you some guidance on how to find your way around.

[150] https://github.com/toddmotto/public-apis
[151] repl.it



Developing Django on Repl.it

- Fork this template to get started
- Simply hit run to start the server
- The server will autoreload as needed. You don't need to restart the server manually.

Add your first view

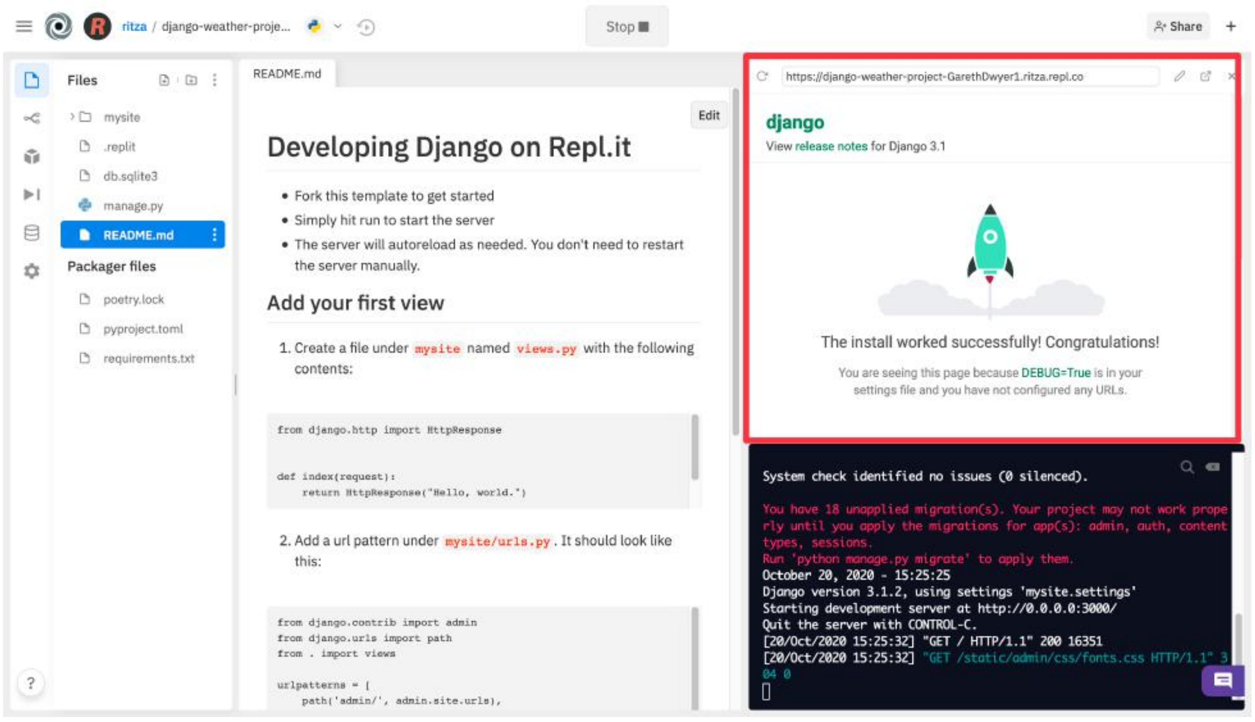
1. Create a file under `mysite` named `views.py` with the following contents:

We won't explain what all these different components are for and how they tie together in this tutorial. If you want to understand Django better, you should go through their official tutorial[152].

In this tutorial, we'll just look at the few files that we need to modify to get our basic application working.

Hit the Run button in the bar at the top and you'll see Repl.it install all of the required packages and start up the default Django app.

[152]https://docs.djangoproject.com/en/2.0/intro/tutorial01/



Developing Django on Repl.it

- Fork this template to get started
- Simply hit run to start the server
- The server will autoreload as needed. You don't need to restart the server manually.

Add your first view

1. Create a file under `mysite` named `views.py` with the following contents:

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world.")
```

2. Add a url pattern under `mysite/urls.py`. It should look like this:

```
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
```

django

View release notes for Django 3.1

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

```
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work prope
rly until you apply the migrations for app(s): admin, auth, content
types, sessions.
Run 'python manage.py migrate' to apply them.
October 20, 2020 - 15:25:25
Django version 3.1.2, using settings 'mysite.settings'
Starting development server at http://0.0.0.0:3000/
Quit the server with CONTROL-C.
[20/Oct/2020 15:25:32] "GET / HTTP/1.1" 200 16351
[20/Oct/2020 15:25:32] "GET /static/admin/css/fonts.css HTTP/1.1" 3
04 0
```

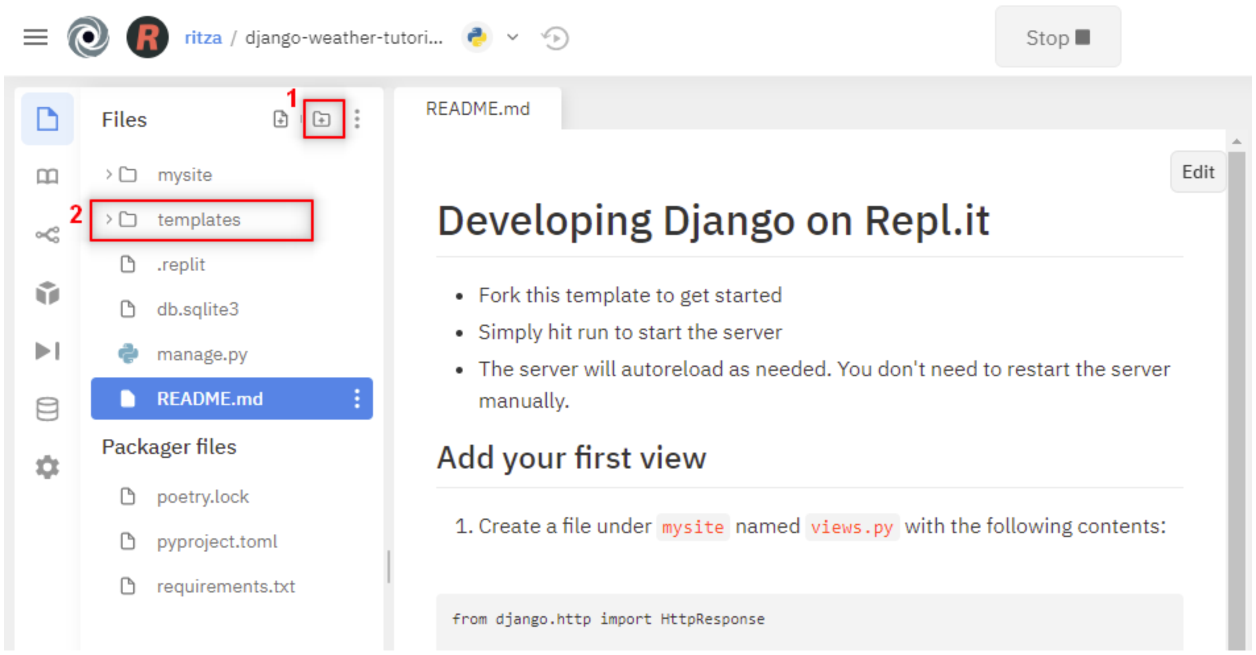
Creating a static home page

Viewing the default website isn't that interesting and you'll notice that there are no files containing the content you currently see, so you can't easily modify it.

To create our own page in place, we'll have to modify several files. If you've created a basic HTML file before, you might be surprised at how complicated this step is. Like other frameworks, Django "makes the easy things hard, and the hard things possible". If you just want to display a basic web page, it's probably over kill, but as your web app grows, you'll find use for all of the extra structure.

Let's set up a basic "Hello world" page to make sure we have the pieces in place. You'll need to create several more folders and files to achieve this.

Create a folder called `templates`. This is where Django will get HTML templates for rendering pages.



Developing Django on Repl.it

- Fork this template to get started
- Simply hit run to start the server
- The server will autoreload as needed. You don't need to restart the server manually.

Add your first view

1. Create a file under `mysite` named `views.py` with the following contents:

```
from django.http import HttpResponse
```

Let's add the HTML templates that we will use to render our static page. Create a file called `base.html` within the newly created templates folder and add the following code.

```
{% load static %}
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
    <link rel="stylesheet" href="{% static "css/style.css" %}">
</head>
<body>
    {% block content %}{% endblock content %}
</body>
</html>
```

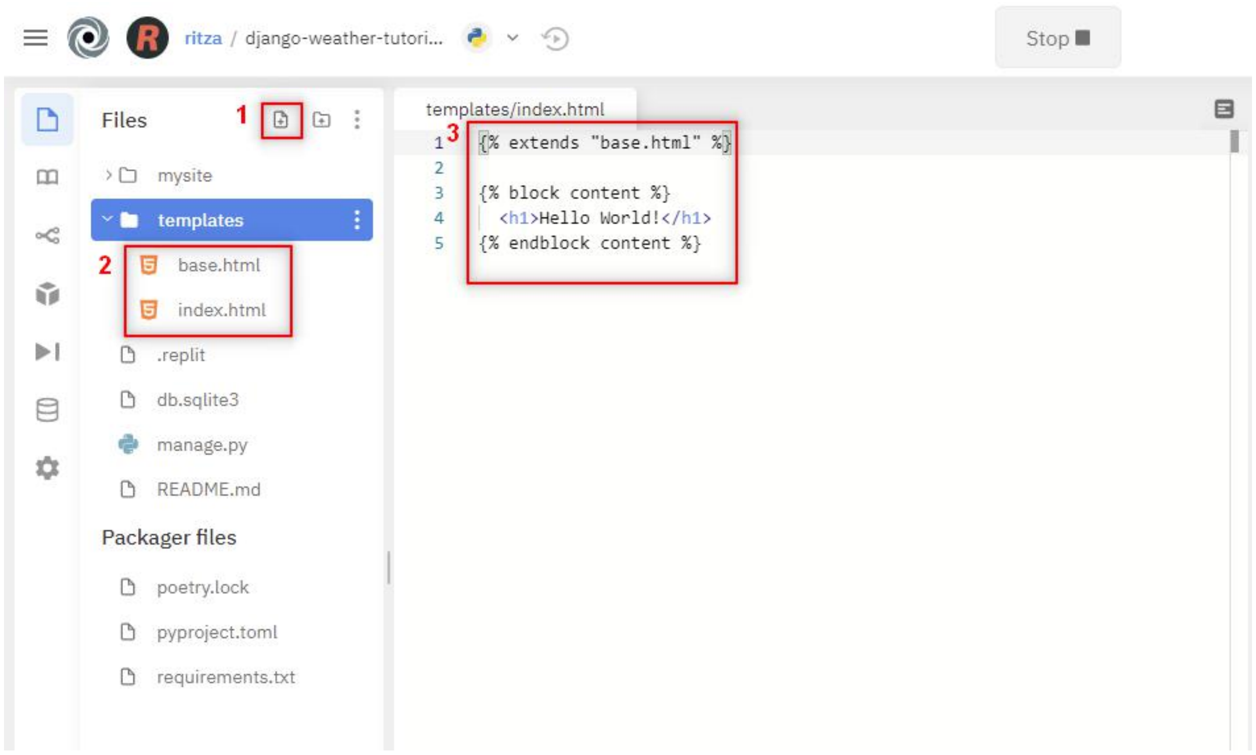
The above is a basic HTML template that our Django app will use when rendering pages. We also link to a stylesheet that Django will get from a folder called `static/css` which we will create soon. Note the `{% load static %}` in the first line, this is to tell Django that we are using static files in this template ie. `style.css`.

Still in the templates folder, create a file called `index.html` and add the following code.

```
{% extends "base.html" %}

{% block content %}
  <h1>Hello World!</h1>
{% endblock content %}
```

This is a file written in Django's template language[153], which often looks very much like HTML (and is usually found in files with a `.html` extension), but which contains some extra functionality to help us load dynamic data from the back end of our web application into the front end for our users to see.

The above extends the `base.html` template and adds the block content to it, in this case "Hello World!".



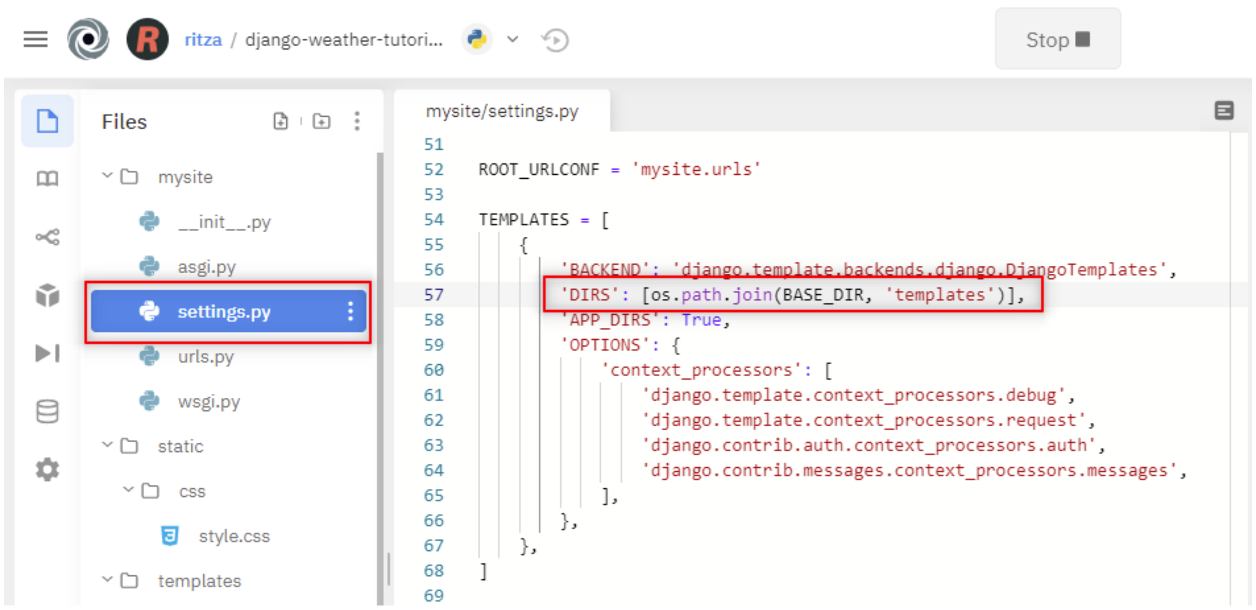
```
{% extends "base.html" %}

{% block content %}
  <h1>Hello World!</h1>
{% endblock content %}
```

Django looks for template folders within "app" folders by default. This is helpful when you have multiple apps within your project but in this case we don't so we need to tell Django where to find our templates folder.

Open the `mysite/settings.py` file, scroll down to `TEMPLATES` and add `os.path.join(BASE_DIR, 'templates')` within the square brackets next to `DIRS:` like below

[153] https://docs.djangoproject.com/en/2.0/topics/templates/

```
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
```



```

ROOT_URLCONF = 'mysite.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

```

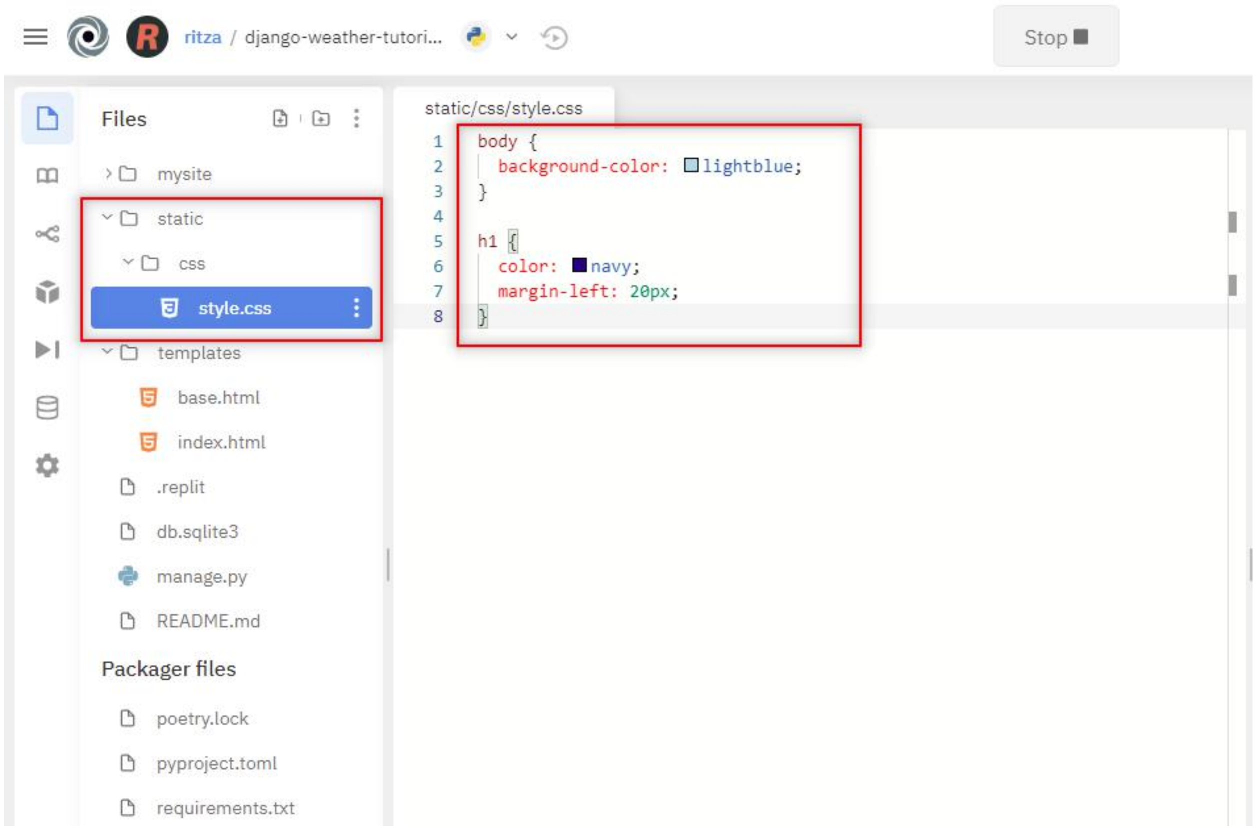
Now that we have our templates added, let's add the folders and files for adding the stylesheet.

Create a folder called `static` and also create a folder called `css` within the `static` folder.

Create a file called `style.css` within the `static/css/` directory and add the following code.

```css
body {
  background-color: lightblue;
}

h1 {
  color: navy;
  margin-left: 20px;
}
```

Above we add basic CSS code to demonstrate how you can modify the look of your site.

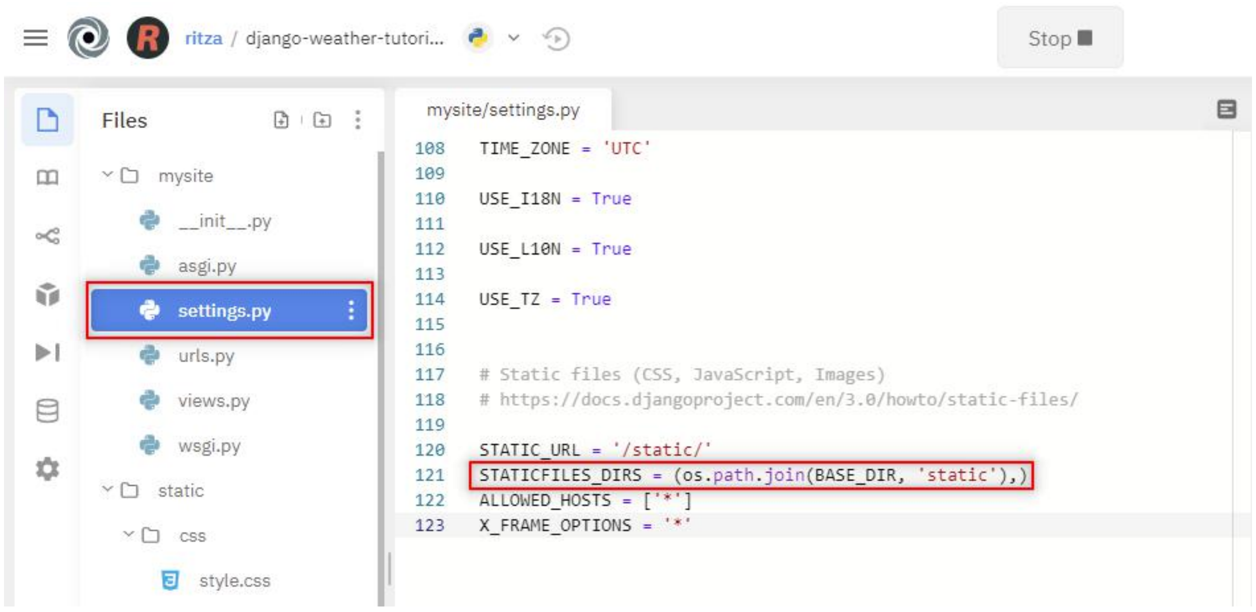


```
body {
  background-color: lightblue;
}

h1 {
  color: navy;
  margin-left: 20px;
}
```

Django handles static files similar to templates where it automatically checks app directories for a directory called `static`. Since we only have a static page instead of apps we need to tell Django where our static directory is located.

Open the `mysite/settings.py` file, scroll all the way down and add the following code right after `STATIC_URL ='/static/'`

```
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
```

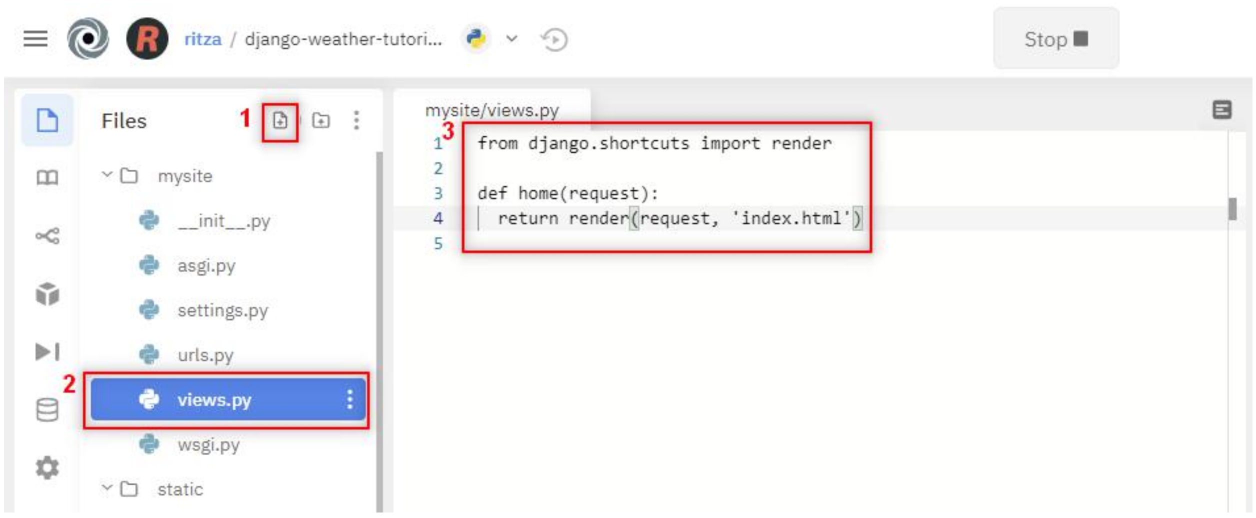


```
TIME_ZONE = 'UTC'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.0/howto/static-files/

STATIC_URL = '/static/'
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
ALLOWED_HOSTS = ['*']
X_FRAME_OPTIONS = '*'
```

Within the `mysite/` directory, create a file called `views.py` and add the following code.

```
from django.shortcuts import render

# Create your views here.
def home(request):
        return render(request, 'index.html')
```

A view function in Django is a Python function that takes Web requests and returns a Web response. This is where you add the logic that will return a certain response when called. In our case we define a view that will return the `index.html` page.



```
from django.shortcuts import render

def home(request):
  return render(request, 'index.html')
```

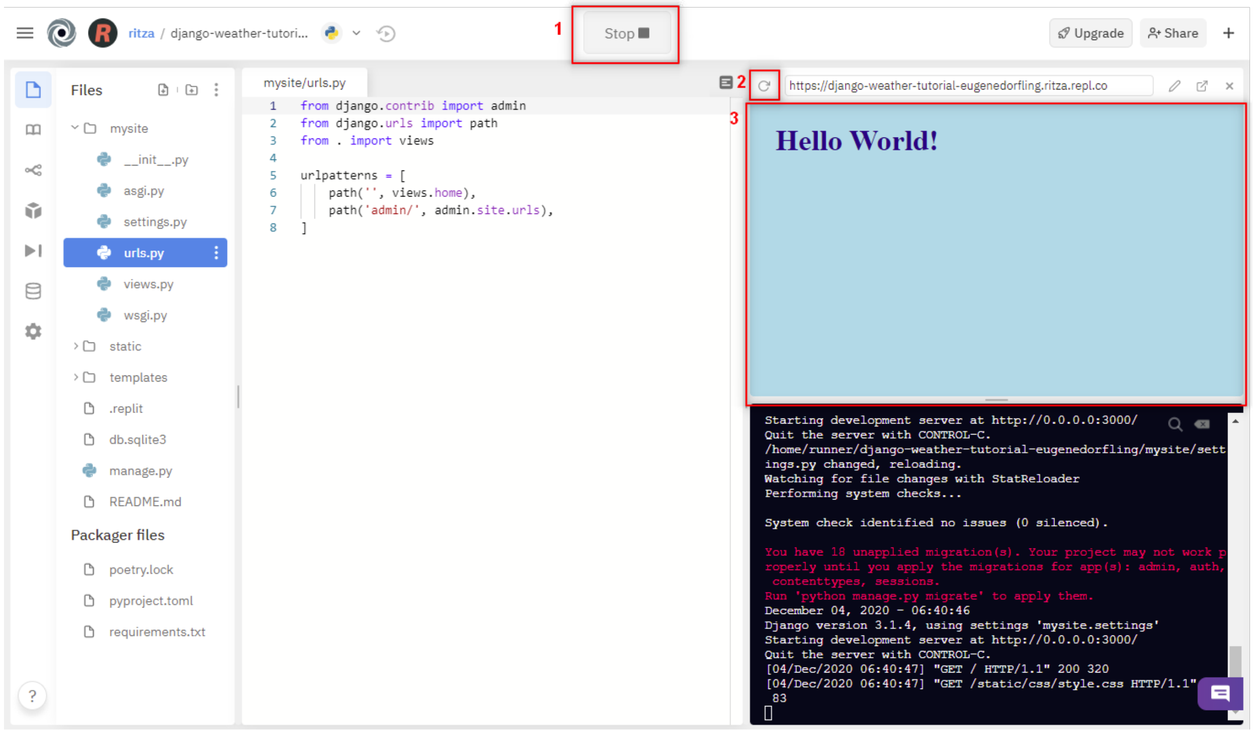
To call this view function we need to add it to our url patterns. Open the `mysite/urls.py` file and replace the contents with the below code.

```python
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
                path('', views.home, name= 'home'),
    path('admin/', admin.site.urls),
]
```

Note that we import the views file created earlier `from . import views`. Then we add the url pattern with an empty path `''` and point it to the `home` view created earlier. The `admin/` path points to the admin page that comes as a default with Django.

When Django receives a request it goes through the `urlpatterns` list until it finds a match. In our case `<url>.com` will match the first path and return the `home` view that will render the `index.html` page. If we navigate to`<url>.com/admin/`, Django will match the pattern of the `admin/` path and return the admin page.

Restart your server and refresh the web page on the right. You should see our "Hello World!" page.



```python
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home),
    path('admin/', admin.site.urls),
]
```

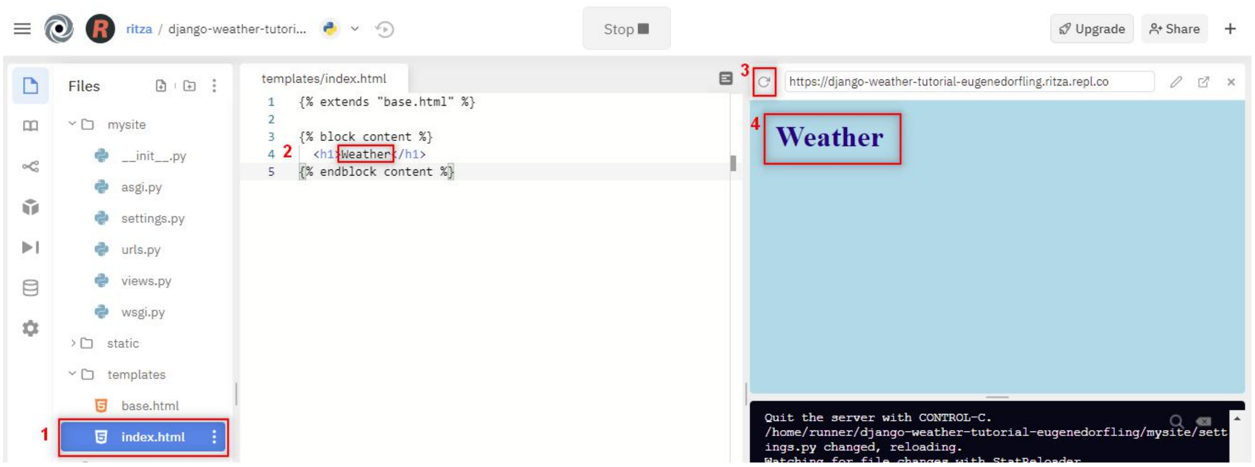
```
Starting development server at http://0.0.0.0:3000/
Quit the server with CONTROL-C.
/home/runner/django-weather-tutorial-eugenedorfling/mysite/sett
ings.py changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work p
roperly until you apply the migrations for app(s): admin, auth,
 contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
December 04, 2020 - 06:40:46
Django version 3.1.4, using settings 'mysite.settings'
Starting development server at http://0.0.0.0:3000/
Quit the server with CONTROL-C.
[04/Dec/2020 06:40:47] "GET / HTTP/1.1" 200 320
[04/Dec/2020 06:40:47] "GET /static/css/style.css HTTP/1.1"
 83
```

Great, we have now put all the pieces in place for our "Hello World!" web page.Let's expand this and start building our weather app.

Open the `templates/index.html` file and change the code where it says "Hello World!" to read "Weather" like below.

```
{% extends "base.html" %}

{% block content %}
  <h1>Weather</h1>
{% endblock content %}
```

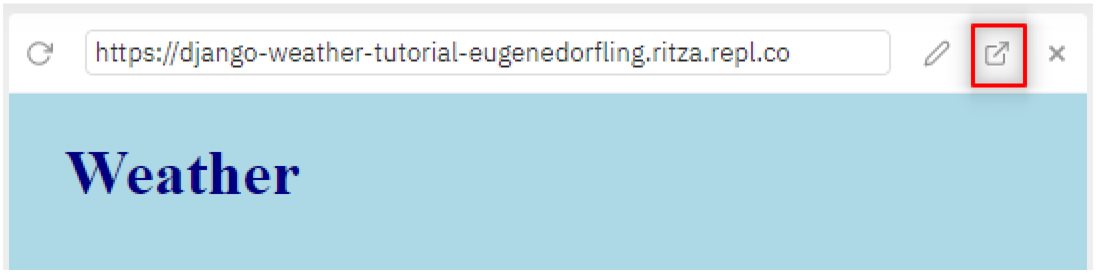
Click the refresh button as indicated below to see the result change from "Hello World!" to "Weather".

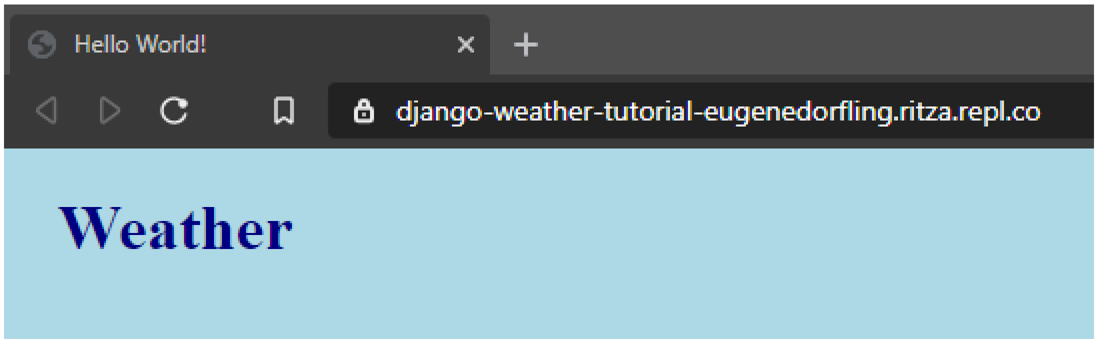


```
{% extends "base.html" %}

{% block content %}
  <h1>Weather</h1>
{% endblock content %}
```

```
Quit the server with CONTROL-C.
/home/runner/django-weather-tutorial-eugenedorfling/mysite/sett
ings.py changed, reloading.
```

You can also press the pop-out button to the right of the the URL bar to open only the resulting web page that we're building, as a visitor would see it. You can share the URL with anyone and they'll be able to see your Weather website already!


https://django-weather-tutorial-eugenedorfling.ritza.repl.co
Weather



Changing the static text that our visitors see is a good start, but our web application still doesn't *do* anything. We'll change that in the next step by using JavaScript to get our user's IP Address.

Calling IPIFY from JavaScript

An IP address is like a phone number. When you visit "google.com", your computer actually looks up the the name google.com to get a resulting IP address that is linked to one of Google's servers. While people find it easier to remember names like "google.com", computers work better with numbers. Instead of typing "google.com" into your browser toolbar, you could type the IP address 216.58.223.46, with the same results. Every device connecting to the internet, whether to serve content (like google.com) or to consume it (like you, reading this tutorial) has an IP address.

IP addresses are interesting to us because it is possible to guess a user's location based on their IP address. (In reality, this is an imprecise and highly complicated[154] process, but for our purposes it will be more than adequate). We will use the web service ipify.org[155] to retrieve our visitors' IP addresses.

In the Repl.it files tab, navigate to `templates/base.html`, which should look as follows.

```
{% load static %}
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
    <link rel="stylesheet" href="{% static "css/style.css" %}">
</head>
<body>
    {% block content %}{% endblock content %}
```

[154] https://dyn.com/blog/finding-yourself-the-challenges-of-accurate-ip-geolocation/
[155] https://www.ipify.org/

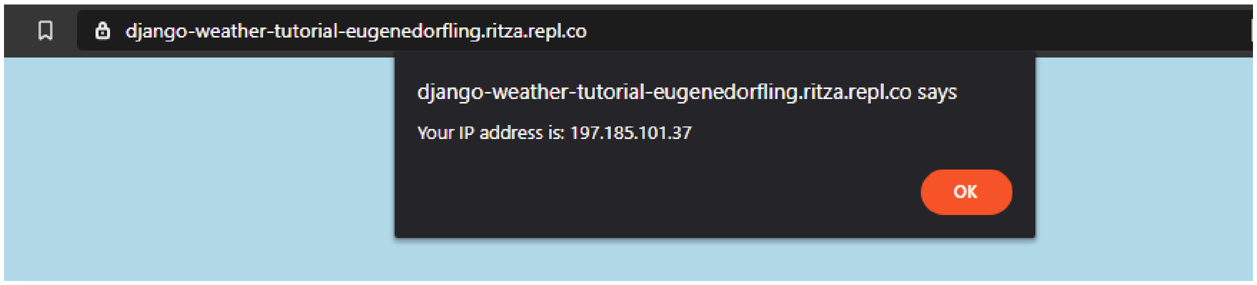
```
14 </body>
15 </html>
```

The "head" section of this template is between lines 5 and 11 – the opening and closing `<head>` tags. We'll add our scripts directly below the `<link ...>` on line 10 and above the closing `</head>` tag on line 11. Modify this part of code to add the following lines:

```
1 <script>
2   function use_ip(json) {
3     alert("Your IP address is: " + json.ip);
4   }
5   </script>
6
7 <script src="https://api.ipify.org?format=jsonp&callback=use_ip"></script>
```

These are two snippets of JavaScript. The first (lines 1-5) is a function that when called will display a pop-up box (an "alert") in our visitor's browser showing their IP address from the json object that we pass in. The second (line 7) loads an external script from ipify's API and asks it to pass data (including our visitor's IP address) along to the `use_ip` function that we provide.

If you open your web app again and refresh the page, you should see this script in action (if you're running an adblocker, it might block the ipify scripts, so try disabling that temporarily if you have any issues).


django-weather-tutorial-eugenedorfling.ritza.repl.co

django-weather-tutorial-eugenedorfling.ritza.repl.co says

Your IP address is: 197.185.101.37

This code doesn't do anything with the IP address except display it to the user, but it is enough to see that the first component of our system (getting our user's IP Address) is working. This also introduces the first *dynamic* functionality to our app – before, any visitor would see exactly the same thing, but now we can show each visitor something related specifically to them (no two people have the same IP address).

Now instead of simply showing this IP address to our visitor, we'll modify the code to rather pass it along to our Repl webserver (the "backend" of our application), so that we can use it to fetch location information through a different service.

Adding a new route and view, and passing data

Currently our Django application only has two routes, the default (`admin/`) route and our home route (`/`) which is loaded as our home page. We'll add another route at `/get_weather_from_ip` where we can pass an IP address to our application to detect the location and get a current weather report.

To do this, we'll need to modify the files at `mysite/views.py` and `mysite/urls.py`.

Edit `urls.py` to look as follows (add line 8, but you shouldn't need to change anything else).

```
 1  from django.contrib import admin
 2  from django.urls import path
 3  from . import views
 4
 5  urlpatterns = [
 6                  path('', views.home, name=home),
 7      path('admin/', admin.site.urls),
 8      path('get_weather_from_ip/', views.get_weather_from_ip, name="get_weather_from_i\
 9  p"),
10  ]
```

We've added a definition for the `get_weather_from_ip` route, telling our app that if anyone visits https://django-weather-tutorial-eugenedorfling.ritza.repl.co/get_weather_from_ip[156] then we should trigger a function in our `views.py` file that is also called `get_weather_from_ip`. Let's write that function now.

In your `views.py` file, add a `get_weather_from_ip()` function beneath the existing `home()` one, and add an import for JsonResponse on line 2. Your whole `views.py` file should now look like this:

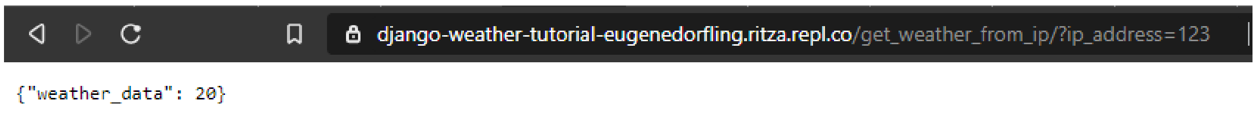
```
 1  from django.shortcuts import render
 2  from django.http import JsonResponse
 3
 4
 5  # Create your views here.
 6  def home(request):
 7      return render(request, 'index.html')
 8
 9  def get_weather_from_ip(request):
10    print(request.GET.get("ip_address"))
11    data = {"weather_data": 20}
12    return JsonResponse(data)
```

[156] https://django-weather-tutorial-eugenedorfling.ritza.repl.co/get_weather_from_ip

By default, Django passes a `request` argument to all views. This is an object that contains information about our user and the connection, and any additional arguments passed in the URL. As our application isn't connected to any weather services yet, we'll just make up a temperature (20) and pass that back to our user as JSON.

In line 10, we print out the IP address that we will pass along to this route from the `GET` arguments (we'll look at how to use this later). We then create the fake data (which we'll later replace with real data) and return a JSON response (the data in a format that a computer can read more easily, with no formatting). We return JSON instead of HTML because our system is going to use this route internally to pass data between the front and back ends of our application, but we don't expect our users to use this directly.

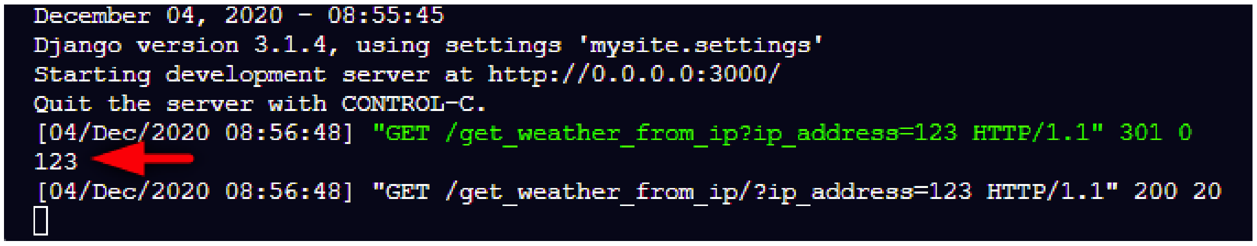
To test this part of our system, open your web application in a new tab and add `/get_weather_-from_ip?ip_address=123` to the URL. Here, we're asking our system to fetch weather data for the IP address `123` (not a real IP address). In your browser, you'll see the fake weather data displayed in a format that can easily be programmatically parsed.


django-weather-tutorial-eugenedorfling.ritza.repl.co/get_weather_from_ip/?ip_address=123

```
{"weather_data": 20}
```

Viewing the fake JSON data

In our Repl's output, we can see that the backend of our application has found the "IP address" and printed it out, between some other outputs telling us which routes are being visited and which port our server is running on:



```
December 04, 2020 - 08:55:45
Django version 3.1.4, using settings 'mysite.settings'
Starting development server at http://0.0.0.0:3000/
Quit the server with CONTROL-C.
[04/Dec/2020 08:56:48] "GET /get_weather_from_ip?ip_address=123 HTTP/1.1" 301 0
123
[04/Dec/2020 08:56:48] "GET /get_weather_from_ip/?ip_address=123 HTTP/1.1" 200 20
```

Django print output of fake IP

The steps that remain now are to:

- pass the user's real IP address to our new route in the background when the user visits our main page
- add more backend logic to fetch the user's location from the IP address
- add logic to fetch the user's weather from their location
- display this data to the user.

Let's start by using Ajax[157] to pass the user's IP address that we collected before to our new route,

[157]https://en.wikipedia.org/wiki/Ajax_(programming)

without our user having to explicitly visit the `get_weather_from_ip` endpoint or refresh their page.

Calling a Django route using Ajax and jQuery

We'll use Ajax through jQuery[158] to do a "partial page refresh" – that is, to update part of the page the user is seeing by sending new data from our backend code without the user needing to reload the page.

To do this, we need to include jQuery as a library.

> Note: usually you wouldn't add JavaScript directly to your base.html template, but to keep things simpler and to avoid creating too many files, we'll be diverging from some good practices. See the Django documentation[159] for some guidance on how to structure JavaScript properly in larger projects.

In your `templates/base.html` file, add the following script above the line where we previously defined the `use_ip()` function.

```
<script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
  integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
  crossorigin="anonymous"></script>
```

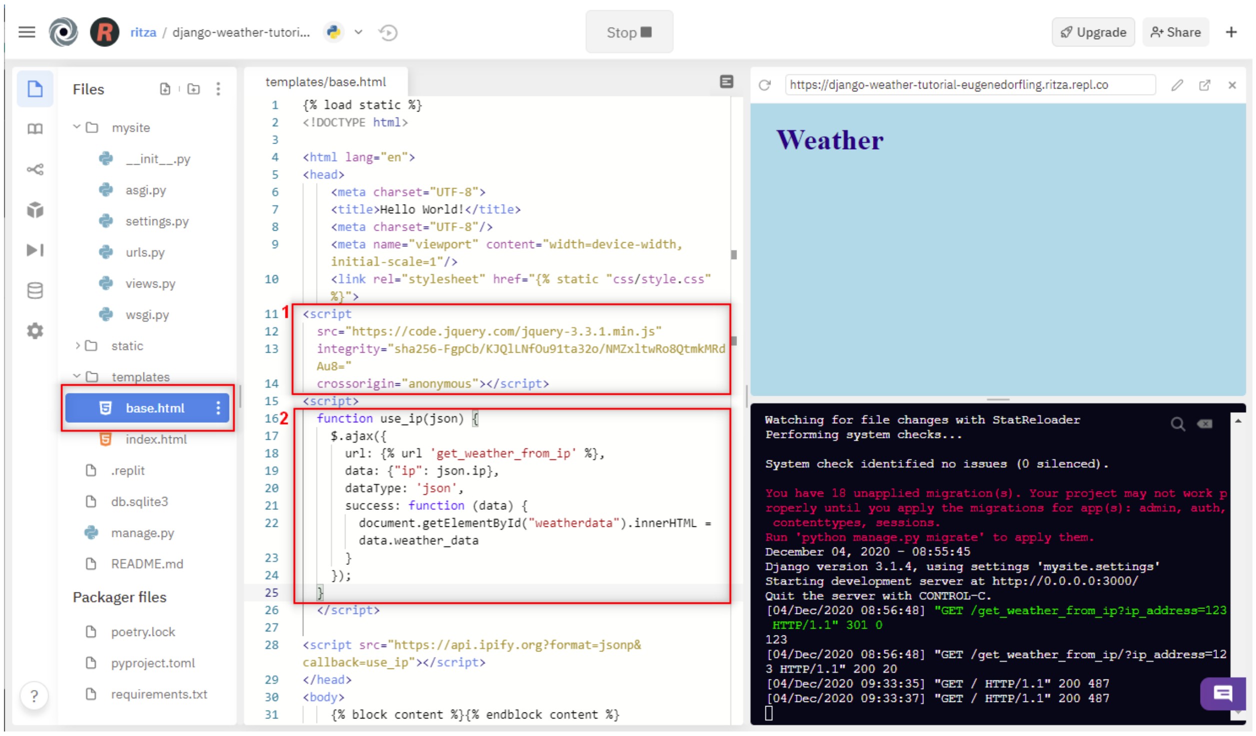
This loads the entire jQuery library from a CDN[160], allowing us to complete certain tasks using fewer lines of JavaScript.

Now, modify the `use_ip()` script that we wrote before to call our backend route using Ajax. The new `use_ip()` function should be as follows:

```
  function use_ip(json) {
    $.ajax({
      url: {% url 'get_weather_from_ip' %},
      data: {"ip": json.ip},
      dataType: 'json',
      success: function (data) {
        document.getElementById("weatherdata").innerHTML = data.weather_data
      }
    });
  }
```

[158]http://api.jquery.com/jquery.ajax/
[159]https://docs.djangoproject.com/en/2.0/howto/static-files/
[160]https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

Our new `use_ip()`function makes an asynchronous[161] call to our `get_weather_from_ip` route, sending along the IP address that we previously displayed in a pop-up box. If the call is successful, we call a new function (in the `success:` section) with the returned data. This new function (line 7) looks for an HTML element with the ID of `weatherdata` and replaces the contents with the `weather_data` attribute of the response that we received from `get_weather_from_ip` (which at the moment is still hardcoded to be "20").



```
{% load static %}
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width,
    initial-scale=1"/>
    <link rel="stylesheet" href="{% static "css/style.css"
    %}">
<script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
  integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRd
  Au8="
  crossorigin="anonymous"></script>
<script>
  function use_ip(json) {
    $.ajax({
      url: {% url 'get_weather_from_ip' %},
      data: {"ip": json.ip},
      dataType: 'json',
      success: function (data) {
        document.getElementById("weatherdata").innerHTML =
        data.weather_data
      }
    });
  }
  </script>

<script src="https://api.ipify.org?format=jsonp&
callback=use_ip"></script>
</head>
<body>
    {% block content %}{% endblock content %}
```

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work p
roperly until you apply the migrations for app(s): admin, auth,
 contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
December 04, 2020 - 08:55:45
Django version 3.1.4, using settings 'mysite.settings'
Starting development server at http://0.0.0.0:3000/
Quit the server with CONTROL-C.
[04/Dec/2020 08:56:48] "GET /get_weather_from_ip?ip_address=123
 HTTP/1.1" 301 0
123
[04/Dec/2020 08:56:48] "GET /get_weather_from_ip/?ip_address=12
3 HTTP/1.1" 200 20
[04/Dec/2020 09:33:35] "GET / HTTP/1.1" 200 487
[04/Dec/2020 09:33:37] "GET / HTTP/1.1" 200 487
```

To see the results, we'll need to add an HTML element as a placeholder with the id `weatherdata`. Do this in the `templates/index.html` file as follows.

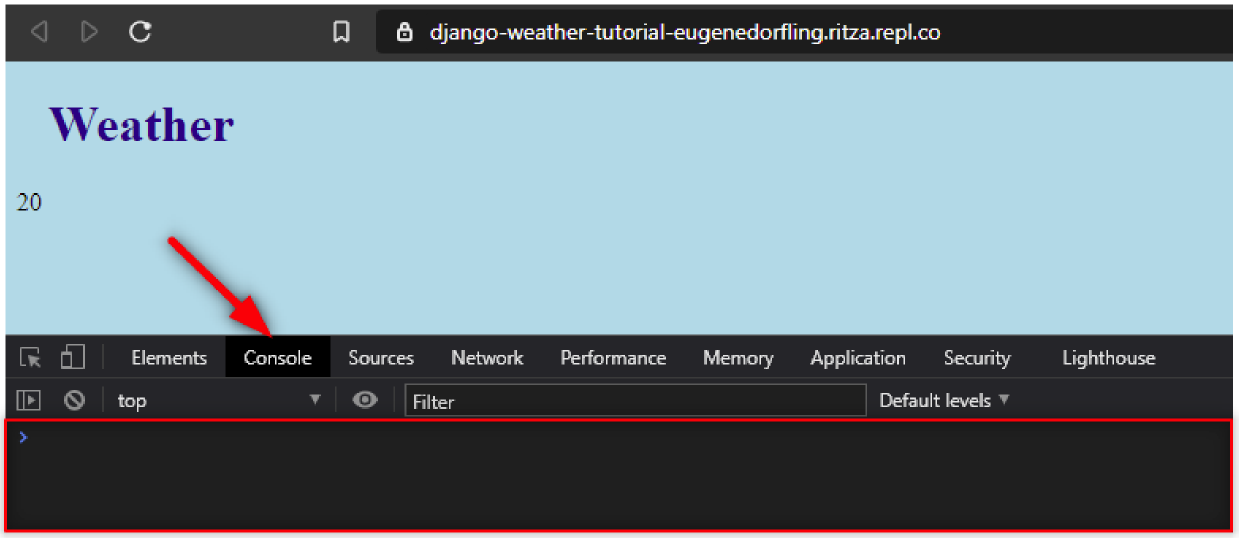
```
{% extends "base.html" %}

{% block content %}
  <h1>Weather</h1>
  <p id=weatherdata></p>
{% endblock %}
```

This adds an empty HTML *paragraph* element which our JavaScript can populate once it has the required data.

Now reload the app and you should see our fake `20` being displayed to the user. If you don't see what you expect, open up your browser's developer tools for Chrome[162] and Firefox[163]) and have a look

[161]http://api.jquery.com/jquery.ajax/
[162]https://developers.google.com/web/tools/chrome-devtools/
[163]https://developer.mozilla.org/son/docs/Tools

at the Console section for any JavaScript errors. A clean console (with no errors) is shown below.



Now it's time to change out our mock data for real data by calling two services backend – the first to get the user's location from their IP address and the second to fetch the weather for that location.

Using ip-api.com for geolocation

The service at ip-api.com[164] is very simple to use. To get the country and city from an IP address we only need to make one web call. We'll use the python `requests` library for this, so first we'll have to add an import for this to our `views.py` file, and then write a function that can translate IP addresses to location information. Add the following import to your`views.py` file:

```
import requests
```

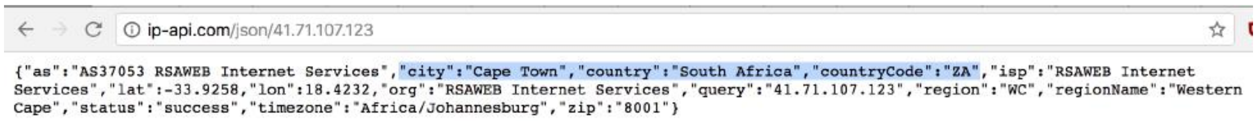
and above the `get_weather_from_ip()` function, add the`get_location_from_ip()` function as follows:

```
def get_location_from_ip(ip_address):
    response = requests.get("http://ip-api.com/json/{}".format(ip_address))
    return response.json()
```

> Note: again we are diverging from best practice in the name of simplicity. Usually whenever you write any code that relies on networking (as above), you should add exception handling[165] so that your code can fail more gracefully if there are problems.

You can see the response that we'll be getting from this service by trying it out in your browser. Visit http://ip-api.com/json/41.71.107.123[166] to see the JSON response for that specific IP address.

[164]http://ip-api.com
[165]https://docs.python.org/3/tutorial/errors.html
[166]http://ip-api.com/json/41.71.107.123



ip-api.com/json/41.71.107.123

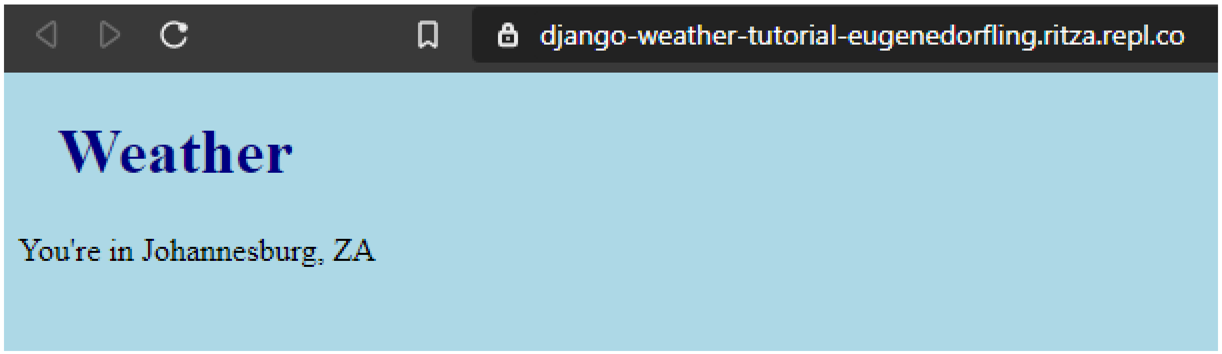
{"as":"AS37053 RSAWEB Internet Services","city":"Cape Town","country":"South Africa","countryCode":"ZA","isp":"RSAWEB Internet Services","lat":-33.9258,"lon":18.4232,"org":"RSAWEB Internet Services","query":"41.71.107.123","region":"WC","regionName":"Western Cape","status":"success","timezone":"Africa/Johannesburg","zip":"8001"}

Take a look specifically at the highlighted location information that we'll need to extract to pass on to a weather service.

Before we set up the weather component, let's display the user's current location data instead of the hardcoded temperature that we had before. Change the `get_weather_from_ip()` function to call our new function and pass along some useful data as follows:

```python
def get_weather_from_ip(request):
  ip_address = request.GET.get("ip")
  location = get_location_from_ip(ip_address)
  city = location.get("city")
  country_code = location.get("countryCode")
  s = "You're in {}, {}".format(city, country_code)
  data = {"weather_data": s}
  return JsonResponse(data)
```

Now, instead of just printing the IP address that we get sent and making up some weather data, we use the IP address to guess the user's location, and pass the city and country code back to the template to be displayed. If you reload your app again, you should see something similar to the following (though hopefully with your location instead of mine).



Weather

You're in Johannesburg, ZA

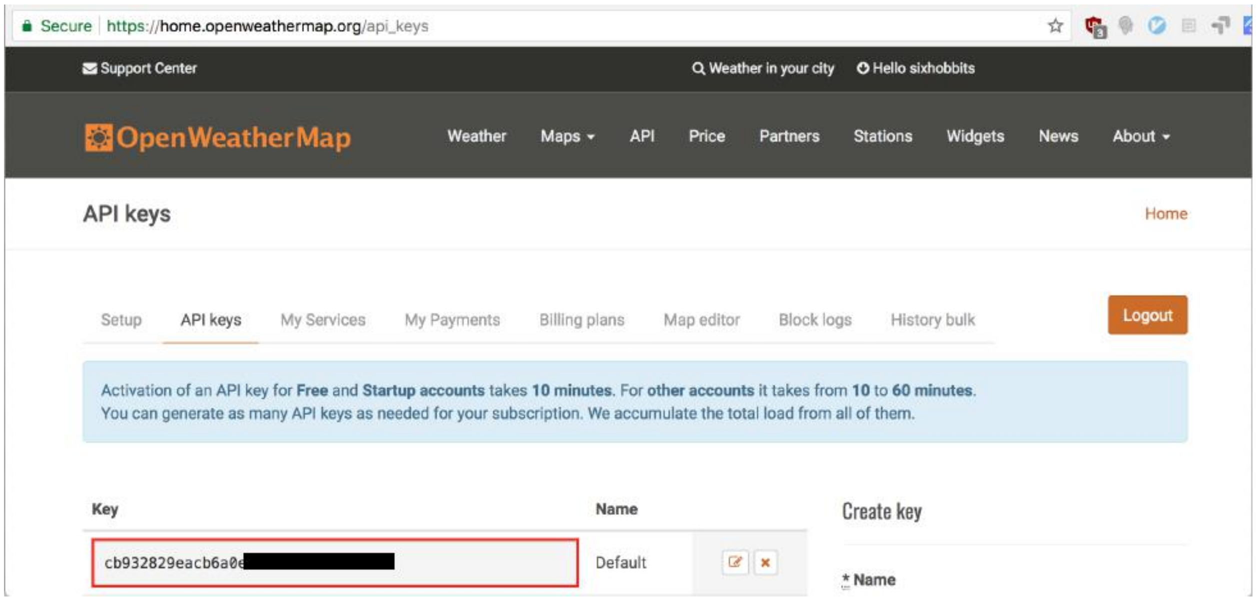
weather app, location showing

That's the location component of our app done and dusted – let's move on to getting weather data for that location now.

Getting weather data from OpenWeatherMap

To get weather data automatically from OpenWeatherMap[167], you'll need an API Key. This is a unique string that OpenWeatherMap gives to each user of their service and it's used mainly to restrict how many calls each person can make in a specified period. Luckily, OpenWeatherMap provides a generous "free" allowance of calls, so we won't need to spend any money to build our app. Unfortunately, this allowance is not quite generous enough to allow me to share my key with every reader of this tutorial, so you'll need to sign up for your own account and generate your own key.

Visit openweathermap.org[168], hit the "sign up" button, and register for the service by giving them an email address and choosing a password. Then navigate to the API Keys[169] section and note down your unique API key (or copy it to your clipboard).



Activation of an API key for **Free** and **Startup accounts** takes **10 minutes**. For **other accounts** it takes from **10** to **60 minutes**.
You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key	Name
cb932829eacb6a0e[redacted]	Default

OpenWeatherMap API Key page

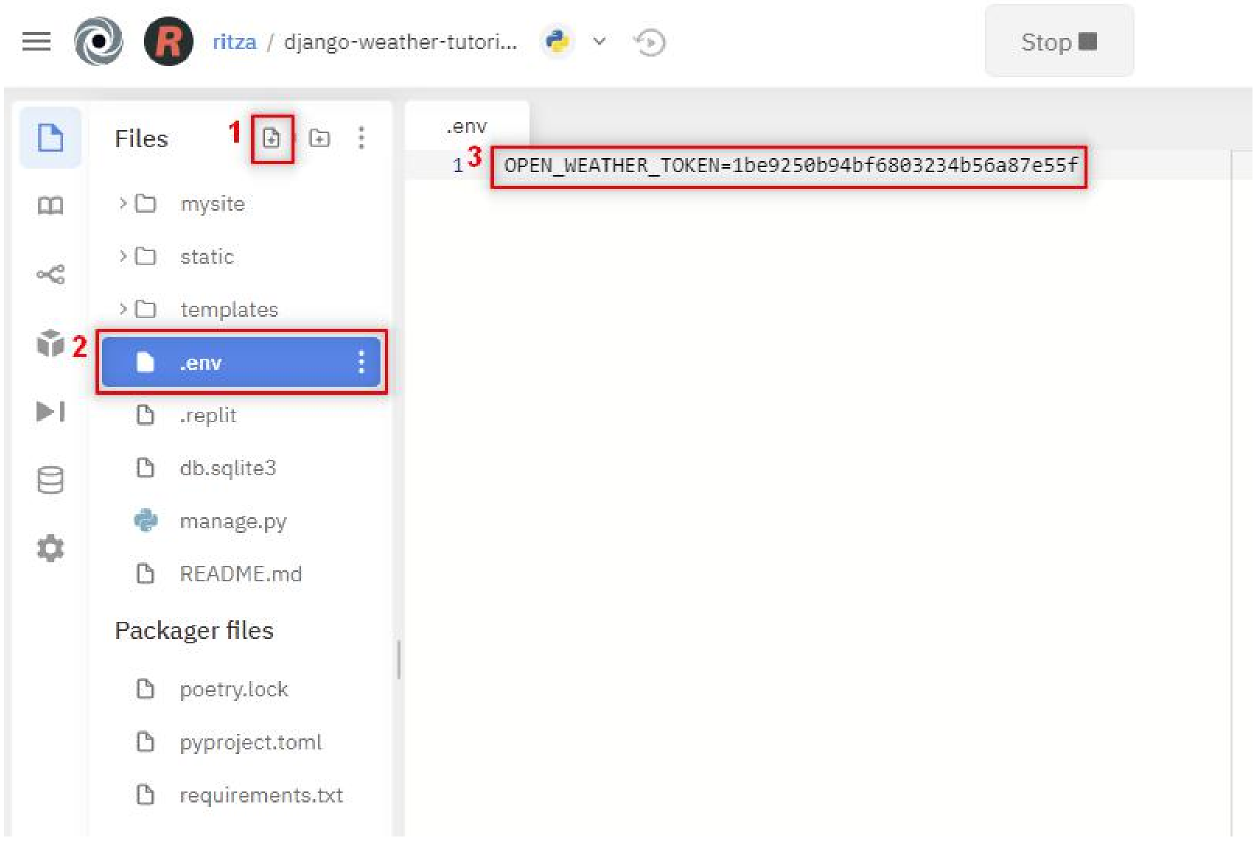
This key is a bit like a password – when we use OpenWeatherMap's service, we'll always send along this key to indicate that it's us making the call. Because Repl.it's projects are public by default, we'll need to be careful to keep this key private and prevent other people making too many calls using our OpenWeatherMap quota (potentially making our app fail when OpenWeatherMap starts blocking our calls). Luckily Repl.it provides a neat way of solving this problem using `.env` files[170].

In your project, create a new file using the "New file" button as shown below. Make sure that the file is in the root of your project and that you name the file `.env` (in Linux, starting a filename with a `.` usually indicates that it's a system or configuration file). Inside this file, define the `OPEN_WEATHER_-`

[167]https://openweathermap.org/
[168]https://openweathermap.org/
[169]https://home.openweathermap.org/api_keys
[170]https://repl.it/site/docs/secret-keys

TOKEN variable as follows, but using your own token instead of the fake one below. Make sure not to have a space on either side of the = sign.

```
OPEN_WEATHER_TOKEN=1be9250b94bf6803234b56a87e55f
```



```
OPEN_WEATHER_TOKEN=1be9250b94bf6803234b56a87e55f
```

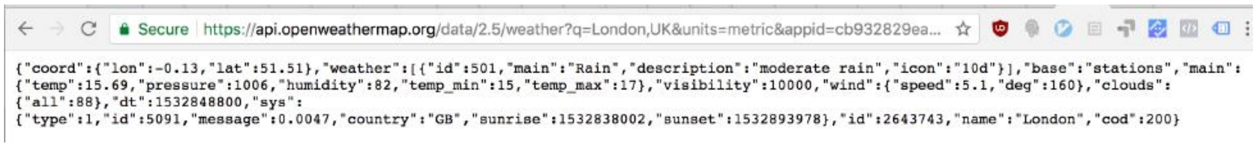
Creating a new file

Repl.it will load the contents of this file into our server's environment variables[171]. We'll be able to access this using the os library in Python, but when other people view or fork our Repl, they won't see the .env file, keeping our API key safe and private.

To fetch weather data, we need to call the OpenWeatherMap api, passing along a search term. To make sure we're getting the city that we want, it's good to pass along the country code as well as the city name. For example, to get the weather in London right now, we can visit (again, you'll need to add your own API key in place of the string after appid=) https://api.openweathermap.org/data/2.5/weather?q=Lond

To test this, you can visit the URL in your browser first. If you prefer Fahrenheit to Celsius, simply change the unit=metric part of the url to units=imperial.

[171]https://wiki.archlinux.org/index.php/environment_variables


Secure https://api.openweathermap.org/data/2.5/weather?q=London,UK&units=metric&appid=cb932829ea...

```
{"coord":{"lon":-0.13,"lat":51.51},"weather":[{"id":501,"main":"Rain","description":"moderate rain","icon":"10d"}],"base":"stations","main":
{"temp":15.69,"pressure":1006,"humidity":82,"temp_min":15,"temp_max":17},"visibility":10000,"wind":{"speed":5.1,"deg":160},"clouds":
{"all":88},"dt":1532848800,"sys":
{"type":1,"id":5091,"message":0.0047,"country":"GB","sunrise":1532838002,"sunset":1532893978},"id":2643743,"name":"London","cod":200}
```

Json response from OpenWeatherMap

Let's write one last function in our `views.py` file to replicate this call for our visitor's city which we previously displayed.

First we need to add an import for the Python `os` (operating system) module so that we can access our environment variables. At the top of `views.py` add:

```python
import os
```

Now we can write the function. Add the following to `views.py`:

```python
1  def get_weather_from_location(city, country_code):
2      token = os.environ.get("OPEN_WEATHER_TOKEN")
3      url = "https://api.openweathermap.org/data/2.5/weather?q={},{}&units=metric&appi\
4  d={}".format(
5          city, country_code, token)
6      response = requests.get(url)
7      return response.json()
```

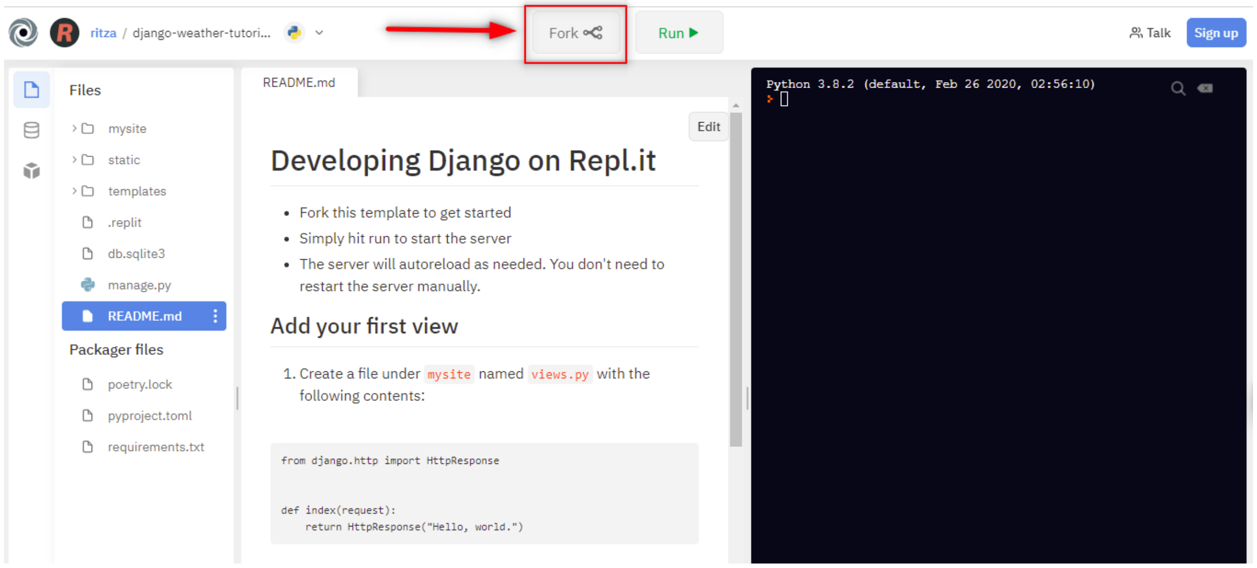
In line 2, we get our API key from the environment variables (note, you sometimes need to refresh the repl.it page with your repl in to properly load in the environment variables), and we then use this to format our URL properly in line 3. We get the response from OpenWeatherMap and return it as json.

We can now use this function in our `get_weather_from_ip()` function by modifying it to look as follows:

```python
def get_weather_from_ip(request):
  ip_address = request.GET.get("ip")
  location = get_location_from_ip(ip_address)
  city = location.get("city")
  country_code = location.get("countryCode")
  weather_data = get_weather_from_location(city, country_code)
  description = weather_data['weather'][0]['description']
  temperature = weather_data['main']['temp']
  s = "You're in {}, {}. You can expect {} with a temperature of {} degrees".format(\
city, country_code, description, temperature)
  data = {"weather_data": s}
  return JsonResponse(data)
```

We now get the weather data in line 6, parse this into a description and temperature in lines 7 and 8, and add this to the string we pass back to our template in line 9. If you reload the page, you should see your location and your weather.

and hit the “Fork” button. If you didn’t create an account at the beginning of this tutorial, you’ll be prompted to create one. (You can even use a lot of Repl functionality without creating an account.)



Developing Django on Repl.it

- Fork this template to get started
- Simply hit run to start the server
- The server will autoreload as needed. You don't need to restart the server manually.

Add your first view

1. Create a file under `mysite` named `views.py` with the following contents:

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world.")
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> 
```

Forking a Repl

If you’re stuck for ideas, some possible extensions are:

- Make the page look nicer by using Bootstrap[172] or another CSS framework in your template files.
- Make the app more customizable by allowing the user to choose their own location if the IP location that we guess is wrong
- Make the app more useful by showing the weather forecast along with the current weather. (This data is also available[173] from Open Weather Map).
- Add other location-related data to the web app such as news, currency conversion, translation, postal codes. See https://github.com/toddmotto/public-apis#geocoding[174] for a nice list of possibilities.

In the next chapter, we’ll be looking at building our own CRM app with NodeJS and Repl.it. This tutorial will also introduce you to setting up a MongoDB database and creating a user interface.

[172]https://getbootstrap.com/
[173]https://openweathermap.org/forecast5
[174]https://github.com/toddmotto/public-apis#geocoding

Building a CRM app with NodeJS, Repl.it, and MongoDB

In this tutorial, we'll use NodeJS on Repl.it, along with a MongoDB database to build a basic CRUD[175] (Create, Read, Update, Delete) CRM[176] (Customer Relationship Management) application. A CRM lets you store information about customers to help you track the status of every customer relationship. This can help businesses keep track of their clients and ultimately increase sales. The application will be able to store and edit customer details, as well as keep notes about them.

This tutorial won't be covering the basics of Node.js, but each line of code will be explained in detail.

Overview and requirements

All of the code will be written and hosted in Repl.it, so you won't need to install any additional software on your computer. In this tutorial, we'll be covering:

- Creating an account on MongoDB Atlas[177]
- Connecting our database to our Repl
- Creating a user interface to insert customer data
- Updating and deleting database entries

By the end of the tutorial, the application you will have created will be able to create, update, and delete documents in a MongoDB database. You will also have used a web application framework called Express[178] and the Pug[179] templating engine.

Creating an account on MongoDB Atlas

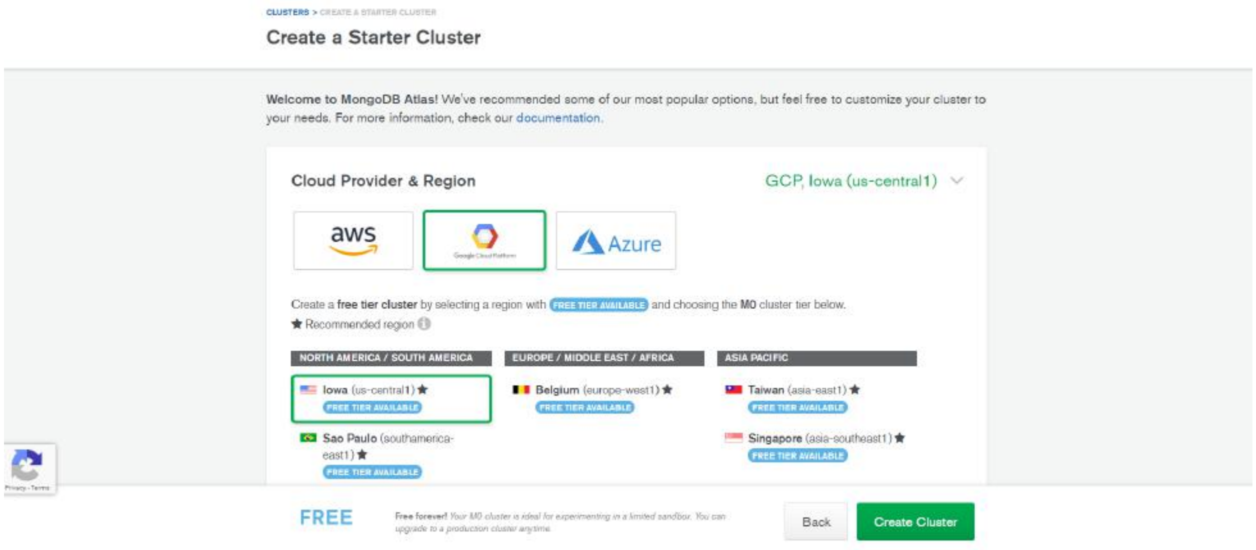
MongoDB Atlas is a fully managed Database-as-a-Service. It provides a document database (often referred to as NoSQL), as opposed to a more traditional relational database like PostgreSQL.

Head over to MongoDB Atlas[180] and hit the "Try free" button. You should then sign up, clicking the "Get started free" button to complete the process.

[175]https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
[176]https://en.wikipedia.org/wiki/Customer_relationship_management
[177]https://www.mongodb.com/cloud/atlas
[178]https://expressjs.com/
[179]https://pugjs.org/api/getting-started.html
[180]https://www.mongodb.com/cloud/atlas

After signing up, under “Shared Clusters”, press the “Create a Cluster” button.

You now have to select a provider and a region. For the purposes of this tutorial, we chose Google Cloud Platform as the provider and Iowa (us-central1) as the region, although it should work regardless of the provider and region.



CLUSTERS > CREATE A STARTER CLUSTER

Create a Starter Cluster

Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our documentation.

Cloud Provider & Region: GCP, Iowa (us-central1)

Create a **free tier cluster** by selecting a region with FREE TIER AVAILABLE and choosing the **M0** cluster tier below.

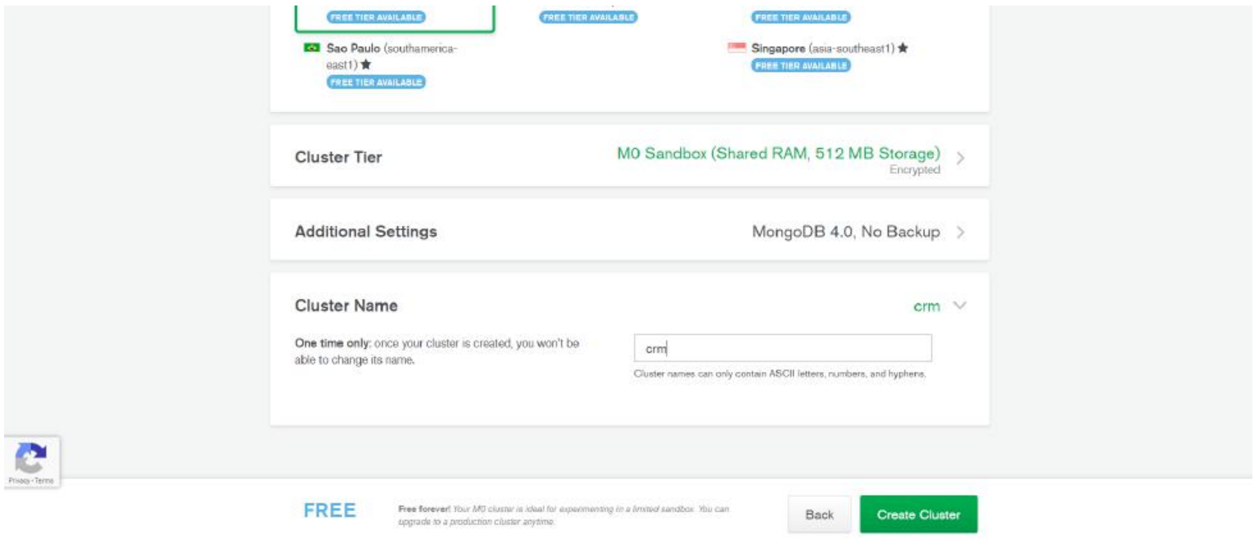
★ Recommended region

NORTH AMERICA / SOUTH AMERICA	EUROPE / MIDDLE EAST / AFRICA	ASIA PACIFIC
Iowa (us-central1) ★ FREE TIER AVAILABLE	Belgium (europe-west1) ★ FREE TIER AVAILABLE	Taiwan (asia-east1) ★ FREE TIER AVAILABLE
Sao Paulo (southamerica-east1) ★ FREE TIER AVAILABLE		Singapore (asia-southeast1) ★ FREE TIER AVAILABLE

FREE — **Free forever!** Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Image: 1 ***Cluster Region***

Under “Cluster Name” you can change the name of your cluster. Note that you can only change the name now - it can’t be changed once the cluster is created. After you’ve done that, click “Create Cluster”.



Sao Paulo (southamerica-east1) ★ FREE TIER AVAILABLE

Singapore (asia-southeast1) ★ FREE TIER AVAILABLE

Cluster Tier: M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted

Additional Settings: MongoDB 4.0, No Backup

Cluster Name: crm

One time only: once your cluster is created, you won't be able to change its name.

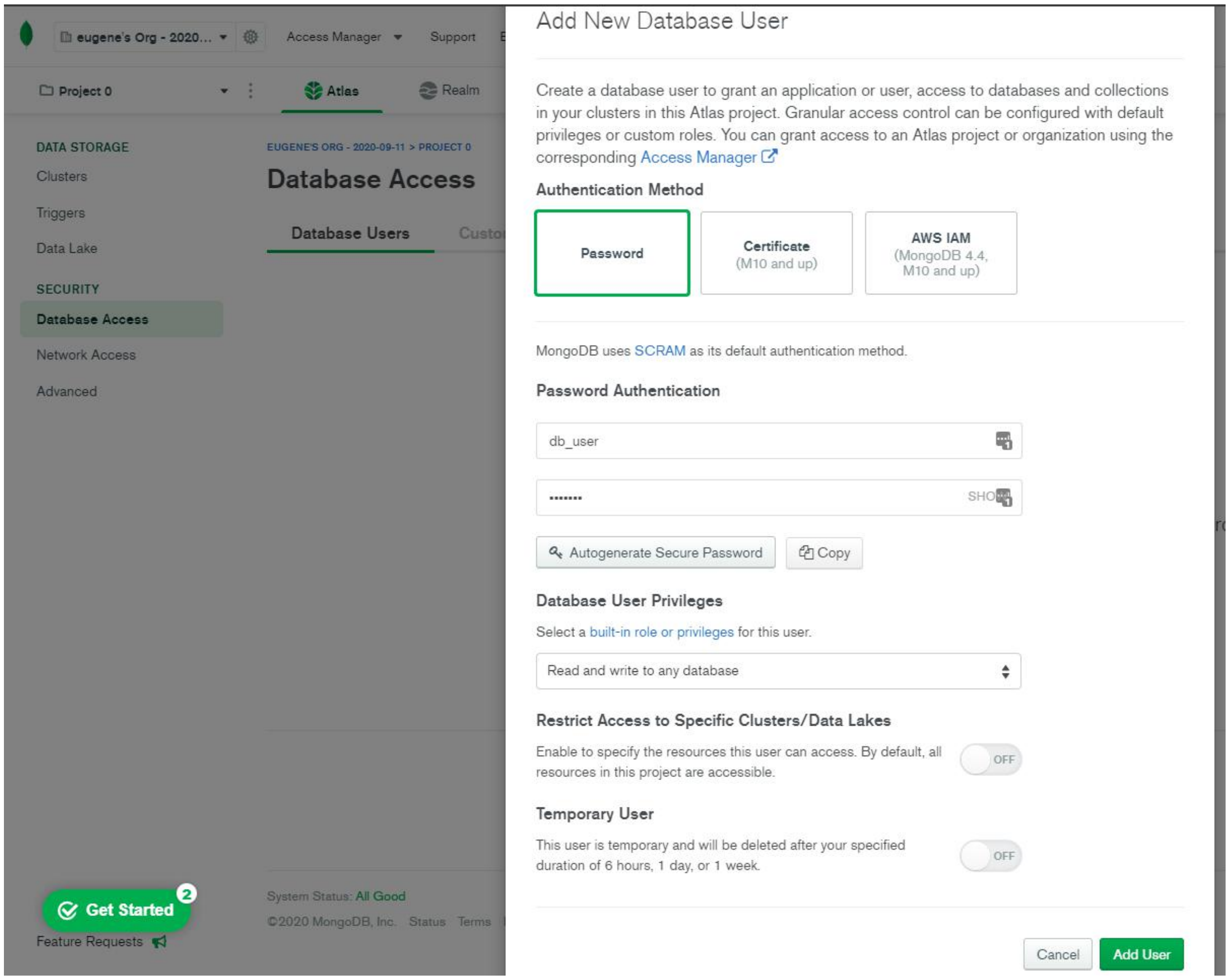
crm

Cluster names can only contain ASCII letters, numbers, and hyphens.

FREE — **Free forever!** Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Image: 2 ***Cluster Name***

After a bit of time, your cluster will be created. Once it’s available, click on “Database Access” under the Security heading in the left-hand column and then click “Add New Database User”. You need a database user to actually store and retrieve data. Enter a username and password for the user and make a note of those details - you'll need them later. Select "Read and write to any database" as the user privilege. Hit "Add User" to complete this step.



Add New Database User

Create a database user to grant an application or user, access to databases and collections in your clusters in this Atlas project. Granular access control can be configured with default privileges or custom roles. You can grant access to an Atlas project or organization using the corresponding Access Manager

Authentication Method

Password

MongoDB uses SCRAM as its default authentication method.

Password Authentication

db_user

•••••••

Database User Privileges

Select a built-in role or privileges for this user.

Read and write to any database

Restrict Access to Specific Clusters/Data Lakes

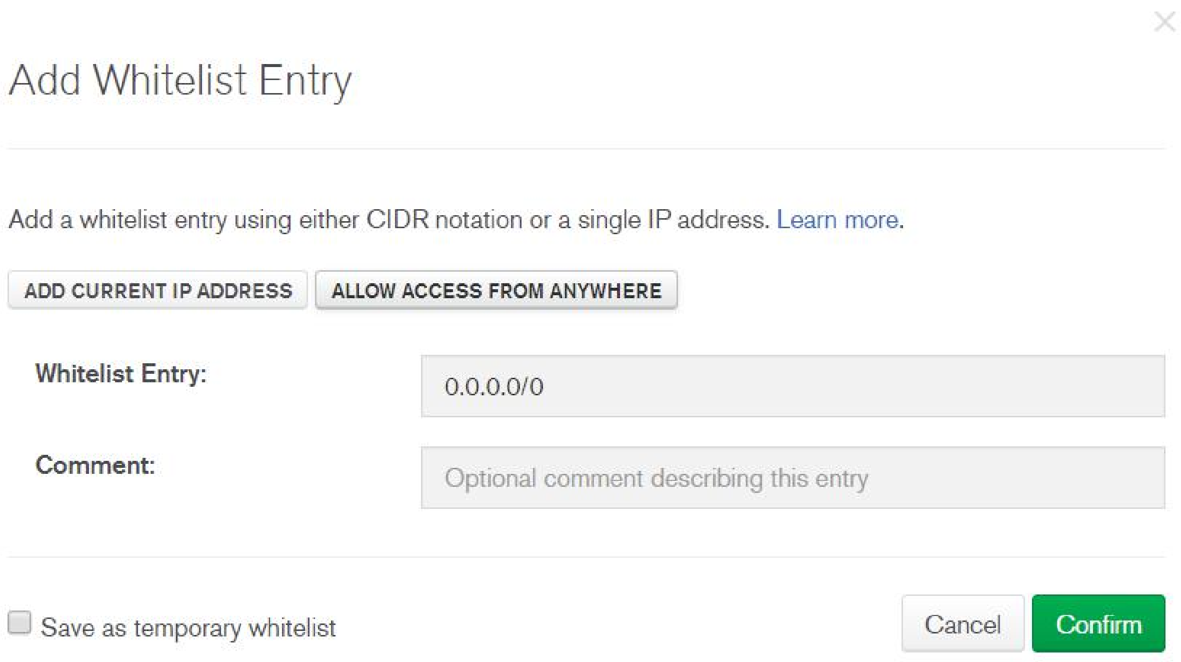
Enable to specify the resources this user can access. By default, all resources in this project are accessible.

Temporary User

This user is temporary and will be deleted after your specified duration of 6 hours, 1 day, or 1 week.

Image: 3 ***Adding a New Database User***

Next, you need to allow network access to the database. Click on "Network Access" in the left-hand column, and "Add IP Address". Because we won't have a static IP from Repl.it, we're just going to allow access from anywhere - don't worry, the database is still secured with the username and password you created earlier. In the popup, click "Allow Access From Anywhere" and then "Confirm".


Add Whitelist Entry

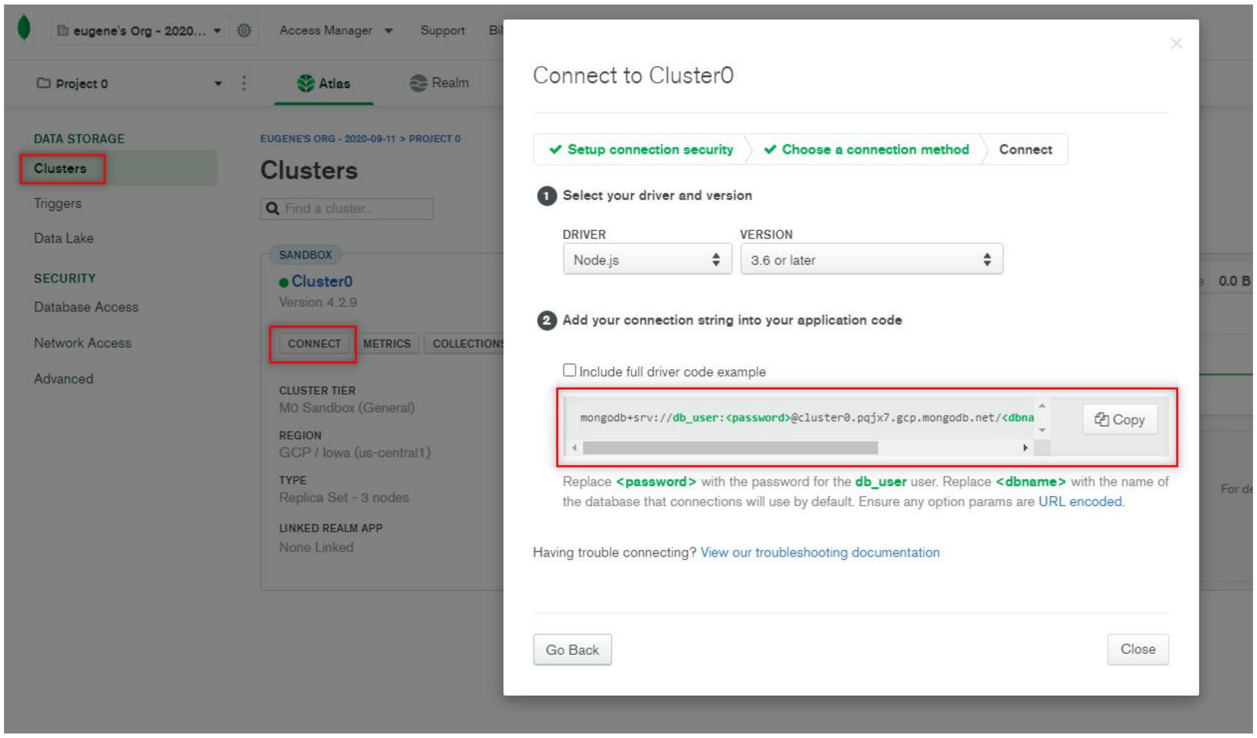
Add a whitelist entry using either CIDR notation or a single IP address. Learn more.

Whitelist Entry: 0.0.0.0/0

Comment: Optional comment describing this entry

Image: 4 ***Allow Access From Anywhere***

Now select "Clusters", under "Data Storage" in the left-hand column. Click on "Connect" and select "Connect Your Application". This will change the pop-up view. Copy the "Connection String" as you will need it shortly to connect to your database from Repl.it. It will look something like this: `mongodb+srv://<username>:<password>@cluster0-zrtwi.gcp.mongodb.net/test?retryWrites=true&w=majority`



```
mongodb+srv://db_user:<password>@cluster0.pqjx7.gcp.mongodb.net/<dbna
```

Replace <password> with the password for the db_user user. Replace <dbname> with the name of the database that connections will use by default. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

Image: 5 ***Retrieve Your Connection String***

Creating a Repl and connecting to our Database

First, we need to create a new Node.js Repl to write the code necessary to connect to our shiny new Database. Navigate to repl.it and create a new Repl, selecting "Node.js" as the language.

A great thing about Repl is that it makes projects public by default. This makes it easy to share and is great for collaboration and learning, but we have to be careful not to make our database credentials available on the open Internet.

To solve this problem, we'll be using `environment variables`, as we have done in previous tutorials. We'll create a special file that Repl.it recognizes and keeps private for you, and in that file we declare variables that become part of our Repl.it development environment and are accessible in our code.

In your Repl, create a file called `.env` by selecting "Files" in the left-hand pane and then clicking the "Add File" button. Note that the spelling has to be exact or the file will not be recognized. Add your MongoDB database username and password (not your login details to MongoDB Atlas) into the file in the below format:

```
MONGO_USERNAME=username
MONGO_PASSWORD=password
```

- **Replace** `username` and `password` with your database username and password

- **Spacing matters**. Make sure that you don't add any spaces before or after the = sign

Now that we have credentials set up for the database, we can move on to connecting to it in our code.

MongoDB is kind enough to provide a client that we can use. To test out our database connection, we're going to insert some customer data into our database. In your `index.js` file (created automatically and found under the Files pane), add the following code:

```
1 const MongoClient = require('mongodb').MongoClient;
2 const mongo_username = process.env.MONGO_USERNAME
3 const mongo_password = process.env.MONGO_PASSWORD
4 
5 const uri = `mongodb+srv://${mongo_username}:${mongo_password}@cluster0-zrtwi.gcp.mo\
6 ngodb.net/crmdb?retryWrites=true&w=majority`;
7 const client = new MongoClient(uri, { useNewUrlParser: true });
```

Let's break this down to see what is going on and what we still need to change:

- **Line 1** adds the dependency for the MongoDB Client. As we have discussed before, Repl.it makes things easy by installing all the dependencies for us, so we don't have to use something like npm to do it manually.
- **Line 2 & 3** we retrieve our MongoDB username and password from the environment variables that we set up earlier.
- **Line 5** has a few very important details that we need to get right.
  - Replace the section between the `@` and the next `/` with the same section of your connection string from MongoDB that we copied earlier. You may notice the `${mongo_username}` and `${mongo_password}` before and after the colon near the beginning of the string. These are called Template Literals. Template Literals allow us to put variables in a string, which Node.js will then helpfully replace with the actual values of the variables.
  - Note `crmdb` after the `/` and before the `?`. This will be the name of the database that we will be using. MongoDB creates the database if it doesn't exist for us. You can change this to whatever you want to name the database, but remember what you changed it to for future sections of this tutorial.
- **Line 6** creates the client that we will use to connect to the database.

Making a user interface to insert customer data

We're going to make an HTML form that will capture the customer data and send it to our Repl.it code, which will then insert it into our database.

In order to actually present and handle an HTML form, we need a way to process HTTP GET and POST requests. The easiest way to do this is to use a web application framework. A web application framework is designed to support the development of web applications - it gives you a standard way to build your application and lets you get to building your application fast without having to do the boilerplate code.

A really simple, fast and flexible Node.js web application framework is Express[181], which provides a robust set of features for the development of web applications.

The first thing we need to do is add the dependencies we need. Right at the top of your index.js file (above the MongoDB code), add the following lines:

```
1 let express = require('express');
2 let app = express();
3 let bodyParser = require('body-parser');
4 let http = require('http').Server(app);
5
6 app.use(bodyParser.json())
7 app.use(bodyParser.urlencoded({ extended: true }));
```

Let's break this down.

- **Line 1** adds the dependency for Express. Repl.it will take care of installing it for us.
- **Line 2** creates a new Express app that will be needed to handle incoming requests.
- **Line 3** adds a dependency for 'body-parser'. This is needed for the Express server to be able to handle the data that the form will send, and give it to us in a useful format to use in the code.
- **Line 4** adds a dependency for a basic HTTP server.
- **Line 6 & 7** tell the Express app which parsers to use on incoming data. This is needed to handle form data.

Next, we need to add a way for the Express to handle an incoming request and give us the form that we want. Add the following lines of code below that which you just added:

```
app.get('/', function (req, res) {
  res.sendFile('/index.html', {root:'.'});
});

app.get('/create', function (req, res) {
  res.sendFile('/create.html', {root:'.'});
});
```

- `app.get` tells Express that we want it to handle a GET request.

[181] https://expressjs.com/

- `'/'` tells Express that it should respond to GET requests sent to the root URL. A root URL looks something like 'https://crm.hawkiesza.repl.co' - note that there are no slashes after the URL.
- `'/create'` tells Express that it should respond to GET requests to /create after the root URL i.e. 'https://crm.hawkiesza.repl.co/create'
- `res.sendFile` tells Express to send the given file as a response.

Before the server will start receiving requests and sending responses, we need to tell it to run. Add the following code below the previous line.

```
1 app.set('port', process.env.PORT || 5000);
2 http.listen(app.get('port'), function() {
3     console.log('listening on port', app.get('port'));
4 });
```

- **Line 1** tells Express to set the port number to either a number defined as an environment variable, or 5000 if no definition was made.
- **Line 2-4** tells the server to start listening for requests.

Now we have an Express server listening for requests, but we haven't yet built the form that it needs to send back if it receives a request.

Make a new file called `index.html` and paste the following code into it:

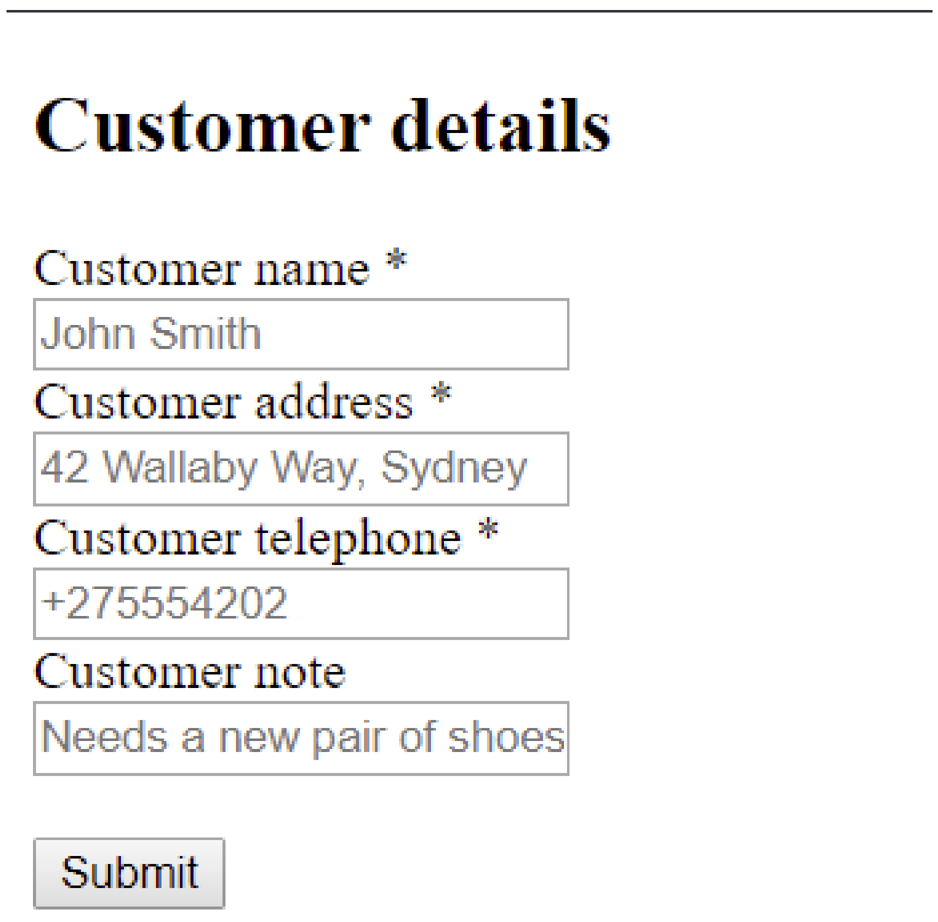
```
<!DOCTYPE html>
<html>
<body>
<form action="/create" method="GET">
  <input type="submit" value="Create">
</form>

</body>
</html>
```

This is just a simple bit of HTML that puts a single button on the page. When this button is clicked it sends a GET request to `/create`, which the server will then respond to according to the code that we wrote above - in our case it will send back the `create.html` file which we will define now.

Make a new file called `create.html` and paste the following into it:

```html
<!DOCTYPE html>
<html>
<body>

<h2>Customer details</h2>

<form action="/create" method="POST">
  <label for="name" >Customer name *</label><br>
  <input type="text" id="name" name="name" class="textInput" placeholder="John Smith\
" required>
  <br>
  <label for="address" >Customer address *</label><br>
  <input type="text" name="address" class="textInput" placeholder="42 Wallaby Way, S\
ydney" required>
  <br>
  <label for="telephone" >Customer telephone *</label><br>
  <input type="text" name="telephone" class="textInput" placeholder="+275554202" req\
uired>
  <br>
  <label for="note" >Customer note</label><br>
  <input type="text" name="note" class="textInput" placeholder="Needs a new pair of \
shoes">
  <br><br>
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

We won't go in-depth into the above HTML. It is a very basic form with 4 fields (name, address, telephone, note) and a Submit button, which creates an interface that will look like the one below.


Customer details

Customer name *

John Smith

Customer address *

42 Wallaby Way, Sydney

Customer telephone *

+275554202

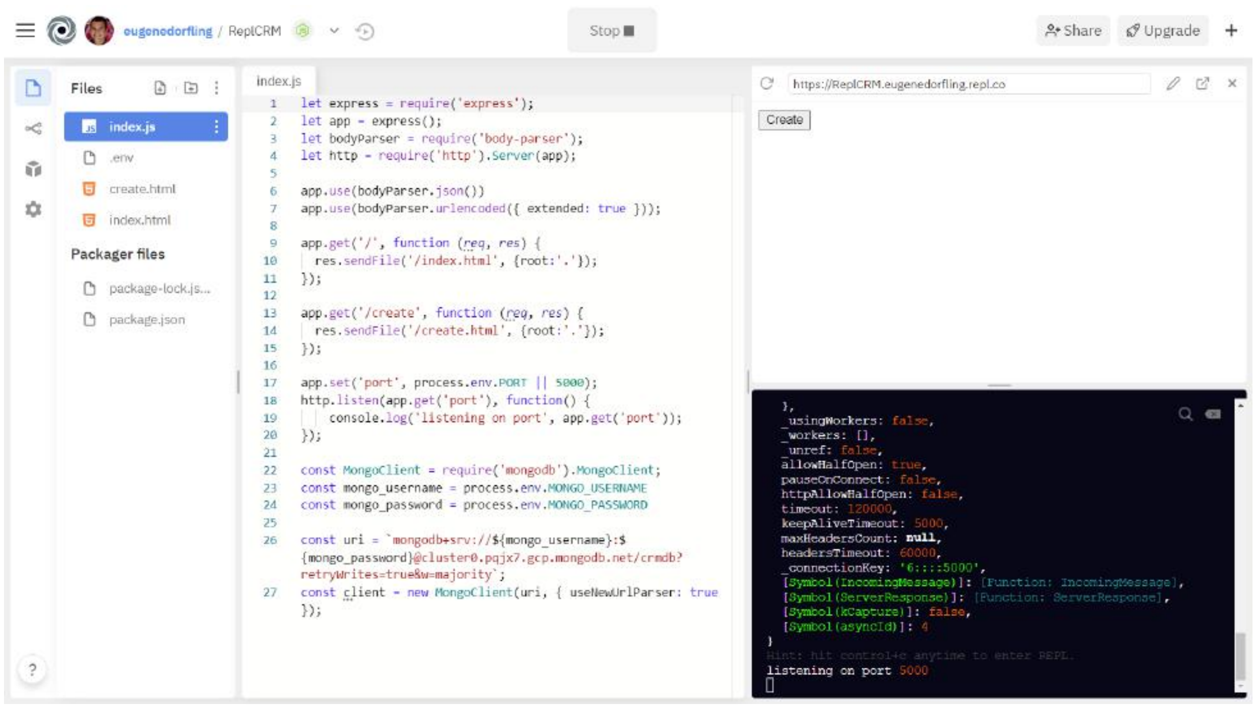
Customer note

Needs a new pair of shoes

Image: 6 ***Customer Details***

When the user presses the submit button a POST request is made to `/create` with the data in the form - we still have to handle this request in our code as we're currently only handling a GET request to `/`.

If you now start up your application (click the "run" button) a new window should appear on the right that displays the "create" button we defined just now in "create.html". You can also navigate to `https://<repl_name>.<your_username>.repl.co` (replace <repl_name> with whatever you named your Repl (but with no underscores or spaces) and <your_username> with your Repl username) to see the form. You will be able to see this URL in your Repl itself.

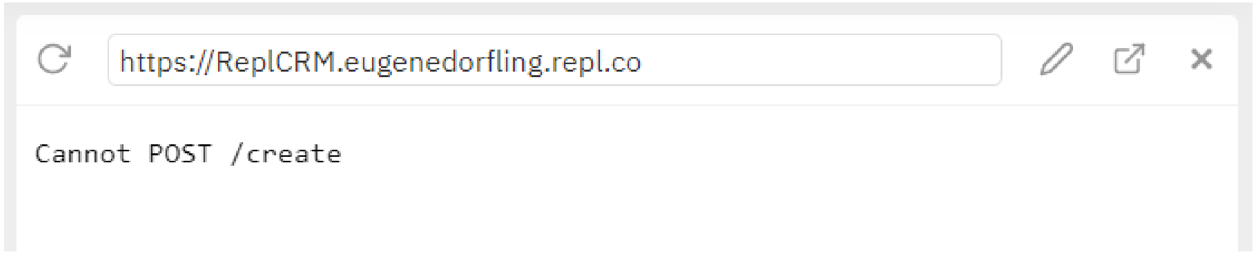


```javascript
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let http = require('http').Server(app);

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', function (req, res) {
  res.sendFile('/index.html', {root:'.'});
});

app.get('/create', function (req, res) {
  res.sendFile('/create.html', {root:'.'});
});

app.set('port', process.env.PORT || 5000);
http.listen(app.get('port'), function() {
    console.log('listening on port', app.get('port'));
});

const MongoClient = require('mongodb').MongoClient;
const mongo_username = process.env.MONGO_USERNAME
const mongo_password = process.env.MONGO_PASSWORD

const uri = `mongodb+srv://${mongo_username}:$
{mongo_password}@cluster0.pqjx7.gcp.mongodb.net/crmdb?
retryWrites=true&w=majority`;
const client = new MongoClient(uri, { useNewUrlParser: true
});
```

```
  },
  _usingWorkers: false,
  _workers: [],
  _unref: false,
  allowHalfOpen: true,
  pauseOnConnect: false,
  httpAllowHalfOpen: false,
  timeout: 120000,
  keepAliveTimeout: 5000,
  maxHeadersCount: null,
  headersTimeout: 60000,
  _connectionKey: '6::::5000',
  [Symbol(IncomingMessage)]: [Function: IncomingMessage],
  [Symbol(ServerResponse)]: [Function: ServerResponse],
  [Symbol(kCapture)]: false,
  [Symbol(asyncId)]: 4
}
Hint: hit control+c anytime to enter REPL.
listening on port 5000
```

Image: 7 ***Run Your Application***

If you select "create" and then fill in the form and hit submit, you'll get a response back that says `Cannot POST /create`. This is because we haven't added the code that handles the form POST request, so let's do that.



```
Cannot POST /create
```

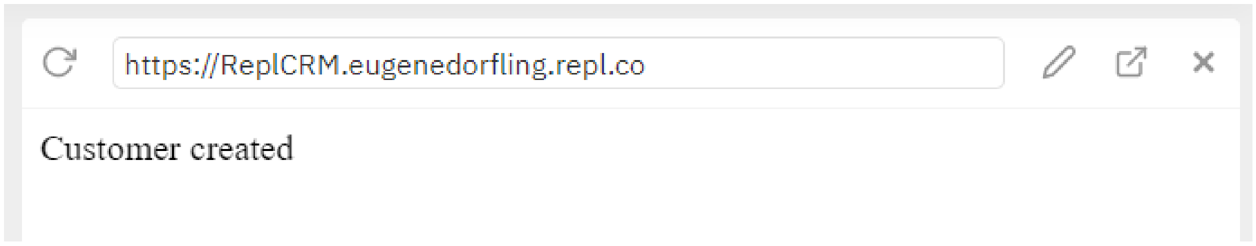
Image: 8 ***Cannot POST/create***

Add the following code into your `index.js` file, below the `app.get` entry that we made above.

```javascript
app.post('/create', function (req, res, next) {
  client.connect(err => {
    const customers = client.db("crmdb").collection("customers");

    let customer = { name: req.body.name, address: req.body.address, telephone: req.\
body.telephone, note: req.body.note };
    customers.insertOne(customer, function(err, res) {
      if (err) throw err;
      console.log("1 customer inserted");
10      });
11    })
12    res.send('Customer created');
13  })
```

- **Line 1** defines a new route that listens for an HTTP ‘POST’ request at `/create`.
- **Line 2** connects to the database. This happens asynchronously, so we define a callback function that will be called once the connection is made.
- **Line 3** creates a new collection of customers. Collections in MongoDB are similar to Tables in SQL.
- **Line 5** defines customer data that will be inserted into the collection. This is taken from the incoming request. The form data is parsed using the parsers that we defined earlier and is then placed in the `req.body` variable for us to use in the code.
- **Line 6** inserts the customer data into the collection. This also happens asynchronously, and so we define another callback function that will get an error if an error occurred, or the response if everything happened successfully.
- **Line 7** throws an error if the above insert had a problem.
- **Line 8** gives us some feedback that the insert happened successfully.

If you now run the Repl (you may need to refresh it) and submit the filled-in form, you’ll get a message back that says “Customer created”. If you then go and look in your cluster in MongoDB and select the “collections” button, you’ll see a document has been created with the details that we submitted in the form.



Customer created

Image: 9 ***Customer Created***

Updating and deleting database entries

As a final step in this tutorial, we want to be able to update and delete database documents in our collection. To make things simpler, we’re going to make a new HTML page where we can request a document and then update or delete it.

First, let’s make the routes to our new page. In your `index.js`, add the following code below the rest of your routing code (ie. before the MongoDB code):

```
 1  app.get('/get', function (req, res) {
 2    res.sendFile('/get.html', {root:'.'});
 3  });
 4
 5  app.get('/get-client', function (req, res) {
 6      client.connect(err => {
 7          client.db("crmdb").collection("customers").findOne({name: req.query.name}, f\
 8  unction(err, result) {
 9            if (err) throw err;
10            res.render('update', {oldname: result.name, oldaddress: result.address, ol\
11  dtelephone: result.telephone, oldnote: result.note, name: result.name, address: resu\
12  lt.address, telephone: result.telephone, note: result.note});
13          });
14        });
15  });
```

- **Line 1-3** as before, this tells Express to respond to incoming GET requests on `/get` by sending the `get.html` file which we will define below.
- **Line 5-12** this tells Express to respond to incoming GET requests on `/get-client`.
  - **Line 7** makes a call to the database to fetch a customer by name. If there are more than 1 with the same name, then the first one found will be returned.
  - **Line 9** tells Express to render the `update` template, replacing variables with the given values as it goes. Important to note here is that we are also replacing values in the hidden form fields we created earlier with the current values of the customer details. This is to ensure that we update or delete the correct customer.

In your `index.html` file, add the following code after the `</form>` tag:

```
<br>
<form action="/get" method="GET">
  <input type="submit" value="Update/Delete">
</form>
```

This adds a new button that will make a GET request to `/get`, which will then return `get.html`.

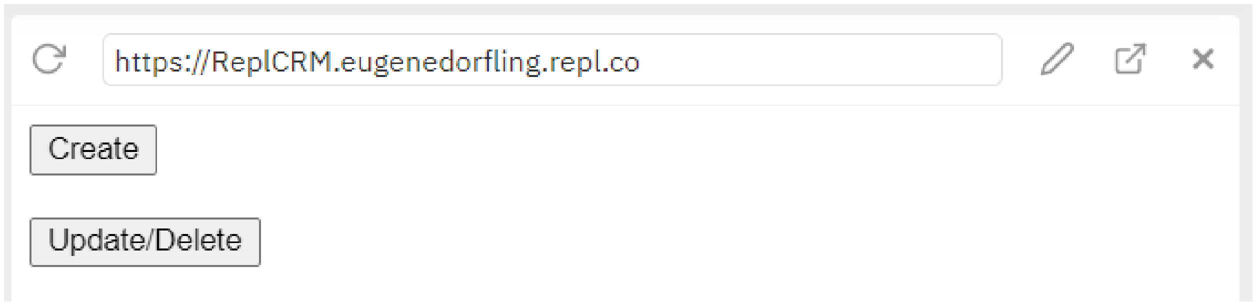


Image:10 ***Index***

Make a new file called `get.html` with the following contents:

```
<!DOCTYPE html>
<html>
<body>
  <form action="/get-client" method="GET">
    <label for="name" >Customer name *</label><br>
    <input type="text" id="name" name="name" class="textInput" placeholder="John Smi\
th" required>
    <input type="submit" value="Get customer">
  </form>
</body>
</html>
```

This makes a simple form with an input for the customer's name and a button.

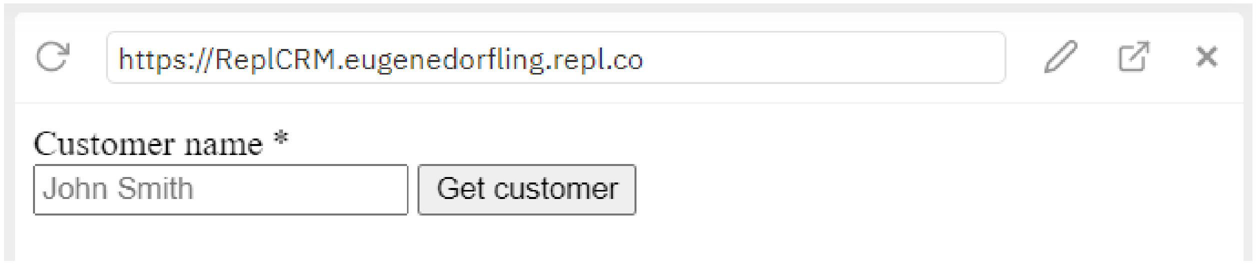


Image:11 ***Get Customer***

Clicking this button will then make a GET call to `/get-client` which will respond with the client details where we will be able to update or delete them.

To actually see the customer details on a form after requesting them, we need a templating engine to render them onto the HTML page and send the rendered page back to us. With a templating engine, you define a template - a page with variables in it - and then give it the values you want to fill into the variables. In our case, we're going to request the customer details from the database and tell the templating engine to render them onto the page.

We're going to use a templating engine called Pug[182]. Pug is a simple templating engine that

[182]https://pugjs.org/api/getting-started.html

integrates fully with Express. The syntax that Pug uses is very similar to HTML. One important difference in the syntax is that spacing is very important as it determines your parent/child hierarchy.

First, we need to tell Express which templating engine to use and where to find our templates. Put the following line above your route definitions (i.e. after the other app. lines in index.js):

```
app.engine('pug', require('pug').__express)
app.set('views', '.')
app.set('view engine', 'pug')
```

Now create a new file called `update.pug` with the following content:

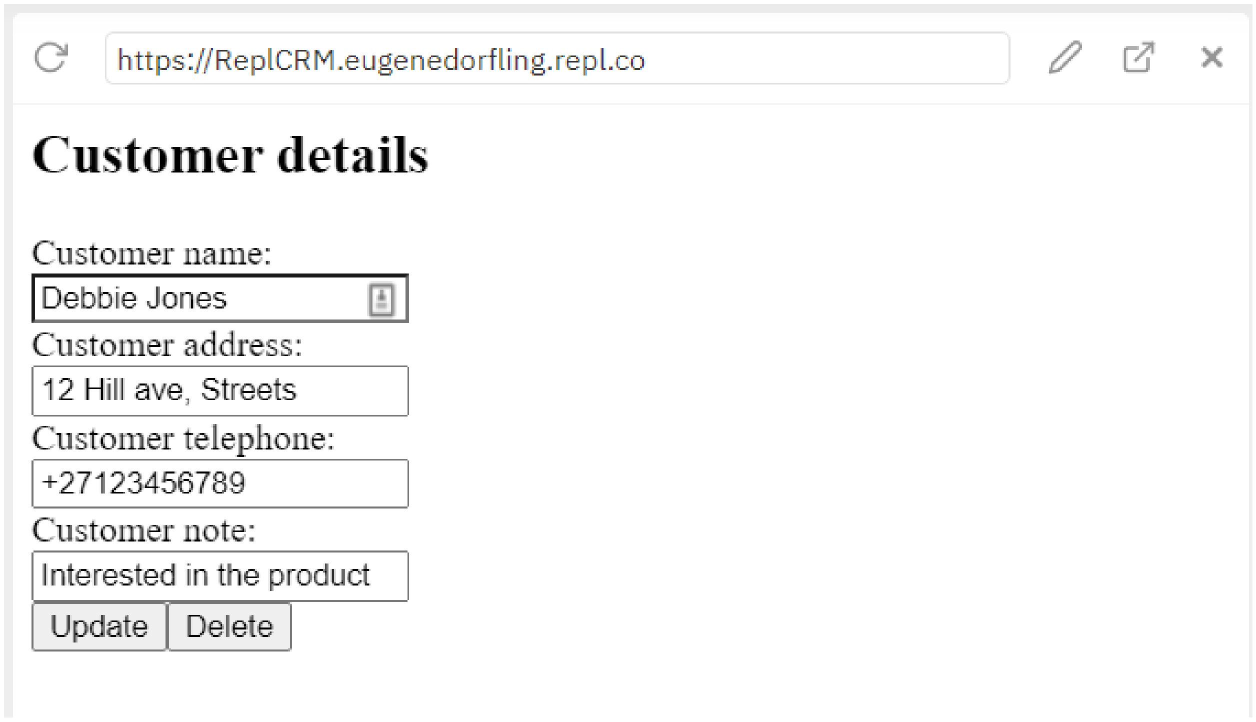
```
html
  body
    p #{message}
    h2= 'Customer details'
    form(method='POST' action='/update')
      input(type='hidden' id='oldname' name='oldname' value=oldname)
      input(type='hidden' id='oldaddress' name='oldaddress' value=oldaddress)
      input(type='hidden' id='oldtelephone' name='oldtelephone' value=oldtelephone)
      input(type='hidden' id='oldnote' name='oldnote' value=oldnote)
      label(for='name') Customer name:
      br
      input(type='text', placeholder='John Smith' name='name' value=name)
      br
      label(for='address') Customer address:
      br
      input(type='text', placeholder='42 Wallaby Way, Sydney' name='address' value=a\
ddress)
      br
      label(for='telephone') Customer telephone:
      br
      input(type='text', placeholder='+275554202' name='telephone' value=telephone)
      br
      label(for='note') Customer note:
      br
      input(type='text', placeholder='Likes unicorns' name='note' value=note)
      br
      button(type='submit' formaction="/update") Update
      button(type='submit' formaction="/delete") Delete
```

This is very similar to the HTML form we created previously for `create.html`, however this is written in the Pug templating language. We're creating a hidden element to store the "old" name, telephone, address, and note of the customer - this is for when we want to do an update.

Using the old details to update the customer is an easy solution, but not the best solution as it makes the query cumbersome and slow. If you add extra fields into your database you would have to remember to update your query as well, otherwise it could lead to updating or deleting the wrong customer if they have the same information. A better, but more complicated way is to use the unique ID of the database document as that will only ever refer to one customer.

We have also put in placeholder variables for name, address, telephone, and note, and we have given the form 2 buttons with different actions.

If you now run the code, you will have an index page with 2 buttons. Pressing the 'Update/Delete' button will take you to a new page that asks for a Customer name. Filling the customer name and pressing 'Get customer' will, after a little time, load a page with the customer's details and 2 buttons below that say 'Update' and 'Delete'. Make sure you enter a customer name you have entered before.


https://ReplCRM.eugenedorfling.repl.co

Customer details

Customer name:
Debbie Jones
Customer address:
12 Hill ave, Streets
Customer telephone:
+27123456789
Customer note:
Interested in the product

Image:12 ***Update-Delete***

Our next step is to add the 'Update' and 'Delete' functionality. Add the following code below your routes in `index.js`:

```
1  app.post('/update', function(req, res) {
2    client.connect(err => {
3      if (err) throw err;
4      let query = { name: req.body.oldname, address: req.body.oldaddress, telephone: r\
5  eq.body.oldtelephone, note: req.body.oldnote };
6      let newvalues = { $set: {name: req.body.name, address: req.body.address, telepho\
7  ne: req.body.telephone, note: req.body.note } };
8      client.db("crmdb").collection("customers").updateOne(query, newvalues, function(\
9  err, result) {
10         if (err) throw err;
11         console.log("1 document updated");
12         res.render('update', {message: 'Customer updated!', oldname: req.body.name, \
13 oldaddress: req.body.address, oldtelephone: req.body.telephone, oldnote: req.body.no\
14 te, name: req.body.name, address: req.body.address, telephone: req.body.telephone, n\
15 ote: req.body.note});
16       });
17   });
18 })
19 
20 app.post('/delete', function(req, res) {
21   client.connect(err => {
22     if (err) throw err;
23     let query = { name: req.body.name, address: req.body.address ? req.body.address \
24 : null, telephone: req.body.telephone ? req.body.telephone : null, note: req.body.no\
25 te ? req.body.note : null };
26     client.db("crmdb").collection("customers").deleteOne(query, function(err, obj) {
27       if (err) throw err;
28       console.log("1 document deleted");
29       res.send(`Customer ${req.body.name} deleted`);
30     });
31   });
32 })
```

This introduces 2 new 'POST' handlers - one for `/update`, and one for `/delete`.

- **Line 2** connects to our MongoDB database.
- **Line 3** throws an error if there was a problem connecting to the database.
- **Line 4** defines a query that we will use to find the document to update. In this case, we are using the details of the customer *before* it was updated. We saved this name earlier in a hidden field in the HTML. Trying to find the customer by its updated name obviously won't work because it hasn't been updated yet. Also, note that we are setting some of the fields to null if they are empty. This is so that the database returns the correct document when we update or

delete - if we search for a document that has no address with an address of ‘’ (empty string), then our query won’t return anything.
- **Line 5** defines the new values that we want to update our customer with.
- **Line 6** updates the customer with the new values using the query
- **Line 7** throws an error if there was a problem with the update.
- **Line 8** logs that a document was updated.
- **Line 9** re-renders the update page with a message saying that the customer was updated, and displays the new values.
- **Line 15** connects to our MongoDB database.
- **Line 16** throws an error if there was a problem connecting to the database.
- **Line 17** defines a query that we will use to find the document to delete. In this case we are using all the details of the customer *before* any changes were made on the form to make sure we delete that specific customer.
- **Line 18** we connect to the database and delete the customer.
- **Line 19** throws an error if there was a problem with the delete.
- **Line 20** logs that a document was deleted.
- **Line 21** sends a response to say that the customer was deleted.

Putting it all together

If you run your application now, you’ll be able to create, update, and delete documents in a MongoDB database. This is a very basic CRUD application, with a very basic and unstyled UI, but it should give you the foundation to build much more sophisticated applications.

Some ideas for this are:
*You could add fields to the database to classify customers according to which stage they are in your sales pipeline[183] so that you can track if a customer is potentially stuck somewhere and contact them to re-engage.
*You could then integrate some basic marketing automation with a page allowing you to send an email or SMS to customers (though don’t spam clients!).
*You could also add fields to keep track of customer purchasing information so that you can see which products do well with which customers.

If you want to start from where this tutorial leaves off, fork the Repl at https://repl.it/@GarethDwyer1/nodejs-crm[184].

In the next chapter, we’ll be introducing using machine learning to classify text in Repl.it.

[183]https://www.bitrix24.com/glossary/what-is-pipeline-management-definition-crm.php
[184]https://repl.it/@GarethDwyer1/nodejs-crm

Introduction to Machine Learning with Python and Repl.it

In this tutorial, we're going to walk through how to set up a basic Python Repl[185] that can learn the difference between two categories of sentences, positive and negative. For example, if you had the sentence "I love it!", we want to train a machine to know that this sentence is associated with happy and positive emotions. If we have a sentence like "it was really terrible", we want the machine to label it as a negative or sad sentence.

The maths, specifically calculus and linear algebra, behind machine learning gets a bit hairy. We'll be abstracting this away with the Python library scikit-learn[186], which makes it possible to do advanced machine learning in a few lines of Python.

At the end of this tutorial, you'll understand the fundamental ideas of automatic classification and have a program that can learn by itself to distinguish between different categories of text. You'll be able to use the same code to learn new categories (e.g. spam/not-spam, or clickbait/non-clickbait).

Overview and requirements

To follow along with this tutorial, you should have at least basic knowledge of Python or a similar programming language. Ideally, you should also sign up for a Repl.it[187] account so that you can modify and extend the bot we build, but it's not completely necessary.

In this tutorial, we will:

- Create some simple mock data - text to classify as positive or negative
- Explain vectorisation of the dataset
- Cover how to classify text using a machine learning classifier
- Compare this to a manual classifier

Setting up

Create a new Python Repl and open the `main.py` file that Repl created for you automatically. Add the following two imports to the top and run the Repl so that these dependencies are installed.

[185] https://repl.it
[186] https://scikit-learn.org/
[187] https://repl.it

```
1 from sklearn import tree
2 from sklearn.feature_extraction.text import CountVectorizer
```

In line 1, we import the `tree` module, which will give us a Decision Tree classifier that can learn from data. In line 2, we import a vectoriser – something that can turn text into numbers. We'll describe each of these in more detail soon!

Throughout the next steps, you can hit the big green "run" button to run your code, check for bugs, and view output along the way (you should do this every time you add new code).

Creating some mock data

Before we get started with the exciting part, we'll create a very simple dataset – too simple in fact. You might not see the full power of machine learning at first as our task will look so easy, but once we've walked through the concepts, it'll be a simple matter of swapping the data out for something bigger and more complicated.

On the next lines of main.py add the following lines of code.

```
positive_texts = [
    "we love you",
    "they love us",
    "you are good",
    "he is good",
    "they love mary"
]

negative_texts =  [
    "we hate you",
    "they hate us",
    "you are bad",
    "he is bad",
    "we hate mary"
]

test_texts = [
    "they love mary",
    "they are good",
    "why do you hate mary",
    "they are almost always good",
    "we are very bad"
]
```

We've created three simple datasets of five sentences each. The first one contains positive sentences; the second one contains negative sentences; and the last contains a mix of positive and negative sentences.

It's immediately obvious to a human which sentences are positive and which are negative, but can we teach a computer to tell them apart?

We'll use the two lists `positive_texts` and `negative_texts` to *train* our model. That is, we'll show these examples to the computer along with the correct answers for the question "is this text positive or negative?". The computer will try to find rules to tell the difference, and then we'll test how well it did by giving it `test_texts` without the answers and ask it to guess whether each example is positive or negative.

Understanding vectorization

The first step in nearly all machine learning problems is to translate your data from a format that makes sense to a human to one that makes sense to a computer. In the case of language and text data, a simple but effective way to do this is to associate each unique word in the dataset with a number, from `0` onwards. Each text can then be represented by an array of numbers, representing how often each possible word appears in the text.

Let's go through an example to see how this works. If we had the two sentences

`["nice pizza is nice"], ["what is pizza"]`

then we would have a dataset with four unique words in it. The first thing we'd want to do is create a vocabulary mapping to map each unique word to a unique number. We could do this as follows:

```
{
    "nice": 0,
    "pizza": 1,
    "is": 2,
    "what": 3
}
```

To create this, we simply go through both sentences from left to right, mapping each new word to the next available number and skipping words that we've seen before. We can now convert our sentences into bag of words vectors as follows, where we indicate the frequency of occurrence of each of the words in our vocabulary:

```
[
    [2, 1, 1, 0], # two "nice", one "pizza", one "is", zero "what"
    [0, 1, 1, 1]  # zero "nice", one "pizza", one "is", one "what"
]
```

Each sentence vector is always the same length as the *total* vocabulary size. We have four words in total (across all of our sentences), so each sentence is represented by an array of length four. Each position in the array represents a word, and each value represents how often that word appears in that sentence.

The first sentence contains the word "nice" twice, while the second sentence does not contain the word "nice" at all. According to our mapping, the zeroth element of each array should indicate how often the word nice appears, so the first sentence contains a `2` in the beginning and the second sentence contains a `0` there.

This representation is called "bag of words" because we lose all of the information represented by the *order* of words. We don't know, for example, that the first sentence starts and ends with "nice", only that it contains the word "nice" twice.

With real data, these arrays get *very* long. There are millions of words in most languages, so for a big dataset containing most words, each sentence needs to be represented by a very long array, where nearly all values are set to zero (all the words not in that sentence). This could take up a lot of space, but luckily scikit-learn uses a clever sparse-matrix implementation to overcome this. This doesn't quite look like the above, but the overall concept remains the same.

Let's see how to achieve the above using scikit-learn's optimised vectoriser.

First we want to combine all of our "training" data (the data that we'll show the computer along with the correct labels of "positive" or "negative" so that it can learn), so we'll combine our positive and negative texts into one array. Add the following code below the datasets you created.

```
training_texts = negative_texts + positive_texts
training_labels = ["negative"] * len(negative_texts) + ["positive"] * len(positive_t\
exts)
```

Our dataset now looks like this:

```
['we hate you', 'they hate us', 'you are bad', 'he is bad', 'we hate mary', 'we love\
 you', 'they love us', 'you are good', 'he is good', 'they love mary']
['negative', 'negative', 'negative', 'negative', 'negative', 'positive', 'positive',\
 'positive', 'positive', 'positive']
```

The two arrays (texts and labels) are associated by index. The first text in the first array is negative, and corresponds to the first label in the second array, and so on.

Now we need a vectoriser to transform the texts into numbers. We can create one in scikit-learn with

```
vectorizer = CountVectorizer()
```

Before we can use our vectorizer, it needs to run once through all the data we have so it can build the mapping from words to indices. This is referred to as "fitting" the vectoriser, and we can do it like this:

```
vectorizer.fit(training_texts)
```

If we want, we can see the mapping it created (which might not be in order, as in the examples we walked through earlier, but each word will have its own index). We can inspect the vectoriser's vocabulary by adding the line

```
print(vectorizer.vocabulary_)
```

(Note the underscore at the end. Scikit-learn uses this as a convention for "helper" attributes. The mapping is explicit only for debugging purposes and you shouldn't need to use it in most cases). My vocabulary mapping looked as follows:

```
{'we': 10, 'hate': 3, 'you': 11, 'they': 8, 'us': 9, 'are': 0, 'bad': 1, 'he': 4, 'i\
s': 5, 'mary': 7, 'love':6, 'good': 2}
```

Behind the scenes, the vectoriser inspected all of our texts, did same basic preprocessing like making everything lowercase, split the text into words using a built-in *tokenization* method, and produced a vocabulary mapping specific to our dataset.

Now that we have a vectorizer that knows what words are in our dataset, we can use it to transform our texts into vectors. Add the following lines of code to your Repl:

```
training_vectors = vectorizer.transform(training_texts)
testing_vectors = vectorizer.transform(test_texts)
```

The first line creates a list of vectors which represent all of the training texts, still in the same order, but now each text is a vector of numbers instead of a string.

The second line does the same with the test vectors. The machine learning part isn't looking at our test texts (that would be cheating) – it's just mapping the words to numbers so that it can work with them more easily. Note that when we called `fit()` on the vectoriser, we only showed it the training texts. Because there are words in the test texts that don't appear in the training texts, these words will simply be ignored and will not be represented in `testing_vectors`.

Now that we have a vectorised representation of our problem, let's take a look at how we can solve it.

Understanding classification

A classifier is a statistical model that tries to predict a label for a given input. In our case, the input is the text and the output is either "positive" or "negative", depending on whether the classifier thinks that the input is positive or negative.

A machine learning classifier can be "trained". We give it labelled data and it tries to learn rules based on that data. Every time it gets more data, it updates its rules slightly to account for the new information. There are many kinds of classifiers, but one of the simplest is called a Decision Tree.

Decision trees learn a set of yes/no rules by building decisions into a tree structure. Each new input moves down the tree, while various questions are asked one by one. When the input filters all the way to a leaf node in the tree, it acquires a label.

If that's confusing, don't worry! We'll walk through a detailed example with a picture soon to clarify. First, let's show how to get some results using Python.

Add the following lines to `main.py`:

```
classifier = tree.DecisionTreeClassifier()
classifier.fit(training_vectors, training_labels)
predictions = classifier.predict(testing_vectors)
print(predictions)
```

Similarly to the vectoriser, we first create a classifier by using the module we imported at the start. Then we call `fit()` on the classifier and pass in our training vectors and their associated labels. The decision tree is going to look at both and attempt to learn rules that separate the two kinds of data.

Once our classifier is trained, we can call the `predict()` method and pass in previously unseen data. Here we pass in `testing_vectors` which is the list of vectorized test data that the computer didn't look at during training. It has to try and apply the rules it learned from the training data to this new "unseen" data. Machine learning is pretty cool, but it's not magic, so there's no guarantee that the rules we learned will be any good yet.

The code above produces the following output:

```
['positive' 'positive' 'negative' 'positive' 'negative']
```

Let's take a look at our test texts again to see if these predictions match up to reality.

```
"they love mary"
"they are good"
"why do you hate mary"
"they are almost always good"
"we are very bad"
```

The output maps to the input by index, so the first output label ("positive") matches up to the first input text ("they love mary"), and the last output label ("negative") matches up to the last input text ("we are very bad").

It looks like the computer got every example right! It's not a difficult problem to solve. The words "bad" and "hate" appear only in the negative texts and the words "good" and "love", only in the positive ones. Other words like "they", "mary", "you" and "we" appear in both good and bad texts. If our model did well, it will have learned to ignore the words that appear in both kinds of texts, and focus on "good", "bad", "love" and "hate".

Decision Trees are not the most powerful machine learning model, but they have one advantage over most other algorithms: after we have trained them, we can look inside and see exactly how they work. More advanced models like deep neural networks are nearly impossible to make sense of after training.

The Scikit-learn `tree` module contains a useful function to assist in visualising trees. Add the following code to the end of your Repl:

```python
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(5,5))
tree.plot_tree(classifier,feature_names = vectorizer.get_feature_names(), rounded = \
True, filled = True)
fig.savefig('tree.png')
```

In the left-hand pane, you should see a file called 'tree.png'. If you open it, your tree graph should look as follows:

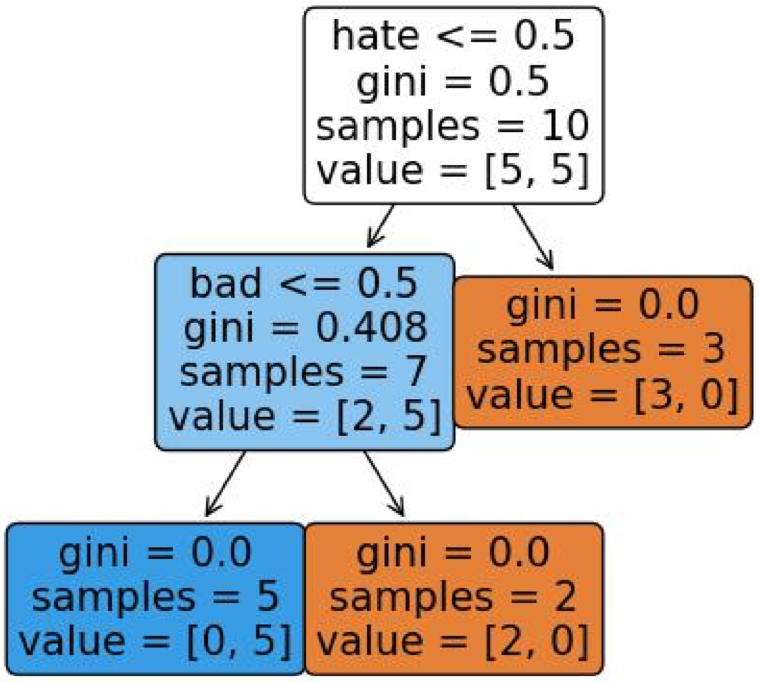


Image: 1 ***A visualised decision tree***

The above shows a decision tree that only learned two rules. The first rule (top square) is about the word "hate". The rule is "is the number of times 'hate' occurs in this sentence less than or equal to 0.5". None of our sentences contain duplicate words, so each rule will really be only about whether the word appears or not (you can think of the `<= 0.5` rules as `< 1` in this case).

For each question in our training dataset, we can ask if the first rule is True or False. If the rule is True for a given sentence, we'll move that sentence down the tree **left**. If not, we'll go **right**.

Once we've asked this first question for each sentence in our dataset, we'll have three sentences for which the answer is "False", because three of our training sentences contain the word "hate". These three sentences go right in the decision tree and end up at first leaf node (an end node with no arrows coming out the bottom). This leaf node has `value = [3,0]` in it, which means that three samples reach this node, and three belong to the negative class and zero to the positive class.

For each sentence where the first rule is "True" (the word "hate" appears less than 0.5 times, or in our case 0 times), we go down the left of the tree, to the node where `value = [2,5]`. This isn't a leaf node (it has more arrows coming out the bottom), so we're not done yet. At this point we have two negative sentences and all five positive sentences still.

The next rule is “bad <= 0.5”. In the same way as before, we’ll go down the right path if we have more than 0.5 occurrences of “bad” and left if we have fewer than 0.5 occurrences of “bad”. For the last two negative sentences that we are still evaluating (the two containing “bad”), we’ll go *right* and end up at the node with `value=[2,0]`. This is another leaf node and when we get here we have two negative sentences and zero positive ones.

All other data will go left, and we’ll end up at `[0,5]`, or zero negative sentences and five positive ones.

As an exercise, take each of the test sentences (not represented in the annotated tree above) and try to follow the set of rules for each one. If it ends up in a bucket with more negative sentences than positive ones (either of the right branches), it’ll be predicted as a negative sentence. If it ends up in the left-most leaf node, it’ll be predicted as a positive sentence.

Building a manual classifier

When the task at hand is this simple, it’s often easier to write a couple of rules manually rather than using Machine Learning. For this dataset, we could have achieved the same result by writing the following code.

```python
def manual_classify(text):
    if "hate" in text:
        return "negative"
    if "bad" in text:
        return "negative"
    return "positive"

predictions = []
for text in test_texts:
    prediction = manual_classify(text)
    predictions.append(prediction)
print(predictions)
```

Here we have replicated the decision tree above. For each sentence, we check if it contains “hate” and if it does we classify it as negative. If it doesn’t, we check if it contains “bad”, and if it does, we classify it as negative. All other sentences are classified as positive.

So what’s the difference between machine learning and traditional rule-based models like this one? The main advantage of learning the rules directly is that it doesn’t really get more difficult as the dataset grows. Our dataset was trivially simple, but a real-world dataset might need thousands or millions of rules, and while we could write a more complicated set of if-statements “by hand”, it’s much easier if we can teach machines to learn these by themselves.

Also, once we've perfected a set of manual rules, they'll still only work for a single dataset. But once we've perfected a machine learning model, we can use it for many tasks, simply by changing the input data!

In the example we walked through, our model was a perfect student and learned to correctly classify all five unseen sentences, this is not usually the case for real-world settings. Because machine learning models are based on probability, the goal is to make them as accurate as possible, but in general you will not get 100% accuracy. Depending on the problem, you might be able to get higher accuracy by hand-crafting rules yourself, so machine learning definitely isn't the correct tool to solve all classification problems.

Try the code on bigger datasets to see how it performs. There is no shortage of interesting data sets to experiment with. For example, you could have a look at positive vs negative movie reviews from IMDB using the dataset here[188]. See if you can load the dataset from there into the classifier we built in this tutorial and compare the results.

You can fork this Repl here: https://repl.it/@GarethDwyer1/machine-learning-intro[189] to keep hacking on it (it's the same code as we walked through above but with some comments added.) If you prefer, the entire program is shown at the end of this tutorial, so you can copy paste it and work from there.

In the next chapter, we'll be investigating the Quicksort algorithm. Whether you're applying for jobs or just like algorithms, it's useful to understand how sorting works. In real projects, most of the time you'll just call .sort(), but here you'll build a sorter from scratch and understand how it works.

—

```
from sklearn import tree
from sklearn.feature_extraction.text import CountVectorizer
import matplotlib.pyplot as plt

positive_texts = [
    "we love you",
    "they love us",
    "you are good",
    "he is good",
    "they love mary"
]

negative_texts =  [
    "we hate you",
    "they hate us",
    "you are bad",
    "he is bad",
```

188https://keras.io/api/datasets/imdb/
189https://repl.it/@GarethDwyer1/machine-learning-intro

```
    "we hate mary"
]

test_texts = [
    "they love mary",
    "they are good",
    "why do you hate mary",
    "they are almost always good",
    "we are very bad"
]

training_texts = negative_texts + positive_texts
training_labels = ["negative"] * len(negative_texts) + ["positive"] * len(positive_t\
exts)

vectorizer = CountVectorizer()
vectorizer.fit(training_texts)
print(vectorizer.vocabulary_)

training_vectors = vectorizer.transform(training_texts)
testing_vectors = vectorizer.transform(test_texts)

classifier = tree.DecisionTreeClassifier()
classifier.fit(training_vectors, training_labels)

print(classifier.predict(testing_vectors))

fig = plt.figure(figsize=(5,5))
tree.plot_tree(classifier,feature_names = vectorizer.get_feature_names(), rounded = \
True, filled = True)
fig.savefig('tree.png')

def manual_classify(text):
    if "hate" in text:
        return "negative"
    if "bad" in text:
        return "negative"
    return "positive"

predictions = []
for text in test_texts:
    prediction = manual_classify(text)
    predictions.append(prediction)
```

```
print(predictions)
```

Quicksort tutorial: Python implementation with line by line explanation

In this tutorial, we'll be going over the Quicksort[190] algorithm with a line-by-line explanation. We'll go through how the algorithm works, build it in Repl.it and then time it to see how efficient it is.

Overview and requirements

We're going to assume that you already know at least something about sorting algorithms[191], and have been introduced to the idea of Quicksort. By the end of this tutorial, you should have a better understanding of how it works.

We're also going to assume that you've covered some more fundamental computer science concepts, especially recursion[192], on which Quicksort relies.

To recap, Quicksort is one of the most efficient and most commonly used algorithms to sort a list of numbers. Unlike its competitor, Mergesort, Quicksort can sort a list in place, without the need to create a copy of the list, and therefore saving on memory requirements.

The main intuition behind Quicksort is that if we can efficiently *partition* a list, then we can efficiently sort it. Partitioning a list means that we pick a *pivot* item in the list, and then modify the list to move all items larger than the pivot to the right and all smaller items to the left.

Once the pivot is done, we can do the same operation to the left and right sections of the list recursively until the list is sorted.

Here's a Python implementation of Quicksort. Have a read through it and see if it makes sense. If not, read on below!

[190]https://en.wikipedia.org/wiki/Quicksort
[191]https://en.wikipedia.org/wiki/Sorting_algorithm
[192]https://en.wikipedia.org/wiki/Recursion#In_computer_science

```python
def partition(xs, start, end):
    follower = leader = start
    while leader < end:
        if xs[leader] <= xs[end]:
            xs[follower], xs[leader] = xs[leader], xs[follower]
            follower += 1
        leader += 1
    xs[follower], xs[end] = xs[end], xs[follower]
    return follower

def _quicksort(xs, start, end):
    if start >= end:
        return
    p = partition(xs, start, end)
    _quicksort(xs, start, p-1)
    _quicksort(xs, p+1, end)

def quicksort(xs):
    _quicksort(xs, 0, len(xs)-1)
```

The Partition algorithm

The idea behind the partition algorithm seems intuitive, but the actual algorithm to do it efficiently is pretty counter-intuitive.

Let's start with the easy part – the idea. We have a list of numbers that isn't sorted. We pick a point in this list, and make sure that all larger numbers are to the *right* of that point and all the smaller numbers are to the *left*. For example, given the random list:

```python
xs = [8, 4, 2, 2, 1, 7, 10, 5]
```

We could pick the *last* element (5) as the pivot point. We would want the list (after partitioning) to look as follows:

```python
xs = [4, 2, 2, 1, 5, 7, 10, 8]
```

Note that this list isn't sorted, but it has some interesting properties. Our pivot element, `5`, is in the correct place (if we sort the list completely, this element won't move). Also, all the numbers to the left are smaller than `5`and all the numbers to the right are greater.

Because `5` is the in the correct place, we can ignore it after the partition algorithm (we won't need to move it again). This means that if we can sort the two smaller sublists to the left and right of

5() `[4, 2, 2, 1]` and `[7, 10, 8]`) then the entire list will be sorted. Any time we can efficiently break a problem into smaller sub-problems, we should think of *recursion* as a tool to solve our main problem. Using recursion, we often don't even have to think about the entire solution. Instead, we define a base case (a list of length `0` or `1` is always sorted), and a way to divide a larger problem into smaller ones (e.g. partitioning a list in two), and almost by magic the problem solves itself!

But we're getting ahead of ourselves a bit. Let's take a look at how to actually implement the partition algorithm on its own, and then we can come back to using it to implement a sorting algorithm.

A bad partition implementation

You could probably easily write your own `partition` algorithm that gets the correct results without referring to any textbook implementations or thinking about it too much. For example:

```python
def bad_partition(xs):
    smaller = []
    larger = []
    pivot = xs.pop()
    for x in xs:
        if x >= pivot:
            larger.append(x)
        else:
            smaller.append(x)
    return smaller + [pivot] + larger
```

In this implementation, we set up two temporary lists (`smaller` and `larger`). We then take the `pivot` element as the last element of the list (`pop` takes the last element and removes it from the original `xs` list).

We then consider each element `x` in the list `xs`. The ones that are smaller than the pivot, we store in the `smaller` temporary list, and the others go to the `larger` temporary list. Finally, we combine the two lists with the pivot item in the middle, and we have partitioned our list.

This is much easier to read than the implementation at the start of this post, so why don't we do it like this?

The primary advantage of Quicksort is that it is an *in place* sorting algorithm. Although for the toy examples we're looking at, it might not seem like much of an issue to create a few copies of our list, if you're trying to sort terabytes of data, or if you are trying to sort any amount of data on a very limited computer (e.g a smartwatch), then you don't want to needlessly copy arrays around.

In Computer Science terms, this algorithm has a space-complexity of `O(2n)`, where `n` is the number of elements in our `xs` array. If we consider our example above of `xs = [8, 4, 2, 2, 1, 7, 10, 5]`, we'll need to store all `8` elements in the original `xs` array as well as three elements (`[7, 10, 8]`] in the `larger` array and four elements (`[4, 2, 2, 1]`) in the `smaller` array. This is a waste of space! With some clever tricks, we can do a series of swap operations on the original array and not need to make any copies at all.

Overview of the actual partition implementation

Let's pull out a few key parts of the good `partition` function that might be especially confusing before getting into the detailed explanation. Here it is again for reference.

```python
1 def partition(xs, start, end):
2     follower = leader = start
3     while leader < end:
4         if xs[leader] <= xs[end]:
5             xs[follower], xs[leader] = xs[leader], xs[follower]
6             follower += 1
7         leader += 1
8     xs[follower], xs[end] = xs[end], xs[follower]
9     return follower
```

In our good `partition` function, you can see that we do some swap operations (lines 5 and 8) on the `xs` that is passed in, but we never allocate any new memory. This means that the storage remains constant to the size of `xs`, or `O(n)` in Computer Science terms. That is, this algorithm has *half* the space requirement of the "bad" implementation above, and should therefore allow us to sort lists that are twice the size using the same amount of memory.

The confusing part of this implementation is that although everything is based around our pivot element (the last item of the list in our case), and although the pivot element ends up somewhere in the middle of the list at the end, we don't actually touch the pivot element *until the very last swap*.

Instead, we have two other counters (`follower` and `leader`) which move around the smaller and bigger numbers in a clever way and implicitly keep track of where the pivot element should end up. We then switch the pivot element into the correct place at the end of the loop (line 8).

The `leader` is just a loop counter. Every iteration it increments by one until it gets to the pivot element (the end of the list). The follower is more subtle, and it keeps count of the number of swap iterations we do, moving up the list more slowly than the leader, tracking where our pivot element should eventually end up.

The other confusing part of this algorithm is on line 4. We move through the list from left to right. All numbers are currently to the *left* of the pivot but we eventually want the "big" items to end up on the *right*.

Intuitively, you would then expect us to do the swapping action when we find an item that is *larger* than the pivot, but in fact, we do the opposite. When we find items that are *smaller* than the pivot, we swap the leader and the follower.

You can think of this as pushing the small items further to the left. Because the leader is always ahead of the follower, when we do a swap, we are swapping a small element with one further left in the list. The follower only looks at "big" items (ones that the leader has passed over without action), so when we do the swap, we're swapping a small item (leader) with a big one (follower), meaning that small items will move towards the left and large ones towards the right.

Line by line examination of partition

We define `partition` with three arguments, `xs` which is the list we want to sort, `start` which is the index of the first element to consider and `end` which is the index of the last element to consider.

We need to define the `start` and `end` arguments because we won't always be partitioning the entire list. As we work through the sorting algorithm later, we are going to be working on smaller and smaller sublists, but because we don't want to create new copies of the list, we'll be defining these sublists by using indexes to the original list.

In line `2`, we start off both of our pointers – `follower`, and `leader` – to be the same as the beginning of the segment of the list that we're interested in. The leader is going to move faster than the follower, so we'll carry on looping until the leader falls off the end of the list segment (`while leader < end`).

We could take any element we want as a pivot element, but for simplicity, we'll just choose the last element. In line `4` then, we compare the `leader` element to the pivot. The leader is going to step through each and every item in our list segment, so this means that when we're done, we'll have compared the partition with every item in the list.

If the `leader` element is smaller or equal to the pivot element, we need to send it further to the left and bring a larger item (tracked by `follower`) further to the right. We do this in lines `4-5`, where if we find a case where the `leader` is smaller or equal to the pivot, we swap it with the `follower`. At this point, the follower is pointing at a small item (the one that was `leader` a moment ago), so we increment `follower` by one in order to track the next item instead. This has a side effect of counting how many swaps we do, which incidentally tracks the exact place that our pivot element should eventually end up.

Whether or not we did a swap, we want to consider the next element in relation to our pivot, so in line `7` we increment `leader`.

Once we break out of the loop (line `8`), we need to swap the pivot item (still on the end of the list) with the `follower` (which has moved up one for each element that was smaller than the pivot). If this is still confusing, look at our example again:

```
xs = [8, 4, 2, 2, 1, 7, 10, 5]
```

In `xs`, there are `4` items that are smaller than the pivot. Every time we find an item that is smaller than the pivot, we increment `follower` by one. This means that at the end of the loop, follower will have incremented `4` times and be pointing at index `4` in the original list. By inspection, you can see that this is the correct place for our pivot element (`5`).

The last thing we do is return the follower index, which now points to our pivot element in its *correct* place. We need to return this as it defines the two smaller sub-problems in our partitioned list - we now want to sort`xs[0:4]` (the first `4` items, which form an unsorted list) and the `xs[5:]` (the last `3` items, which form an unsorted list).

```python
xs = [4, 2, 2, 1, 5, 7, 10, 8]
```

If you want another way to visualise exactly how this works, going over some examples by hand (that is, writing out a short randomly ordered list with a pen and paper, and writing out the new list at each step of the algorithm) is very helpful. You can also watch this detailed YouTube video[193] where KC Ang demonstrates every step of the algorithm using paper cups in under 5 minutes!

The Quicksort function

Once we get the partition algorithm right, sorting is easy. We'll define a helper `_quicksort` function first to handle the recursion and then implement a prettier public function after.

```python
1 def _quicksort(xs, start, end):
2     if start >= end:
3         return
4     p = partition(xs, start, end)
5     _quicksort(xs, start, p-1)
6     _quicksort(xs, p+1, end)
```

To sort a list, we partition it (line 4), sort the left sublist (line 5: from the start of the original list up to the pivot point), and then sort the right sublist (line 6: from just after the pivot point to the end of the original list). We do this recursively with the `end` boundary moving left, closer to `start`, for the left sublists and the `start` boundary moving right, closer to `end`, for the right sublists. When the start and end boundaries meet (line 2), we're done!

The first call to Quicksort will always be with the entire list that we want sorted, which means that `0` will be the start of the list and `len(xs)-1` will be the end of the list. We don't want to have to remember to pass these extra arguments in every time we call Quicksort from another program (e.g. in any case where it is not calling itself), so we'll make a prettier wrapper function with these defaults to get the process started.

```python
def quicksort(xs):
    return _quicksort(xs, 0, len(xs)-1)
```

Now we, as users of the sorting function, can call `quicksort([4,5,6,2,3,9,10,2,1,5,3,100,23,42,1])`, passing in only the list that we want sorted. This will in turn go and call the `_quicksort` function, which will keep calling itself until the list is sorted.

Testing our algorithm

We can write some basic driver code to take our newly implemented Quicksort out for a spin. Create a new Python Repl and add the following code to `main.py`. Then insert the code listed at the beginning of this tutorial after the imports.

[193] https://www.youtube.com/watch?v=MZaf_9IZCrc

```python
from datetime import datetime
import random

# create 100000 random numbers between 1 and 1000
xs = [random.randrange(1000) for _ in range(10)]

# look at the first few and last few
print(xs[:10])
#apply the algorithm
quicksort(xs)
# have a look at the results
print(xs[:10])
```

If you run this code, you will see the sorted list. This does what we expect, but it doesn't tell us about how efficient Quicksort is - so let's take a closer look. Replace the code in `main.py` with the following, and again add the code listed at the beginning of this tutorial after the imports on line 3.

```python
 1  from datetime import datetime
 2  import random
 3
 4  # create 100000 random numbers between 1 and 1000
 5  xs = [random.randrange(1000) for _ in range(100000)]
 6
 7  # look at the first few and last few
 8  print(xs[:10], xs[-10:])
 9
10  # start the clock
11  t1 = datetime.now()
12  quicksort(xs)
13  t2 = datetime.now()
14  print("Sorted list of size {} in {}".format(len(xs), t2 - t1))
15
16  # have a look at the results
17  print(xs[:10], xs[-10:])
```

The code generates a random list of 100 000 numbers and sorts this list in around 5 seconds. You can compare the performance of Quicksort to some other common sorting algorithms using this Repl[194].

If you want to try the code from the tutorial out, visit the Repl at https://repl.it/@GarethDwyer1/quicksort[195]. You'll be able to run the code, see the results, and even fork it to continue developing or testing it on your own.

[194]https://repl.it/@GarethDwyer1/sorting
[195]https://repl.it/@GarethDwyer1/quicksort?language=python3

If you need help, the folk over at the Repl discord server[196] are very friendly and keen to help people learn.

[196]https://repl.it/discord

Hiding messages in images: steganography with Python and Repl.it

In this tutorial, we'll build a steganography tool in Python. Steganography is the practice of hiding information within other data. Unlike encryption, where the goal is to secure *the contents* of communication between two parties, steganography aims to obscure the fact that the parties are communicating at all.

Our tool will enable the user to hide secret text within a normal-looking `.png` image file. The receiver of the image will use the same tool to reveal the hidden message.

We'll use Python to build the tool. The most popular Python image processing libraries are Pillow[197] and OpenCV[198], but these are heavy libraries with many dependencies. We'll avoid these and instead use the lightweight PyPNG[199] library which is written in pure Python, and therefore easier to run on various platforms.

A quick background on steganography

Let's imagine three people: Alice, Bob and Eve. Alice wants to send a private message to Bob, while Eve wants to intercept this message. While modern-day encryption can help Alice and Bob ensure that Eve doesn't know the *contents* of their message, Eve can possibly still deduce interesting information just from knowing that Alice and Bob are communicating at all, and how frequently they communicate.

To obscure the communication channel completely, Alice and Bob can exploit the fact that hundreds of millions of photos are uploaded and shared across the internet daily. Instead of communicating directly, Alice can leave her message hidden in an image at a pre-agreed location and Bob can access this message. From Eve's perspective, there is now no direct communication between the two.

A single image is made up of millions of pixels. While many formats exist, a pixel is most simply represented by a group of three numbers between 0 and 255, one number each for the red, blue, and green values of that pixel. Using this Red-Green-Blue scheme we can represent any colour in the RGB color model[200].

[197]https://pypi.org/project/Pillow/
[198]https://pypi.org/project/opencv-python/
[199]https://pypi.org/project/pypng/
[200]https://en.wikipedia.org/wiki/RGB_color_model

Digital text, like images, is also represented internally by numbers, so the differences between a text file and an image file are not as large as you might assume. Any digital data can be represented as a binary string[201], a bunch of 1s and 0s, and we can make tiny modifications to an image to encode a binary string within it. As an example, consider the following:

```
image = [(255, 0, 0), (0, 255, 0), (0, 0, 255)]
```

This is a representation of an image with three pixels: one red, one green, and one blue. If we encode this as an image and open it in an image viewer, we'll see the three pixel image, but if we read this data with Python, it is simply a list of tuples, each containing three integers.

We could also look at each value making up each pixel and calculate whether it is *odd* or *even*. We could encode odd numbers as `1` and even values as `0`. This would give us the binary string "100 010 001" (as the 255 values are odd and the 0s are even).

If we made a small modification to the image as follows:

```
image = [(254, 1, 1), (1, 255, 1), (1, 0, 254)]
```

The image would look almost identical in any image viewer (we have just added or subtracted a minuscule amount of color from some values), but the binary string – using our odd/even method – would look completely different: "011 111 100".

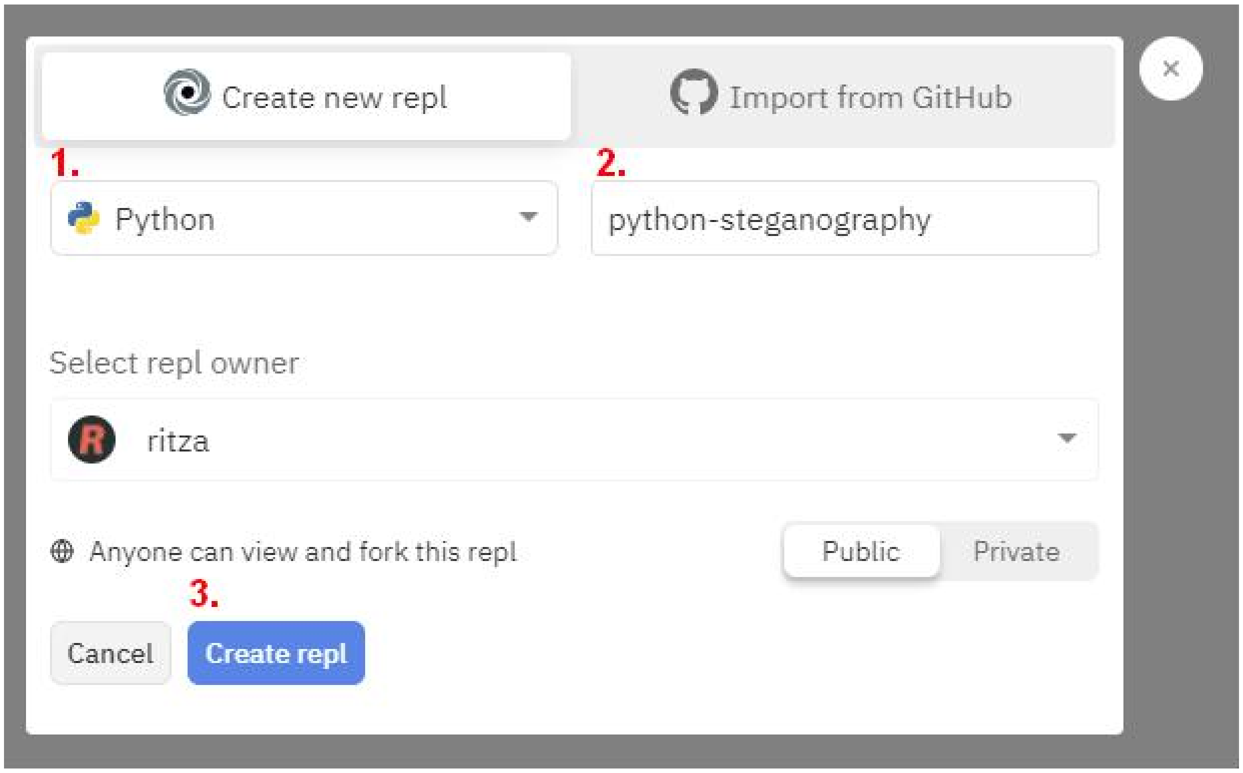
Using this technique but extending it over an entire image (millions of pixels), we can hide a large amount of text data in any image.

Creating the project on Repl.it

If you were serious about keeping your messages as secret as possible, you'd want to do all of these steps on an offline computer that you fully control. As a learning exercise though, we'll set the project up on repl.it[202]. Navigate to their site and sign up for an account if you don't have one.

Create a new project, choosing "Python" as the language, and give your project a name.

[201]https://thehelloworldprogram.com/computer-science/what-is-binary/
[202]https://repl.it



Creating a new repl

The first piece we need to build is a function to encode any text message as a binary string.

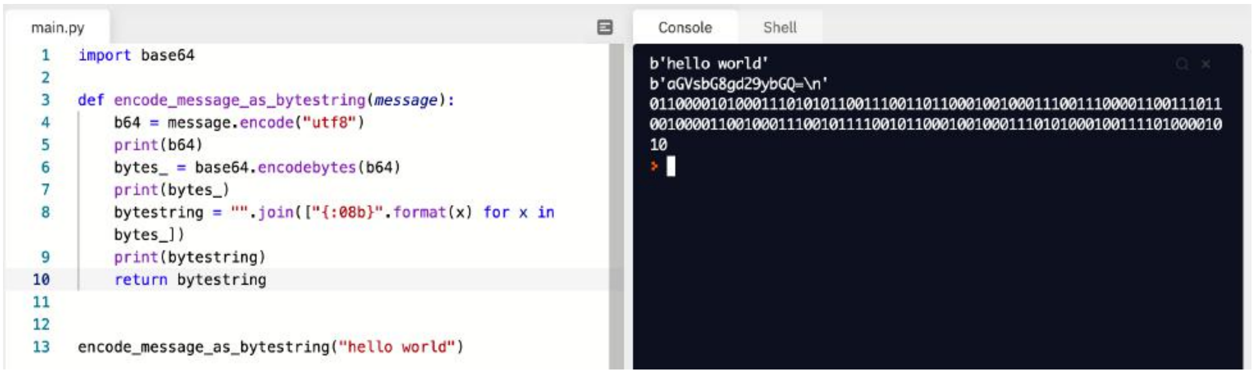
Encoding a text message as a binary string

Open the `main.py` file and add the following code

```python
import base64

def encode_message_as_bytestring(message):
    b64 = message.encode("utf8")
    bytes_ = base64.encodebytes(b64)
    bytestring = "".join(["{:08b}".format(x) for x in bytes_])
    return bytestring
```

This first encodes our text as base64[203] and then as a binary string. You can add some print statements to see how the message is transformed in the different steps, as shown below.

[203] https://en.wikipedia.org/wiki/Base64



```python
import base64

def encode_message_as_bytestring(message):
    b64 = message.encode("utf8")
    print(b64)
    bytes_ = base64.encodebytes(b64)
    print(bytes_)
    bytestring = "".join(["{:08b}".format(x) for x in
    bytes_])
    print(bytestring)
    return bytestring


encode_message_as_bytestring("hello world")
```

```
b'hello world'
b'aGVsbG8gd29ybGQ=\n'
0110000101000111010101100111001101100010010001110011100001100111011
0010000110010001110010111100101100010010001110101000100111101000010
10
> 
```

Encoding a message as a binary string

The base64 step is not strictly necessary, but it is useful as any file or data can be encoded as base64. This opens our project up to future extensions such as hiding other kinds of files within image files instead of just text strings.

Adding an 'end of message' delimeter

We'll assume that our message will always 'fit' in our image. We can fit three binary digits per pixel (one for each of the RGB values), so our resulting binary string should be shorter than the the number of pixels in the image multiplied by three.

We'll also need to know when the message *ends*. The message will only be encoded in the beginning of the image file, but if we don't know how long the message is, we'll keep looking at normal pixels and trying to encode them as text data. Let's add an "end of string" delimiter to the end of our message: this should be something that wouldn't appear half way through our actual message by chance. We'll use the binary representation of '!ENDOFMESSAGE!' for this.

Modify your function to look as follows, which adds this delimeter at the end.

```python
import base64

ENDOFMESSAGE = "0100100101010101010101011001001111010100100100010100111001010001110101\
01000101010101010101100101010001010101001100000100011001001000010100100101001101000101\
00111101"

def encode_message_as_bytestring(message):
    b64 = message.encode("utf8")
    bytes_ = base64.encodebytes(b64)
    bytestring = "".join(["{:08b}".format(x) for x in bytes_])
    bytestring += ENDOFMESSAGE
    return bytestring
```

Now that we can handle some basic text encoding, let's look at images.

Getting pixels from an image

Find a PNG image somewhere - either one you've taken yourself or from a site like unsplash. You can use any online JPG to PNG converter if you only have `.jpg` files available.

Upload your PNG file by clicking on the three dot menu in the repl sidebar, in the top right corner of the files pane to the left, and selecting `upload file` or by simply dragging and dropping your file within the files pane.

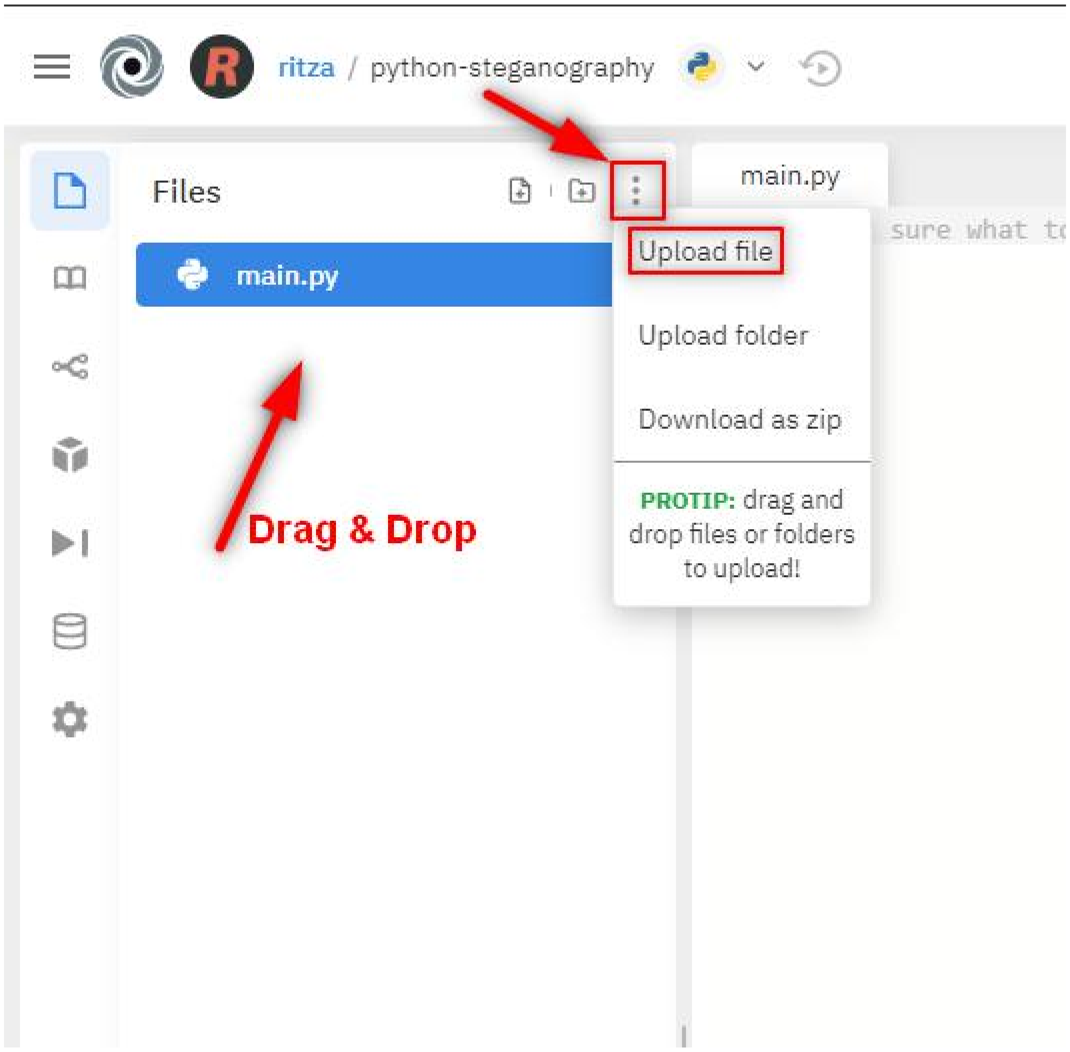


Image showing file upload

We're going to write a function that extracts the raw pixel data from this image file. Add an import to the top of the file.

```python
import png
```

And then add a new function to the bottom of `main.py`:

```python
def get_pixels_from_image(fname):
    img = png.Reader(fname).read()
    pixels = img[2]
    return pixels
```

The `read()` method returns a 4-tuple consisting of:

- width: Width of PNG image in pixels
- height: Height of PNG image in pixels
- rows: A sequence or iterator for the row data
- info: An info dictionary containing some meta data

We are primarily interested in the third item, “rows”, which is an iterator containing all the pixels of the image, row by row. If you’re not familiar with Python generators take a look at this guide[204], but they are essentially memory-efficient lists.

Encoding the image with the message

Now that we have the encoded message and pixels of the image ready we can combine them to form our secret encoded image.

Add the following function to the bottom of the `main.py` file. This function takes in the outputs from the previous functions (our raw pixels and our message encoded as a binary string), and combines them.

```python
def encode_pixels_with_message(pixels, bytestring):
    '''modifies pixels to encode the contents from bytestring'''

    enc_pixels = []
    string_i = 0
    for row in pixels:
        enc_row = []
        for i, char in enumerate(row):
            if string_i >= len(bytestring):
                pixel = row[i]
            else:
                if row[i] % 2 != int(bytestring[string_i]):
                    if row[i] == 0:
                        pixel = 1
                    else:
```

[204]https://realpython.com/introduction-to-python-generators/

```
                        pixel = row[i] - 1
                    else:
                        pixel = row[i]
                enc_row.append(pixel)
                string_i += 1

            enc_pixels.append(enc_row)
    return enc_pixels
```

This is the most complicated part of our project, but most of the code is there to handle edge cases. The important insight is that we want to control whether each pixel has an odd value (representing a 1 in our binary string) or an even one (to represent a 0). By chance, half of the pixel values will already have the correct value.

We simply loop through the binary string and the pixel and 'bump' each value that isn't correct by one. That is, we subtract 1 from the value if we need to change it from odd to even or vice versa. We don't want any negative numbers, so if we need to change any of the `0` values, we add 1 instead.

Writing our modified pixels back to an image

We now have all the image data, including the encoded message but it is still just a list of pixels. Let's add a function that will compile our pixels back into a PNG image.

Add the following function to the bottom of the `main.py` file.

```
def write_pixels_to_image(pixels, fname):
    png.from_array(pixels, 'RGB').save(fname)
```

The above function takes the array `pixels` and uses the `png` module to write these to a brand new `.png` file.

Play around with these functions to make sure you understand how they work. Before we write some wrapper code to actually use these, we're going to do everything backwards so that we can also extract hidden messages from previously encoded PNG files.

Decoding messages from image files

First we need a function that can turn a binary string back into readable text. As before, we'll go via base64 for better compatability. Add the following function to the bottom of the `main.py` file.

```python
def decode_message_from_bytestring(bytestring):
    bytestring = bytestring.split(ENDOFMESSAGE)[0]
    message = int(bytestring, 2).to_bytes(len(bytestring) // 8, byteorder='big')
    message = base64.decodebytes(message).decode("utf8")
    return message
```

Remember how we added a special `ENDOFMESSAGE` delimiter above? Here we first split our string on that so we don't look for text in random data (pixels from the unmodified part of the image) and then go backwards through our encoding pipe: first to base64 and then to text.

We also need a way to extract the bytestring from an image. Add the following function to `main.py` to do this.

```python
def decode_pixels(pixels):
    bytestring = []
    for row in pixels:
        for c in row:
            bytestring.append(str(c % 2))
    bytestring = ''.join(bytestring)
    message = decode_message_from_bytestring(bytestring)
    return message
```

Once again, this is just the reverse of what we did before. We grab the remainder of each value to get `1` for each odd value and `0` for each even one and keep them in a string. We then call our decode function to get the plaintext.

That's it for our encoding and decoding functions; next we'll put everything together in our `main()` function.

Adding a command line wrapper script

At this point, we could create a web application with a UI for people to add text to their images. Given the fact that people who want to do steganography probably won't trust a web application with their data, we'll rather create a command line application that people can run on their own machines.

Add the following to the top of your `main.py` file, right below the imports.

```python
PROMPT = """
Welcome to basic steganography. Please choose:

1. To encode a message into an image
2. To decode an image into a message
q. To exit
"""
```

Now let's write the `main()` function that puts it all together. Add the following to the end of the `main.py` file.

```python
def main():
    print(PROMPT)
    user_inp = ""
    while user_inp not in ("1", "2", "q"):
        user_inp = input("Your choice: ")

    if user_inp == "1":
        in_image = input("Please enter filename of existing PNG image: ")
        in_message = input("Please enter the message to encode: ")

        print("-ENCODING-")
        pixels = get_pixels_from_image(in_image)
        bytestring = encode_message_as_bytestring(in_message)
        epixels = encode_pixels_with_message(pixels, bytestring)
        write_pixels_to_image(epixels, in_image + "-enc.png")

    elif user_inp == "2":
        in_image = input("Please enter the filename of an existing PNG image: ")
        print("-DECODING-")
        pixels = get_pixels_from_image(in_image)
        print(decode_pixels(pixels))

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

The `main()` function above creates a prompt flow for the user to interact with the program. Depending on the input from the user, the program will call the relevant functions in order to either encode or decode a message. We also included a `q` for the user to close the program.

Where next?

If you have followed along you'll have your own repl to expand; if not you can fork our repl[205] and work from there.

[205]https://repl.it/@ritza/python-steganography

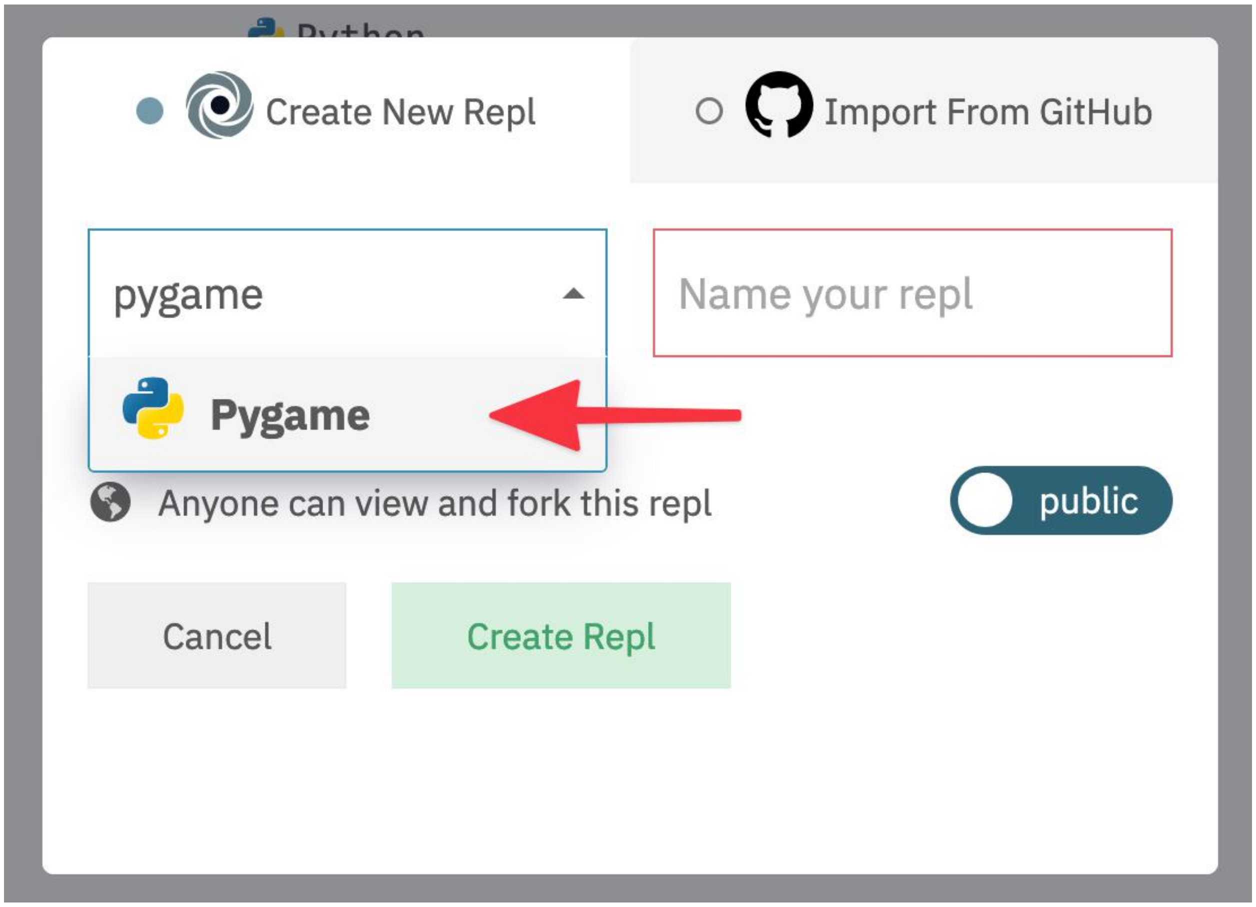
Build a 2D Platform Game with PyGame and Repl.it

In chapter 7(Building a game with PyGame) we introduced graphical game development with PyGame, covering how to develop a 2D game with animated sprites and user interaction. In this tutorial, we'll go a step further and create a 2D platformer, where you can have an alien walk and jump around a room full of boxes. The previous PyGame tutorial is not a prerequisite for trying this one.

We're going to focus on basic animation and movement to create a solid base from which you can continue on to build an entire platform game, complete with enemies, power-ups and multiple levels.

Getting Started

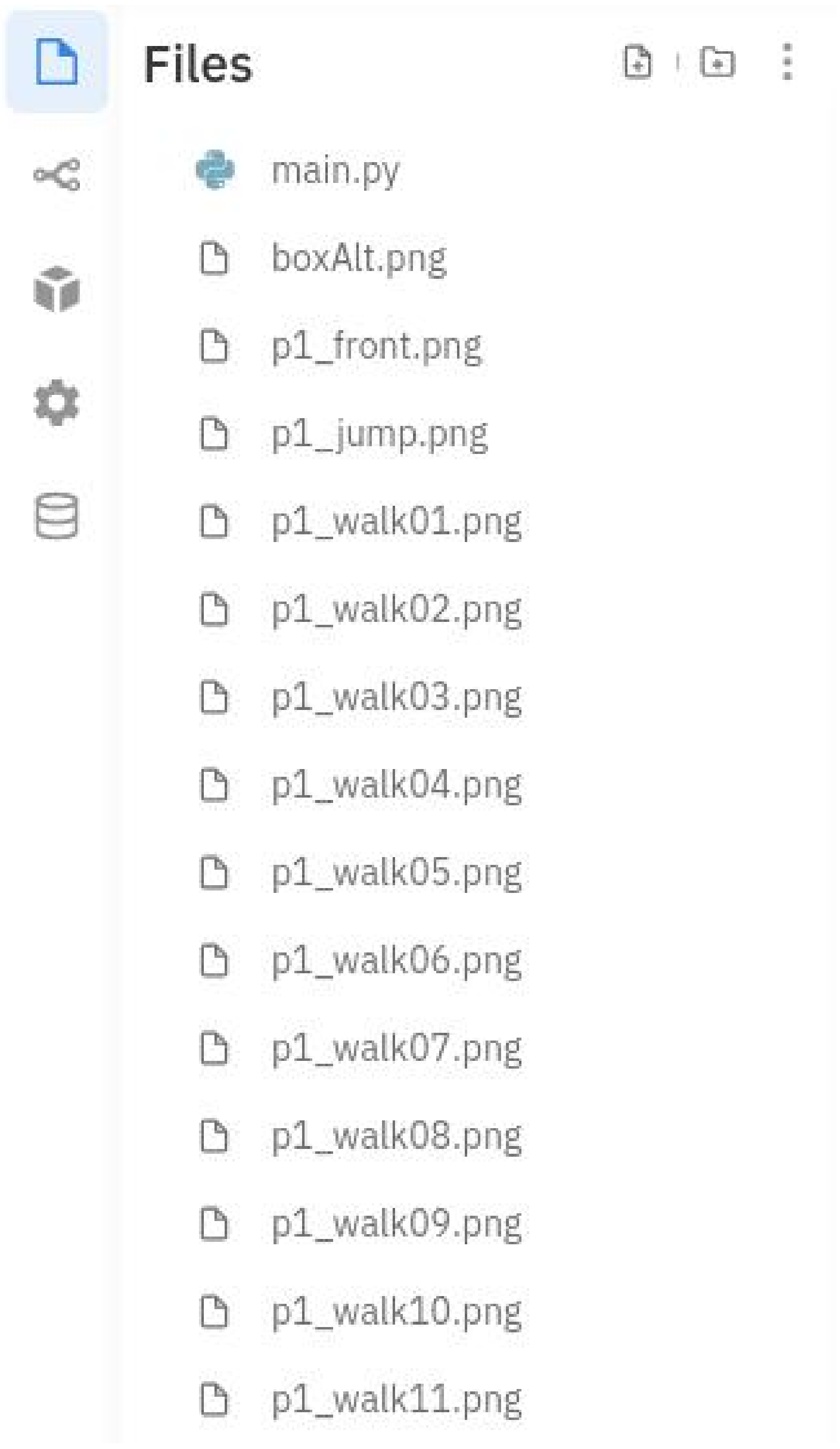
Create a new repl and select "PyGame" from the language dropdown.



You'll see "Python3 with PyGame" displayed in the default console and a separate pane in the Repl.it IDE where you will be able to see and interact with the game you will create.

Before we start writing code, we're going to need a few sprites, which we've made available here[206]. Extract this ZIP file and add the files inside to your repl using the `upload file` function. You can select multiple files to upload at once. Your repl's file pane should now look like this:

[206]/tutorial-files/2d-platform-game/2d-platform-game-sprites.zip



main.py
boxAlt.png
p1_front.png
p1_jump.png
p1_walk01.png
p1_walk02.png
p1_walk03.png
p1_walk04.png
p1_walk05.png
p1_walk06.png
p1_walk07.png
p1_walk08.png
p1_walk09.png
p1_walk10.png
p1_walk11.png

In this tutorial, we will be gradually building up the `main.py` file, adding code in different parts of the file as we go along. Each code snippets will contain some existing code to give you an idea of where in the file the new additions should be placed. The line `# ...` will be used to represent existing code that has been left out for brevity.

Setting up the scaffolding

We will start with the following code in `main.py`, which draws a black screen:

```python
import pygame

WIDTH = 400
HEIGHT = 300
BACKGROUND = (0, 0, 0)

def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    clock = pygame.time.Clock()

    while True:
        screen.fill(BACKGROUND)
        pygame.display.flip()

        clock.tick(60)

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

At the top of the file, we import `pygame`. Following that, we set the width and height of the screen in pixels, and the background color. This last value is an RGB[207] tuple, and will make our background black. To use a white background instead, we would write `(255, 255, 255)`.

Then, in the `main` method, we initiate PyGame, create both the screen and the clock, and start the **game loop**, which is this code:

```python
    while True:
        screen.fill(BACKGROUND)
        pygame.display.flip()

        clock.tick(60)
```

The game loop is where everything happens. Because our game runs in real time, our code needs to constantly poll for the user's keystrokes and mouse movements, and constantly redraw the screen in

[207] https://www.google.com/search?q=rgb+color+picker

response to those keystrokes and mouse movements, and to other events[208] in the game. We achieve this with an infinite while loop. PyGame uses the final `clock.tick(60)` line in this loop to adjust the game's framerate in line with how long each iteration of the loop takes, in order to keep the game running smoothly.

Now let's draw something on this black screen. Our game is going to have two sprites: an alien, which will be the player, and a box. To avoid code duplication, let's create a `Sprite` parent class before we create either of those. This class will inherit from the `pygame.sprite.Sprite` class, which gives us useful methods for collision detection – this will become important later on.

```python
class Sprite(pygame.sprite.Sprite):
    def __init__(self, image, startx, starty):
        super().__init__()

        self.image = pygame.image.load(image)
        self.rect = self.image.get_rect()

        self.rect.center = [startx, starty]

    def update(self):
        pass

    def draw(self, screen):
        screen.blit(self.image, self.rect)

def main():
```

As this class will be the parent for all other objects in our game, we're keeping it quite simple. It has three methods:

- `__init__`, which will create the sprite with a given image and a PyGame rectangle[209] based on that image. This rectangle will initially be placed at the position specified by `startx` and `starty`. The sprite's rectangle is what PyGame will use for sprite movement and collision detection.
- `update`, which we'll use in child classes to handle events, such as key presses, gravity and collisions.
- `draw`, which we use to draw the sprite. We do this by blitting[210] it onto the screen.

Now we can create our `Player` and `Box` objects as child classes of `Sprite`:

[208] https://www.pygame.org/docs/ref/event.html
[209] https://www.pygame.org/docs/ref/rect.html
[210] https://en.wikipedia.org/wiki/Bit_blit

```python
class Sprite(pygame.sprite.Sprite):
    # ...
class Player(Sprite):
    def __init__(self, startx, starty):
        super().__init__("p1_front.png", startx, starty)

class Box(Sprite):
    def __init__(self, startx, starty):
        super().__init__("boxAlt.png", startx, starty)

def main():
```

We'll add more code to the player later, but first let's draw these sprites on the screen.

Drawing the sprites

Let's go back to our `main` function and create our sprites. We'll start with the player:

```python
def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    clock = pygame.time.Clock()

    player = Player(100, 200)
```

Then we need to put boxes under the player's feet. As we will be placing multiple sprites, we'll create a PyGame sprite group[211] to put them in.

```python
    player = Player(100, 200)

    boxes = pygame.sprite.Group()
```

Our box sprites are 70 pixels wide, and we need to span over the screen width of 400 pixels. We can do this in a `for` loop using Python's `range`[212]:

[211]https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group
[212]https://docs.python.org/3/library/functions.html#func-range

```
    player = Player(100, 200)

    boxes = pygame.sprite.Group()
    for bx in range(0,400,70):
        boxes.add(Box(bx,300))
```

Now we need to go back to the game loop and add some code to make things happen. First, we'll have PyGame put new events on the event queue, and then we'll call the player's `update` function. This function will handle the events[213] generated by `pygame.event.pump()`.

```
    while True:
        pygame.event.pump()
        player.update()
        # ...
```

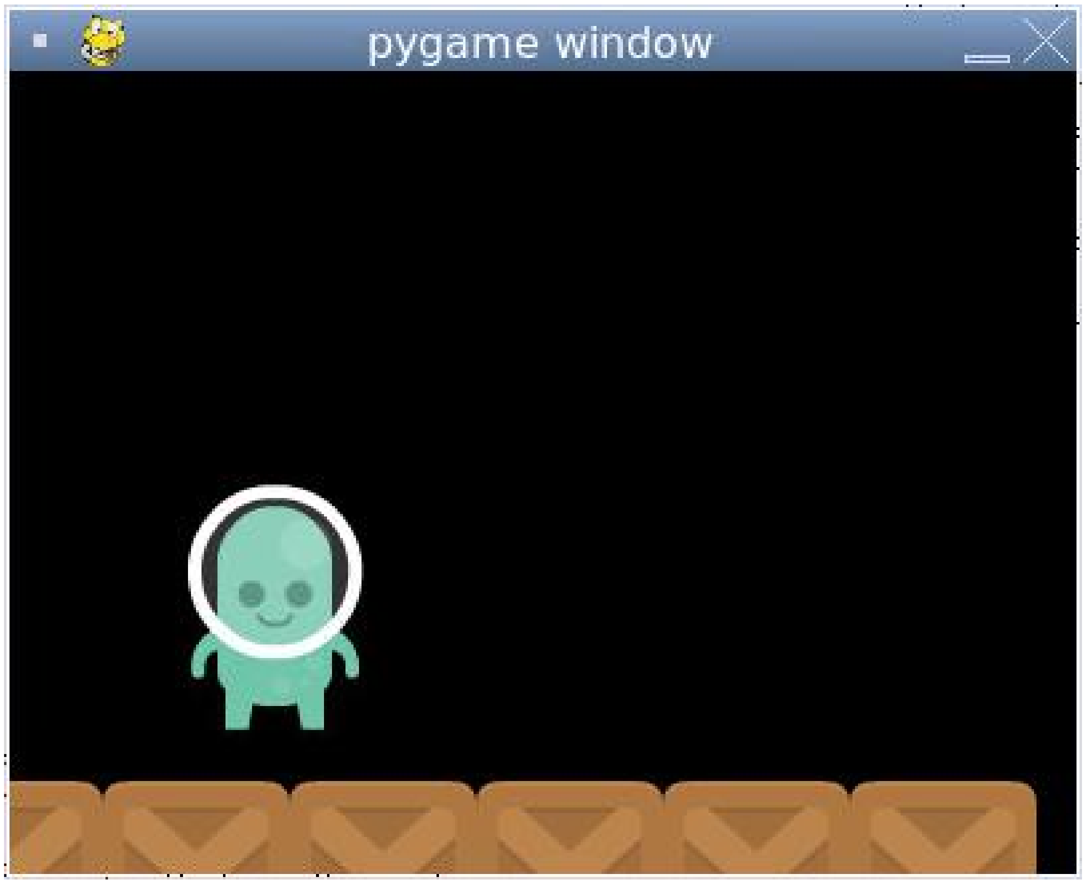
For a more complex game, we would want to loop through a number of sprites and call each one's `update` method, but for now just doing this with the player is sufficient. Our boxes won't have any dynamic behavior, so there's no need to call their `update` methods.

In contrast to `update`, we need all our sprites to draw themselves. After drawing the background, we'll add a call to the player's draw method. To draw the boxes, we can call PyGame's `Group.draw`[214] on our `boxes` group.

```
    while True:
        pygame.event.pump()
        player.update()

        # Draw loop
        screen.fill(BACKGROUND)
        player.draw(screen)
        boxes.draw(screen)
        pygame.display.flip()

        clock.tick(60)
```

Our game loop is now set up to update and draw every sprite in the game in each cycle of the game loop. If you run the game now, you should see both the player and the line of boxes on the screen.

[213]https://www.pygame.org/docs/ref/event.html
[214]https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.draw



Next, we're going to add some user interaction.

Making the Player Walk

Let's return to the `Player` object and make it mobile. We'll move the player using `pygame.Rect.move_ip`, which moves a given rectangle by a given vector. This will be wrapped in a `move` method, to simplify our code. Create this method now:

```python
class Player(Sprite):
    # ...
    def move(self, x, y):
        self.rect.move_ip([x,y])
```

Now that we have a way to move the player, it's time to add an `update` method so that this movement can be triggered by key presses. Add an empty `update` method now:

```
class Player(Sprite):
    def __init__(self, startx, starty):
        super().__init__("p1_front.png", startx, starty)

    def update(self):
        pass

    def move(self, x, y):
        self.rect.move_ip([x,y])
```

PyGame provides a couple of different ways to check the state of the keyboard. By default, its event queue collects `KEY_DOWN` and `KEY_UP` events when particular keys are pressed and released. Using a `KEY_DOWN` event to move the player seems like the logical thing to do, but because the event is only triggered in same update loop in which the key is first pressed, this would force us to rapidly tap an arrow key to keep moving in a single direction.

We need a way to move the player whenever an arrow key is held down, not just after it's pressed. So instead of relying on events, we will query the current status of all keyboard keys with `pygame.key.get_pressed()`:

```
    def update(self):
        # check keys
        key = pygame.key.get_pressed()
```

This method returns a tuple of `0`s and `1`s showing the pressed status of each key on the keyboard. We can thus detect whether the left or right arrow key is currently pressed by indexing the tuple with PyGame's keyboard constants[215], like so:

```
    def update(self):
        # check keys
        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.move(-1,0)
        elif key[pygame.K_RIGHT]:
            self.move(1,0)
```

Run the game. You should now be able to move the player left and right, albeit very slowly. Let's speed things up and reduce our code's reliance on magic numbers at the same time by giving the player a `speed` variable.

[215]https://www.pygame.org/docs/ref/key.html

```python
class Player(Sprite):
    def __init__(self, startx, starty):
        super().__init__("p1_front.png", startx, starty)

        self.speed = 4

    def update(self):
        # check keys
        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.move(-self.speed,0)
        elif key[pygame.K_RIGHT]:
            self.move(self.speed,0)
```

Right now the player glides from side to side, but we have already uploaded images for a walk cycle[216] animation, so let's implement that now. First, we'll add some image loading code to our player's `__init__` method:

```python
    def __init__(self, startx, starty):
        super().__init__("p1_front.png", startx, starty)
        self.stand_image = self.image

        self.walk_cycle = [pygame.image.load(f"p1_walk{i:0>2}.png") for i in range(1\
,12)]
        self.animation_index = 0
        self.facing_left = False

        self.speed = 4
```

In this code, we first designate our initial alien image as `stand_image`. This will allow us to use it for the player when he's standing still. We then load our walking images into a list called `walk_cycle`, using Python's string formatting[217] to get the correct filename format (`p1_walk01.png` -> `p1_walk11.png`). We then create `self.animation_index`, which will record which frame of the walk cycle the player is on, and `self.facing_left` which will help us to flip the right-facing walking images when the player is walking left.

Now let's implement the actual animation. Create a new method called `walk_animation`:

[216]https://en.wikipedia.org/wiki/Walk_cycle
[217]https://docs.python.org/3/library/string.html#string-formatting

```python
class Player(Sprite):
    # ...
    def walk_animation(self):
        self.image = self.walk_cycle[self.animation_index]
        if self.facing_left:
            self.image = pygame.transform.flip(self.image, True, False)

        if self.animation_index < len(self.walk_cycle)-1:
            self.animation_index += 1
        else:
            self.animation_index = 0
```

Here we're setting the player's current image to the frame of the walk cycle we're currently on. If the player is facing left, we use `pygame.transform.flip` to horizontally flip his sprite (the last two arguments are for horizontal and vertical flipping, respectively). Then we animate the player by incrementing the `animation_index`, unless the animation is in its penultimate frame, in which case we return to the start of the animation.

Let's add this to our `update` method now:

```python
    def update(self):
        # ...
        # check keys
        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.facing_left = True
            self.walk_animation()
            self.move(-self.speed,0)
        elif key[pygame.K_RIGHT]:
            self.facing_left = False
            self.walk_animation()
            self.move(self.speed,0)
        else:
            self.image = self.stand_image
```

If we're moving left or right, we set `self.facing_left` appropriately and call `self.walk_animation`. Otherwise, we set the player's image to `self.stand_image`.

Run the game now to see the player's walk cycle in motion. After that, it's time to make him jump.

Making the Player Jump

For our player to be able to jump, we need to implement four things:

1. Upward motion triggered by the up arrow key.
2. Gravity, to bring the player back down after reaching the top of his jump.
3. Collision detection, so the player doesn't fall through the ground.
4. A jumping animation.

Triggering the jump

To simply make the player move up, we can just add another `elif`, like so:

```
    def update(self):
        # ...
        if key[pygame.K_LEFT]:
            self.facing_left = True
            self.walk_animation()
            self.move(-self.speed,0)
        elif key[pygame.K_RIGHT]:
            self.facing_left = False
            self.walk_animation()
            self.move(self.speed,0)
        elif key[pygame.K_UP]:
            self.move(0,-self.speed)
        else:
            self.image = self.stand_image
```

If you try the game now, you should notice a couple of problems with this approach. Besides the lack of gravity, we can only jump straight up, and must release the left and right arrow keys before we may do so. Much of the gameplay in platformers is reliant on the player's ability to jump to the left or right, so this won't do. To fix this, we'll change our last `elif` to a separate `if` statement:

```
        if key[pygame.K_LEFT]:
            self.facing_left = True
            self.walk_animation()
            self.move(-self.speed,0)
        elif key[pygame.K_RIGHT]:
            self.facing_left = False
            self.walk_animation()
            self.move(self.speed,0)
        else:
            self.image = self.stand_image

        if key[pygame.K_UP]:
            self.move(0,-self.speed)
```

We also probably want to be able to jump at a different speed to our walking pace, so let's define another variable and use it.

```python
    def __init__(self, startx, starty):
        # ...
        self.speed = 4
        self.jumpspeed = 20

    def update(self):
        # ...
        if key[pygame.K_UP]:
            self.move(0,-self.jumpspeed)
```

That's better, but now we really need some gravity!

Adding gravity

Up until now, we've had only a single operation manipulating our horizontal or vertical speed per update loop. With the addition of gravity, this will change, so we need to restructure our code to calculate our net horizontal and vertical movement before calling `move`. Let's change the `update` function like so:

```python
    def update(self):
        hsp = 0 # horizontal speed
        vsp = 0 # vertical speed

        # check keys
        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.facing_left = True
            self.walk_animation()
            hsp = -self.speed
        elif key[pygame.K_RIGHT]:
            self.facing_left = False
            self.walk_animation()
            hsp = self.speed
        else:
            self.image = self.stand_image

        if key[pygame.K_UP]:
            vsp = -self.jumpspeed

        # movement
        self.move(hsp,vsp)
```

We've added two variables, `hsp` and `vsp`, to represent our horizontal speed and vertical speed. Instead of calling `move` when each key is pressed, we work with these variables throughout the `update` method and then pass their final values into a single `move` call at the end.

But wait! It makes sense for horizontal speed to be set to 0 at the start of every update loop, because it is directly controlled by the player's key presses. When the left arrow is held down, the player moves left at a speed of 4 pixels per loop; when the left arrow is released, the player instantly stops. Vertical speed will be less controllable – while pressing the up arrow will initiate a jump, releasing it should not stop the player in mid-air. Therefore, vertical speed must persist between loops.

We can accomplish this by moving the `vsp` definition into `__init__` and making it an instance variable.

```
    def __init__(self, startx, starty)
        # ...
        self.vsp = 0 # vertical speed

    def update(self):
        hsp = 0 # horizontal speed
        # ...
        if key[pygame.K_UP]:
            self.vsp = -self.jumpspeed

        # movement
        self.move(hsp,self.vsp)
```

Now we can implement gravity. We'll do this by adding a small constant to the player's vertical speed (`vsp`) until it reaches terminal velocity[218].

```
    def __init__(self, startx, starty)
        # ...
        self.gravity = 1

    def update(self):
        # ...
        if key[pygame.K_UP]:
            self.vsp = -self.jumpspeed

        # gravity
```

[218]https://en.wikipedia.org/wiki/Terminal_velocity

```
        if self.vsp < 10: # 9.8 rounded up
            self.vsp += self.gravity

        # movement
        self.move(hsp,self.vsp)
```

Start up the game now, and the player will fall straight down, through the ground and off the screen. Gravity's working, but we need somewhere for the player to land.

Adding collision detection

Collision detection is a key element of most graphical games. In PyGame, the bulk of collision detection involves checking whether rectangles intersect with each other. Luckily, PyGame provides a number of useful built-ins for doing this, so we won't have to think too much about the internal workings of collisions.

Let's add some collision detection now, near the top of our `update` method. We'll create a variable called `onground` and set it to the result of `pygame.sprite.spritecollideany()`.

```
    def update(self):
        hsp = 0 # horizontal speed
        onground = pygame.sprite.spritecollideany(self, boxes)
```

This PyGame method takes two arguments: a single sprite and a group of sprites. It returns whether the sprite given as the first argument, i.e. the player, has a collision with any of the sprites in the group given as the second argument, i.e. the boxes. So we'll know that the player is on a box when it returns `True`.

We can pass the `boxes` group into the player's `update` method by making a couple of code changes:

```
    def update(self, boxes):
        hsp = 0 # horizontal speed
        onground = pygame.sprite.spritecollideany(self, boxes)
        # ...

def main():
    # ...
    while True:
        pygame.event.pump()
        player.update(boxes)
```

Now that we can tell whether the player is on the ground, we can prevent jumping in mid-air by adding a condition to our jump code:

```
    def update(self, boxes):
        # ...
        if key[pygame.K_UP] and onground:
            self.vsp = -self.jumpspeed
```

To stop the player from falling through the ground, we'll add the following code to our gravity implementation:

```
    def update(self, boxes):
        # ...
        # gravity
        if self.vsp < 10 and not onground: # 9.8: rounded up
            self.vsp += self.gravity

        # stop falling when the ground is reached
        if self.vsp > 0 and onground:
            self.vsp = 0
```

Adding a jumping animation

Lastly, we'll use our last alien image (`p1_jump.png`) to give our player a jumping animation. First create `self.jump_image` in `__init__`:

```
    def __init__(self, startx, starty):
        super().__init__("p1_front.png", startx, starty)
        self.stand_image = self.image
        self.jump_image = pygame.image.load("p1_jump.png")
        # ...
```

Then create the following `Player` method:

```
    def jump_animation(self):
        self.image = self.jump_image
        if self.facing_left:
            self.image = pygame.transform.flip(self.image, True, False)
```

Our jump animation only has one frame, so the code is much simpler than what we used for our walking animation. To trigger this method when the player is in the air, alter the gravity implementation like so:

```
    def update(self, boxes):
        # ...
        # gravity
        if self.vsp < 10 and not onground: # 9.8 rounded up
            self.jump_animation()
            self.vsp += self.gravity
```

Run the game, and you should be able to run and jump! Be careful not to fall off the edge.

Refining the Game

At this point, we have our game working on a basic level, but it could use some refinements. First, the jumping is quite unresponsive to user input: pressing the up arrow for any length of time results in the same size jump. Second, our collision detection will only prevent the player from falling through the floor, not walking through walls or jumping through the ceiling.

We're going to iterate on our code to fix both of these shortcomings.

Making jumps variable

It would be nice if the player could control the height of their jump by holding the jump key down for different lengths of time. This is fairly simple to implement – we just need a way to reduce the speed of a jump if the player releases the jump key while the player is still moving up. Add the following code to the player's `__init__` method.

```
class Player(Sprite):
    def __init__(self, startx, starty):
        # ...
        self.min_jumpspeed = 3
        self.prev_key = pygame.key.get_pressed()
```

Here we've added a `prev_key` instance variable that will track the state of the keyboard in the previous update loop, and a `min_jumpspeed` variable, which will be the smallest jump we'll allow the player to do, by just tapping the jump key.

Now let's add variable jumping to the `update` method, between the code that handles the up arrow key and the code that handles gravity:

```
    def update(self, boxes)
        # ...
        if key[pygame.K_UP] and onground:
            self.vsp = -self.jumpspeed

        # variable height jumping
        if self.prev_key[pygame.K_UP] and not key[pygame.K_UP]:
            if self.vsp < -self.min_jumpspeed:
                self.vsp = -self.min_jumpspeed

        self.prev_key = key

        # gravity
        if self.vsp < 10: # 9.8 rounded up
            self.vsp += self.gravity
```

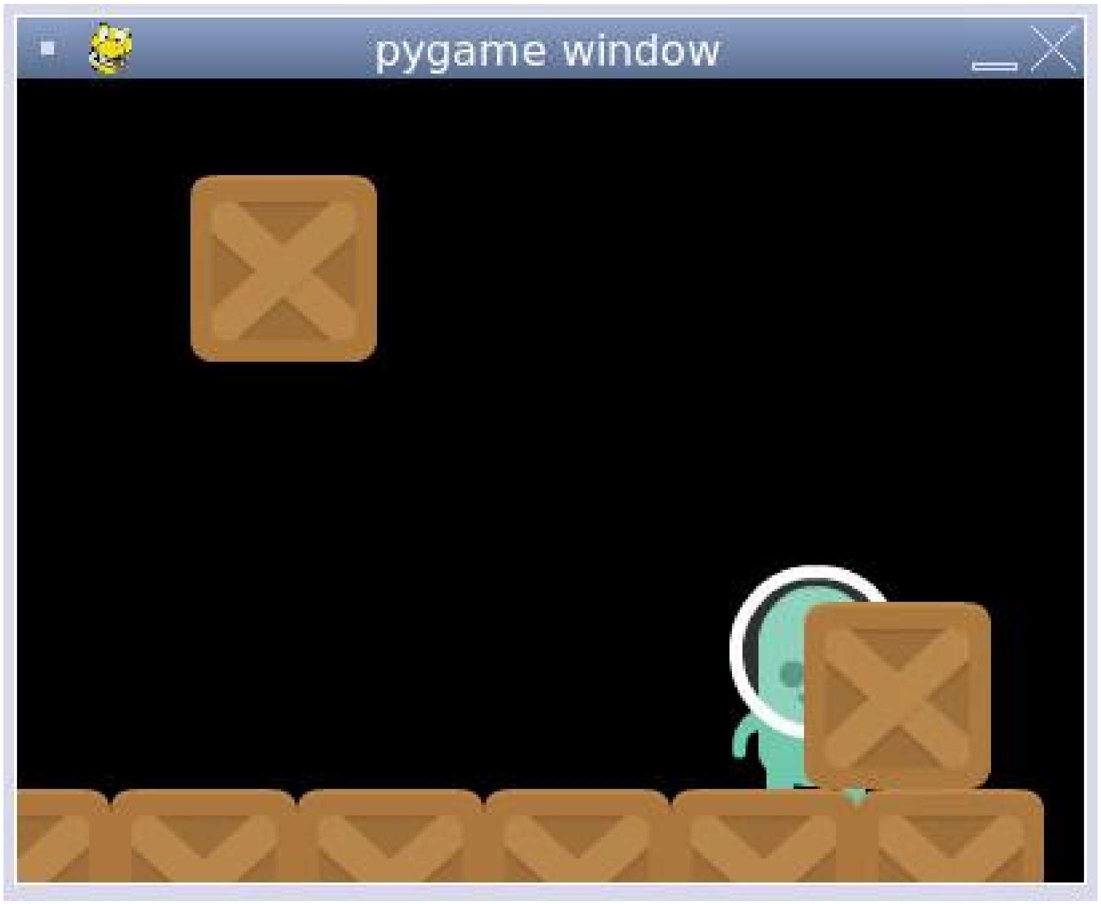
The `if` statement we've just added will evaluate to `True` if the up arrow key was pressed in the previous loop but is not longer pressed, i.e. it has been released. When that happens, we cut off the player's jump by reducing its speed to the `min_jumpspeed`. We then set `self.prev_key` to the current keyboard state in preparation for the next loop.

Try the game now, and you should notice a different height of jump when lightly tap the up arrow key versus when you hold it down. Play around with the value of `min_jumpspeed` and see what difference it makes.

Refining collision detection

As mentioned above, the only collision detection we've implemented applies to the ground beneath the player's feet, so he will be able to walk through walls and jump through ceilings. See this for yourself by adding some boxes above and next to the player in the `main` method.

```
def main():
    # ...
    boxes = pygame.sprite.group()
    for bx in range(0,400,70):
        boxes.add(Box(bx,300))

    boxes.add(Box(330,230))
    boxes.add(Box(100,70))
```



Another issue that you may have already noticed is that the player sinks into the ground after some jumps – this results from the imprecision of our collision detection.

We're going to fix these problems by making a subtle change to how we deal with collisions with boxes. Rather than deciding that we're on the ground when the player sprite is in collision with a box, we'll check whether the player is 1 pixel above a collision with a box. We'll then apply the same principle for left, right and up, stopping the player just before a collision.

First, let's give the player a `check_collision` method to make these checks:

```python
class Player(Sprite):
    # ...
    def check_collision(self, x, y, boxes):
        self.rect.move_ip([x,y])
        collide = pygame.sprite.spritecollideany(self, boxes)
        self.rect.move_ip([-x,-y])
        return collide
```

Here, we're moving the player by a specified amount, checking for a collision, and then moving the player back. This back and forth movement happens before the player is drawn to the screen, so the user won't notice anything.

Let's change our onground check to use this method:

```
    def update(self, boxes):
        hsp = 0
        onground = self.check_collision(0, 1, boxes)
```

Run the game now, and you may be able to notice a very slight difference in how the player stands on the ground from before.

This doesn't yet solve our horizontal and upward collisions problems, though. For that, we'll need to implement our new `check_collision` method directly into the player's `move` method. The first thing we'll need to do is prepare the `x` and `y` parameters for additional processing:

```
    def move(self, x, y):
        dx = x
        dy = y
        self.rect.move_ip([dx,dy])
```

Then we're going to check for collisions, so we need to start passing `boxes` into `move`. We're going to do this for `x` and `y` separately, starting with `y`. We'll check for a collision after moving `dy` pixels vertically, decrementing `dy` until we no longer collide with a box:

```
    def update(self, boxes):
        # ...
        # movement
        self.move(hsp, self.vsp, boxes)

    def move(self, x, y, boxes):
        dx = x
        dy = y

        while self.check_collision(0, dy, boxes):
            dy -= 1

        self.rect.move_ip([dx,dy])
```

But wait! This code will only work as intended if we're moving down and `dy` is positive. If `dy` is negative, this will just move us further into a collision, not away from it. To fix this, we'll need to import `numpy` at the top of our file, so we can use `numpy.sign`.

```python
import pygame, numpy
# ...
```

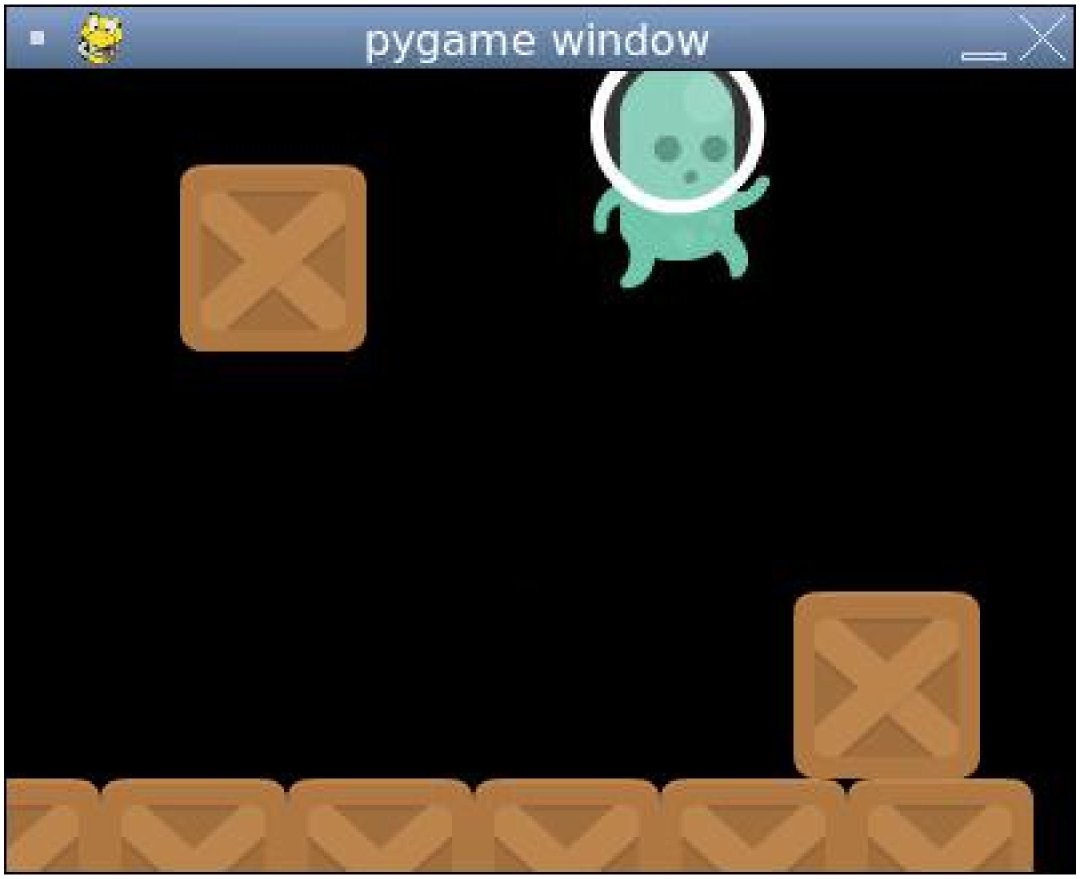
`numpy.sign` takes an integer and returns 1 if it's positive, -1 if it's negative, and `0` if it's `0`. This is exactly the functionality we need!

```python
    def move(self, x, y, boxes):
        dx = x
        dy = y

        while self.check_collision(0, dy, boxes):
            dy -= numpy.sign(dy)

        self.rect.move_ip([dx,dy])
```

Now do the same for `dx`. As we've already figured out the appropriate `dy` for our movement, we'll include that in the collision check.

```python
    def move(self, x, y, boxes):
        dx = x
        dy = y

        while self.check_collision(0, dy, boxes):
            dy -= numpy.sign(dy)

        while self.check_collision(dx, dy, boxes):
            dx -= numpy.sign(dx)

        self.rect.move_ip([dx,dy])
```

Run the game. The player should now stop when he runs into a wall or jumps into a ceiling.



Where Next?

If you'd like to continue working on this game, you can find a large number of matching art assets here[219]. Try implementing some of these features:

- More jump refinements, such as jump grace time and input buffering[220].
- Moving platforms.
- Slopes.
- Water and swimming mechanics.
- Hazards like spike pits and enemies who are also subject to gravity.
- Double jumping[221].

You can find our game repl and fork it here[222].

[219]https://www.kenney.nl/assets/platformer-art-pixel-redux
[220]https://www.gamasutra.com/blogs/LisaBrown/20171005/307063/GameMaker_Platformer_Jumping_Tips.php
[221]https://en.wikipedia.org/wiki/Glossary_of_video_game_terms#Double_jump
[222]https://repl.it/@ritza/2D-platform-game

Next Tutorial

Up next, we'll learn a simple and flexible implementation of a static site generator. The tutorial will walk you through advanced file handling and by the end of it you'll have a fully functional SSG that you can use as is or customise to add your desired functionality.

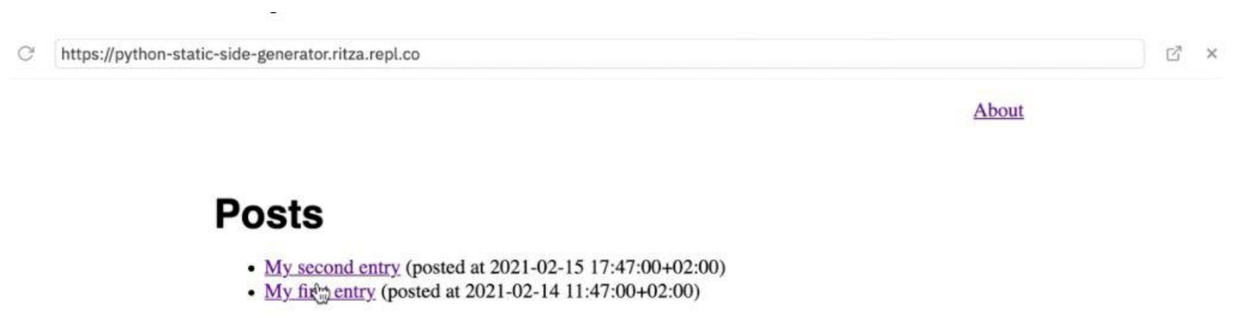
Create a static site generator with Python and Replit

A static site generator (SSG) is a tool for building informational websites such as blogs and documentation repositories. SSGs allow technical users to build websites that are faster and more secure than ones running on dynamic platforms such as Wordpress, without having to write each HTML page.

There are many SSGs out there already, such as Jekyll and Hugo, but many people opt to write their own – either so that they fully understand it and can be more productive, or to meet custom needs.

After this tutorial, you'll:

- Be able to build a simple but flexible SSG in Python in under 100 lines of code.
- Understand advanced file and directory handling.
- Know how to build a configurable tool for technical users.


https://python-static-side-generator.ritza.repl.co
About
Posts
My second entry (posted at 2021-02-15 17:47:00+02:00)
My first entry (posted at 2021-02-14 11:47:00+02:00)

*click to open gif[223]

At the end, you'll have a full SSG that you can use as is or extend for your own requirements.

[223]https://docs.replit.com/images/tutorials/static-site-generator/generator_functionality.gif

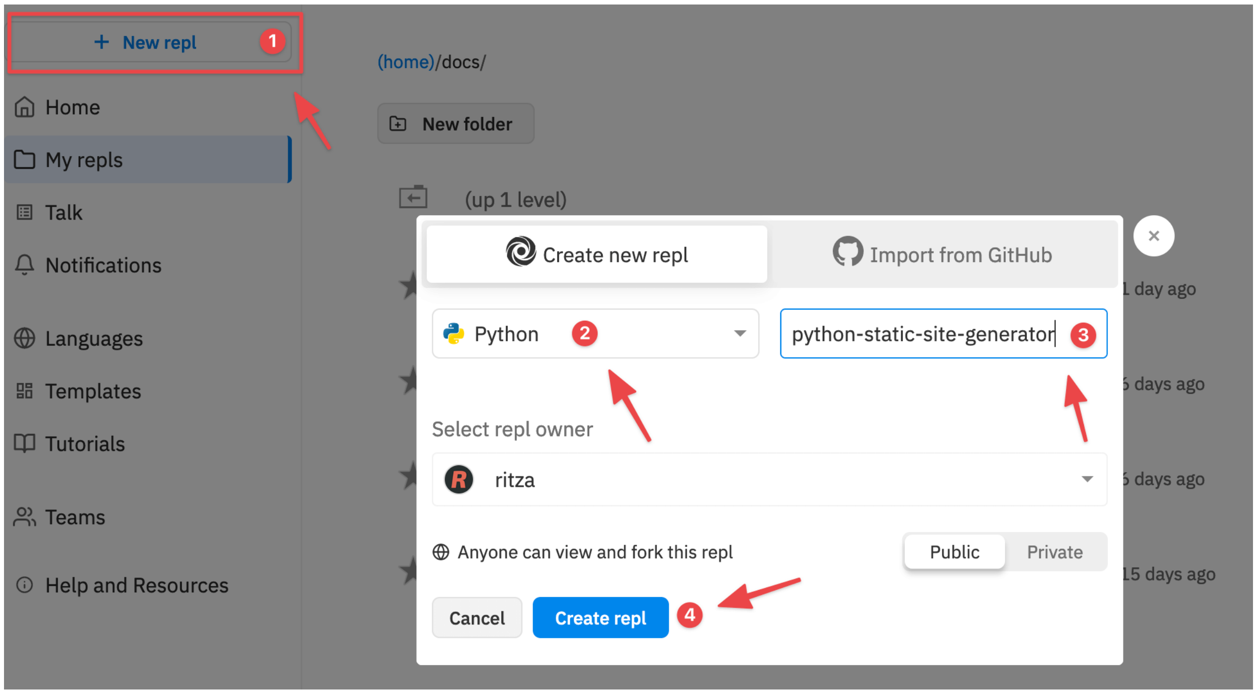
Building a proof of concept

A basic SSG takes in a list of Markdown[224] files, converts them to HTML, and inserts them into different predefined HTML templates. Beyond that, most SSGs have the concept of frontmatter[225] to define metadata such as title and publish date for each Markdown file. SSGs also usually have global configuration files, containing general information about the site such as its name and domain.

Before we start dealing with files, we're going to implement our SSG using strings. This will serve as an initial proof of concept.

Setting up and defining the flow

We'll start by defining the main functions we'll use. Create a new Python repl and enter the following code in `main.py`.



Python 2

python-static-site-generator 3

Select repl owner

ritza

Anyone can view and fork this repl

Public Private

224https://en.wikipedia.org/wiki/Markdown
225https://jekyllrb.com/docs/front-matter/

```python
def load_config():
    pass

def load_content_items():
    pass

def load_templates():
    pass

def render_site(config, content, templates):
    pass

def main():
    config = load_config()
    content = load_content_items()
    templates = load_templates()
    render_site(config, content, templates)

main()
```

This skeleton defines the program flow:

- Load the global site configuration.
- Load the content files containing Markdown and frontmatter.
- Load the HTML templates.
- Render the site using everything we've loaded above.

Throughout this tutorial, we will keep to this flow, even as we expand and refine its individual elements.

Parsing content and templates

Now we need to import some modules. At the top of main.py, enter the following line.

```python
import markdown, jinja2, toml, re
```

All four of these modules are essentially parsers:

- `markdown`: This module will render Markdown.
- `jinja2`: The Jinja templating language, which we will use to create HTML templates that we can enhance with Python-esque code.

- `toml`: We will use TOML (Tom's Obvious, Minimal Language) for post frontmatter and global configuration.
- `re`: We'll use Python's regular expressions (regex) module for some additional, very light, parsing not provided by the three packages above.

Now that we have our parsers, let's add some content to parse. Add a TOML string for global site configuration at the top of the `main` function.

```
def main():
    config_string = """
    title = "My blog"
    """
```

For now, this just defines the title of our site. Change it to whatever you want. To load this config, we'll use `toml.loads` on its content. Go to the `load_config` function at the top of `main.py` and give it the following parameter and content.

```
def load_config(config_string):
    return toml.loads(config_string)
```

To use this function, go back to the main function and pass config_string to this line in the main function.

```
config = load_config(config_string)
```

Now let's create a couple of content strings below the config string. We're going to format these strings with a block of TOML metadata terminated by a row of five plus signs (`+++++`). The rest of the string will contain Markdown-formatted text. Add this block of code below the definition of `config_string` in the `main` function.

```
content_strings = ["""
title = "My first entry"
date = 2021-02-14T11:47:00+02:00
+++++

Hello, welcome to my **blog**
""",
"""
title = "My second entry"
date = 2021-02-15T17:47:00+02:00
+++++

This is my second post.
"""]
```

We'll parse these strings in our `load_content_items` function. Give the function a `content_strings` parameter and add the following code.

```python
def load_content_items(content_strings):
    items = []
    for item in content_strings:
        frontmatter, content = re.split("^\s*\+\+\+\+\+\s*$", item, 1, re.MULTILINE)
        item = toml.loads(frontmatter)
        item['content'] = markdown.markdown(content)

        items.append(item)

    # sort in reverse chronological order
    items.sort(key=lambda x: x["date"],reverse=True)

    return items
```

Here we use a for loop to construct a list of items from our item strings. For each one, we split up the frontmatter and content on a regular expression that will match a line of text containing five plus signs. We pass in `1` as `re.split`'s `maxsplit` parameter to ensure that we only split on the first matched line, and `re.MULTILINE` so that our regex will work correctly in a multiline string.

We then use `toml.loads()` to convert the frontmatter into a dictionary. Finally, we convert the Markdown in `content` into HTML and add it to the dictionary we just created. The result will be a dictionary that looks something like this:

```
{
    'title': 'My first entry',
    'date': datetime.datetime(2021, 2, 14, 11, 47, tzinfo=<toml.tz.TomlTz object at \
0x7f4032da6eb0>),
    'content': '<p>Hello, welcome to my <strong>blog</strong>.</p>'
}
```

Finally, since this is a blog site, we're sorting our `items` dictionary in reverse chronological order. We do this by using Python's `list.sort` method's custom sort functionality to sort by each list entry's `date` value. The `key` parameter takes a function which it will pass each value into and use the return value to sort the list. For brevity, we've created an in-line anonymous function using a lambda expression[226].

Back in our `main` function, let's pass `content_strings` to the `load_content_items` function call.

[226]https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

```
content = load_content_items(content_strings)
```

Now let's create a template string below the content strings. This is just some HTML with Jinja code in `{{ }}` and `{% %}` blocks. Add this code block beneath the definition of `content_strings` in the `main` function.

```
template_string = """
<!DOCTYPE html>
<html>
    <body>
        <h1>{{ config.title }}</h1>
        {% for post in content %}
        <article>
            <h2>{{ post.title }}</h2>
            <p>Posted at {{ post.date }}</p>
            {{ post.content }}
        </article>
        {% endfor %}
    </body>
</html>
"""
```

Each of the values inside `{{ }}` blocks is something we've assembled in the preceding code: `config.title` from the config strings, `content` from the content strings, and the individual values inside the Jinja for loop from each item in the `content` list. Note that in Jinja, `post.title` is equivalent to `post["title"]`.

To load this template, we will add the following parameter and code to the `load_templates` function.

```
def load_templates(template_string):
    return jinja2.Template(template_string)
```

We'll also change the `load_templates` function invocation in the `main` function.

```
templates = load_templates(template_string)
```

Rendering the site

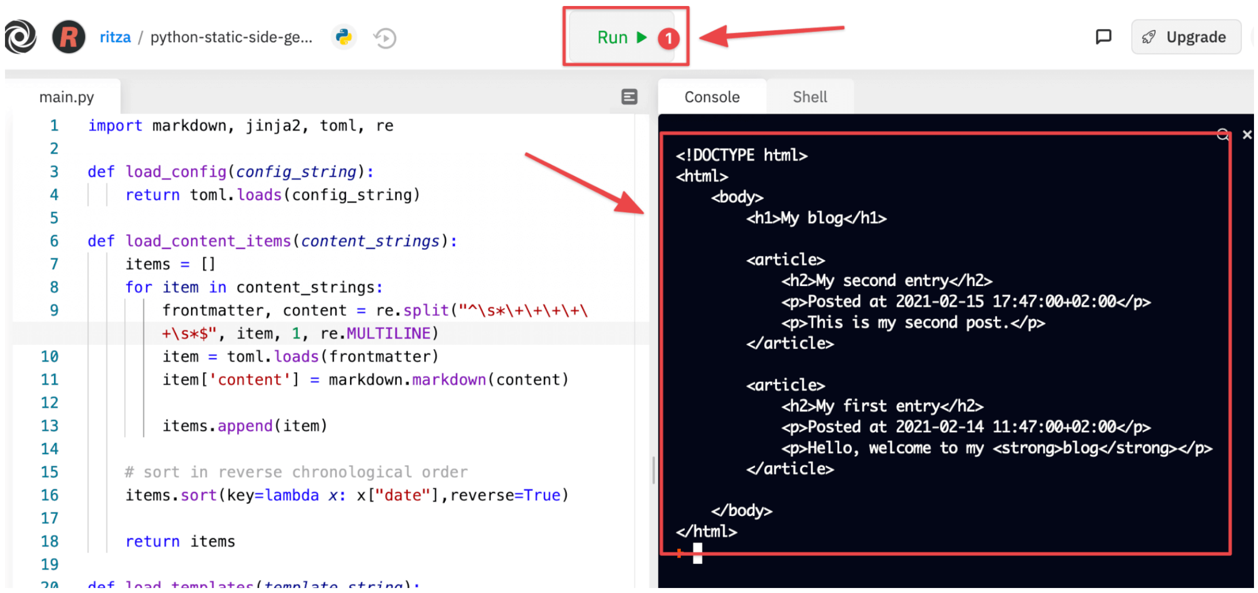
Now let's populate the template with our config and content data. We'll do this using the template's `render()` method. This method takes a list of keyword arguments which it will use to resolve the variable references template's `{{ }}` and `{% %}` blocks.

In the `render_site` function, add the following code:

```python
def render_site(config, content, template):
    print(template.render(config=config, content=content))
```

As our `render_site` invocation in `main` already takes the correct arguments, we can run our code now. The result should look like this:

```html
<!DOCTYPE html>
<html>
    <body>
        <h1>My blog</h1>

        <article>
            <h2>My second entry</h2>
            <p>Posted at 2021-02-15 17:47:00+02:00</p>
            <p>This is my second post.</p>
        </article>

        <article>
            <h2>My first entry</h2>
            <p>Posted at 2021-02-14 11:47:00+02:00</p>
            <p>Hello, welcome to my <strong>blog</strong></p>
        </article>

    </body>
</html>
```



```python
import markdown, jinja2, toml, re

def load_config(config_string):
    return toml.loads(config_string)

def load_content_items(content_strings):
    items = []
    for item in content_strings:
        frontmatter, content = re.split("^\s*\+\+\+\+\
+\s*$", item, 1, re.MULTILINE)
        item = toml.loads(frontmatter)
        item['content'] = markdown.markdown(content)

        items.append(item)

    # sort in reverse chronological order
    items.sort(key=lambda x: x["date"],reverse=True)

    return items

def load_templates(template_string):
```

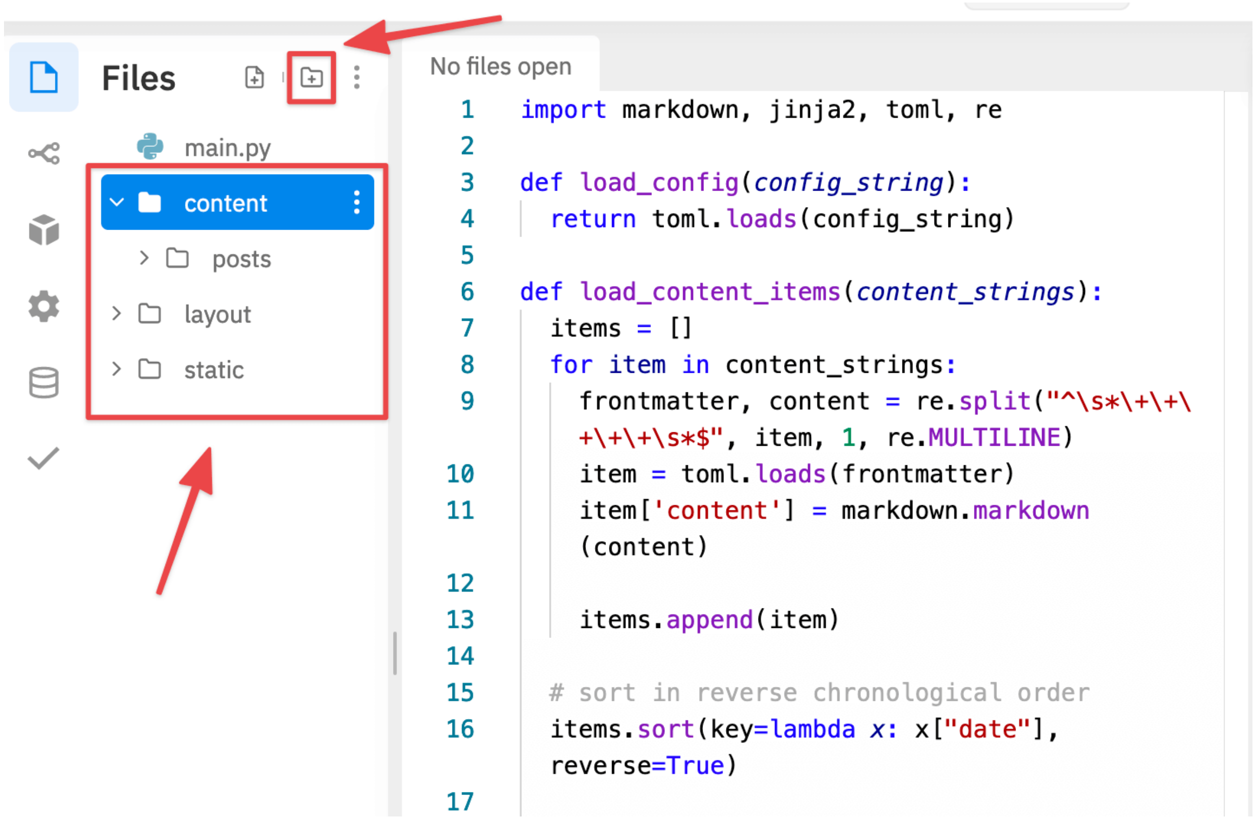
```
<!DOCTYPE html>
<html>
    <body>
        <h1>My blog</h1>

        <article>
            <h2>My second entry</h2>
            <p>Posted at 2021-02-15 17:47:00+02:00</p>
            <p>This is my second post.</p>
        </article>

        <article>
            <h2>My first entry</h2>
            <p>Posted at 2021-02-14 11:47:00+02:00</p>
            <p>Hello, welcome to my <strong>blog</strong></p>
        </article>

    </body>
</html>
```

We now have the core of our SSG. Modify the content of one of the content strings and the output will change. Add new variables to each content file's frontmatter and the template, and they will propagate through without any changes to the Python code.

Next, let's create and ingest some files.

Blog generator

First, we need to create a directory structure. In the file pane of your repl, create four directories: `content`, `content/posts`, `layout` and `static`. Your file pane should now look like this:



```
import markdown, jinja2, toml, re

def load_config(config_string):
    return toml.loads(config_string)

def load_content_items(content_strings):
    items = []
    for item in content_strings:
        frontmatter, content = re.split("^\s*\+\+\
+\+\+\s*$", item, 1, re.MULTILINE)
        item = toml.loads(frontmatter)
        item['content'] = markdown.markdown
(content)

        items.append(item)

    # sort in reverse chronological order
    items.sort(key=lambda x: x["date"],
reverse=True)

```

We will put our Markdown files in `content/posts`, our Jinja files in `layout` and unprocessed files like CSS stylesheets and images in `static`. We're using `content/posts` so we can create different content types later on, such as undated pages like "About".

Creating input files

First, we'll create our config file `config.toml`. In addition to the title value, we'll give it a base URL based on our repl's URL.

`config.toml`

```
title = "My blog"
baseURL = "https://YOUR-REPL-NAME-HERE.YOUR-REPLIT-USERNAME.repl.co"
```

Replace the all-caps text with the relevant values.

Now let's put our content strings into post files. Create two files with the following content:

`content/posts/first-post.md`

```
title = "My first entry"
date = 2021-02-14T11:47:00+02:00
+++++

Hello, welcome to my **blog**.
```

`content/posts/second-post.md`

```
title = "My second entry"
date = 2021-02-15T17:47:00+02:00
+++++

This is my second post.
```

Make as many additional posts as you want. Just remember to give each one a title, correctly formatted datestamp and some Markdown content. File names should be lowercase with no spaces, ending in the `.md` file extension.

In contrast to our proof of concept, this will be a multi-page website, so we're going to create three HTML files in our `layout` directory: `index.html`, `post.html` and `macros.html`.

- `index.html` will be the template for our homepage, showing a list of blog posts in reverse chronological order.
- `post.html` will be the template for post pages, containing their rendered Markdown content.
- `macros.html` will not be a template, but a container file for Jinja macros[227]. These are reusable snippets of HTML that we can use in our templates.

Create three files and populate them as follows.

`layout/index.html`

[227]https://jinja.palletsprojects.com/en/2.10.x/templates/#macros

```
<!DOCTYPE html>
<html>
    {% import "macros.html" as macros %}
    {{ macros.head(config.title) }}
    <body>
        <h1>Posts</h1>
        <ul>
        {% for post in content.posts %}
            <li><a href="{{ post.url }}">{{ post.title }}</a> (posted at {{ post.dat\
e }})</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

`layout/post.html`

```
<!DOCTYPE html>
<html>
    {% import "macros.html" as macros %}
    {{ macros.head(this.title) }}
    <body>
        <h1>{{ this.title }}</h1>
        <p>Posted at {{ this.date }}</p>
        {{ this.content }}
        <p><a href="{{ config.baseURL }}">Return to the homepage &#10558;</a></p>
    </body>
</html>
```

(`&#10558;` is the HTML entity[228] for "⤾".)

`layout/macros.html`

```
{% macro head(page_title) -%}
<head>
    <title>{{ page_title }}</title>
    <link rel="stylesheet" href="/css/style.css">
</head>
{% endmacro -%}
```

The only macro we've defined is `head`, which will generate an HTML `<head>` tag containing an appropriate title for the page as well as a link to our website's stylesheet. Let's create that now.

[228]https://developer.mozilla.org/en-US/docs/Glossary/Entity

In the `static` directory, create a subdirectory called `css`. Then create a file called `style.css` in this subdirectory and add the following code.

`static/css/style.css`

```css
h1 {
    font-family: sans-serif;
    margin-top: 2em;
}

body {
    font-family: serif;
    margin: 0 auto;
    max-width: 40em;
    line-height: 1.2em;
}
```

These are a couple of small style adjustments to improve readability and differentiate our site from an unstyled page. Feel free to add your own touches.

Ingesting input files

Now that we've created our input files, let's write some code in `main.py` to read them and create our website. To do this, we'll be iterating our proof-of-concept code.

First, at the top of the file, let's import some new modules for dealing with reading and writing files and directories. Add the second line below the first in `main.py`.

```python
import jinja2, markdown, toml, re
import os, glob, pathlib, shutil, distutils.dir_util
```

Then delete the `config_string`, `content_strings` and `template_string` definitions from the `main` function.

Ingesting site configuration

First, let's ingest the configuration file. Change the `load_config` function as follows.

```python
def load_config(config_filename):
    with open(config_filename, 'r') as config_file:
        return toml.loads(config_file.read())
```

Now change this line in the `main` function:

```
config = load_config(config_string)
```

To this:

```
config = load_config("config.toml")
```

Ingesting posts

Next, we will ingest the `content/posts` directory. Change the content of the `load_content_items` function as follows.

```
def load_content_items(content_directory):
    items = []
    for fn in glob.glob(f"{content_directory}/*.md"):
        with open(fn, 'r') as file:
            frontmatter, content = re.split("^\+\+\+\+\+$", file.read(), 1, re.MULTI\
LINE)
        item = toml.loads(frontmatter)
        item['content'] = markdown.markdown(content)

        items.append(item)

    # sort in reverse chronological order
    items.sort(key=lambda x: x["date"],reverse=True)

    return items
```

Instead of looping through a list of strings, we're now looping through all files ending in `.md` in the `content/posts` directory using the glob method and parsing their contents.

Since we're now building a real site with multiple pages, we'll need to add a couple of additional attributes to our `post` dictionary. Namely, `slug` and `url`.

- `slug` will be the name of the post's Markdown file without the .md extension.
- `url` will be a partial URL including the post's date and slug. For the first post, it will look like this: `/2021/02/14/first-post/`

Let's create the slug by using `os.path.basename` to get our file's filename without its full path (i.e. `first-post.md` rather than `content/posts/first-post.md`). Then we'll use `os.path.splitext` on the result to split the filename and extension, and we'll discard the extension. Add the following line to the for loop, below where we define `item['content']`.

```
item['slug'] = os.path.splitext(os.path.basename(file.name))[0]
```

We'll then use this slug along with our post's date to construct the full URL. We'll use Python's string formatting[229] to ensure correct zero-padding of single-digit values for months and days. Add this line below the one we just added:

```
item['url'] = f"/{item['date'].year}/{item['date'].month:0>2}/{item['date'].day:0>2}\
/{item['slug']}/"
```

Now we can update our function invocation in `main`. Change this line:

```
content = load_content_items(content_strings)
```

To this:

```
content = { "posts": load_content_items("content/posts") }
```

Using a dictionary instead of a plain list will allow us to add additional content types in a later section of this tutorial.

Ingesting templates

Now that we have a list of posts, let's ingest our templates so we have somewhere to put them. Jinja works quite differently from the file system and from strings, so we're going to change our `load_templates` function to create a Jinja `Environment` with a `FileSystemLoader` that knows to look for templates in a particular directory. Change the function code as follows.

```
def load_templates(template_directory):
    file_system_loader = jinja2.FileSystemLoader(template_directory)
    return jinja2.Environment(loader=file_system_loader)
```

Then, in the `main` function, change this line:

```
template = load_templates(template_string)
```

To this:

```
environment = load_templates("layout")
```

In the next section, we'll pass this environment to our render_site function where we'll load individual templates as we need them.

[229]https://docs.python.org/3/library/string.html#string-formatting

Writing output files

Now let's render the site by writing some output files. We'll be using a directory named `public` for this, but you don't need to create this in your file pane – we'll do so in code. Go to the `render_site` function and replace its code with the following (remember to change the function parameters).

```python
def render_site(config, content, environment, output_directory):
    if os.path.exists(output_directory):
        shutil.rmtree(output_directory)
    os.mkdir(output_directory)
```

We do two things here: remove the output directory and all of its content if it exists, and create a fresh output directory. This will avoid errors when running our code multiple times.

Now let's write our home page by adding this code to the bottom of the function.

```python
# Homepage
index_template = environment.get_template("index.html")
with open(f"{output_directory}/index.html", 'w') as file:
    file.write(index_template.render(config=config,content=content))
```

Here we use our Jinja environment to load the template at `layout/index.html`. We then open the `public/index.html` file and write to it the results of rendering `index_template` with our `config` and `content` dictionaries passed in.

The code for writing individual post files is a bit more complex. Add the for loop below to the bottom of the function.

```python
# Post pages
post_template = environment.get_template("post.html")
for item in content["posts"]:
    path = f"{output_directory}/{item['url']}"
    pathlib.Path(path).mkdir(parents=True, exist_ok=True)
    with open(path+"index.html", 'w') as file:
        file.write(post_template.render(this=item, config=config, content=content))
```

First we create the directories necessary to show our post URLs. To display a URL such as `2021/02/14/first-post/`, we need to create a directory named `2021` inside public, and then nested directories named `02`, `14` and `first-post`. Inside the final directory, we create a file named `index.html` and write our rendered template to it.

Note the values we pass to `render`: variables for this post are contained in `this` and site-wide configuration variables are contained in `config`. We also pass in `content` to allow us to access other posts. Although we aren't using this in the `post.html` template right now, it's good to have the option for future template updates.

Now we need to load our static files. Add this code to the bottom of the `render_site` function:

```python
# Static files
distutils.dir_util.copy_tree("static", "public")
```

All this code does is copy the file tree from our static directory into our public directory. This means that our CSS file at `static/css/style.css` can be accessed in our HTML templates as `css/style.css`. Similarly, if we create a file at `static/my-picture.jpg`, we can reference that in our HTML or Markdown as `my-picture.jpg` and it will be found and loaded.

Now we just need to update the function invocation in our `main` function. Change this line:

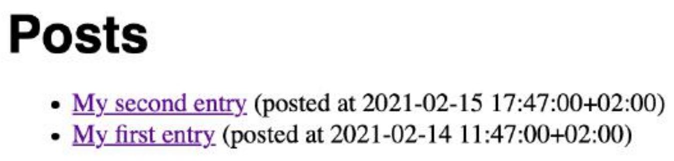
```python
render_site(config, content, templates)
```

To this:

```python
render_site(config, content, environment, "public")
```

Now run the code. You should see the `public` directory appear in your file pane. Look inside, and you'll see the directories and files we just created. To see your site in action, run the following commands in Replit's "Shell" tab.

```
cd public
python -m http.server
```



Posts

- My second entry (posted at 2021-02-15 17:47:00+02:00)
- My first entry (posted at 2021-02-14 11:47:00+02:00)

This server will need to be restarted periodically as you work on your site.

Generic site generator

In addition to chronological blog posts, our site could do with undated pages, such as an "About" or "Contact" page. Depending on the kind of site we want to build, we may also want photo pages, or pages including podcast episodes, or any number of other things. If we give this SSG to someone else to use, they may have their own ideas as well – for example, they may want to make a site organised as a book with numbered chapters rather than as a blog. Rather than trying to anticipate everyone's needs, let's make it so we can create multiple types of content pages, and allow the user to define those types and how they should be ordered.

This is simpler than it sounds, but will require some refactoring.

Expanding the config file

First, let's add some content to our `config.toml` file to give this customization a definite shape. Add these lines below the definition of `baseURL`.

`config.toml`

```
title = "My site"
baseURL = "https://YOUR-REPL-NAME-HERE.YOUR-REPLIT-USERNAME.repl.co"

types = ["post", "page"]

post.dateInURL = true
post.sortBy = "date"
post.sortReverse = true

page.dateInURL = false
page.sortBy = "title"
page.sortReverse = false
```

Here we've told our site generator we want two kinds of pages – a post type, which we will use for blog posts, and a page type, which we will use for evergreen content such as contact details and general site information. Below that, we've used TOML's dictionary syntax to specify some characteristics of each type.

- Posts will have a date in their URLs and will be sorted in reverse date order when listed.
- Pages will not have a date in their URLs and will be sorted alphabetically by their title.

By creating these settings, we'll make it possible to sort a content type by any attribute in its frontmatter.

Ingesting user-defined content

To implement this, let's first import a new module at the top of `main.py`. Add the third line to your file, below the first two.

```python
import jinja2, markdown, toml, re
import glob, pathlib, os, shutil, distutils.dir_util
import inflect
```

The `inflect` module allows us to turn singular words into plurals and vice versa. This will be useful for working with the `types` list from our configuration file. Change the `load_config` function to resemble the following.

```python
def load_config(config_filename):

    with open(config_filename, 'r') as config_file:
        config = toml.loads(config_file.read())

    ie = inflect.engine()
    for content_type in config["types"]:
        config[content_type]["plural"] = ie.plural(content_type)

    return config
```

This code will expand the dictionaries we load from our config file with a key containing the type's plural. If we were to print out our config dictionary at this point, it would look like this:

```
{
    "title": "My site"
    "baseURL": "https://YOUR-REPL-NAME-HERE.YOUR-REPLIT-USERNAME.repl.co"
    "types": ["post", "page"]
    "post": {
        "plural": "posts",
        "dateInURL": true,
        "sortBy": "date",
        "sortReverse": true
    },
    "page": {
        "plural": "pages",
        "dateInURL": true,
        "sortBy": "title",
        "sortReverse": false
    }
}
```

Now let's modify `load_content_items` to deal with multiple, user-defined content types. First, we need to change the function to take our `config` dictionary as an additional parameter. Second, we'll put all of our function's current content in an inner function named `load_content_type`. Your function should now look like this:

```
def load_content_items(config, content_directory):

    def load_content_type(content_type):
        items = []
        for fn in glob.glob(f"{content_directory}/*.md"):
            with open(fn, 'r') as file:
                frontmatter, content = re.split("^\+\+\+\+\+$", file.read(), 1, re.M\
ULTILINE)

            item = toml.loads(frontmatter)
            item['content'] = markdown.markdown(content)
            item['slug'] = os.path.splitext(os.path.basename(file.name))[0]
            item['url'] = f"/{item['date'].year}/{item['date'].month:0>2}/{item['dat\
e'].day:0>2}/{item['slug']}/"

            items.append(item)

        # sort in reverse chronological order
        items.sort(key=lambda x: x["date"],reverse=True)

        return items
```

To load from the correct directory, we will need to change this line:

```
for fn in glob.glob(f"{content_directory}/*.md"):
```

To this:

```
for fn in glob.glob(f"{content_directory}/{config[content_type]['plural']}/*.md"):
```

Here we're using the plural of the content type we defined earlier. This will ensure that items of type "post" can be found in "content/posts" and items of type "page" can be found in "content/pages".

We now need to add code to respect our configuration settings. We'll do this by changing this line:

```
item['url'] = f"/{item['date'].year}/{item['date'].month:0>2}/{item['date'].day:0>2}\
/{item['slug']}/"
```

To this:

```
if config[content_type]["dateInURL"]:
    item['url'] = f"/{item['date'].year}/{item['date'].month:0>2}/{item['date'].day:\
0>2}/{item['slug']}/"
else:
    item['url'] = f"/{item['slug']}/"
```

Now we'll sort according to the configuration file by changing this line:

```
# sort in reverse chronological order
items.sort(key=lambda x: x["date"],reverse=True)
```

To this:

```
# sort according to config
items.sort(key=lambda x: x[config[content_type]["sortBy"]],
           reverse=config[content_type]["sortReverse"])
```

We can complete this `load_content_items` function by writing some code to iterate through our site's configured content types, calling `load_content_type` for each one. Add the following code below the definition of `load_content_type` (ensure that it's de-indented so as to be part of `load_-content_items`).

```
content_types = {}
for content_type in config["types"]:
    content_types[config[content_type]['plural']] = load_content_type(content_type)

return content_types
```

Then in the `main` function, change this line:

```
content = { "posts": load_content_items("content/posts") }
```

To this:

```
content = load_content_items(config, "content")
```

Rendering user-defined content

Now we need to change our output code in `render_site` to render each content type with its own template. As we did with `load_content_items`, we'll start by moving the post-creating for loop into an inner function, this time named `render_type`. Alter your `render_site` function so that it resembles the following.

```python
def render_site(config, content, environment, output_directory):

    def render_type(content_type): # <-- new inner function
        # Post pages
        post_template = environment.get_template("post.html")
        for item in content["posts"]:
            path = f"public/{item['url']}"
            pathlib.Path(path).mkdir(parents=True, exist_ok=True)
            with open(path+"index.html", 'w') as file:
                file.write(post_template.render(this=item, config=config))

    if os.path.exists(output_directory):
        shutil.rmtree(output_directory)
    os.mkdir(output_directory)

    for content_type in config["types"]: # <-- new for loop
        render_type(content_type)

    # !!! post for loop moved to inner function above

    # Homepage
    index_template = environment.get_template("index.html")
    with open("public/index.html", 'w') as file:
        file.write(index_template.render(config=config, content=content))


    # Static files
    distutils.dir_util.copy_tree("static", "public")
```

Then change this line in the `render_type` inner function that loads the post template:

```python
post_template = environment.get_template("post.html")
```

Into this line that loads a template for the provided content type:

```python
template = environment.get_template(f"{content_type}.html")
```

Alter the for loop below that line to use the content type's plural.

```python
for item in content[config[content_type]["plural"]]:
```

Finally, change `post_template` in the loop's final line to `template`.

```
file.write(template.render(this=item, config=config, content=content))
```

Adding a new content type

Now that we've done all that work to generify our code, all that's left is to create our pages. First, let's create a page template at `layout/page.html`. Use the following code.

```
<!DOCTYPE html>
<html>
    {% import "macros.html" as macros %}
    {{ macros.head(this.title) }}
    <body>
        <h1>{{ this.title }}</h1>
        {{ this.content }}
        <p><a href="{{ config.baseURL }}">Return to the homepage &#10558;</a></p>
    </body>
</html>
```

This is just our `post.html` template without the date.

Now create a new subdirectory in `content` called `pages`. Inside that subdirectory, create a file named `about.md` and put the following content in it.

```
title = "About"
+++++

This website is built with Python, Jinja, TOML and Markdown.
```

This is sufficient to create a new page at `/about/`, but it won't be linked anywhere. For that, we'll need to create a global navigation bar for our site. Create the following additional macro in `layout/macros.html`.

```
{% macro navigation(pages) -%}
<nav><ul>
    {% for page in pages %}
        <li><a href="{{ page.url }}">{{ page.title }}</a></li>
    {% endfor %}
</ul></nav>
{% endmacro -%}
```

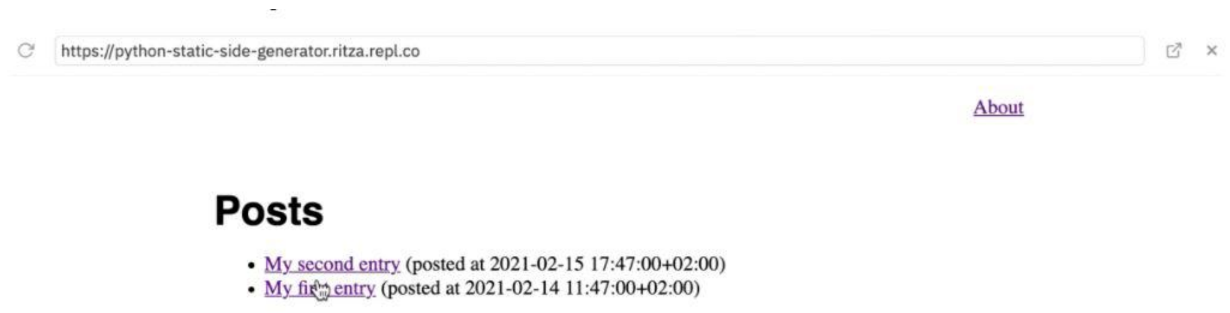
Then include the macro in `index.html`, `page.html` and `post.html` by inserting the following code just underneath `{{ macros.head(this.title) }}`.

```
{{ macros.navigation(content.pages) }}
```

Finally, add the CSS below to `static/css/style.css` to apply light styling to the navigation bar.

```
nav ul
{
    list-style-type: none;
    text-align: right;
}
```

Run your code and preview your site with `cd public && python -m http.server` in the repl shell, and you should see something like this:



Posts

- My second entry (posted at 2021-02-15 17:47:00+02:00)
- My first entry (posted at 2021-02-14 11:47:00+02:00)

*click to open gif[230]

Where to next?

We've created a flexible SSG capable of generating many different types of HTML pages, which can be served from any web server. Apart from fleshing out the templates and adding new content types, you might want to expand the generator's functionality to allow things like:

- Categories or tags for content items.
- Ability to generate an RSS or Atom feed for people to subscribe to.
- A way to mark items as drafts, so they won't be included when the site is compiled.
- Navigation features like next and previous item links.
- Useful error messages for malformed directory structures and configuration files.

[230]https://docs.replit.com/images/tutorials/static-site-generator/generator_functionality.gif

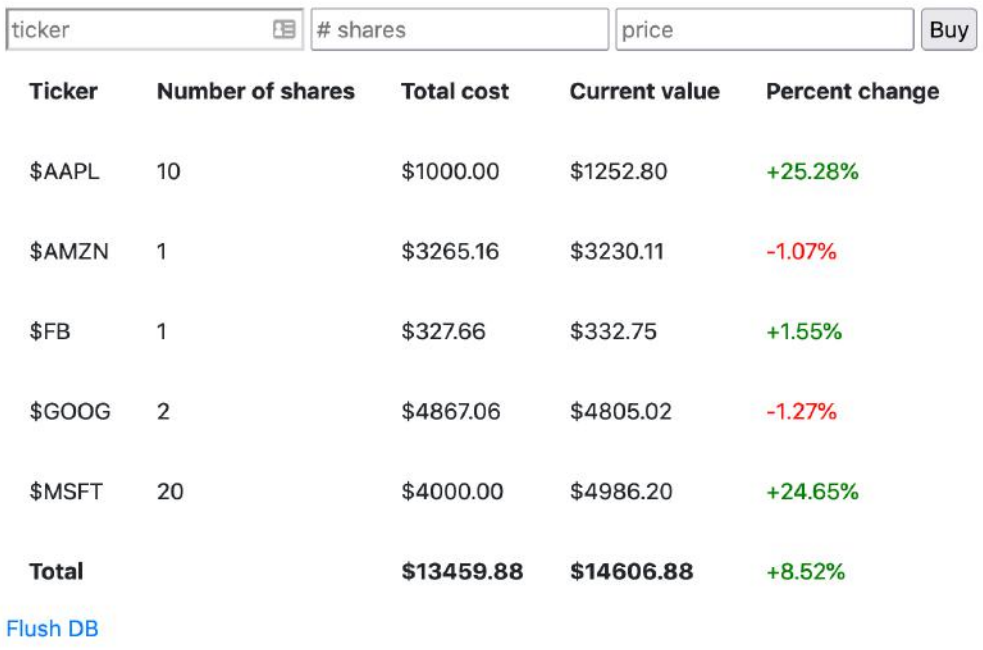
Next Tutorial

In the next tutorial we take a look at how to build a personal stock market dashboard app to keep track of your portfolio's performance. The application will allow you to record stock purchases and then show you the percentage gain or loss on the stock by making use of web scraping. At the end of it, you should have a good understanding of web scraping using Requests and Beautiful Soup.

Build a Personal Stock Market Dashboard

In this tutorial, we will be building a single-page web dashboard for tracking a real or imaginary stock portfolio. This dashboard will:

- Allow the user to record stock purchases.
- Track the current price of all stocks held through web scraping.
- Show the user the percentage gain or loss on their holdings, for each stock and in total.



Ticker	Number of shares	Total cost	Current value	Percent change
$AAPL	10	$1000.00	$1252.80	+25.28%
$AMZN	1	$3265.16	$3230.11	-1.07%
$FB	1	$327.66	$332.75	+1.55%
$GOOG	2	$4867.06	$4805.02	-1.27%
$MSFT	20	$4000.00	$4986.20	+24.65%
Total		**$13459.88**	**$14606.88**	+8.52%

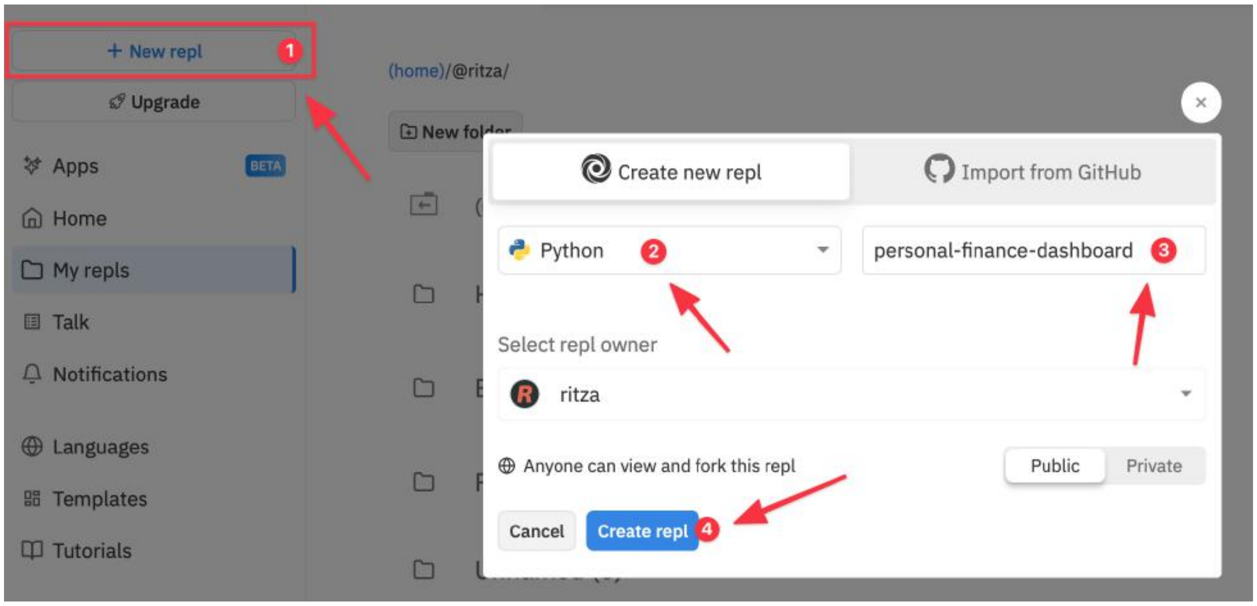
Dashboard functionality

After working through this tutorial, you'll:

- Be able to build a single-page application with Flask and JavaScript.
- Understand web scraping with `Requests` and `Beautiful Soup`.
- Know how to manage persistent data with Replit's database.

Creating the Dashboard

We're going to build our dashboard using Python (plus a bit of JavaScript). Sign in to Replit[231] or create an account[232] if you haven't already. Once logged in, create a new Python repl.



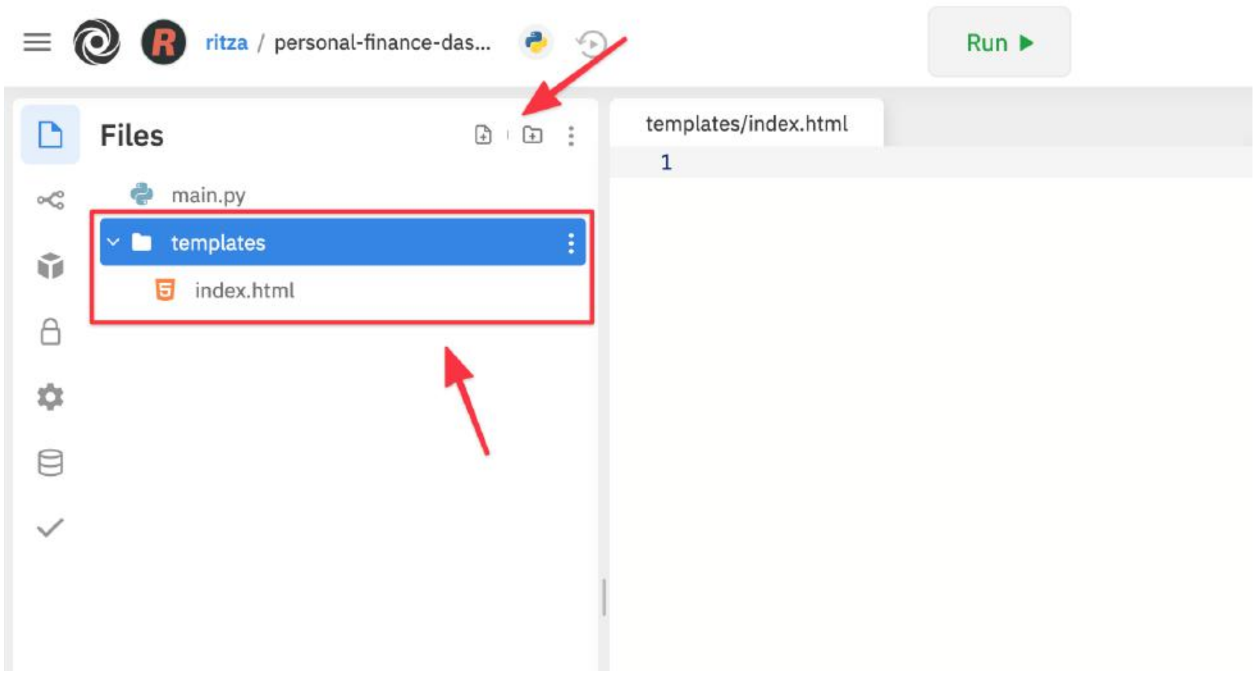
Creating a Python repl

Our dashboard will need to have three functions:

- Displaying the current state of the stock portfolio.
- Recording share purchases.
- Flushing the database (this will be useful for testing).

Let's start by creating an HTML frontend with the basic data and form elements necessary to enable this functionality. In your repl's file pane, create a directory named `templates`, and in that folder, create a file called `index.html` (this file structure is necessary for building Flask applications).

[231]https://replit.com
[232]https://replit.com/signup



Creating the file structure

Then enter the following HTML into the `index.html` file:

```html
<!DOCTYPE html>
<html>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/\
css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQ\
UOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
    <style>
        .positive:before { content: "+"; color: green; }
        .positive { color: green; }
        .negative { color: red; }
        th, td { padding: 1em; }
        body { margin: 2em auto; max-width: 80em; }
    </style>
    <body>
        <form action="/buy" method="post">
            <input type="text" placeholder="ticker" name="ticker"/>
            <input type="text" placeholder="# shares" name="shares"/>
            <input type="text" placeholder="price" name="price"/>
            <input type="submit" value="Buy"/>
        </form>

        <table id="portfolio">
            <tr>
```

```
                <th>Ticker</th>
                <th>Number of shares</th>
                <th>Total cost</th>
                <th>Current value</th>
                <th>Percent change</th>
            </tr>
            <tr>
                <td>Loading...</td>
            <tr>
        </table>
        <a href="/flush">Flush DB</a>
    </body>
</html>
```

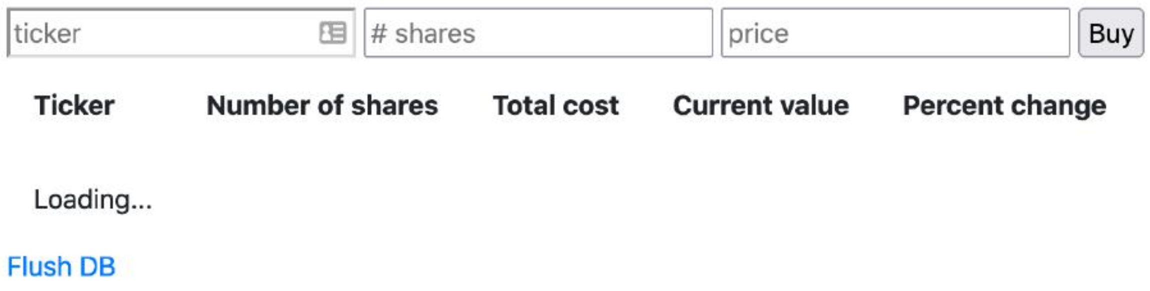
In this file, we've imported Bootstrap CSS[233] and applied some minimal styles of our own to make the default content look a little better. We've also created a form for recording share purchases, and a table that will display our share portfolio.

Now let's write some Python code so we can display this page in our Flask app. Enter the following code in `main.py`:

```python
from flask import Flask, render_template, request, jsonify, url_for, redirect
import json

site = Flask(__name__)

@site.route('/')
def index():
    return render_template('index.html')

site.run(host='0.0.0.0', port=8080)
```

Now run your repl. The resulting page should look like this:

[233]https://getbootstrap.com/



Initial dashboard

The first thing we need to implement to get this page functional is the share purchase form. Let's do that now.

Accessing the Database and Recording Purchases

We can allow users to "buy" stock by entering the ticker symbol[234], the number of shares purchased, and the price per share. While it would also make sense to record the purchase time, we will leave that out for the sake of simplicity (but you can add it later on your own).

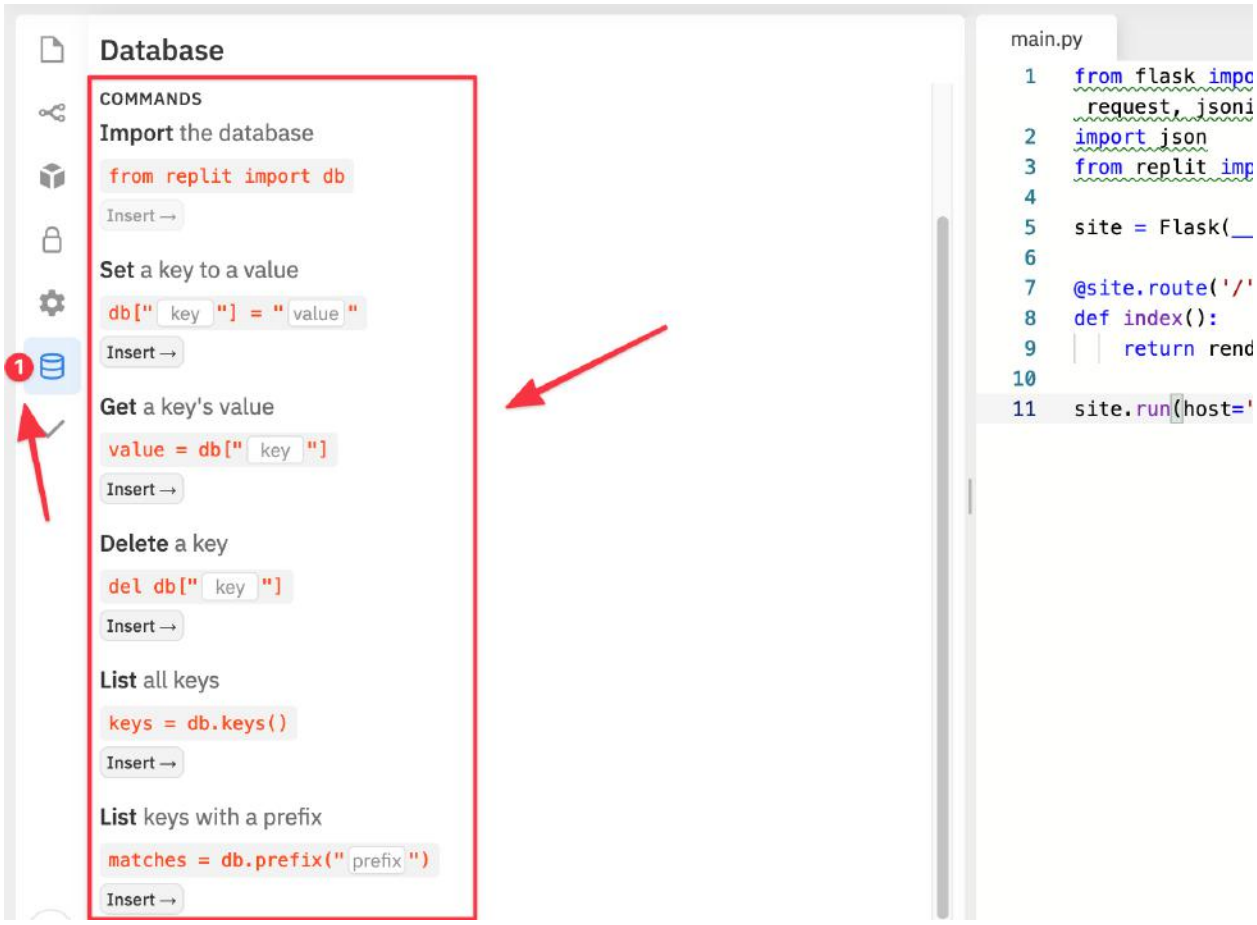
We will record these purchases by adding them to the Replit database[235]. This is a simple key-value store that you can think of as a big Python dictionary which retains its state between runs of a repl. Using this database will save us from having to re-enter all of our stock information every time we restart our dashboard.

To use the Replit Database, all we have to do is add the following import statement at the top of `main.py`:

```python
from replit import db
```

Now we can use the globally scoped variable `db` like a Python dictionary, keeping in mind that whatever we store in it will persist between executions of our application. A cheat sheet for using the database is available from your repl's database tab on the sidebar.

[234]https://en.wikipedia.org/wiki/Ticker_symbol
[235]https://docs.replit.com/misc/database



```
from replit import db
db["key"] = "value"
value = db["key"]
del db["key"]
keys = db.keys()
matches = db.prefix("prefix")
```

```
from flask import
  request, jsoni
import json
from replit imp

site = Flask(_

@site.route('/'
def index():
    return rend

site.run(host='
```

Database sidebar

Let's give it a spin and write the function for buying shares. Add the following code just above the line beginning with `site.run`:

```
@site.route('/buy', methods=['POST'])
def buy():
    # Create shares key if it doesn't exist
    if 'shares' not in db.keys():
        db['shares'] = {}

    # Extract values from form
    ticker = request.form['ticker'][:5].upper()
    price = float(request.form['price'])
    shares = int(request.form['shares'])

    if ticker not in db['shares']: # buying these for the first time
        db['shares'][ticker] = { 'total_shares': shares,
                                 'total_cost': shares * price }

        db['shares'][ticker]['purchases'] = [{ 'shares': shares,
                                   'price': price }]
    else: # buying more
        db['shares'][ticker]['total_shares'] += shares
        db['shares'][ticker]['total_cost'] += shares * price
        db['shares'][ticker]['purchases'].append({ 'shares': shares,
                                           'price': price})

    return redirect(url_for("index"))
```

First, if necessary, we create an empty dictionary at the “shares” key of our `db` database. This code will only need to run the first time we buy shares.

Then, we extract the ticker, price and number of shares from the form data, coercing each one into an appropriate format. We want stock tickers to be uppercase and a maximum of five characters long[236], prices to include fractional amounts, and number of shares to be integers (though you could change that later to support fractional shares[237]).

Finally, we add our share purchase to the “shares” dictionary. This dictionary is made up of ticker symbol keys mapped to inner dictionaries, which in turn contain the following information:

- The total number of shares owned.
- The total cost of all shares purchased.
- A list of individual purchases. Each purchase is a dictionary containing the number of shares purchased, and their unit price.

This may seem like a complicated structure, but it is necessary to allow users to buy shares in the same company at different prices. With some data added, our dictionary could look like this:

```
{
    "AAPL": {
        "total_shares": 15,
        "total_cost": 1550,
        "purchases": [
            {
                "shares": 10,
                "price": 100
            },
            {
                "shares": 5,
```

[236]https://www.investopedia.com/terms/s/stocksymbol.asp
[237]https://www.investopedia.com/terms/f/fractionalshare.asp

```
                "price": 110
            }
        ]
    },
    "MSFT": {
        "total_shares": 20,
        "total_cost": 4000,
        "purchases": [
            {
                "shares": 20,
                "price": 200
            }
        ]
    }
}
```

In the data above, we've bought 10 shares of Apple Inc (AAPL) at $100 per share, and 5 at $110 per share. We've also bought 20 shares of Microsoft Corporation (MSFT) at $200 per share. The `total_shares` and `total_cost` values could be derived from the list of purchases, but we're storing them in the database to avoid having to recalculate them unnecessarily.

Run your code now, and add a few stocks with random values. You can use some example tickers: AAPL, MSFT, AMZN, FB, GOOG. While our purchases won't show up on our dashboard, you can determine whether they're getting added to the database by visiting the database tab on your repl's sidebar. If your code is correct, you should see non-zero values under the "STORAGE" heading.

At the moment, our dashboard will also allow you to add any value as a ticker symbol, even if it's not a real public company. And, needless to say, our dashboard also doesn't show us the current value of our stocks. We'll fix those issues soon, but first we need to implement some functionality to help us test.

Flushing the Database

A persistent database is vital for creating most useful web applications, but it can get messy in the early stages of development when we're adding a lot of different test data and experimenting with different structures. That's why it's useful to have a quick and easy way to delete everything.

We've already included a link to flush the database in our `index.html` file, so now let's create the backend functionality in Flask. Add this code to `main.py`, below the `buy` function:

```python
@site.route('/flush')
def flush_db():
    del db["shares"]
    return redirect(url_for("index"))
```

Here we're deleting the `shares` key from our database and then redirecting the user to the dashboard. As `shares` is the only key in our database, this code will suffice for now, but if we add more keys, we'll have to change it accordingly.

Test this new functionality out by flushing your database before moving on to the next section, especially if you have invalid stock tickers. You can confirm whether the flush worked by checking the database tab of your repl's sidebar, where the values under "STORAGE" should now be zero. Note that deletion may take a few seconds to reflect.

Serving Our Portfolio Data

We want our dashboard to be a live display that fetches new stock prices periodically, without us having to refresh the page. It would also be nice to unload calculations such as percentage gain or loss to the client's web browser, so we can reduce load on our server. To this end, we will be structuring our portfolio viewing functionality as an API endpoint that is queried by JavaScript code, rather than using Jinja templates to build it on the server-side.

The first thing we must do to achieve this is to create a Flask endpoint that returns the user's portfolio. We'll do this at `/portfolio`. Add the following code to `main.py`, below the `buy` function:

```python
@site.route('/portfolio')
def portfolio():
    if "shares" not in db.keys():
        return jsonify({})

    portfolio = json.loads(db.get_raw("shares"))

    # Get current values
    for ticker in portfolio.keys():
        current_price = float(get_price(ticker))
        current_value = current_price * portfolio[ticker]['total_shares']
        portfolio[ticker]['current_value'] = current_value

    return jsonify(**portfolio)
```

The purpose of this function is to serve a JSON object[238] of the shares portfolio to the client. Later, we'll write JavaScript for our dashboard which will use this object to construct a table showing our portfolio information.

[238]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

In the above code, if no stocks have been added, we return an empty JSON object. Otherwise, we set `portfolio` to a copy of the `shares` dictionary in our Replit database. The Python Replit library uses custom list and dictionary types that cannot be directly serialized into JSON, so we use `db.get_raw`[239] to convert the whole thing into a string and `json.loads` to convert that string into a standard Python dictionary.

Then we need to get the current values for each of our stock holdings. To do so, we loop through `portfolio.keys()`, call `get_price(ticker)` and multiply the return value by the total shares we're holding for this stock. We then add this value under the new `current_value` key in our stock's dictionary.

Finally, we convert our portfolio dictionary to JSON using Flask's `jsonify`[240] and return it.

There's just one problem: we haven't implemented `get_price` yet! Let's do that now, before we try to run this code.

Fetching Current Prices

We'll fetch the current prices of our stocks by scraping[241] the Yahoo Finance[242] website. While the more traditional and foolproof way of consuming structured data such as share prices is to use an API[243] that provides structured data in a computer-ready format, this is not always feasible, as the relevant APIs may be limited or even non-existent. For these and other reasons, web scraping is a useful skill to have.

A quick disclaimer before we jump in: Copyright law and web scraping laws are complex and differ by country. As long as you aren't blatantly copying their content or doing web scraping for commercial gain, people generally don't mind web scraping. However, there have been some legal cases involving scraping data from LinkedIn, and media attention from scraping data from OKCupid. Web scraping can violate the law, go against a particular website's terms of service, or breach ethical guidelines – so take care with where you apply this skill.

Additionally, from a practical perspective, web scraping code is usually brittle and likely to break in the event that a scraped site changes its appearance.

With those considerations in mind, let's start scraping. We'll use Python Requests[244] to fetch web pages and Beautiful Soup[245] to parse them and extract the parts we're interested in. Let's import those at the top of `main.py`.

[239]https://replit-py.readthedocs.io/en/latest/db_tutorial.html#advanced-usage
[240]https://flask.palletsprojects.com/en/2.0.x/api/#module-flask.json
[241]https://en.wikipedia.org/wiki/Web_scraping
[242]https://finance.yahoo.com/
[243]https://en.wikipedia.org/wiki/API
[244]https://docs.python-requests.org/en/master/
[245]https://www.crummy.com/software/BeautifulSoup/

```python
from bs4 import BeautifulSoup
import requests
```

Now we can create our `get_price` function. Enter the following code near the top of `main.py`, just below `site = Flask(__name__)`:

```python
def get_price(ticker):
    page = requests.get("https://finance.yahoo.com/quote/" + ticker)
    soup = BeautifulSoup(page.text, "html5lib")

    price = soup.find('span', {'class':'Trsdu(0.3s) Fw(b) Fz(36px) Mb(-4px) D(ib)'})\
.text

    # remove thousands separator
    price = price.replace(",", "")

    return price
```

The first line fetches the page on Yahoo Finance that shows information about our stock share price. For example, the link below will show share price information for Apple Inc:

https://finance.yahoo.com/quote/AAPL[246]

We then load the page into a Beautiful Soup object, parsing it as HTML5 content. Finally, we need to find the price. If you visit the above page in your browser, right-click on the price near the top of the page and select "Inspect". You'll notice that it's inside a `span` element with a class value containing `Trsdu(0.3s) Fw(b) Fz(36px) Mb(-4px) D(ib)`. If the market is open, and the price is changing, additional classes may be added and removed as you watch, but the previously mentioned value should still be sufficient.

We use Beautiful Soup's `find`[247] method to locate this `span`. The `text` attribute of the object returned is the price we want. Before returning it, we remove any comma thousands separators to avoid float conversion errors later on.

Although we've implemented this functionality for the sake of portfolio viewing, we can also use it to improve our share buying process. We'll make a few additional quality-of-life changes at the same time. Find your `buy` function code and modify it to look like this:

[246] https://finance.yahoo.com/quote/AAPL
[247] https://www.crummy.com/software/BeautifulSoup/bs3/documentation.html#find(name,%20attrs,%20recursive,%20text,%20**kwargs)

```python
@site.route('/buy', methods=['POST'])
def buy():
    # Create shares key if it doesn't exist
    if 'shares' not in db.keys():
        db['shares'] = {}

    ticker = request.form['ticker']

    # remove starting $
    if ticker[0] == '$':
        ticker = ticker[1:]

    # uppercase and maximum five characters
    ticker = ticker.upper()[:5]

    current_price = get_price(ticker)
    if not get_price(ticker): # reject invalid tickers
        return f"Ticker $'{ticker}' not found"

    if not request.form['price']: # use current price if price not specified
        price = float(current_price)
    else:
        price = float(request.form['price'])

    if not request.form['shares']: # buy one if number not specified
        shares = 1
    else:
        shares = int(request.form['shares'])

    if ticker not in db['shares']: # buying these for the first time
        db['shares'][ticker] = { 'total_shares': shares,
                                 'total_cost': shares * price }

        db['shares'][ticker]['purchases'] = [{ 'shares': shares,
                                'price': price }]
    else: # buying more
        db['shares'][ticker]['total_shares'] += shares
        db['shares'][ticker]['total_cost'] += shares * price
        db['shares'][ticker]['purchases'].append({ 'shares': shares,
                                        'price': price})

    return redirect(url_for("index"))
```

The first change we've made to this function is to strip leading `$s` on ticker symbols, in case users include those. Then, by calling `get_price` in this function, we can both prevent users from adding invalid stock tickers and allow users to record purchases at the current price by leaving the price field blank. Additionally, we'll assume users want to buy just one share if they leave the number of shares field blank.

We can now test out our code. Run your repl, add some stocks, and then, in a separate tab, navigate to this URL (replacing the two ALL-CAPS values first):

```
https://YOUR-REPL-NAME.YOUR-USERNAME.repl.co/portfolio
```

You should now see a JSON object similar to the database structure detailed above, with the current value of each stock holding as an additional field. In the next section, we'll display this data on our dashboard.

Showing Our Portfolio

We will need to write some JavaScript to fetch our portfolio information, assemble it into a table, and calculate the percentage changes for each stock as well as our portfolio's total cost, current value and percentage change.

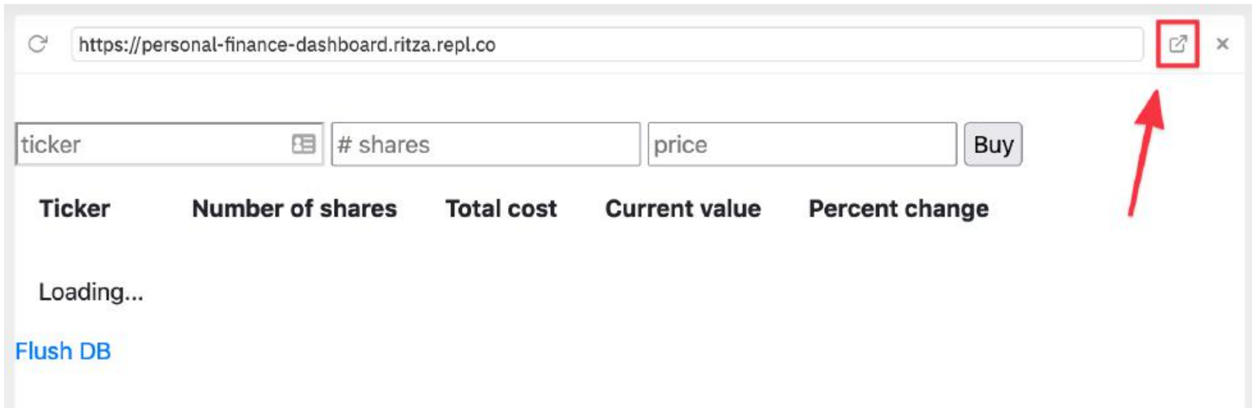
Add the following code just above the closing `</body>` tag in `templates/index.html`:

```
<script>
function getPortfolio() {
    fetch("/portfolio")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });

}

getPortfolio();
</script>
```

This code uses the Fetch API[248] to query our `/portfolio` endpoint and returns a `Promise`[249], which we feed into two `then`[250] methods. The first one extracts the JSON data from the response, and the second one logs the data to JavaScript console. This is a common pattern in JavaScript, which provides a lot of asynchronous[251] functionality.

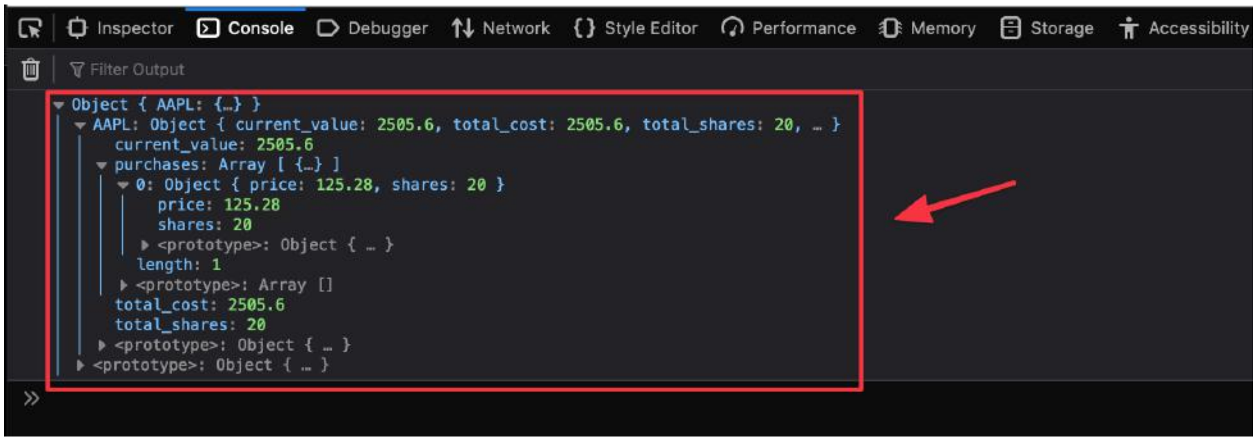
[248]https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[249]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[250]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
[251]https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts

Run your repl and open its web page in a new tab.



Opn in new tab

Then open your browser's devtools with F12, and you should see your portfolio JSON object in the console. If you don't, give it a few seconds.



```
▼ Object { AAPL: {…} }
  ▼ AAPL: Object { current_value: 2505.6, total_cost: 2505.6, total_shares: 20, … }
      current_value: 2505.6
    ▼ purchases: Array [ {…} ]
      ▼ 0: Object { price: 125.28, shares: 20 }
          price: 125.28
          shares: 20
        ▶ <prototype>: Object { … }
        length: 1
      ▶ <prototype>: Array []
      total_cost: 2505.6
      total_shares: 20
    ▶ <prototype>: Object { … }
  ▶ <prototype>: Object { … }
```

In browser console

Now let's add the rest of our JavaScript code. Delete `console.log(data);` and add the following code in its place:

```
var table = document.getElementById("portfolio");
var tableHTML = `<tr>
    <th>Ticker</th>
    <th>Number of shares</th>
    <th>Total cost</th>
    <th>Current value</th>
    <th>Percent change</th>
</tr>`;

var portfolioCost = 0;
var portfolioCurrent = 0;

for (var ticker in data) {
    var totalShares = data[ticker]['total_shares'];
    var totalCost = data[ticker]['total_cost'];
    var currentValue = data[ticker]['current_value'];
    var percentChange = percentChangeCalc(totalCost, currentValue);

    row = "<tr>";
    row += "<td>$" + ticker + "</td>";
    row += "<td>" + totalShares + "</td>";
    row += "<td>$" + totalCost.toFixed(2)  + "</td>";
    row += "<td>$" + currentValue.toFixed(2) + "</td>";
    row += percentChangeRow(percentChange);
    row += "</tr>";
    tableHTML += row;

    portfolioCost += totalCost;
    portfolioCurrent += currentValue;
}

portfolioPercentChange = percentChangeCalc(portfolioCost, portfolioCurrent);

tableHTML += "<tr>";
tableHTML += "<th>Total</th>";
tableHTML += "<th>&nbsp;</th>";
tableHTML += "<th>$" + portfolioCost.toFixed(2) + "</th>";
tableHTML += "<th>$" + portfolioCurrent.toFixed(2) + "</th>";
tableHTML += percentChangeRow(portfolioPercentChange);
tableHTML += "</tr>"

table.innerHTML = tableHTML;
```

This code constructs an HTML table[252] containing the values queried from our portfolio endpoint, as well as the extra calculated values we mentioned above. We use the `toFixed`[253] method to cap the number of decimal places for financial values to two.

We also use a couple of helper functions for calculating and displaying percentage changes. Add the code for these above the `getPortfolio` function declaration:

[252]https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table
[253]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed

```
function percentChangeCalc(x, y) {
    return (x != 0 ? (y - x) * 100 / x : 0);
}

function percentChangeRow(percentChange) {
    if (percentChange > 0) {
        return "<td class='positive'>" + percentChange.toFixed(2) + "%</td>";
    }
    else if (percentChange < 0) {
        return "<td class='negative'>" + percentChange.toFixed(2) + "%</td>";
    }
    else {
        return "<td>" + percentChange.toFixed(2) + "%</td>";
    }
}
```

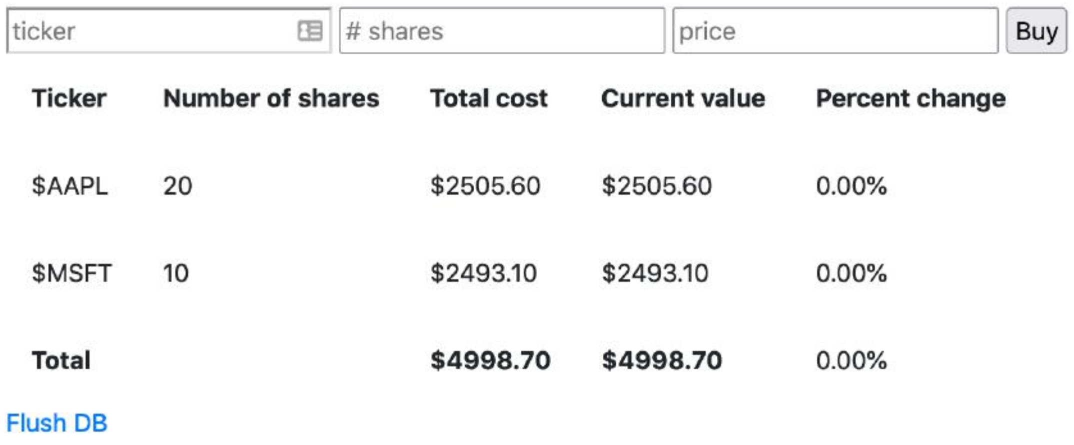
The `percentChangeCalc` function calculates the percentage difference between two numbers, avoiding division by zero. The `percentChangeRow` function allows us to style gains and losses differently by adding classes that we've already declared in the page's CSS.

Finally, we need to add some code to periodically refetch our portfolio, so that we can see the newest price data. We'll use JavaScript's `setInterval`[254] function for this. Add the following code just above the closing `</script>` tag.

```
// refresh portfolio every 60 seconds
setInterval(function() {
    getPortfolio()
}, 60000)
```

Run your repl, add some stocks if you haven't, and you should see something like this:

[254]https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Timeouts_and_intervals#setinterval



Ticker	Number of shares	Total cost	Current value	Percent change
$AAPL	20	$2505.60	$2505.60	0.00%
$MSFT	10	$2493.10	$2493.10	0.00%
Total		**$4998.70**	**$4998.70**	0.00%

Dashboard with portfolio

From this point on, we highly recommend viewing your application in a new browser tab rather than Replit's in-page browser, to get the full-page dashboard experience.

Caching

Our dashboard is feature-complete, but a bit slow. As we're rendering it with client-side JavaScript that has to execute in the user's browser, we won't be able to make it load instantly with the rest of the page, but we can do some server-side caching to speed it up a little and reduce the load on our repl.

Currently, whenever we send a request to the `/portfolio` endpoint, we execute `get_price` on each of our stocks and rescrape Yahoo Finance to find their prices. Under normal conditions, stock prices are unlikely to change significantly moment-to-moment, and our dashboard is not a high-frequency trading[255] platform, so we should write some logic to store the current share price and only renew it if it's more than 60 seconds old. Let's do this now.

As we're going to be modifying the database structure in this section, it's a good idea to flush your repl's database before going any further, so as to avoid errors.

First, we'll import the `time` module, near the top of `main.py`.

```
import time
```

[255]https://www.investopedia.com/ask/answers/09/high-frequency-trading.asp

This allows us to use `time.time()`, which returns the current Unix Epoch[256], a useful value for counting elapsed time in seconds. Add the following code to the `buy` function, just above the `return` statement:

```
    db['shares'][ticker]['current_price'] = current_price
    db['shares'][ticker]['last_updated'] = time.time()
```

This code will add the current share price for each ticker and when it was last updated to our database.

Now we need to modify the `get_price` function to resemble the code below:

```
def get_price(ticker):

    # use cache if price is not stale
    if ticker in db["shares"].keys() and time.time() < db["shares"][ticker]["last_up\
dated"]+60:
        return db["shares"][ticker]["current_price"]

    page = requests.get("https://finance.yahoo.com/quote/" + ticker)
    soup = BeautifulSoup(page.text, "html5lib")

    price = soup.find('span', {'class':'Trsdu(0.3s) Fw(b) Fz(36px) Mb(-4px) D(ib)'})\
.text

    # remove thousands separator
    price = price.replace(",", "")

    # update price in db
    if ticker in db["shares"].keys():
        db["shares"][ticker]["current_price"] = price
        db["shares"][ticker]["last_updated"] = time.time()

    return price
```

The if statement at the top will cause the function to return the current price recorded in our database if it has been fetched recently, and the two new lines near the bottom of the function will ensure that when a new price is fetched, it gets recorded in the database, along with an updated timestamp.

You can play around with different caching time periods in this function and different refresh intervals in the JavaScript code to find the right tradeoff between accurate prices and fast load times.

[256]https://www.epochconverter.com/

Where Next?

Our stock dashboard is functional, and even useful to an extent, but there's still a lot more we could do with it. The following features would be good additions:

- Support for fractional shares.
- The ability to record the sale of shares.
- Timestamps for purchase (and sale) records.
- Support for cryptocurrencies, perhaps using data from CoinMarketCap[257].
- The ability to create multiple portfolios or user accounts.
- Graphs.

Next Tutorial

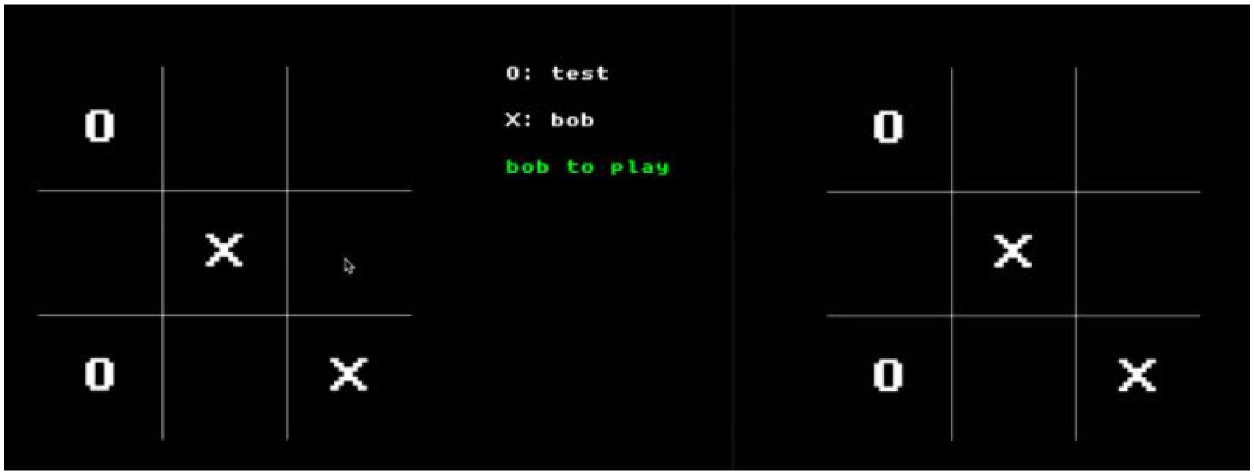
Up next, we're going to learn how to build a tic-tac-toe game using Kaboom.js for the interface and WebSocket to allow the players to play over the internet. The tutorial will leave you with a good understanding of gaming development principles and valuable knowledge on how to use sockets to communicate between clients in real time.

[257]https://coinmarketcap.com/

Building tic-tac-toe with WebSocket and Kaboom.js

Tic-tac-toe, or noughts and crosses, or Xs and Os, is a simple classic game for 2 players. It's usually played with paper and pen, but it also makes a good first game to write for networked multiplayer.

In this tutorial, we'll create a 2-player online tic-tac-toe game using a Node.js[258] server. Socket.IO[259] will enable realtime gameplay across the internet. We'll use Kaboom.js to create the game interface.



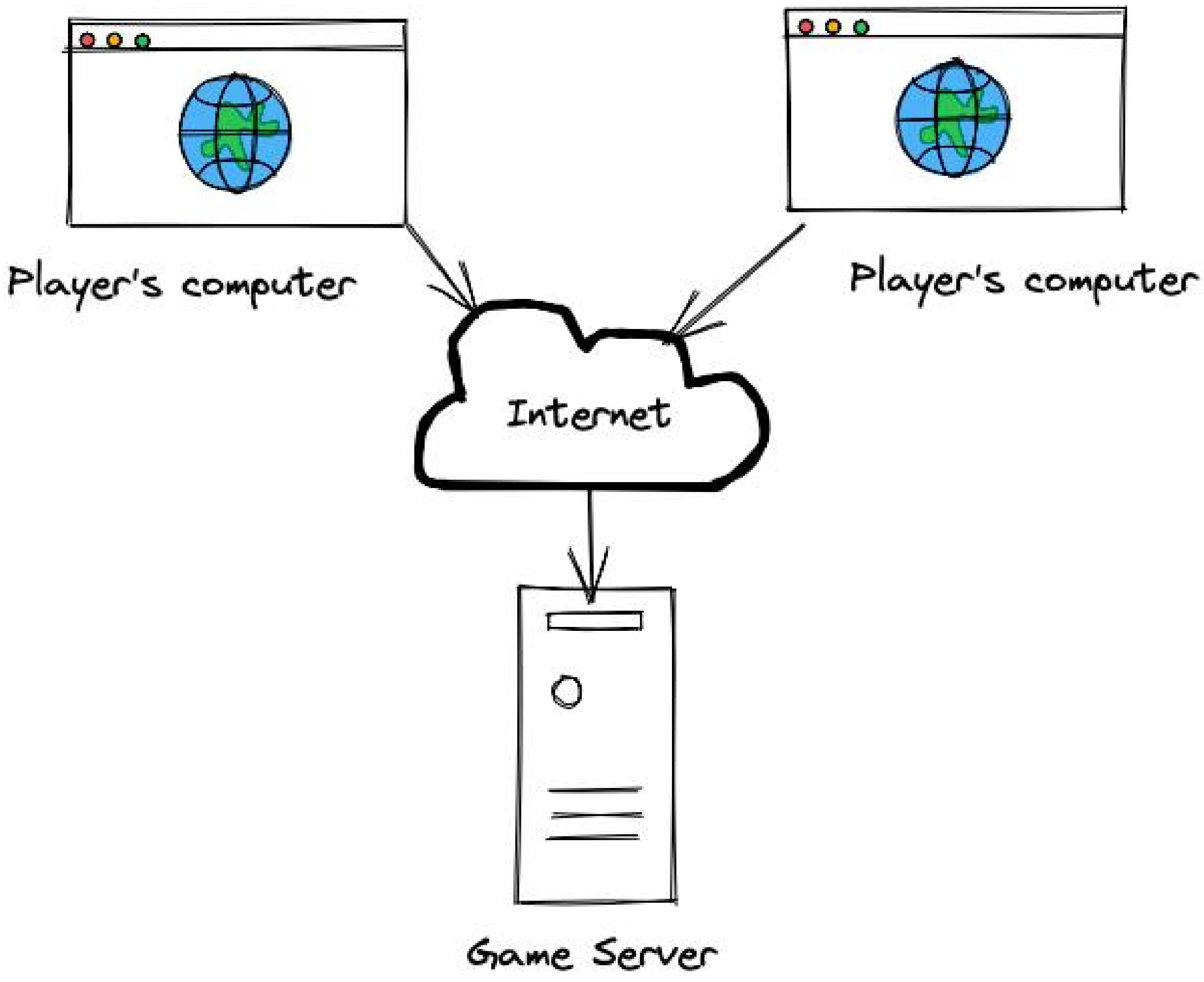
Game play

click to open gif[260]

How do multiplayer games work?

Multiplayer games have an architecture that typically looks something like this:

[258]https://nodejs.org/en/
[259]https://socket.io
[260]https://docs.replit.com/images/tutorials/27-tictactoe-kaboom/gameplay.gif



Game server architecture

Players (clients) connect to a game server over the internet. The game runs on the game server, where all the game rules, scores and other data are processed. The players' computers render the graphics for the game, and send player commands (from the keyboard, mouse, gamepad, or other input device) back to the game server. The game server checks if these commands are valid, and then updates the game state. The game state is a representation of all the variables, players, data and information about the game. This game state is then transmitted back to all the players and the graphics are updated.

A lot of communication needs to happen between a player's computer and the game server in online multiplayer games. This generally requires a 2-way, or bidirectional, link so that the game server can send data and notify players of updates to the game state. This link should ideally be quick too, so a more permanent connection is better.

With the HTTP[261] protocol that websites usually use, a browser opens a connection to a server, then

[261]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

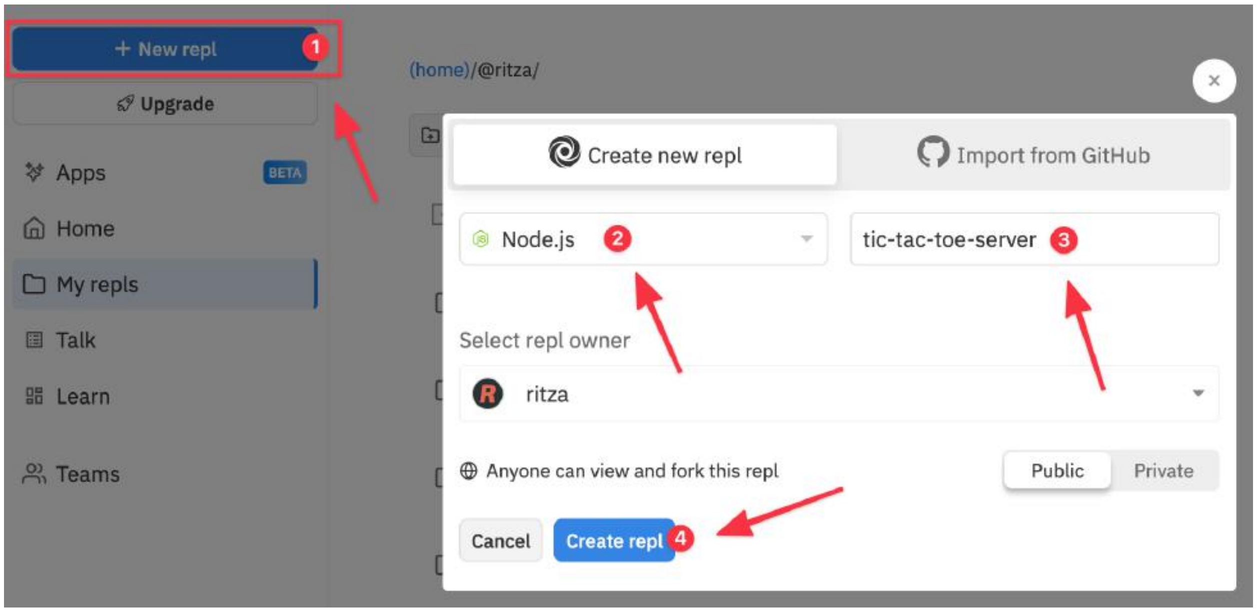
makes a request to the server, and the server sends back data, and closes the connection. There is no way for the server to initiate sending data to the browser. HTTP is also heavy on overhead, since it opens and closes a connection each time data is requested and sent.

WebSocket[262] is an advanced internet protocol that allows us to create a 2-way, persistent connection between a browser and a server. We'll use the Socket.IO[263] package to help us manage WebSocket connections in this project.

Creating a new project

For this project, we'll need to create 2 repls - 1 using Node.js for the game server, and 1 using Kaboom for the players. Head over to Replit[264] and create a two new repls:

- To create the server project, choose "Node.js" as your project type. Give this repl a name, like "tic-tac-toe-server".



Node.js 2

tic-tac-toe-server 3

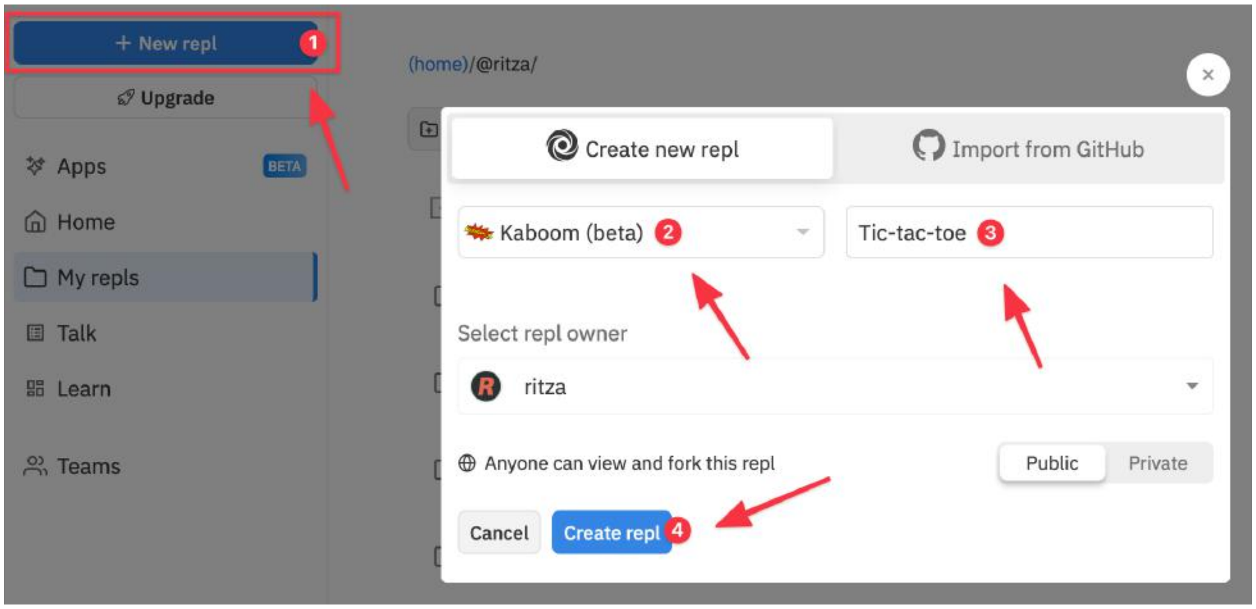
ritza

Anyone can view and fork this repl

Server repl

- To create the player project, choose "Kaboom" as your project type. Give this repl a name, like "tic-tac-toe".

[262]https://en.wikipedia.org/wiki/WebSocket
[263]https://socket.io
[264]https://replit.com



New Player repl

We'll code in the server repl to start, and then switch between repls as we build the game.

Setting up Socket.IO on the server

Add the following code to the file called `index.hs` in the server project to import Socket.IO:

```
const http = require('http');
const sockets = require('socket.io')

const server = http.createServer();
const io = sockets(server,  {
  cors: {
    origin: "https://tic-tac-toe.<YOUR-USER-NAME>.repl.co",
    methods: ["GET", "POST"]
  }
});

server.listen(3000, function() {
  console.log('listening on 3000');
});
```

In the first 2 lines, we import the built-in node `http`[265] package and the `socket.io`[266] package. The

[265]https://nodejs.org/api/http.html
[266]https://socket.io

`http` package enables us to run a simple HTTP server. The `socket.io` package extends that server to add WebSocket[267] functionality.

To create the HTTP server, we use the `http.createServer();` method. Then we set up Socket.IO by creating a new `io` object. We pass in the HTTP server object along with some configuration for CORS[268]. CORS stands for “Cross Origin Resource Sharing”, and it’s a system that tells the server which other sites are allowed to connect to it and access it. We set `origin` to allow our player repl to connect. Replace the `origin` value with the URL of the player project repl you set up earlier.

In the last `2` lines, we start the server up by calling its `listen` method, along with a port to listen on. We use 3000, as this is a standard for Node.js. If it starts successfully, we write a message to the console to let us know.

We’ll add the rest of the server code above the `server.listen` line, as we only want to start the server up after all the other code is ready.

Tracking the game state

Now that we have a server, lets think a bit about how we will represent, or model, the game. Our tic-tac-toe game will have a few different properties to track:

- The status of the game: What is currently happening? Are we waiting for players to join, are the players playing, or is the game over?
- The current positions on the tic-tac-toe board. Is there a player in a grid block, or is it empty?
- All the players. What are their names, and which symbol are they using, X or O?
- The current player. Whose turn is it to go?
- If the game ends in a win, who won it?

For the status of the game, we’ll add an enumeration[269] of the possible states we can expect. This makes it easier to track and use them as we go through the different phases of the game.

```
const Statuses = {
  WAITING: 'waiting',
  PLAYING: 'playing',
  DRAW: 'draw',
  WIN: 'win'
}
```

- WAITING: We are waiting for all the players to join the game.
- PLAYING: The players can make moves on the board.

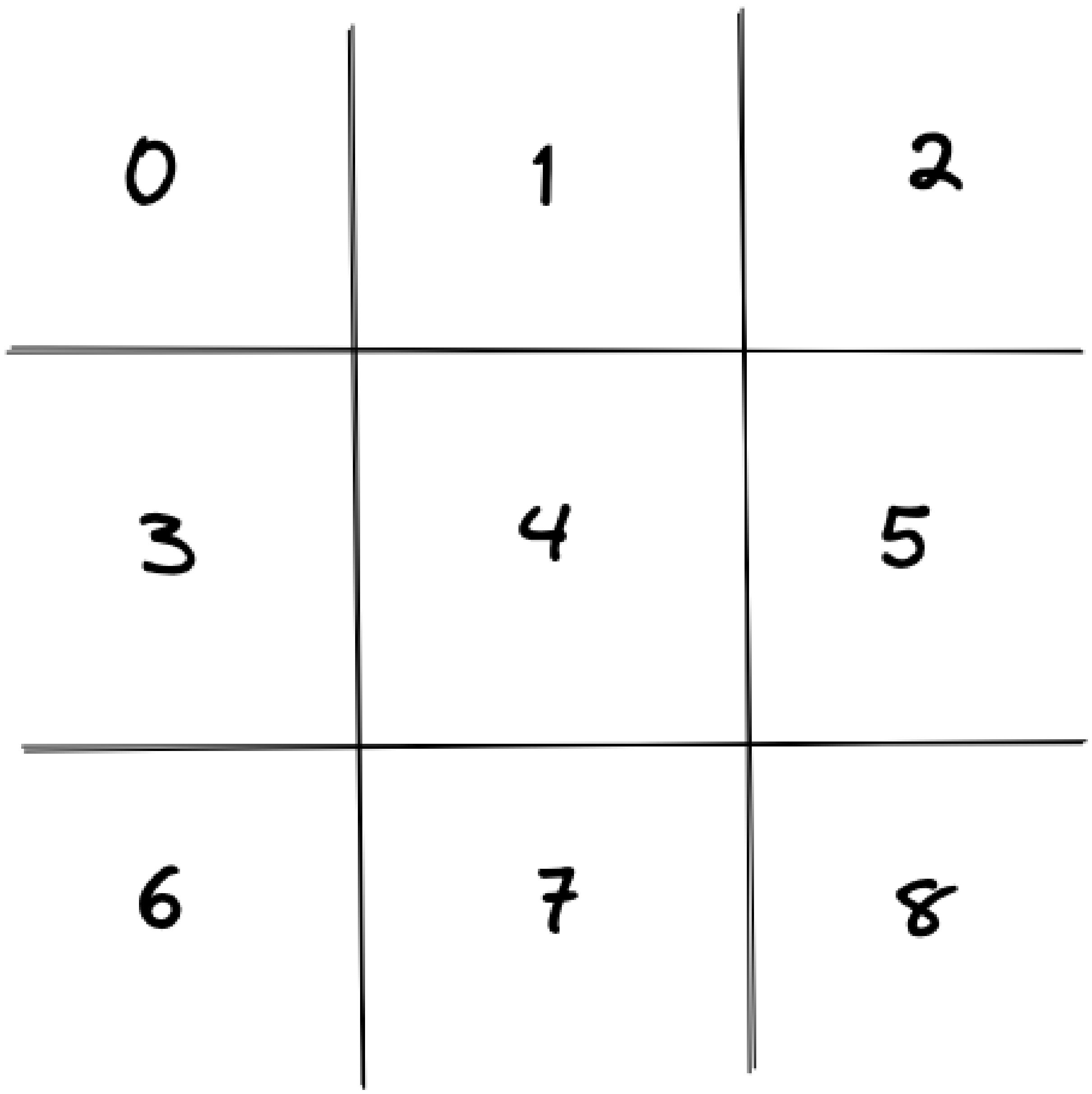
[267] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[268] https://developer.mozilla.org/en-US/docs/Glossary/CORS
[269] https://masteringjs.io/tutorials/fundamentals/enum

- DRAW: The game has ended in a draw.
- WIN: A player has won the game.

Now let's add a game state object to track everything:

```
let gameState = {
  board: new Array(9).fill(null),
  currentPlayer: null,
  players : [],
  result : {
    status : Statuses.WAITING
  }
}
```

First, we have a representation of the tic-tac-toe board as an array with 9 elements. This is how the array elements are mapped to the board:



Tic Tac Toe board mapped to array indices

Each number in the blocks represents the index at which the board position is represented in the array. Initially, we fill all the elements of the array with `null` to indicate that the block is open. When players make a move to occupy an open space, we'll add a reference to the player instead. That way we can keep track of which blocks are empty, and which are occupied by which player.

Next, we have `currentPlayer`, which we will alternately set to each player when it's their turn to move.

Then there is an array called `players`, which will hold references to both of the players in the game. This will allow us to show the names of the players on screen, as well as generally keep track of the players.

The `result` field is updated after every move. This field will contain the status of the game (as we defined above). As it's represented as an object, it will also be able to hold extra fields. We'll use that functionality to add a reference to the winner of the game, if the game ends in a win.

Accepting connections

When a player connects via WebSocket, Sockets.IO will fire a `connection`[270] event. We can listen for this event and handle tracking the connection, as well as creating listeners for other custom events. There are a few custom events[271] we can define here, that our players will emit:

- `addPlayer`: We'll use this event for a player to request joining the game.
- `action`: This is used when a player wants to make a move.
- `rematch`: Used when a game is over, but the players want to play again.

We can also listen for the built-in `disconnect`[272] event, which will alert us if a player leaves the game (for example, by closing the browser window or if their internet connection is lost).

Let's add the code that will hook up our listeners to the events:

```
io.on('connection', function (connection) {
  connection.on('addPlayer', addPlayer(connection.id));
  connection.on('action', action(connection.id));
  connection.on('rematch', rematch(connection.id));
  connection.on('disconnect', disconnect(connection.id));
});
```

Next we'll implement each of these listener functions, starting with `addPlayer`.

Side Note: Normally in examples for custom listeners, you'll see the handler code added immediately with an anonymous function, like this:

[270] https://socket.io/docs/v4/server-instance/#connection
[271] https://socket.io/docs/v4/emitting-events/
[272] https://socket.io/docs/v4/server-socket-instance/#disconnect

```
io.on('connection', function (connection) {
  connection.on('addPlayer', (data)=> {
          // some code here
  });

   connection.on('action', (data)=> {
          // some code here
  });

  // etc ...
});
```

This is convenient, especially when there are a couple of handlers, each with only a small amount of code. It's also handy because in each of the handler functions, you still have access to the `connection` object, which is not passed on each event. However, it can get a little messy and unwieldy if there are many event handlers, with more complex logic in each.

We're doing it differently so that we can separate the handlers into functions elsewhere in the code base. We do have one problem to solve though: if they are separate functions, how will they access the `connection` parameter in such a way that we can tell which player sent the command? With the concept of closures[273], which are well-supported in Javascript, we can make functions that return another function. In this way, we can pass in the `connection.id` parameter to the first wrapping function, and it can return another function that takes the data arguments from the Socket.IO event caller. Because the second function is within the closure of the first, it will have access to the `connection.id` parameter. The pattern looks like this:

```
io.on('connection', function (connection) {
  connection.on('addPlayer', addPlayer(connection.id));
  connection.on('action', action(connection.id)) ;
  // etc ...
});


function addPlayer(socketId){
        return (data)=>{
                // code here
        }
}

function action(socketId){
        return (data)=>{
                // code here
```

[273] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

```
        }
}
```

Handling new players

Add the following function to handle adding players:

```
function addPlayer(socketId){
  return (data)=>{
    const numberOfPlayers = gameState.players.length;
    if (numberOfPlayers >= 2){
      return;
    }

    let nextSymbol = 'X';
    if (numberOfPlayers === 1){
      if (gameState.players[0].symbol === 'X'){
        nextSymbol = 'O';
      }
    }

    const newPlayer = {
      playerName: data.playerName,
      id: socketId,
      symbol: nextSymbol
    };

    gameState.players.push(newPlayer);
    if (gameState.players.length === 2){
      gameState.result.status = Statuses.PLAYING;
      gameState.currentPlayer = newPlayer;
    }
    io.emit('gameState', gameState);

  }
}
```

This function does quite a bit. Let’s go through the main features.

- First it checks to see how many players are already in the game. If there are already 2 players, it returns early without changing anything. If this check passes, it goes on to add a new player. Note that even when there is no space in the game for a new player, we don’t disconnect the player - they still get updates and can watch the match.

- Next, the function figures out which symbol, X or O, the new player should be. It will assign X to the first player. If there is already a player, and the existing player's symbol is X, then it will assign O to the new player. Note that there is a possible case where there is only one player, and their symbol is O. This would occur if there are 2 players, and the player with the X symbol disconnects from the game, leaving only the player with the O symbol. This is why we always check what symbol the existing player in the game has.
- Then the function constructs a new player object with some identifying information, including the name that the player sends through, the `socketId` they connected on, and their symbol. When a new player requests to join, we expect them to send an object with a field `playerName` to tell us their handle.
- Now we add the new player to the player array in our `gameState` object, so that they are part of the game.
- We go on to check if we have 2 players, and start playing if we do. We begin by updating the status of the game to `PLAYING`, and set the `currentPlayer`, i.e. the player who is first to go, as the latest player to have joined.
- Finally, we use the Socket.IO `emit`[274] function to send the updated `gameState` to all connections. This will allow them to update the players' displays.

Handling player actions

The next handler takes care of the moves players make. We expect that the incoming data from the player will have a property called `gridIndex` to indicate which block on the board the player wants to mark. This should be a number that maps to the numbers for each block in the board, as in the picture earlier on.

```
function action(socketId){
  return (data)=> {
    if (gameState.result.status === Statuses.PLAYING && gameState.currentPlayer.id =\
== socketId){
      const player = gameState.players.find(p => p.id === socketId);
      if (gameState.board[data.gridIndex] == null){
        gameState.board[data.gridIndex] = player;
        gameState.currentPlayer = gameState.players.find(p => p !== player);
        checkForEndOfGame();
      }
    }
    io.emit('gameState', gameState);
  }
}
```

In this function, we check a couple of things first:

[274] https://socket.io/docs/v4/emitting-events/

- The game status must be `PLAYING` - players can't make moves if the game is in any other state.
- The player attempting to make the move must be the `currentPlayer`, i.e. the player whose turn it is to go.

If these conditions are met, we find the player in the `gameState.players` array using the built-in `find`[275] method on arrays, by looking for the player by their `socketId`.

Now we can check if the board position (`gridIndex`) requested by the player is available. We check that the value for that position in the `gameState.board` array is `null`, and if it is, we assign the player to it.

The player has made a successful move, so we give the other player a turn. We switch the `gameState.currentPlayer` to the other player by using the array `find`[276] method again, to get the player who does not match the current player.

We also need to check if the move the player made changed the status of the game. Did that move make them win the game, or is it a draw, or is the game still in play? We call out to a function `checkForEndOfGame` to check for this. We'll implement this function a little later, after we're done with all the handlers.

Finally, we send out the latest `gameState` to all the players (and spectators) to update the game UI.

Handling a rematch request

Let's make it possible for a player to challenge their opponent to a rematch when the game has ended:

```
function rematch(socketId){
  return (data) => {
    if (gameState.players.findIndex(p=> p.id === socketId) < 0) return; // Don't let\
 spectators rematch
    if (gameState.result.status === Statuses.WIN || gameState.result.status === Stat\
uses.DRAW){
      resetGame();
      io.emit('gameState', gameState);
    }
  }
}
```

This function first checks if the connection sending the rematch request is actually one of the players, and not just a spectator. If we can't find a match for a player, we return immediately, making no changes.

[275]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
[276]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

Then we check if the game is in one of the final states, either `WIN` or `DRAW`. If it is, we call out to a function `resetGame` to set up the game again. Finally, we send out the latest `gameState` to all the players.

Let's implement the `resetGame` function:

```
function resetGame(){
  gameState.board = new Array(9).fill(null);

  if (gameState.players.length === 2){
    gameState.result.status = Statuses.PLAYING;
    const randPlayer = Math.floor(Math.random() * gameState.players.length);
    gameState.currentPlayer = gameState.players[randPlayer];
  } else {
    gameState.result.status = Statuses.WAITING;
    gameState.currentPlayer = null;
  }
}
```

Let's take a look at what we're doing here:

- First, our function creates a new array for the `gameState` board. This effectively clears the board, setting all the positions back to `null`, or empty.
- Then it checks that there are still `2` players connected. If there are, it sets the game status back to `PLAYING` and chooses at random which player's turn it is to go. We choose the first player randomly so that there isn't one player getting an advantage by going first every time.

If there is only one player remaining, we set the game status to `WAITING` instead, and listen for any new players who want to join. We also set the `currentPlayer` to null, as we will choose which player should go once the new player has joined.

Handling disconnects

The last handler we need to implement is if a connection to a player is lost. This could be because the player has exited the game (by closing the browser tab), or has other internet issues.

```
function disconnect(socketId){
  return (reason) => {
    gameState.players = gameState.players.filter(p => p.id != socketId);
    if (gameState.players !== 2){
      resetGame();
      io.emit('gameState', gameState);
    }
  }
}
```

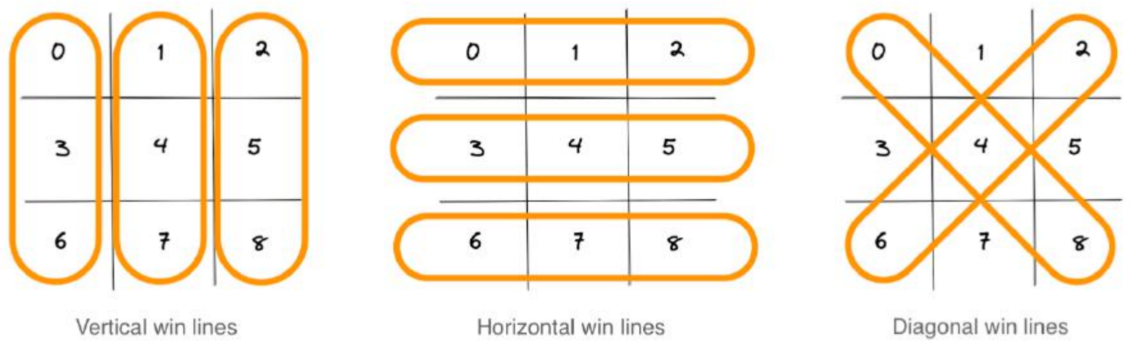
This function uses the built-in array `filter`[277] function to remove the player that disconnected from the server. Since it's possible that the disconnect event isn't from a player but from a spectator, we check the number of players left after filtering the disconnecting socket from the player list. If there aren't 2 players remaining after filtering, we reset the game and send out the updated game state.

Checking for the end of the game

Now we can get back to implementing the `checkForEndOfGame()` function we referenced in the `action` handler.

We're only interested in detecting 2 cases: A win or a draw.

There are just 8 patterns that determine if a player has won at tic-tac-toe. Let's map them to our board with its indexed blocks:



All possible win lines

We can encode each of these winning patterns into an array of 3 numbers each. Then we can add each of those patterns to a larger array, like this:

[277] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

```javascript
const winPatterns = [
  [0, 1, 2],
  [3, 4, 5],
  [6, 7, 8],
  [0, 3, 6],
  [1, 4, 7],
  [2, 5, 8],
  [0, 4, 8],
  [2, 4, 6]
]
```

Now that we have each winning pattern in an array, we can loop through each of them to see if there is a player that has positions that match any of the patterns.

Since the players are also in an array in `gameState.players`, we can loop through that array, and check each player against the winning pattern array. If a player matches any of these patterns, we can change the game status to `WIN` and set that player as the winner in the results.

Here is the code to do that:

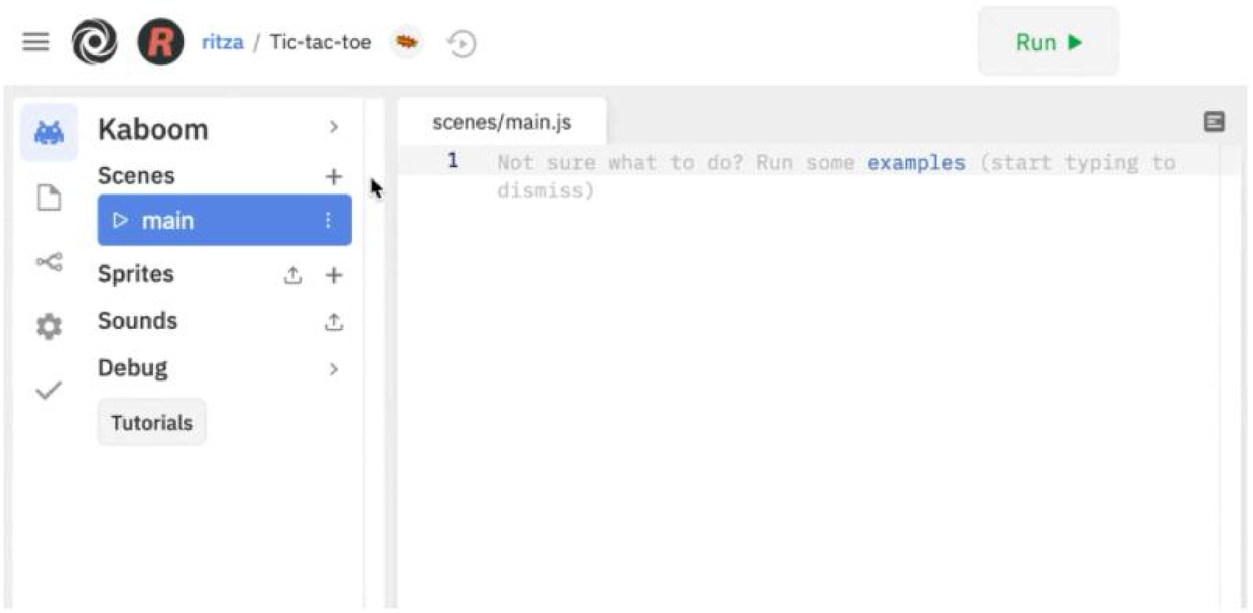
```javascript
function checkForEndOfGame(){

  // Check for a win
  gameState.players.forEach(player => {
    winPatterns.forEach(seq => {
      if (gameState.board[seq[0]] == player
          && gameState.board[seq[1]] == player
          && gameState.board[seq[2]] == player){
            gameState.result.status = Statuses.WIN;
            gameState.result.winner = player;
         }
    });
  });

  // Check for a draw
  if (gameState.result.status != Statuses.WIN){
    const emptyBlock = gameState.board.indexOf(null);
    if (emptyBlock == -1){
      gameState.result.status = Statuses.DRAW;
    }
  }
}
```

We also check for a draw in this function. A draw is defined as when all the blocks are occupied (no more moves can be made), but no player has matched one of the win patterns. To check if there are no more empty blocks, we use the array method `indexOf`[278] to find any `null` values in `gameState.board` array. Remember that `null` means an empty block here. The `indexOf` method will return `-1` if it can't find any `null` values. In that case, we set the game status to `DRAW`, ending the game.

Now we have all the functionality we need on the server, let's move on to building the Kaboom website the players will use to play the game.

Setting up Kaboom with Socket.IO

The first thing we need to do is create an opening scene in Kaboom that will prompt for the player's name, and setup Socket.IO so we can connect to the server. Head over to the Kaboom repl we created earlier, and add a new scene called `startGame`:



```
Not sure what to do? Run some examples (start typing to
dismiss)
```

add new scene

click to open gif[279]

Now we can add a reference to Socket.IO. Normally, in a plain HTML project, we could add a `<script>`[280] tag and reference the Socket.IO client script[281], hosted automatically on our game server. However, in the Kaboom project type on Replit, we don't have direct access to change the underlying HTML files. Therefore, we need to add the script programmatically. We can do it by accessing the `document`[282] object available in every browser, and insert a new element with our script.

[278]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf
[279]https://docs.replit.com/images/tutorials/27-tictactoe-kaboom/startGameScene.gif
[280]https://www.w3schools.com/tags/tag_script.asp
[281]https://socket.io/docs/v4/client-installation/#Installation
[282]https://developer.mozilla.org/en-US/docs/Web/API/Document

```
let script = document.createElement("script");
script.src = 'https://tic-tac-toe-server.<YOUR_USER_NAME>.repl.co' + '/socket.io/soc\
ket.io.js'
document.head.appendChild(script);
```

Replace the `<YOUR_USER_NAME>` part of the URL with your Replit username. This code inserts the new `<script>` tag into the `<head>`[283] section of the underlying HTML page that Kaboom runs in.

Let's move on to the code to prompt the player to enter their name.

```
const SCREEN_WIDTH = 1000;
const SCREEN_HEIGHT = 600;

add([
  text("What's your name? ",20),
  pos(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 3),
  origin("center")
]);

const nameField = add([
  text("",20),
  pos(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2),
  origin("center")
]);

charInput((ch) => {
    nameField.text += ch;
});

keyRelease("enter", ()=>{
    go("main", {playerName: nameField.text} );
})
```

To keep the calculations for the UI layout simpler, we'll use a fixed size for the screen. That's where the 2 constants for the screen width and height come in.

We use the Kaboom `add`[284] function to display the prompt "What's your name?" on the screen, using the `text`[285] component. We choose a position halfway across the screen, `SCREEN_WIDTH / 2`, and about a third of the way down the screen, `SCREEN_HEIGHT / 3`. We add the `origin`[286] component, set to `center`, to indicate that the positions we set must be in the center of the text field.

[283] https://www.w3schools.com/tags/tag_head.asp
[284] https://kaboomjs.com/#add
[285] https://kaboomjs.com/#text
[286] https://kaboomjs.com/#origin

Then we add another object with an empty `""` text component. This will display the characters the player types in. We position it exactly halfway down and across the screen. We also hold a reference to the object in the constant `nameField`.

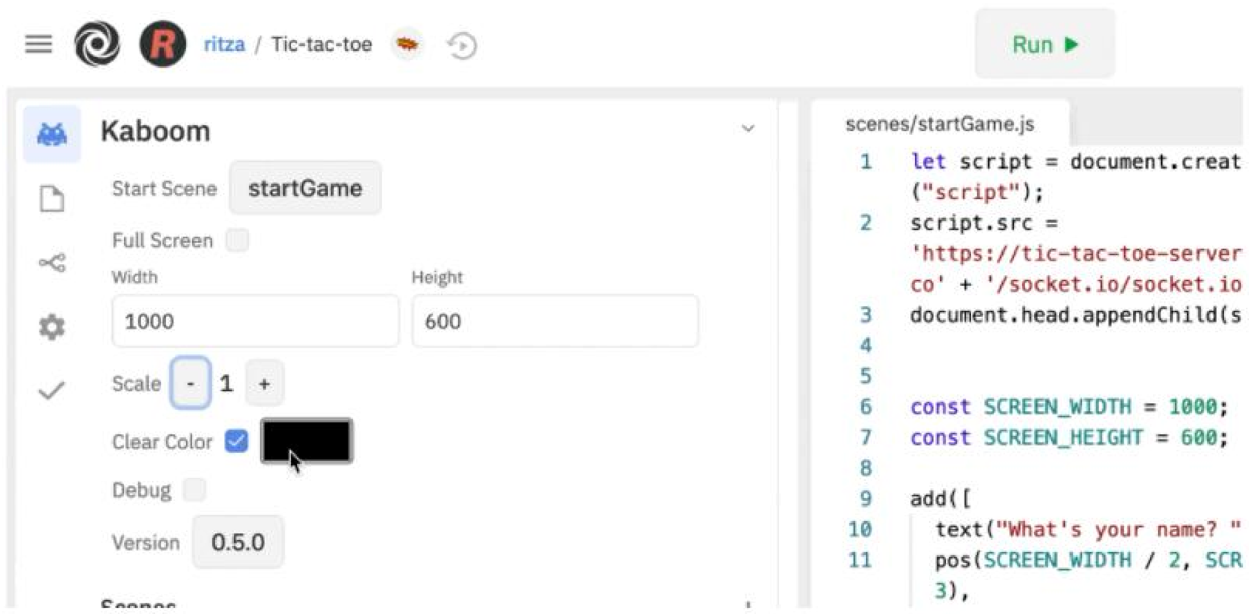
To get the user's keyboard input, we use the Kaboom function `charInput`[287]. This function calls an event handler each time a key on the keyboard is pressed. We take that character and append it to the text in the `nameField` object. Now, when a player presses a key to enter their name, it will show up on the screen.

Finally, we use the Kaboom function `keyRelease`[288] to listen for when the player pushes the `enter` key. We'll take that as meaning they have finished entering their name and want to start the game. In the handler, we use the Kaboom `go`[289] function to redirect to the main scene of the game.

Setting Kaboom parameters

We've created a new starting scene, and we have a fixed size for the game window. Now we need to update the Kaboom settings so that the game play environment reflects those choices.

Click the dropdown next to the Kaboom menu. Set the "Start Scene" to "StartGame". Uncheck "Full Screen", and set the Width to 1000 and Height to 600. Set the scale to "1". Then choose dark blue or black as the "Clear Color".



Kaboom

Start Scene: startGame
Full Screen: ☐
Width: 1000
Height: 600
Scale: - 1 +
Clear Color: ✓ ■
Debug: ☐
Version: 0.5.0

```
let script = document.creat
("script");
script.src =
'https://tic-tac-toe-server
co' + '/socket.io/socket.io
document.head.appendChild(s


const SCREEN_WIDTH = 1000;
const SCREEN_HEIGHT = 600;

add([
  text("What's your name? "
  pos(SCREEN_WIDTH / 2, SCR
  3),
```

Kaboom setup

click to open gif[290]

[287] https://kaboomjs.com/#charInput
[288] https://kaboomjs.com/#keyRelease
[289] https://kaboomjs.com/#go
[290] https://docs.replit.com/images/tutorials/27-tictactoe-kaboom/kaboomSettings.gif

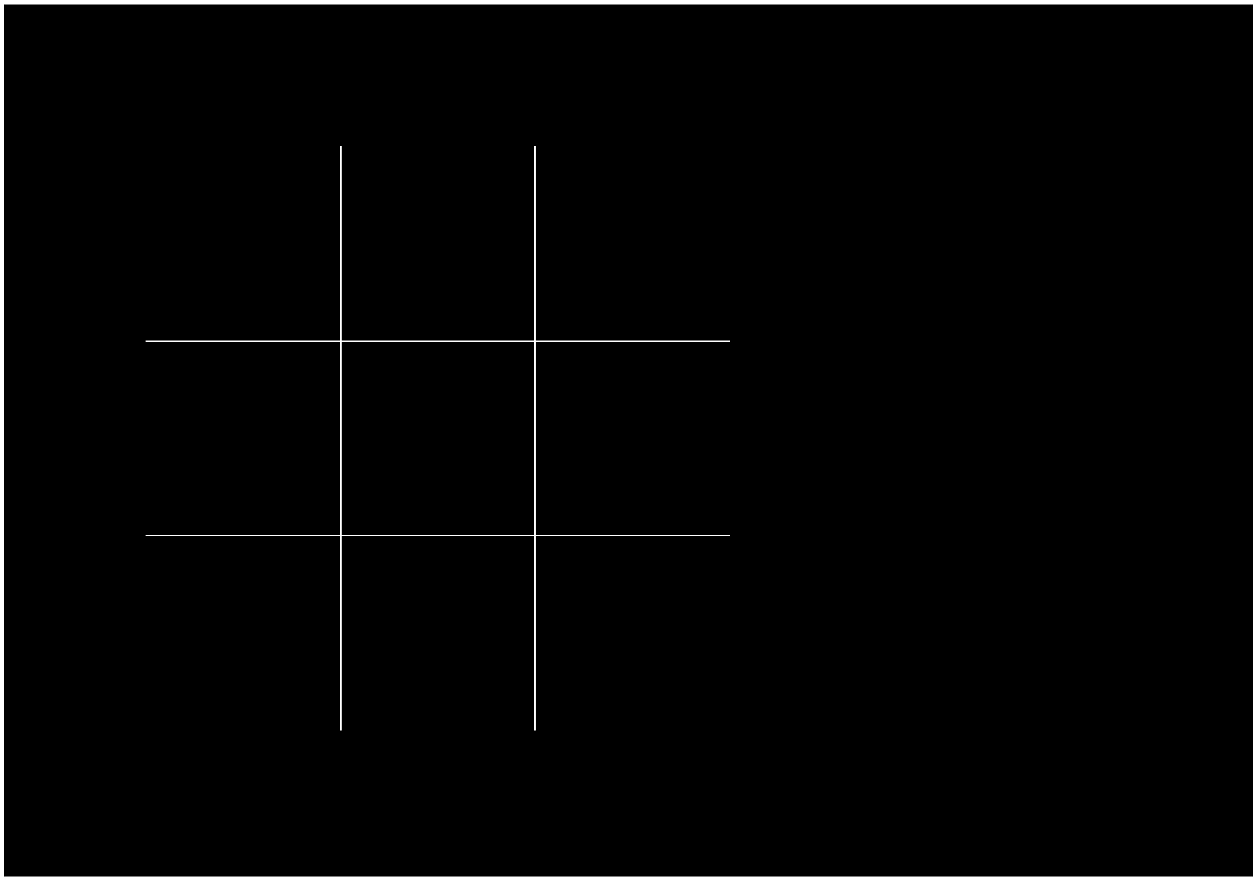
Adding the game board

Now we can add the UI elements for the game itself. Open the "main" scene in your Kaboom repl, and add the following code to draw the tic-tac-toe board:

```
    // Board
    add([
      rect(1,400),
      pos(233,100),
    ]);

    add([
      rect(1,400),
      pos(366,100),
    ]);

    add([
      rect(400,1),
      pos(100, 233),
    ]);

    add([
      rect(400,1),
      pos(100, 366),
    ]);
```

This adds 4 rectangles with a width of 1 pixel and length of 400 pixels to the screen - each rectangle is more like a line. This is how we draw the lines that create the classic tic-tac-toe board shape. The first 2 rectangles are the vertical lines, and the second 2 are the horizontal lines. We place the board closer to the left side of the screen, instead of the center, to save space for game information to be displayed on the right hand side of the screen.

If you run the game, and enter your name, you should see the board layout like this:



board layout

Now we need to add a way to draw the X and O symbols in each block. To do this, we'll add objects with text components in each block of the board. First, we'll make an array containing the location and size of each block:

```
const boardSquares = [
  {index: 0, x: 100, y: 100, width:133, height: 133 },
  {index: 1, x: 233, y: 100, width:133, height: 133 },
  {index: 2, x: 366, y: 100, width:133, height: 133 },
  {index: 3, x: 100, y: 233, width:133, height: 133 },
  {index: 4, x: 233, y: 233, width:133, height: 133 },
  {index: 5, x: 366, y: 233, width:133, height: 133 },
  {index: 6, x: 100, y: 366, width:133, height: 133 },
  {index: 7, x: 233, y: 366, width:133, height: 133 },
  {index: 8, x: 366, y: 366, width:133, height: 133 }
];
```

We can run through this array and create a text object that we can write to when we want to update the symbols on the board. Let's create a function to do that.

```javascript
function createTextBoxesForGrid(){
  boardSquares.forEach((square)=>{
    let x = square.x + square.width*0.5;
    let y = square.y + square.height*0.5;
    square.textBox = add([
      text('', 40),
      pos(x, y),
      origin('center')
    ]);
  })
}

createTextBoxesForGrid();
```

This function uses the array `forEach`[291] method to loop through each "square" definition in the `boardSquares` array. We then find the center `x` and `y` of the square, and `add`[292] a new text object to the screen, and also add it to the square definition on the field `textBox` so we can access it later to update it. We use the `origin` component to ensure the text is centered in the square.

Finally, we call the function to create the text boxes.

Adding player names and game status

Now let's add some areas for the player's names and for the current status of the game (whose turn it is to play, if someone has won, or if it's a draw).

```javascript
// Players and game status elements
const playerOneLabel = add([
  text('', 16),
  pos(600, 100),
]);

const playerTwoLabel = add([
  text('', 16),
  pos(600, 150),
]);

const statusLabel = add([
  text('', 16),
  pos(600, 200),
```

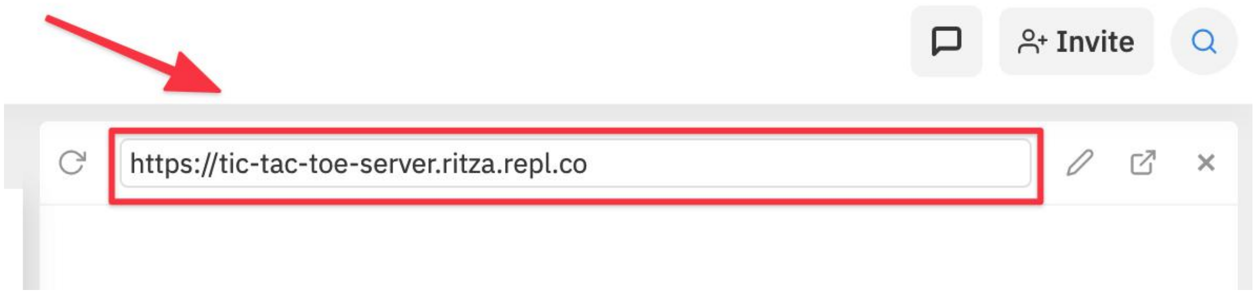
[291] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach
[292] https://kaboomjs.com/#add

```
  color(0,1,0)
]);
```

Here we add 3 objects with `text`[293] components. The first 2 are placeholders for the player names and symbols. The third one is for the game status. They are positioned to the right of the screen, and contain empty text to start. We'll change the contents as we receive new game states from the server. The last object has a `color` component to set the color of the text to green. This is to make the status message stand out from the rest of the text.

Connecting to the server

To connect to the game server, we need to initialize the Socket.IO library we dynamically added earlier. We need to provide the URL to the server repl, so copy that from the output window:



Copying server url

Now add this code along with the server URL:

```
var socket = io('https://tic-tac-toe-server.<YOUR_USER_NAME>.repl.co');

socket.on('connect', function(){
  socket.emit("addPlayer", {
    playerName: args.playerName
  });
});
```

In the first line, we initialize the Socket.IO client library[294] to connect to the server. Then we add a listener to the `connect`[295] event. This lets us know when we have established a connection to the server.

If we have a connection, we then `emit`[296] an event to the server, with our custom event type `addPlayer`. We also add in the player name, which we passed to this scene from the `startGame`

[293]https://kaboomjs.com/#text
[294]https://socket.io/docs/v4/client-initialization/
[295]https://socket.io/docs/v4/client-socket-instance/#Socket-connected
[296]https://socket.io/docs/v4/emitting-events/

scene. Any arguments passed between scenes are accessible through the `args` parameter within the scene. Emitting the `addPlayer` event to the server will cause the `addPlayer` event handler to fire on the server side, adding the player to the game, and emitting back the game state.

Handling updated game state

Remember that our server emits a `gameState` event whenever something changes in the game. We'll listen for that event, and update all the UI elements in an event handler.

First, we need to add the definitions of each status as we have done on the server side, so that we can easily reference them in the code:

```
const Statuses = {
        WAITING: 'waiting',
        PLAYING: 'playing',
        DRAW: 'draw',
        WIN: 'win'
}
```

Now we can add a listener and event handler:

```
socket.on('gameState', function(state){
  for (let index = 0; index < state.board.length; index++) {
    const player = state.board[index];
    if (player != null){
      boardSquares[index].textBox.text = player.symbol;
    } else
    {
      boardSquares[index].textBox.text = '';
    }
  }

  statusLabel.text = '';
  switch (state.result.status) {
    case Statuses.WAITING:
      statusLabel.text = 'Waiting for players....';
      break;
    case Statuses.PLAYING:
      statusLabel.text = state.currentPlayer.playerName + ' to play';
    break;
    case Statuses.DRAW:
      statusLabel.text = 'Draw!';
    break;
    case Statuses.WIN:
      statusLabel.text = state.result.winner.playerName + ' Wins! \nPress R for rema\
tch';
    break;
    default:
      break;
  }

  playerOneLabel.text = '';
  playerTwoLabel.text = '';
  if (state.players.length > 0){
    playerOneLabel.text = state.players[0].symbol + ': ' + state.players[0].playerNa\
me;
  }

  if (state.players.length > 1){
    playerTwoLabel.text = state.players[1].symbol + ': ' + state.players[1].playerNa\
me;
  }

});
```

This function looks quite long, but it's mainly just updating the text boxes we added.

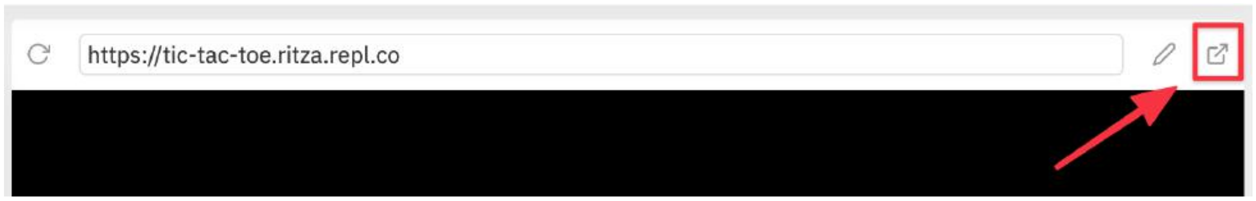
First, we loop through the board positions array that is passed from the server on the `state` payload, to check each block for a player positioned on it. If there is a player on a block, we write that player's symbol to the corresponding text box, found in the `boardSquares` array we created above. If there is no player in the block, i.e it's a `null` value, we write an empty string to the text block.

Then we update the `statusLabel` to show what is currently happening in the game. We use a `switch`[297] statement to create logic for each of the possibilities. We write a different message to the `statusLabel` text box depending on the status, drawing from data in the `gameState` object.

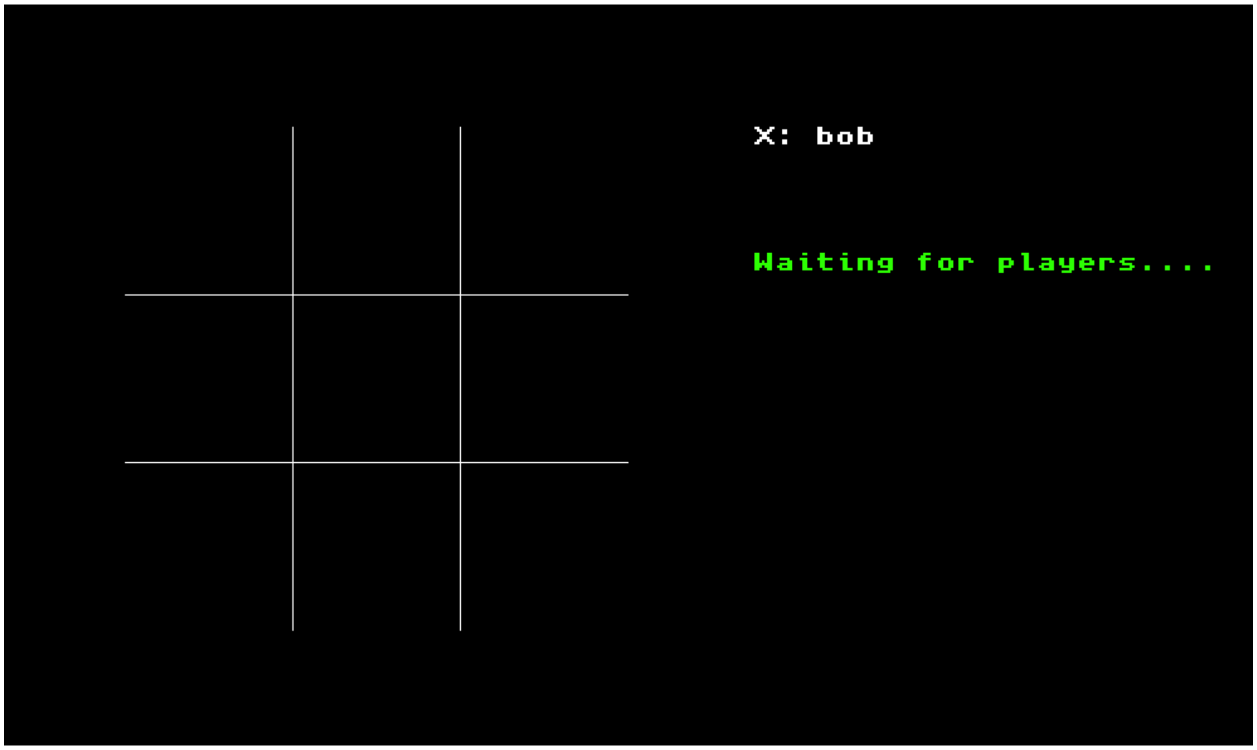
Next we update the player name text boxes. First we reset them, in case one of the players has dropped out. Then we update the text boxes with the players' symbols and names. Note that we first check if there are the corresponding players in the array.

Now that we're done with updating from the game state, let's try running the game again. Open the game window in a new tab so that requests to the repl server don't get blocked by the browser due to the CORS header 'Access-Control-Allow-Origin' not matching in the embedded window. Make sure the server is also running, and enter your name. You should see something like this:

[297] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch



Open in new tab



Waiting for another player

You can connect to your game in another browser tab, and enter another name. Then you should see both names come up, and the status message change to allow a player to make a move. Of course, we haven't yet implemented the code to enable making a move from the UI, so let's do that now.

Handling player moves

We want a player to be able to click on a block to place their move. Kaboom has a function `mouseRelease`[298] that we can use to handle mouse click events. All we need then is the position the mouse cursor is at, and we can map that to one of the board positions using our `boardSquares` array to do the lookup. We'll use the Kaboom function `mousePos`[299] to get the coordinates of the mouse:

[298]https://kaboomjs.com/#mouseRelease
[299]https://kaboomjs.com/#mousePos

```
mouseRelease(() => {
  const mpos = mousePos();
  // find the square we clicked on
  for (let index = 0; index < boardSquares.length; index++) {
    const square = boardSquares[index];
    if (mpos.x > square.x
        && mpos.x < square.x + square.width
        && mpos.y > square.y
        && mpos.y < square.y + square.height){
          socket.emit("action", {
            gridIndex: square.index
          });
          break;
        }
  }
});
```

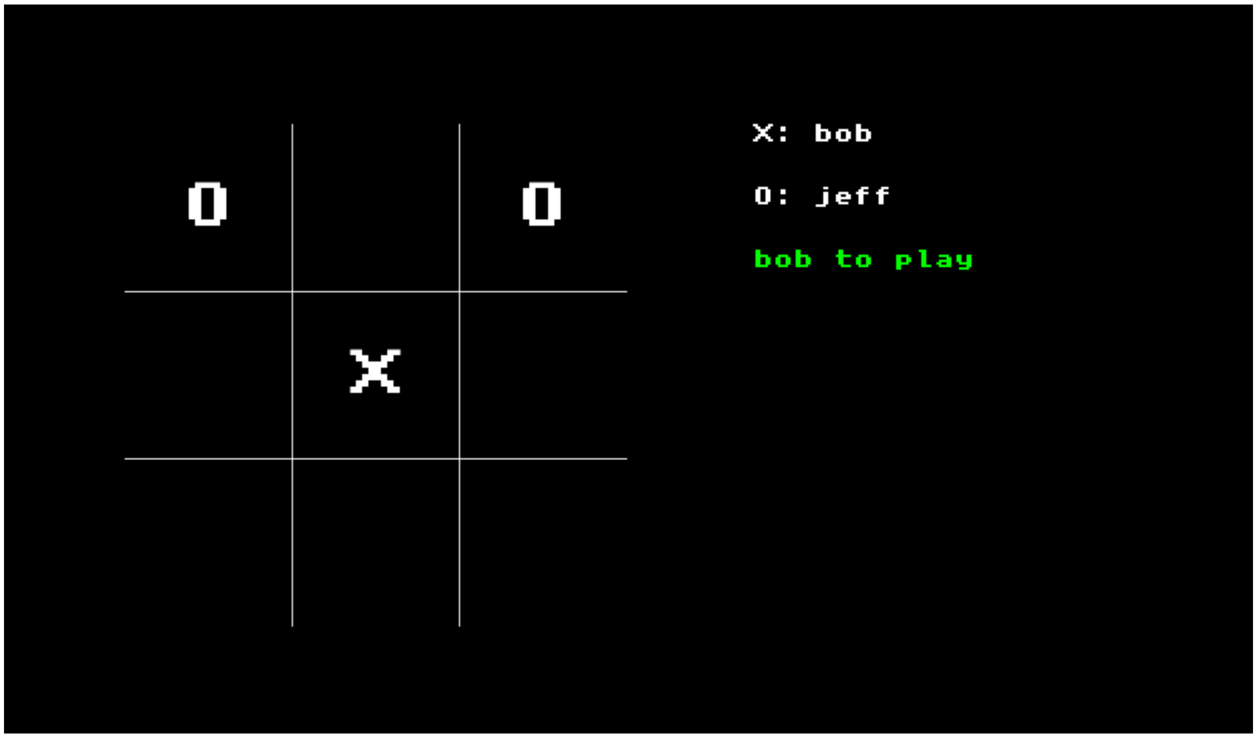
If we find a 'hit' on one of the board squares, we emit our `action` event. We pass the index of the square that was clicked on as the payload data. The server listens for this event, and runs the logic we added for the `action` event on the server side. If the action changes the game state, the server will send back the new game state, and the UI elements update.

The only other input we need to implement is to check if the player wants a rematch. To do that, we'll assign the `r` key as the rematch command. We can use the Kaboom function `charInput`[300] to listen for key press events. We'll check if the key is `r`, or `R`, then emit the `rematch` event. We don't have any data to pass with that, so we'll just pass `null`.

```
charInput((ch) => {
  if (ch === 'r' || ch === 'R'){
    socket.emit("rematch", null)
  }
});
```

Now you can run the game (and the server), and open the game in another tab, and you should be able to play tic-tac-toe against yourself! Send a link to the game to a friend, and see if they can join and play against you.

[300]https://kaboomjs.com/#charInput



playing tic tac toe

Next Steps

Now that you know the basics of creating a multiplayer online game, try your hand at making some different games, like checkers or chess or go.

Happy coding!

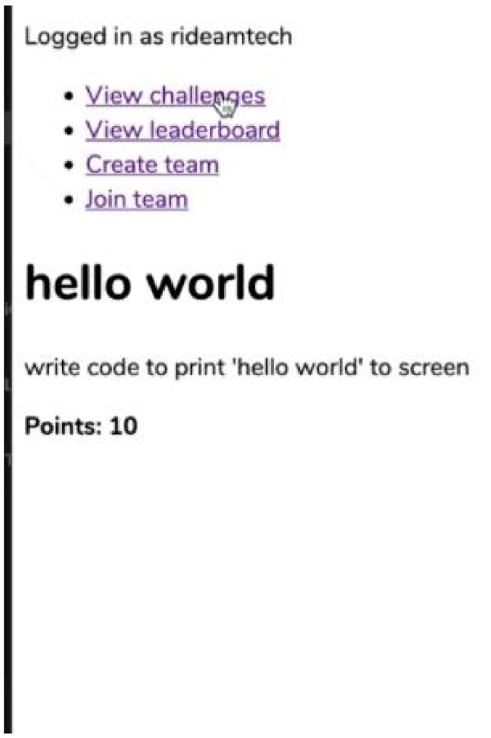
Next Tutorial

In the next tutorial, we take a look at how to build a team technical challenge website with `replit.web`. When complete, the site will serve as a leaderboard that can be used to track competition progress for hackathons and other similar challenges. The resulting application can also be further customized to give it a personal look or additional functionality.

Build a team technical challenge website with `replit.web`

Code competitions and hackathons are a fun way to expand your programming skills, get exposed to new ideas, and work together to solve difficult problems. The time-limited, competitive nature of these competitions provides an additional challenge.

In this tutorial, we'll use the `replit.web` framework to build a leaderboard website for an online technical challenge in the vein of Advent of Code[301] or Hackasat[302]. We'll focus on the generic aspects of the site, such as teams, challenges and scores, so once we're done, you can use the site for your own competition.



Logged in as rideamtech

- View challenges
- View leaderboard
- Create team
- Join team

hello world

write code to print 'hello world' to screen

Points: 10

Challenge site functionality

click to open gif[303]

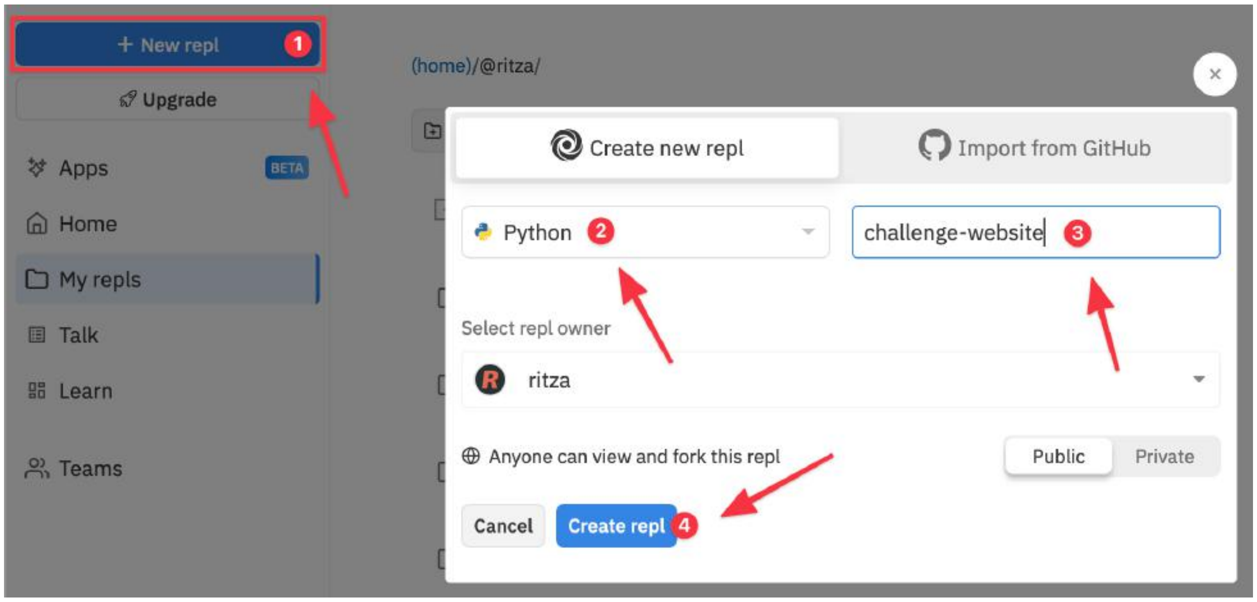
By the end of this tutorial, you'll be able to:

[301]https://adventofcode.com/
[302]https://www.hackasat.com/
[303]https://docs.replit.com/images/tutorials/28-technical-challenge-site/site-functionality.gif

- Use Replit's Flask-based web framework to rapidly develop authenticated web applications with persistent storage.
- Use WTForms to create sophisticated web forms.
- Use custom function decorators to handle multiple user roles.

Getting started

To get started, sign into Replit[304] or create an account[305] if you haven't already. Once logged in, create a Python repl.



Creating a new repl

Our competition website will have the following functionality:

- Users can sign in with their Replit accounts and either create a team or join an existing team. To join an existing team, a team password will be required.
- Once they're in a team, users will be able to view challenges and submit challenge solutions. To keep things simple, we will validate challenge solutions by requiring users to submit a unique code per challenge.
- A designated group of admin users will have the ability to add and remove challenges, start and end the competition, and clear the database for a new competition.

Let's start off our competition application with the following module imports in `main.py`:

[304] https://replit.com
[305] https://replit.com/signup

```
from flask import Flask, render_template, flash, redirect, url_for, request
from replit import db, web
```

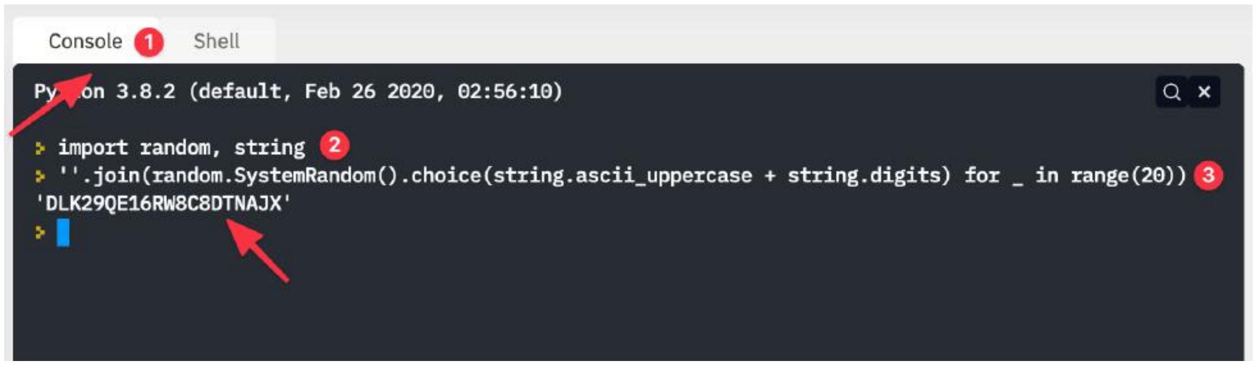
Here we're importing a number of Flask features we'll need. We could just use `import flask` to import everything, but we'll be using most of these functions often enough that having to prepend them with `flask.` would quickly become tiresome. We're also importing Replit's `db` and `web` modules, which will give us data persistence and user authentication.

Now let's create our app and initialize its database. Add the following code just below the import statements in `main.py`:

```
app = Flask(__name__)

# Secret key
app.config['SECRET_KEY'] = "YOUR-SECRET-KEY-HERE"

# Database setup
db_init()
users = web.UserStore()

ADMINS = ["YOUR-REPLIT-USERNAME-HERE"]
```

Here we initialize our application, our general and user databases, and our list of admins. Make sure to replace the string in `ADMINS` with your Replit username before proceeding. Also replace the secret key with a long, random string. You can generate one in your repl's Python console with the following two lines of code:

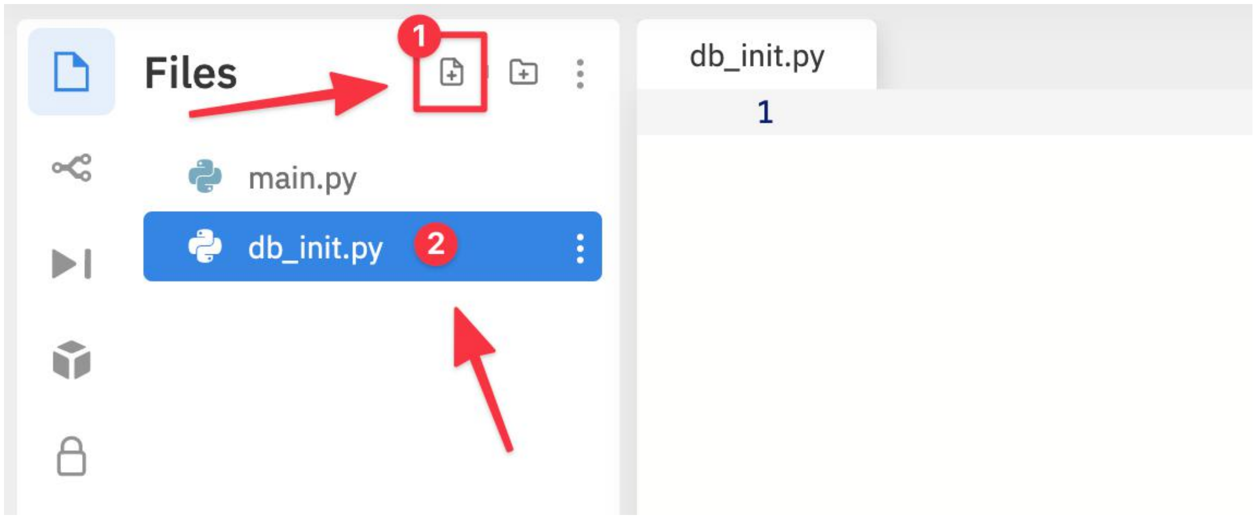
```
import random, string
''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ i\
n range(20))
```



```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> import random, string
> ''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ in range(20))
'DLK29QE16RW8C8DTNAJX'
>
```

Generating a random string

You'll notice that `db_init()` is undefined. As this is going to be a fairly large codebase, we're going to put it in a separate file. Create the file `db_init.py` in your repl's files tab:



Database init file

Add the following code to this file:

```python
from replit import db

def db_init():
    if "teams" not in db.keys():
        db["teams"] = {}

    if "challenges" not in db.keys():
        db["challenges"] = {}

    if "competition_started" not in db.keys():
        db["competition_started"] = False
```

Replit's Database[306] can be thought of and used as one big Python dictionary that we can access with `db`[307]. Any values we store in `db` will persist between repl restarts.

To import this file in `main.py`, we can use an `import` statement in much the same way as we would for a module. Add this line in `main.py`, below your other imports:

```python
from db_init import db_init
```

We've also defined a secondary database `users` in `main.py`. While `db` only contains what we put

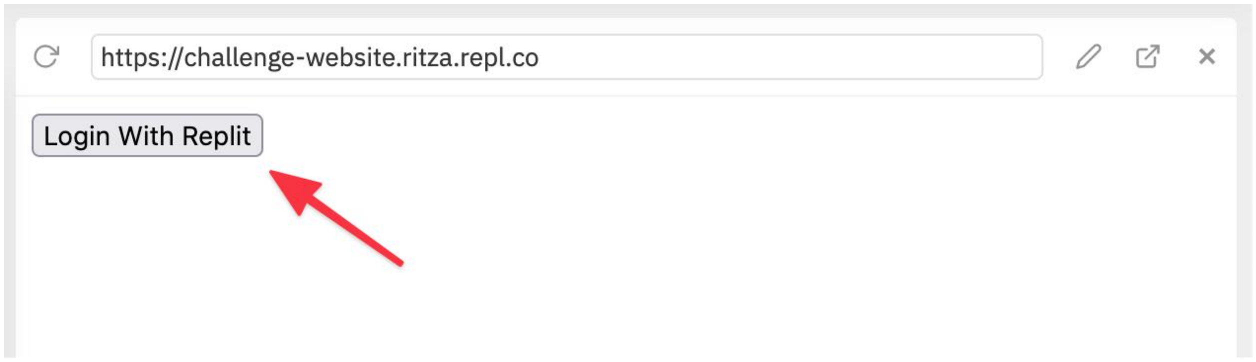
[306]https://docs.replit.com/hosting/database-faq
[307]https://replit-py.readthedocs.io/en/latest/api.html#module-replit.database

into it, `users` is a UserStore[308] that will automatically have the names of users who sign into our application added as keys, so we can easily store and retrieve information about them.

Now let's create some test content and run our app. Add the following code, and then run your repl.

```
# Routes
@app.route("/")
@web.authenticated
def index():
    return f"Hello {web.auth.name}"

web.run(app)
```

Because we've added the `@web.authenticated` function decorator[309] to our index page, it will only be available to logged in users. You should see this now, as your app will show a login button. Click on that button, and authorize your application to use Replit authentication in the window that pops up.



Login Button

Having done that, you should now see the greeting we implemented above. If you send your repl to a friend, they will also be able to log in, and see their own Replit username on the greeting message.

Creating user roles

Function decorators like `@web.authenticated`, which prevent a function from executing unless certain conditions are met, are very useful for web applications like this one, in which we want to restrict certain pages based on who's attempting to view them. `@web.authenticated` restricts users based on *authentication* – who a user is. We can now create our own decorators to restrict users based on *authorization* – what a user is allowed to do.

For this site, we're concerned about three things:

[308]https://replit-py.readthedocs.io/en/latest/api.html
[309]https://realpython.com/primer-on-python-decorators/

- Is the user in a team? Users who aren't need to be able to create or join a team, and users who are need to be able to submit challenge solutions.
- Is the user an admin? Users who are need to be able to create challenges, and perform other administrative tasks. For the sake of fairness, they should not be allowed to join teams themselves.
- Is the competition running? If not, we don't want non-admin users to be able to view challenge pages or attempt to submit solutions.

First, we'll create two helper functions to answer these questions. Add the following code to `main.py`, just below your ADMINS list:

```python
# Helper functions
def is_admin(username):
    return username in ADMINS

def in_team(username):
    if "team" in users[username].keys():
        return users[username]["team"]
```

The `is_admin()` function will return `True` if the provided user is an admin, or `False` otherwise. The function `in_team()` will return the name of the team the user is in, or `None` if they aren't in a team.

Now we can create our authorization function decorators. Add the following import function to the top of `main.py`:

```python
from functools import wraps
```

Then add this code below our helper functions:

```python
# Authorization decorators
def admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if not is_admin(web.auth.name):
            flash("Permission denied.", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

This code may look a bit strange if you haven't written your own decorators before. Here's how it works: `admin_only` is the name of our decorator. You can think of decorators as functions which take other functions as arguments. (The code coming up is example code for the purpose of illustration, and not part of our program.) Therefore, if we write the following:

```python
@admin_only
def admin_function():
    return f"Hello admin"

admin_function()
```

it will be roughly equivalent to:

```python
def admin_function():
    return f"Hello admin"

admin_only(admin_function)
```

So whenever `admin_function` gets called, the code we've defined in `decorated_function` will execute before anything we define in `admin_function`. This means we don't have to include an `if not is_admin` check in every piece of admin functionality. As per the code, if a non-admin attempts to access restricted functionality, our app will flash[310] a warning message and redirect them to the home page.

We also need to define a decorator for the opposite case, where we need to ensure that the current user is not an admin. Add the following code just below the `# Authorization decorators` code you added above:

```python
def not_admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if is_admin(web.auth.name):
            flash("Admins can't do that.", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

We will do much the same thing for `team_only` and `not_team_only`:

[310]https://flask.palletsprojects.com/en/2.0.x/patterns/flashing/

```python
def team_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if not in_team(web.auth.name):
            flash("Join a team first!", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function

def not_team_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if in_team(web.auth.name):
            flash("You've already joined a team!", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

Finally, we need to add a decorator to check whether our competition is running. This is mainly for challenge description pages, so we'll add an exception for non-admin users:

```python
def competition_running(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if not (is_admin(web.auth.name) or db["competition_started"]):
            flash("The competition has not started yet.")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

Now that we've added our authorization controls, it's time to give them something to authorize. In the next sections, we'll define all of our app's functionality and build its front-end.

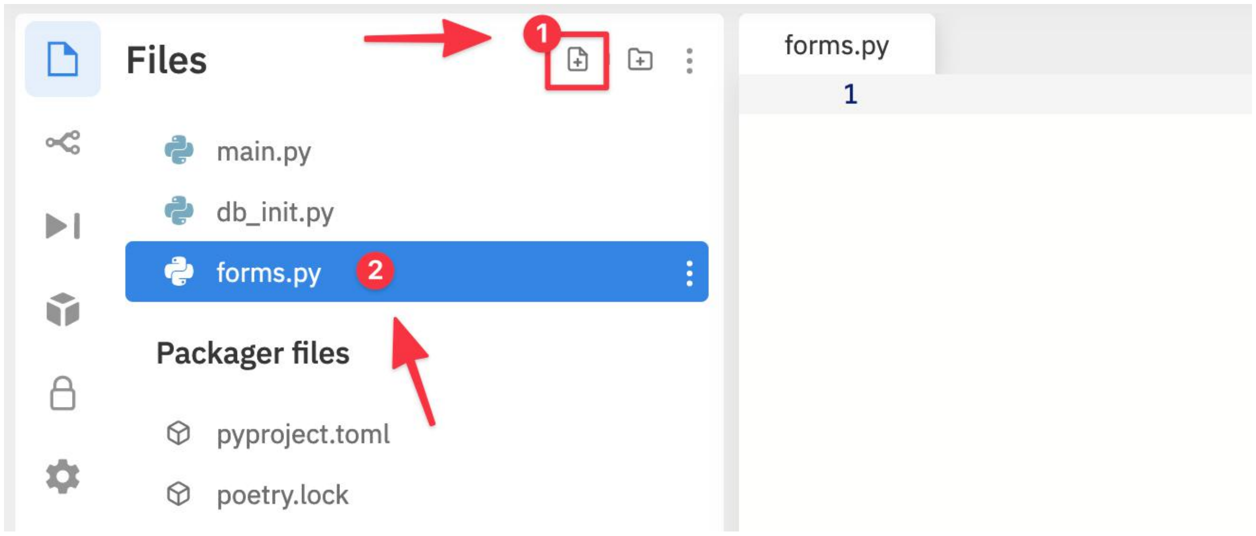
Building forms

The bulk of interactivity in our application will be enabled through forms. Users will be able to create and join teams, as well as submit challenge solutions. When we work with web forms, there's a lot to consider, including:

- Which users should be able to submit which forms (authorization)?
- What validation do we want on different fields? For example, length requirements, or ensuring a given value is an integer rather than a string.
- How do we give feedback on data that doesn't pass our validations?
- Security concerns around user input, such as SQL injection[311], cross-site scripting[312] and cross-site request forgery[313]. While the first one won't be relevant to our app, the second two are.

We could build all of this ourselves using Flask's `request.form` as a basis, but fortunately someone else has already done the hard work and built the WTForms[314] library, as well as Flask WTF[315], which integrates `WTForms` with Flask. We'll be using both of these to construct our application's various forms.

To keep our codebase navigable, we'll put all our form code in a separate file, like we did with our database initialization code. Create `forms.py` in your repl's files tab now:



Building forms

We'll start this file off with some imports:

[311]https://owasp.org/www-community/attacks/SQL_Injection
[312]https://owasp.org/www-community/attacks/xss/
[313]https://owasp.org/www-community/attacks/csrf
[314]https://wtforms.readthedocs.io/en/2.3.x/
[315]https://flask-wtf.readthedocs.io/en/0.15.x/

```python
from replit import db
from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, SubmitField, PasswordField, SelectFi\
eld, IntegerField, ValidationError
from wtforms.validators import InputRequired, NumberRange, Length
```

Here we import our Replit database, which we'll need for uniqueness validations, as well as everything we'll be using from WTForms and Flask WTF.

Before we get started with our forms, it's worth thinking about how we're going to lay out the data structures they'll be used to create and modify. In `db_init.py`, we've defined two dictionaries – "challenges" and "teams". Each of these will contain a dictionary for each challenge or team, keyed by an ID. Our data structure will look something like this:

```json
{
    "challenges": {
        "ID": {
            "name": "NAME",
            "description": "DESCRIPTION",
            "points": 10,
            "code": "CHALLENGE SOLUTION CODE"
        }
    },
    "teams": {
        "ID": {
            "name": "NAME",
            "team_leader": "LEADER NAME",
            "team_members": ["LEADER NAME", "ADDITIONAL MEMBER"],
            "score": 0,
            "password": "TEAM PASSWORD",
            "challenges_solved": ["CHALLENGE ID", "ANOTHER CHALLENGE ID"]
        }
    }
}
```

The ID value for both our challenges and teams will be the challenge or team name, all-lowercase, with spaces replaced by hyphens, so we can use it in our app's URLs. Let's create a function that turns names into IDs, in `forms.py`, just below our imports:

```python
def name_to_id(name):
    return name.lower().replace(" ", "-")
```

Now we can start creating our forms. With Flask WTF, we model each form as a class inheriting from `FlaskForm`. These classes take in the value of Flask's `request.form` and apply validations to the fields therein. We'll create our `TeamCreateField` first, with the following code:

```python
class TeamCreateForm(FlaskForm):
    name = StringField(
        "Team name",
        validators=[
            InputRequired(),
            Length(3)
            ]
    )

    password = PasswordField(
        "Team password",
        validators=[
            InputRequired(),
            Length(8)
        ]
    )

    submit = SubmitField("Create team")

    def validate_name(form, field):
        if name_to_id(field.data) in db["teams"].keys():
            raise ValidationError("Team name already taken.")
```

When users create teams, they'll specify a team name and team password. In our WTForms field specifications above, we've defined minimum lengths for both of these fields, ensured that the team password is entered in a password field, and written a custom validator to reject new teams with IDs that match existing teams. Because we're validating on ID rather than name, users won't be able to create teams with the same name but different capitalization (e.g. "Codeslingers" and "codeslingers").

Every field in all of our forms includes the `InputRequired` validator, which will ensure that users do not submit blank values. This validator can be left out for optional fields.

Our `ChallengeCreateForm` is similar to `TeamCreateForm`, and can be added below it:

```python
class ChallengeCreateForm(FlaskForm):
    name = StringField(
        "Challenge name",
        validators=[
            InputRequired(),
            Length(3)
        ]
    )

    description = TextAreaField(
        "Challenge description",
        validators=[InputRequired()]
    )

    points = IntegerField(
        "Challenge points",
        validators=[
            InputRequired(),
            NumberRange(1)
        ]
    )

    code = StringField("Challenge code",
        validators=[
            InputRequired(),
            Length(8)
        ]
    )

    submit = SubmitField("Create challenge")

    def validate_name(form, field):
        if name_to_id(field.data) in db["challenges"].keys():
            raise ValidationError("Challenge name already used.")
```

Here we've used the `TextAreaField` to give a bit more space for our users to write challenge descriptions, and `IntegerField` to specify the number of points a challenge is worth. We're also requiring that challenges be worth at least 1 point, using the `NumberRange` validator.

Next up is our `TeamJoinForm`:

```
class TeamJoinForm(FlaskForm):
    name = SelectField(
        "Team to join",
        choices= [
            (team_id, team["name"]) for team_id, team in db["teams"].items()
        ],
        validators=[InputRequired()]
    )

    password = PasswordField(
        "Team password",
        validators=[InputRequired()]
    )

    submit = SubmitField("Join team")
```

In this form, we're creating a drop-down box with the names of existing teams. The list comprehension in `choices` constructs a tuple for each team, consisting of the team's ID and name. This way, we can use the ID to identify teams on the backend while displaying the name to the user.

Our last form is `ChallengeSolveForm`, which users will use to submit challenge solutions. Add it to the bottom of `forms.py`:

```
class ChallengeSolveForm(FlaskForm):
    code = StringField("Challenge code",
        validators=[
            InputRequired(),
        ]
    )

    submit = SubmitField("Submit solution code")
```

As we'll be including this form on the individual challenge pages, we don't need to ask the user to specify which challenge they're solving.

Finally, we'll need to import our forms and helper function into `main.py` so we can use them in the rest of our app. Add the following line to the import statements in `main.py`:

```
from forms import TeamCreateForm, TeamJoinForm, ChallengeCreateForm, ChallengeSolveF\
orm, name_to_id
```

Now that we have our form logic, we need to integrate them into both the front-end and back-end of the application. We'll deal with the back-end first.

Building back-end functionality

Back-end functionality is the heart of our application. Below, we'll define our application's routes and build the logic for creating and joining teams, as well as creating and solving challenges.

Team functionality

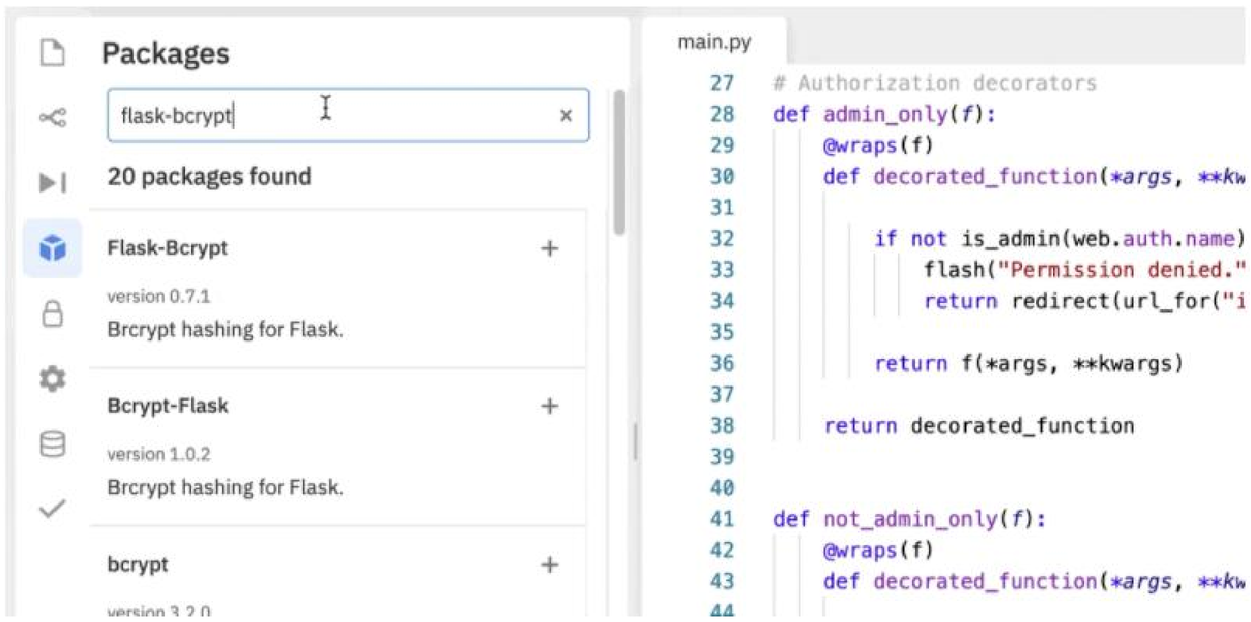
Let's start with teams. We'll define the following routes and functions in `main.py`, below our `index()` function:

```python
# Teams
@app.route("/team-create", methods=["GET", "POST"])
@web.authenticated
@not_admin_only
@not_team_only
def team_create():
    pass

@app.route("/team-join", methods=['GET', 'POST'])
@web.authenticated
@not_admin_only
@not_team_only
def team_join():
    pass

@app.route("/team/<team_id>")
def team(team_id):
    pass
```

The `/team-create` and `/team-join` routes will use their respective forms. Users already in teams and admins will not be permitted to create or join teams. The `/team/<team_id>` page will be an informational page, showing the team's name, score, and which challenges they've solved. We're using part of the URL as a parameter here, so, for example, `/team/codeslingers` will take us to the team page for that team. We won't require authentication for this page.

Because we'll be dealing with passwords, we're going to store them as one-way encrypted hashes[316]. This will prevent anyone with access to our repl's database from easily seeing all team passwords. We'll use Flask's `Bcrypt` extension for this, which you can install by searching for "flask-bcrypt" in the Packages tab on the Replit IDE sidebar.

[316]https://en.wikipedia.org/wiki/Hash_function



Packages

flask-bcrypt

20 packages found

Flask-Bcrypt
version 0.7.1
Brcrypt hashing for Flask.

Bcrypt-Flask
version 1.0.2
Brcrypt hashing for Flask.

bcrypt
version 3.2.0

main.py

```
# Authorization decorators
def admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kw

        if not is_admin(web.auth.name)
            flash("Permission denied."
            return redirect(url_for("i

        return f(*args, **kwargs)

    return decorated_function


def not_admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kw
```

Flask bcrypt package

click to open gif[317]

While Replit usually automatically installs packages based on our import statements, this one must be manually installed, as its package name is slightly different on Pypi and on disk. Once it's installed, we import it with the following additional line at the top of `main.py`:

```
from flask_bcrypt import Bcrypt
```

Then we initialize a `Bcrypt` object for our app by adding the following line just below `app = Flask(__name__)`:

```
bcrypt = Bcrypt(app)
```

Now let's add some code to our `team_create` function:

[317] https://docs.replit.com/images/tutorials/28-technical-challenge-site/bcrypt-package.gif

```python
@app.route("/team-create", methods=["GET", "POST"])
@web.authenticated
@not_admin_only
@not_team_only
def team_create():

    form = TeamCreateForm(request.form)

    if request.method == "POST" and form.validate():
        team_name = form.name.data
        team_id = name_to_id(team_name)

        hashed_password = bcrypt.generate_password_hash(form.password.data).decode("\
utf-8")
        team_leader = web.auth.name

        # Construct team dictionary
        db["teams"][team_id] = {
            "name": team_name,
            "password": hashed_password,
            "leader": team_leader,
            "members": [team_leader],
            "score": 0,
            "challenges_solved": []
        }

        # Set user team
        users.current["team"] = team_id

        flash("Team created!")
        return redirect(url_for('team', team_id=team_id))

    return render_template("team-create.html",
        form = form,
        **context())
```

First, we create an instance of `TeamCreateForm` using the values in `request.form`. We then check whether the current request is an HTTP `POST`, and we call `validate()` on the form. Behind the scenes, this method will run all of our field validators, and return error messages to the user for fields that fail validation. It will only return `True` once all fields validate.

Once we know we've got valid form input, we can save its data to our database. We construct our team's ID using the helper function from `forms.py`, hash our team password, and then define our team's dictionary.

After that, we set the current user's team in our user database and redirect the user to their new team's page. We use `users.current` as an alias for `users[web.auth.name]`.

At the bottom of the function, we render our `team-create` page and tell it which form to use. This will happen regardless of whether the initiating request was a `GET` or a `POST`. We'll create the template and define the `context` function when we build the front-end.

Now we can add the code for joining a team, in the `team_join` function:

```python
@app.route("/team-join", methods=['GET', 'POST'])
@web.authenticated
@not_admin_only
@not_team_only
def team_join():

    form = TeamJoinForm(request.form)

    if request.method == "POST" and form.validate():
        team_id = form.name.data
        team_name = db["teams"][team_id]["name"]

        if bcrypt.check_password_hash(
                db["teams"][team_id]["password"],
                form.password.data
            ):
            db["teams"][team_id]["members"].append(web.auth.name)
            users.current["team"] = team_id

            flash(f"You joined {team_name}!")
            return redirect(url_for('team', team_id=team_id))
        else:
            flash(f"Wrong password for {team_name}!")
            return redirect(url_for("index"))

    return render_template("team-join.html",
        form = form,
        **context())
```

If our form validates, we check the provided team password, and if it's correct, we add the current user to the team and send them to the team page. If it's incorrect, we redirect them to the home page.

Finally, we can define our `/team/<team_id>` route, by adding this code to the `team()` function:

```python
@app.route("/team/<team_id>")
def team(team_id):
    return render_template("team.html",
        team_id = team_id,
        **context())
```

Admin functionality

We're going to let admin users add challenges to the front-end so that we can keep our code generic and re-use it for multiple competitions, if we wish. We'll add the other admin functionality we need at the same time.

We'll start with the challenge creation route. Add this code below your team routes:

```python
# Admin functions
@app.route("/admin/challenge-create", methods=["GET", "POST"])
@web.authenticated
@admin_only
def admin_challenge_create():

    form = ChallengeCreateForm(request.form)

    if request.method == "POST" and form.validate():
        challenge_name = form.name.data
        challenge_id = name_to_id(challenge_name)
        hashed_code = bcrypt.generate_password_hash(form.code.data).decode("utf-8")

        # Construct challenge dictionary
        db["challenges"][challenge_id] = {
            "name": challenge_name,
            "description": form.description.data,
            "points": int(form.points.data),
            "code": hashed_code
        }

        flash("Challenge created!")
        return redirect(url_for('challenge', challenge_id=challenge_id))

    return render_template("admin/challenge-create.html",
        form = form,
        **context())
```

This code is almost identical to our team creation functionality. While hashing challenge codes may not be strictly necessary, it will prevent any users with access to our repl from cheating by viewing the database.

Challenge removal is a bit simpler:

```python
@app.route("/admin/challenge-remove/<challenge_id>")
@web.authenticated
@admin_only
def admin_remove_challenge(challenge_id):

    # Remove challenge from team solutions
    for _, team in db["teams"].items():
        if challenge_id in team["challenges_solved"]:
            team["challenges_solved"].remove(challenge_id)
            team["score"] -= db["challenges"][challenge_id]["points"]

    # Delete challenge dictionary
    del db["challenges"][challenge_id]

    flash("Challenge removed!")
    return redirect(url_for('index'))
```

We'll allow admins to start and stop the competition with two routes that toggle a value in our database:

```python
@app.route("/admin/competition-start")
@web.authenticated
@admin_only
def admin_start_competition():
    db["competition_started"] = True

    flash("Competition started!")
    return redirect(url_for('index'))

@app.route("/admin/competition-stop")
@web.authenticated
@admin_only
@competition_running
def admin_end_competition():
    db["competition_started"] = False

    flash("Competition ended!")
    return redirect(url_for('index'))
```

Finally, we'll define an admin route that deletes and reinitializes the application's general and user databases. This will be useful for running multiple competitions on the same app, and for debugging!

```
@app.route('/admin/db-flush')
@web.authenticated
@admin_only
def flush_db():
    del db["challenges"]
    del db["teams"]
    del db["competition_started"]

    for _, user in users.items():
        user["team"] = None

    db_init()

    return redirect(url_for("index"))
```

If we add any additional keys or values to either of our databases, we will need to remember to delete them in this function.

Challenge functionality

Finally, we need to add functionality that will allow users to solve challenges and score points. Add the following code below your admin routes:

```
# Challenge functionality
@app.route("/challenge/<challenge_id>", methods=["GET", "POST"])
@web.authenticated
@competition_running
def challenge(challenge_id):

    form = ChallengeSolveForm(request.form)

    if request.method == "POST" and form.validate():

        if bcrypt.check_password_hash(
                db["challenges"][challenge_id]["code"],
                form.code.data
            ):
            db["teams"][users.current["team"]]["challenges_solved"].append(challenge\
_id)
```

```
            db["teams"][users.current["team"]]["score"] += db["challenges"][challeng\
e_id]["points"]
            flash("Challenge solved!")
        else:
            flash("Wrong challenge code!")

    return render_template("challenge.html",
        form = form,
        challenge_id = challenge_id,
        **context())
```

This function is very similar to `team_join()`. The main difference is that we will be hosting this form on the challenge description page, so we can fetch the `challenge_id` from the URL rather than asking the user which challenge they're submitting a code for in the form.

Building the web application front-end

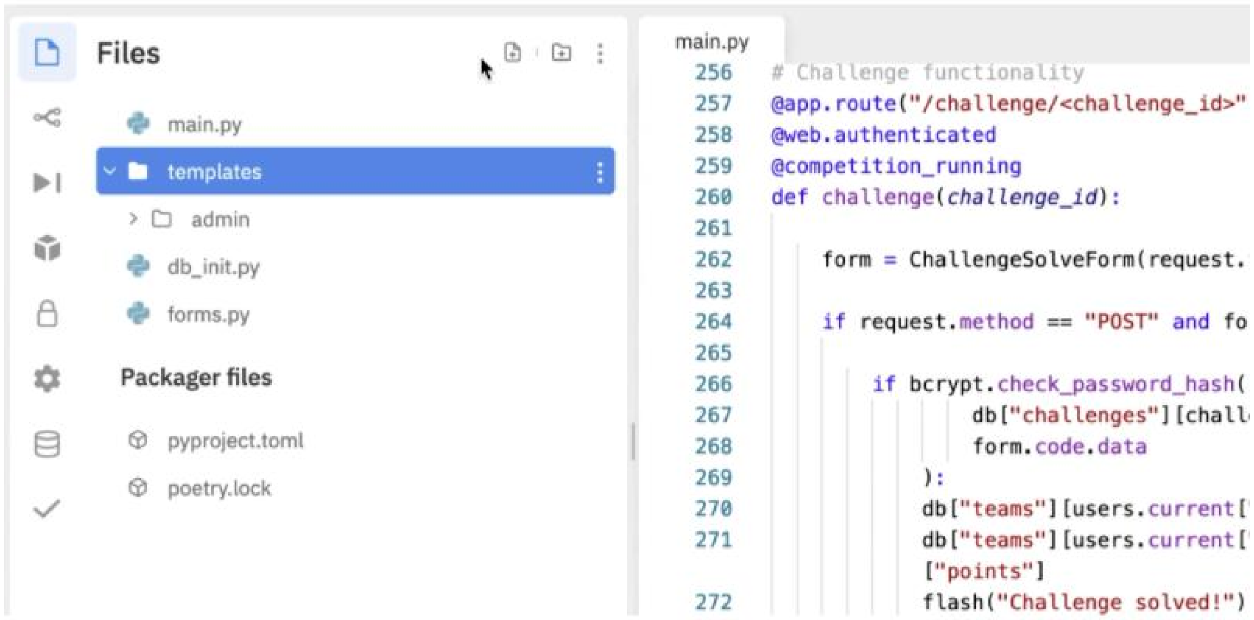
We have a fully functional application back-end, but without some front-end pages, our users will have to join teams and submit challenge solutions using `curl`[318]. So let's create an interface for our back-end using HTML and Jinja[319], Flask's powerful front-end templating language.

Creating the HTML templates

First, we'll need the following HTML files in a new directory called `templates`:

```
templates/
    |__ admin/
    |      |__  challenge-create.html
    |__  _macros.html
    |__  challenge.html
    |__  index.html
    |__  layout.html
    |__  leaderboard.html
    |__  team-create.html
    |__  team-join.html
    |__  team.html
```

[318]https://curl.se/
[319]https://jinja.palletsprojects.com/en/3.0.x/templates/



```
# Challenge functionality
@app.route("/challenge/<challenge_id>"
@web.authenticated
@competition_running
def challenge(challenge_id):

    form = ChallengeSolveForm(request.

    if request.method == "POST" and fo

        if bcrypt.check_password_hash(
                db["challenges"][chall
                form.code.data
            ):
            db["teams"][users.current['
            db["teams"][users.current['
            ["points"]
            flash("Challenge solved!")
```

HTML templates

Once you've created these files, let's populate them, starting with `templates/layout.html`:

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Challenge Leaderboard</title>
    </head>
    <body>
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <ul class=flashes>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}

    {% if name != None %}
    <p>Logged in as {{ username }}</p>
    {% endif %}

    <ul>
        <li><a href="/">View challenges</a></li>
        <li><a href="/leaderboard">View leaderboard</a></li>
```

```
            <li><a href="/team-create">Create team</a></li>
            <li><a href="/team-join">Join team</a></li>
        </ul>


        {% block body %}{% endblock %}
        </body>
</html>
```

We'll use this file as the base of all our pages, so we don't need to repeat the same HTML. It contains features we want on every page, such as flashed messages, an indication of who's currently logged in, and a global navigation menu. All subsequent pages will inject content into the `body` `block`[320]:

~~~
Next, we need to populate another helper file, `templates/_macros.html`:

```html
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
~~~

This file defines the Jinja macro[321] `render_field`, which we'll use to give all our form fields their own error-handling, provided by WTForms.

Let's define our home page now, with a list of challenges. Add the following code to `templates/index.html`:

[320]https://jinja.palletsprojects.com/en/3.0.x/templates/#child-template
[321]https://jinja.palletsprojects.com/en/3.0.x/templates/#macros

```
{% extends "layout.html" %}
{% block body %}
    <h1>Challenges</h1>
    <ul>
    {% for id, challenge in challenges.items()|sort(attribute='1.points') %}
        <li>
            <a href="/challenge/{{ id }}">{{ challenge.name }}</a> ({{ challenge.poi\
nts }} points)
            {% if admin %}
            | <a href="/admin/challenge-remove/{{ id }}">DELETE</a>
            {% endif %}
        <li>
    {% endfor %}
    {% if admin %}
        <li><a href="/admin/challenge-create">NEW CHALLENGE...</a></li>
    {% endif %}
    </ul>

    {% if admin %}
        <h1>Admin functions</h1>
        <ul>
            {% if competition_running %}
                <li><a href="/admin/competition-stop">End competition</a></li>
            {% else %}
                <li><a href="/admin/competition-start">Start competition</a></li>
            {% endif %}
            <li><a href="/admin/db-flush">Flush database</a></li>
        </ul>
    {% endif %}
{% endblock %}
```

Here, `{% extends "layout.html" %}` tells our templating engine to use `layout.html` as a base template, and `{% block body %} ... {% endblock %}` defines the code to place inside `layout.html`'s body block.

The following line will sort challenges in ascending order of points:

```
    {% for id, challenge in challenges.items()|sort(attribute='1.points') %}
```

In addition, we use `{% if admin %}` blocks to include links to admin functionality that will only display when an admin is logged in.

Next we define our team pages:

`templates/team-create.html`

```
{% extends "layout.html" %}
{% block body %}
    {% from "_macros.html" import render_field %}
    <h1>Create team</h1>
    <form action="/team-create" method="post" enctype="multipart/form-data">
        {{ render_field(form.name) }}
        {{ render_field(form.password) }}
        {{ form.csrf_token }}

        {{ form.submit }}
    </form>
{% endblock %}
```

templates/team-join.html

```
{% extends "layout.html" %}
{% block body %}
    {% from "_macros.html" import render_field %}
    <h1>Join team</h1>
    <form action="/team-join" method="post">
        {{ render_field(form.name) }}
        {{ render_field(form.password) }}
        {{ form.csrf_token }}

        {{ form.submit }}
    </form>
{% endblock %}
```

templates/team.html

```
{% extends "layout.html" %}
{% block body %}
    <h1>{{ teams[team_id].name }}</h1>

    <h2>Team members</h2>
    <ul>
    {% for user in teams[team_id].members %}
        <li>{{ user }}</li>
    {% endfor %}
    </ul>

    <h2>Challenges solved</h2>
    <ul>
    {% for id in teams[team_id].challenges_solved %}
        <li>
            <a href="/challenge/"{{ id }}>{{ challenges[id].name }}</a>
        <li>
    {% endfor %}
    </ul>
{% endblock %}
```

You'll notice that we've imported our `render_function` macro on these pages and used it to show our various form fields. Each form also has a hidden field specified by `{{ form.csrf_token }}`. This is a security feature WTForms provides to prevent cross-site request forgery[322] vulnerabilities.

Now we can create our challenge page:

`templates/challenge.html`

```
{% extends "layout.html" %}
{% block body %}
    {% from "_macros.html" import render_field %}
    <h1>{{ challenges[challenge_id].name }}</h1>

    <p>{{ challenges[challenge_id].description }}</p>

    <p><b>Points: {{ challenges[challenge_id].points }}</b></p>

    {% if user_team != None and challenge_id not in teams[user_team]["challenges_sol\
ved"] %}
    <form action="/challenge/{{challenge_id}}" method="post">
        {{ render_field(form.code) }}
        {{ form.csrf_token }}

        {{ form.submit }}
    </form>
    {% endif %}
{% endblock %}
```

Then our challenge creation page (inside the `templates/admin` directory):

`templates/admin/challenge-create.html`

[322] https://owasp.org/www-community/attacks/csrf

```
{% extends "layout.html" %}
{% block body %}
    {% from "_macros.html" import render_field %}
    <h1>Create challenge</h1>
    <form action="/admin/challenge-create" method="post" enctype="multipart/form-dat\
a">
        {{ render_field(form.name) }}
        {{ render_field(form.description) }}
        {{ render_field(form.points) }}
        {{ render_field(form.code) }}
        {{ form.csrf_token }}

        {{ form.submit }}
    </form>
{% endblock %}
```

We've referred to a lot of different variables in our front-end templates. Flask's Jinja templating framework allows us to pass the variables we need into `render_template()`, as we did when building the application backend. Most pages needed a form, and some pages, such as `challenge` and `team`, needed a challenge or team ID. In addition, we unpack[323] the return value of a function named `context` to all of our rendered pages. Define this function now with our other helper functions in `main.py`, just below `in_team`:

```python
def context():
    return {
        "username": web.auth.name,
        "user_team": in_team(web.auth.name),
        "admin": is_admin(web.auth.name),
        "teams": db["teams"],
        "challenges": db["challenges"],
        "competition_running": db["competition_started"]
    }
```

This will give every page most of the application's state. If we find we need another piece of state later, we can add it to the `context` helper function, and it will be available to all our pages.

Importantly, we're using a function rather than a static dictionary so that we can get the most up-to-date application state every time we serve a page.

Before we move on, we should change our app's home page from the initial demo version we made at the beginning of this tutorial to a proper page. Find the `index()` function and replace it with this code:

[323]https://realpython.com/python-kwargs-and-args/#unpacking-with-the-asterisk-operators

```
@app.route("/")
def index():
    return render_template("index.html",
        **context())
```

You'll notice we've removed the `@web.authenticated` decorator. This will allow unauthenticated users to get a glimpse of our site before being asked to log in. `replit.web` will prompt them to log in as soon as they attempt to access an authenticated page.

Building the leaderboard

We've left out a key part of our application: the leaderboard showing which team is winning! Let's add the leaderboard frontend now, with the following HTML and Jinja code in `templates/leaderboard.html`:

```
{% extends "layout.html" %}
{% block body %}
    <h1>Leaderboard</h1>
    <ul>
    {% for id, team in teams.items()|sort(attribute='1.score', reverse=True) %}
        <li {% if id == user_team %}style="font-weight: bold"{% endif %}>
            <a href="/team/{{ id }}">{{ team.name }}</a>: {{ team.score }} points
        <li>
    {% endfor %}
    </ul>
{% endblock %}
```

Similar to the list of challenges on our home page, we use Jinja's sort[324] filter to order the teams from highest to lowest score.

```
    {% for id, team in teams.items()|sort(attribute='1.score', reverse=True) %}
```

We also use an `if` block to show the name of the current user's team in bold.

Finally, we can add one last route to `main.py`, just above the line `web.run(app)`:

```
@app.route("/leaderboard")
def leaderboard():
    return render_template("leaderboard.html",
        **context())
```

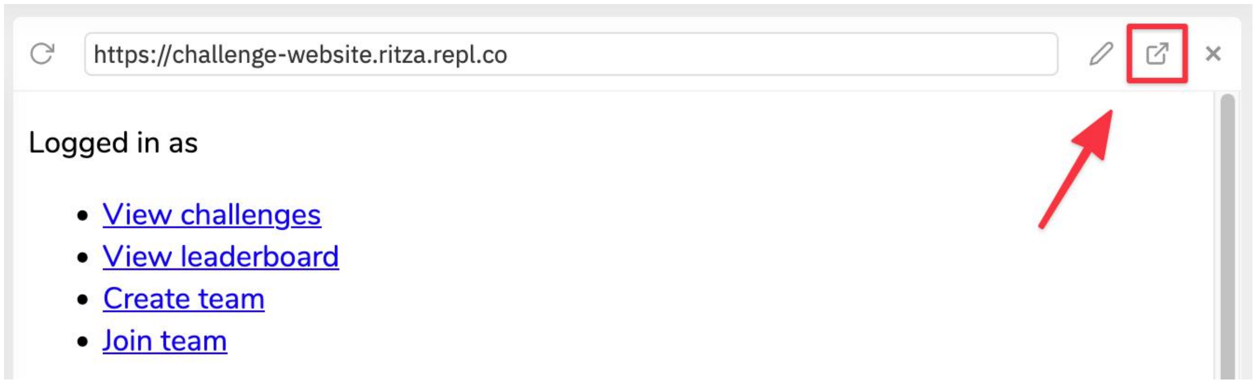
We're leaving this one unauthenticated as well, so that spectators can see how the competition's going.

[324] https://jinja.palletsprojects.com/en/3.0.x/templates/#jinja-filters.sort

Using the app

We're done! Run your repl now to see your app in action. As your user account will be a site admin, you may need to enlist a couple of friends to test out all the app's functionality.

For best results, open your repl's web page in a new tab.



Open in new tab button

If you run into unexplained errors, you may need to clear your browser cookies, or flush the database.

Where next?

We've built a CRUD[325] application with a fair amount of functionality, but there's still room for improvement. Some things you might want to add include:

- CSS styling.
- More admin functionality, such as adjusting scores, banning users and teams, and setting team size limitations.
- File upload, for challenge files and/or team avatars.
- Time-limited competitions, with a countdown.
- Badges/achievements for things like being the first team to solve a given challenge.
- A place for teams to submit challenge solution write-ups.

And of course, you can also use your site to host a competition right now.

Next Tutorial

Up next, we're going to take a look at how to build a paid content site with `replit.web` and Stripe for checkout. When complete, the site will serve as an e-commerce store that can be used to sell a variety of goods online. Users of the store will also be able to keep a record of their past purchases on the site.

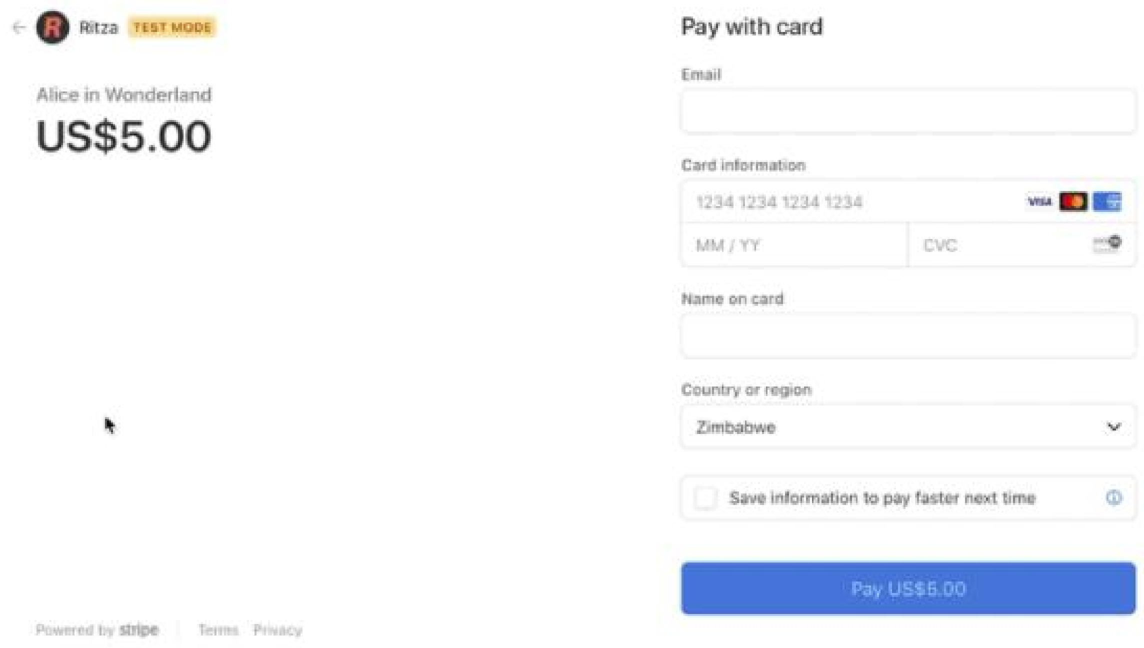
[325] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

Build a paid content site with `replit.web` and Stripe

In this tutorial, we'll combine `replit.web` and Stripe to build a digital content storefront. Anyone with a Replit account will be able to log into our website and purchase premium PDFs. Our site will also keep track of what each user has purchased, so they can build up a library.

By the end of this tutorial, you'll be able to:

- Build a dynamic web application with `replit.web`.
- Use Stripe[326] to sell digital content.



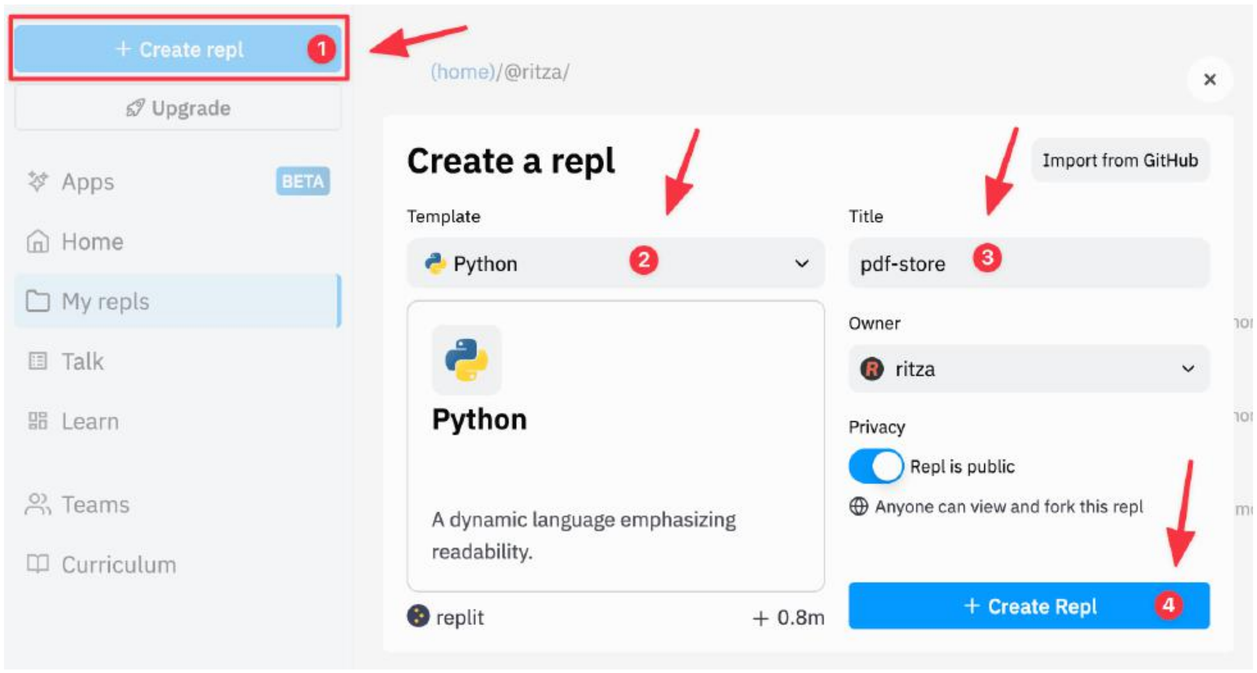
Paid content site functionality

click to open gif[327]

[326] https://stripe.com/
[327] https://docs.replit.com/images/tutorials/29-paid-content-site/site-functionality.gif

Getting started

To get started, create a Python repl.



Create python repl

Our application will have the following functionality:

- Users can log in with their Replit accounts.
- Users can purchase PDFs.
- Users can view free PDFs and PDFs that they've previously purchased.
- Administrators can upload new PDFs.

We've covered both `replit.web` and Stripe in previous tutorials, so some aspects of the following may be familiar if you've built a brick shop[328] or a technical challenge website[329].

We'll start our app off with the following import statements in `main.py`:

[328] https://docs.replit.com/tutorials/20-online-store-checkout-process
[329] https://docs.replit.com/tutorials/28-technical-challenge-site

```python
import os, shutil
import stripe
from flask import Flask, render_template, render_template_string, flash, redirect, u\
rl_for, request, jsonify
from flask.helpers import send_from_directory
from werkzeug.utils import secure_filename
from replit import db, web
from functools import wraps
```

Here we're importing most of what we'll need for our application:

1. Python's `os` and `shutil` packages, which provide useful functions for working with files and directories.
2. Stripe's Python library.
3. Flask, our web framework and the heart of the application.
4. A Flask helper function `send_from_directory`, which will allow us to send PDFs to users.
5. A function `secure_filename` from the Werkzeug WSGI (which Flask is built on) that we'll use when admins upload PDFs and other files.
6. Replit's web framework and Replit DB integration, which we'll use for user authentication and persistent data storage.
7. The `wraps` tool from Python's `functools`, which we'll use to make authorization decorators for restricting access to sensitive application functionality.

Now that the imports are out of the way, let's start on our application scaffold. Add the following code to `main.py`:

```python
app = Flask(__name__,
            static_folder='static',
            static_url_path='')
```

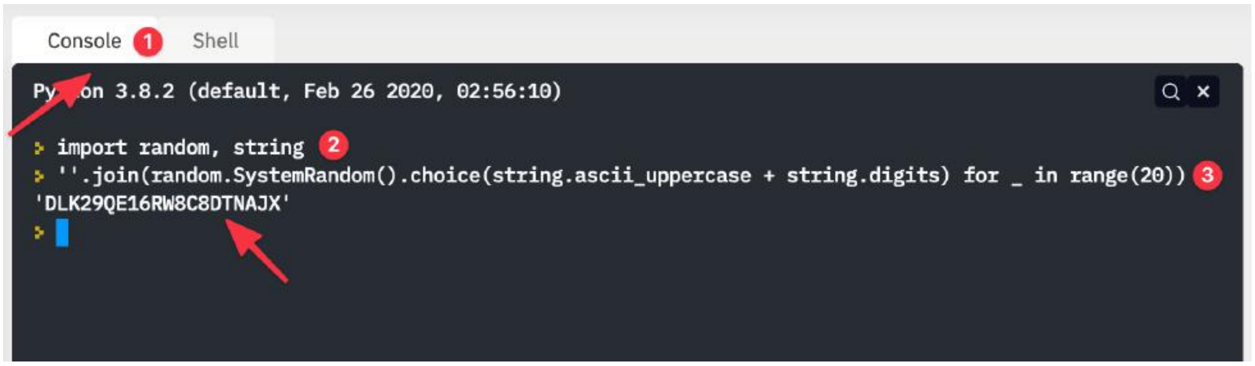
This code initializes our Flask application. We've added a `static_folder` and `static_url_path` so that we can serve static files directly from our repl's file pane without writing routing code for each file. This will be useful for things like images and stylesheets.

Add the following code to initialize your application's secret key:

```python
# Secret key
app.config["SECRET_KEY"] = os.environ["SECRET_KEY"]
```

Our secret key will be a long, random string. You can generate one in your repl's Python console with the following two lines of code:

```
import random, string
''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ i\
n range(20))
```

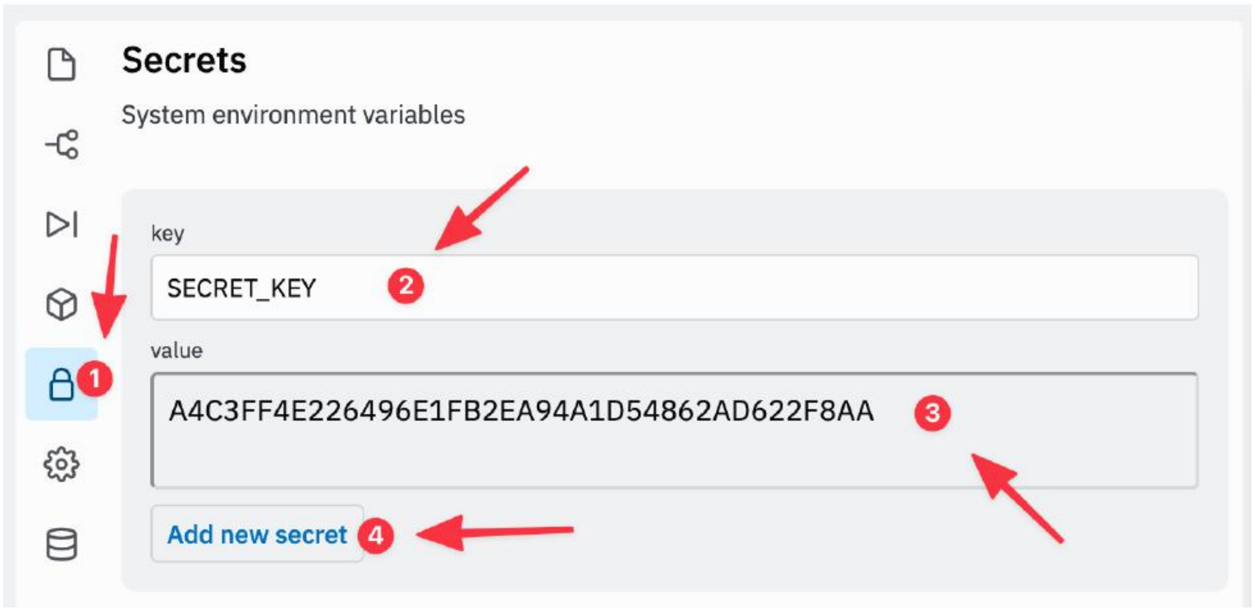


```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> import random, string
> ''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ in range(20))
'DLK29QE16RW8C8DTNAJX'
>
```

Random string

Rather than putting this value directly into our code, we'll retrieve it from an environment variable[330]. This will keep it out of source control and is good practice for sensitive data.

In your repl's Secrets tab, add a new key named `SECRET_KEY` and enter the random string you just generated as its value.



Secrets

System environment variables

key

SECRET_KEY

value

A4C3FF4E226496E1FB2EA94A1D54862AD622F8AA

Repl secrets

Once that's done, return to `main.py` and add the code below to initialize our Replit database:

[330] https://en.wikipedia.org/wiki/Environment_variable

```python
# Database setup
def db_init():
    if "content" not in db.keys():
        db["content"] = {}

    if "orders" not in db.keys():
        db["orders"] = {}

    # Create directories
    if not os.path.exists("static"):
        os.mkdir("static")

    if not os.path.exists("content"):
        os.mkdir("content")

db_init()
```

Replit's Database[331] can be thought of and used as one big Python dictionary that we can access with `db`. Any values we store in `db` will persist between repl restarts.

We've written a function to initialize the database as we may want to do it again if we need to refresh our data during testing. Whenever we initialize our database, we will also create the `content` and `static` directories, which will contain user-uploaded files.

Next we need to create our UserStore[332] (a secondary database keyed by username), and list of admins:

```python
users = web.UserStore()

ADMINS = ["YOUR-REPLIT-USERNAME-HERE"]
```

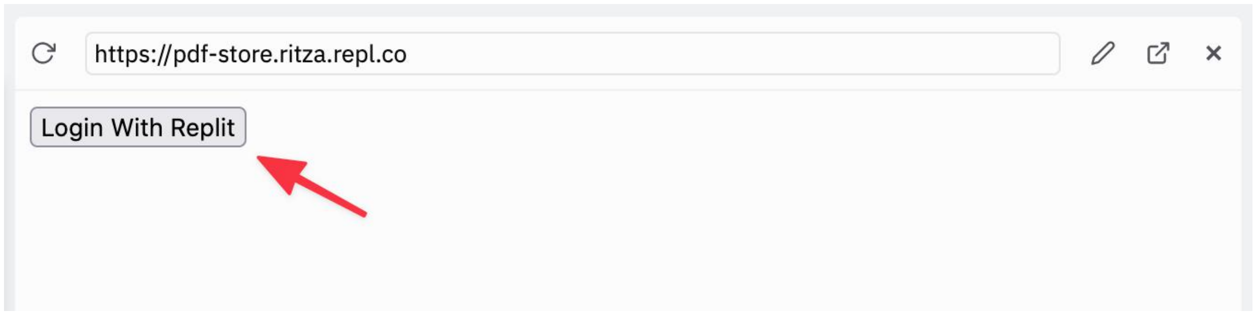
Make sure to replace the contents of the `ADMINS` list with your Replit username.

Finally, let's make our root page. Add the following code, and then run your repl.

[331] https://docs.replit.com/hosting/database-faq
[332] https://replit-py.readthedocs.io/en/latest/api.html

```python
# Main app
@app.route("/")
@web.authenticated
def index():
    return f"Hello {web.auth.name}"

web.run(app)
```

Because we've added the `@web.authenticated` function decorator[333] to our index page, it will only be available to logged-in users. You should see this now, as your app will show a login button. Click on that button, and authorize your application to use Replit authentication in the window that pops up.



Login button

Having done that, you should now see the greeting we implemented above. If you send your repl to a friend, they will also be able to log in and see their Replit username on the greeting message.

Content upload and other admin functionality

Before we do anything else with our site, we need to have some PDFs to sell. While we could manually upload our PDFs to our repl and write code to add each one to the database, it will make our site more user-friendly if we include an upload form for this purpose.

This upload form should only be accessible by admins, so we can enforce some level of quality control. We'll also create a route that allows admins to refresh the application database.

Access control

Add the following functions to `main.py`, just below the line where you've assigned `ADMINS`:

[333] https://realpython.com/primer-on-python-decorators/

```python
# Helper functions
def is_admin(username):
    return username in ADMINS

# Auth decorators
def admin_only(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if not is_admin(web.auth.name):
            flash("Permission denied.", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

The code in the second function may look a bit strange if you haven't written your own decorators before. Here's how it works: `admin_only` is the name of our decorator. You can think of decorators as functions that take other functions as arguments. (The two code snippets below are for illustration and not part of our program.) Therefore, if we write the following:

```python
@admin_only
def admin_function():
    return f"Hello admin"

admin_function()
```

it will be roughly equivalent to:

```python
def admin_function():
    return f"Hello admin"

admin_only(admin_function)
```

So whenever `admin_function` gets called, the code we've defined in `decorated_function` will execute before anything we define in `admin_function`. This means we don't have to include an `if not is_admin` check in every piece of admin functionality. As per the code, if a non-admin attempts to access restricted functionality, our app will flash[334] a warning message and redirect them to the home page.

Now we can create the following admin routes below the definition of the `index` function:

[334]https://flask.palletsprojects.com/en/2.0.x/patterns/flashing/

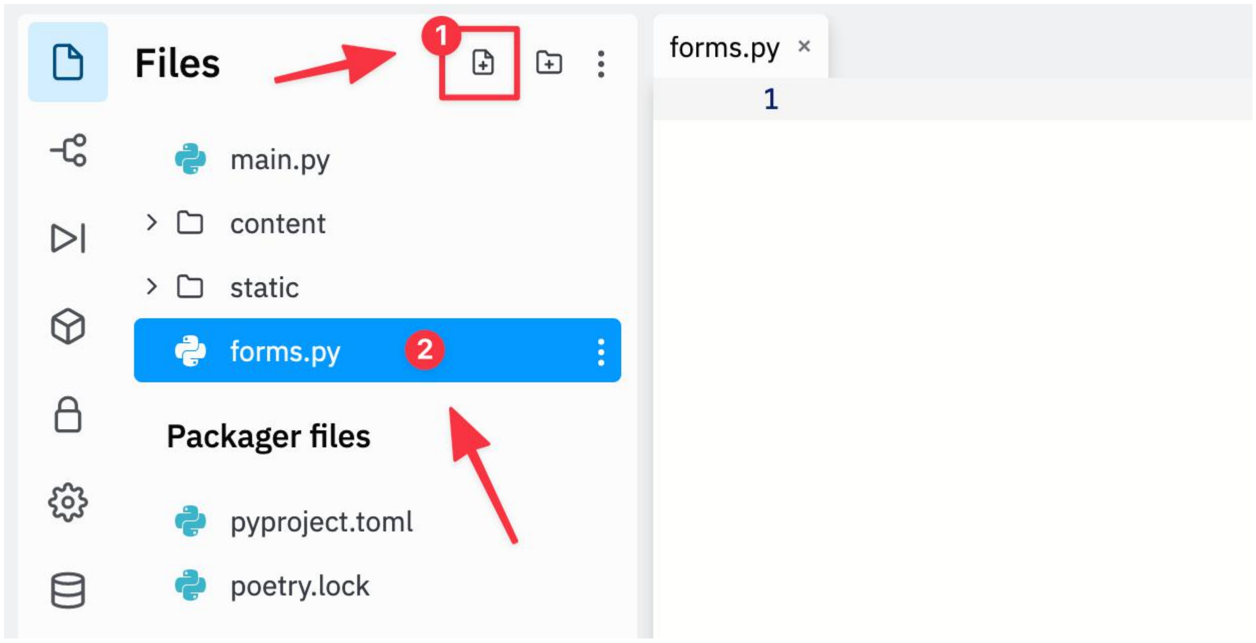
```python
# Admin functionality
@app.route('/admin/content-create', methods=["GET", "POST"])
@web.authenticated
@admin_only
def content_create():
    pass

@app.route('/admin/db-flush')
@web.authenticated
@admin_only
def flush_db():
    pass
```

Note that both of these functions are protected with the `@web.authenticated` and `@admin_only` decorators, restricting their use to logged-in admins.

The first function will let our admins create content, and the second will allow us to flush the database. While the second function will be useful during development, it's not something we'd want to use in a finished application, as our database will contain records of user payments.

Content creation form

Before we can fill in the code for content creation, we need to create the web form our admins will use. As the form creation code will include a lot of information and functionality and require several special imports, we're going to put it in its own file so we can keep a navigable codebase. In your repl's files pane, create `forms.py`.



Create forms.py file

Enter the following `import` statements at the top of `forms.py`:

```python
from flask_wtf import FlaskForm
from flask_wtf.file import FileField, FileRequired, FileAllowed
from wtforms import StringField, TextAreaField, SubmitField, FloatField, ValidationE\
rror
from wtforms.validators import InputRequired, NumberRange, Length
from replit import db
```

Here we're importing from WTForms[335], an extensive library for building web forms, and Flask WTF[336], a library which bridges WTForms and Flask. We're also importing our Replit database, which we'll need for uniqueness validations.

The structure of our forms is dictated by the structure of our database. In our `db_init` function, we defined two dictionaries, "content" and "orders". The former will contain entries for each of the PDFs we have for sale. These entries will contain the PDF's filename as well as general metadata. Thus, our "content" data structure will look something like this:

[335]https://wtforms.readthedocs.io/en/2.3.x/
[336]https://flask-wtf.readthedocs.io/en/0.15.x/

```
{
    "content": {
        "ID": {
            "name": "NAME",
            "description": "DESCRIPTION",
            "file": "PDF_FILENAME",
            "preview_image": "IMAGE_FILENAME",
            "price": 5,
        }
    }
}
```

The ID value will be the content's name, all-lowercase, with spaces replaced by hyphens, so we can use it in our app's URLs. Let's create a function that turns names into IDs, in `forms.py`, just below our imports:

```
def name_to_id(name):
    return name.lower().replace(" ", "-")
```

Now we can create our form. With Flask WTF, we model a form as a class inheriting from FlaskForm. This class takes in the value of Flask's `request.form` and applies validations to the fields therein. Add the following class definition to the bottom of `forms.py`:

```
class ContentCreateForm(FlaskForm):
    name = StringField(
        "Title",
        validators=[
            InputRequired(),
            Length(3)
            ]
    )

    description = TextAreaField(
        "Description",
        validators=[InputRequired()]
    )

    file = FileField(
        "PDF file",
        validators=[
            FileRequired(),
            FileAllowed(['pdf'], "PDFs only.")
        ]
    )

    image = FileField(
        "Preview image",
        validators=[
            FileRequired(),
            FileAllowed(['jpg', 'jpeg', 'png', 'svg'], "Images only.")
        ]
    )

    price = FloatField(
        "Price in USD (0 = free)",
        validators=[
            InputRequired(),
            NumberRange(0)
        ]
    )

    submit = SubmitField("Create content")

    def validate_name(form, field):
        if name_to_id(field.data) in db["content"].keys():
            raise ValidationError("Content name already taken.")
```

When admins create content, they'll specify a name, a description, and a price, as well as upload both the PDF and a preview image. We've used WTForm's validators to restrict the file types that can be uploaded for each. Should we decide to branch out from selling PDFs in the future, we can add additional file extensions to the `file` field's `FileAllowed` validator. We could also make individual fields optional by removing their `InputRequired()` or `FileRequired()` validators.

The final part of our form is a custom validator to reject new PDFs with IDs that match existing PDFs. Because we're validating on ID rather than name, admins won't be able to create PDFs with the same name but different capitalization (e.g. "Sherlock Holmes" and "SHERLOCK HOLMES").

We've finished creating our form class. Now we can return to `main.py` and import the class with the following `import` statement, which you can add just below the other imports at the top of the file.

```
from forms import name_to_id, ContentCreateForm
```

Note that we've also imported `name_to_id`, which we'll use when populating the database.

Admin routes

We can now use our form to implement our content creation route. Populate the `content_create` function with this code:

```python
# Admin functionality
@app.route('/admin/content-create', methods=["GET", "POST"])
@web.authenticated
@admin_only
def content_create():

    form = ContentCreateForm()

    if request.method == "POST" and form.validate():
        content_name = form.name.data
        content_id = name_to_id(content_name)
        content_price = form.price.data

        content_file = form.file.data
        content_filename = secure_filename(content_file.filename)
        content_file.save(os.path.join('content', content_filename))

        image_file = form.image.data
        image_filename = secure_filename(image_file.filename)
        image_file.save(os.path.join('static', image_filename))

        content_paywalled = content_price > 0

        # Construct content dictionary
        db["content"][content_id] = {
            "name": content_name,
            "description": form.description.data,
            "filename": content_filename,
            "preview_image": image_filename,
            "paywalled": content_paywalled,
            "price": content_price,
        }

        flash("Content created!")
        return redirect(url_for('content', content_id=content_id))

    return render_template("admin/content-create.html",
        form = form,
        **context())
```

First, we create an instance of ContentCreateForm. This will automatically use the values in request.form, including the uploaded files. We then check whether the current request is an HTTP POST, and we call validate() on the form. Behind the scenes, this method will run all of our field validators, and return error messages to the user for fields that fail validation. It will only return True once all fields validate.

Once we know we've got valid form input, we can save its data to our database. We construct our content's ID using the helper function from forms.py, store our content's price value, and then save our PDF and image files to the content and static directories. Saving images to static will allow Flask to serve them without us writing additional code. We'll need custom code for PDFs, however, as we need to ensure they're only accessible to paying customers.

We use the variable content_paywalled to determine whether this PDF should be available for free or behind a paywall.

Finally, we save our content's details to the database and redirect the creator to the content page, which we'll build in the next section.

At the bottom of the function, we render our content-create page and tell it which form to use. This will happen regardless of whether the initiating request was a GET or a POST. We'll create the template and define the context function when we build the application front-end.

Next, we need to create our database flushing functionality. Populate the flush_db function with the following code:

```python
@app.route('/admin/db-flush')
@web.authenticated
@admin_only
def flush_db():
    # clear db
    del db["content"]
    del db["orders"]

    # clear users
    for _, user in users.items():
        user["content_library"] = []

    # delete content and images
    shutil.rmtree("content")
    shutil.rmtree("static")

    # reinit
    db_init()

    return redirect(url_for("index"))
```

After deleting all database content and uploaded files, we call `db_init()` to start afresh. Keep in mind that this function should not be used if you're storing real user data unless you've made a backup.

Content viewing and paywalls

Now that our site admins can upload PDFs, we need a way for users to view them. We'll start by creating another helper function, just below the definition of `is_admin`:

```
def owns_content(username, content_id):
    if "content_library" in users[username].keys() and users[username]["content_libr\
ary"] is not None:
        return content_id in users[username]["content_library"]
```

We have to do several checks on our user's `content_library`, as it can be in a few different states – the key might not exist, or it might be set to None, or it might be a list. We'll use this function to determine which content has been purchased by a given user and thus avoid writing all these checks again.

Now we need to create our application's content-viewing routes. We'll start by rewriting the / route so that it renders a template rather than a greeting string. This page will contain a list of PDFs. Change the code in `index` to the following:

```
# Main app
@app.route("/")
@web.authenticated
def index():
    return render_template("index.html",
        **context())
```

Then we'll write a route that displays individual PDF metadata, by adding this function just below the definition of `index`:

```
@app.route("/content/<content_id>")
@web.authenticated
def content(content_id):
    return render_template("content.html",
        content_id=content_id,
        **context())
```

The content_id value will be the same ID that we're using in our database. This page will contain the content's name, preview image, description, and either a download link, or a purchase link, depending on whether the PDF is paywalled, and whether the current user has purchased it.

Lastly, we need a route that handles downloading actual PDFs. Add the following code just below the content function definition:

```
@app.route("/content-file/<content_id>")
@web.authenticated
def content_file(content_id):

    content = db["content"][content_id]

    if not content["paywalled"] or owns_content(web.auth.name, content_id):
        return send_from_directory("content", path=content["filename"])
    else:
        return "Access denied."
```

If the current user owns this PDF, or it's not paywalled, we use Flask's send_from_directory[337] to allow them to download it. Otherwise, we return an error message.

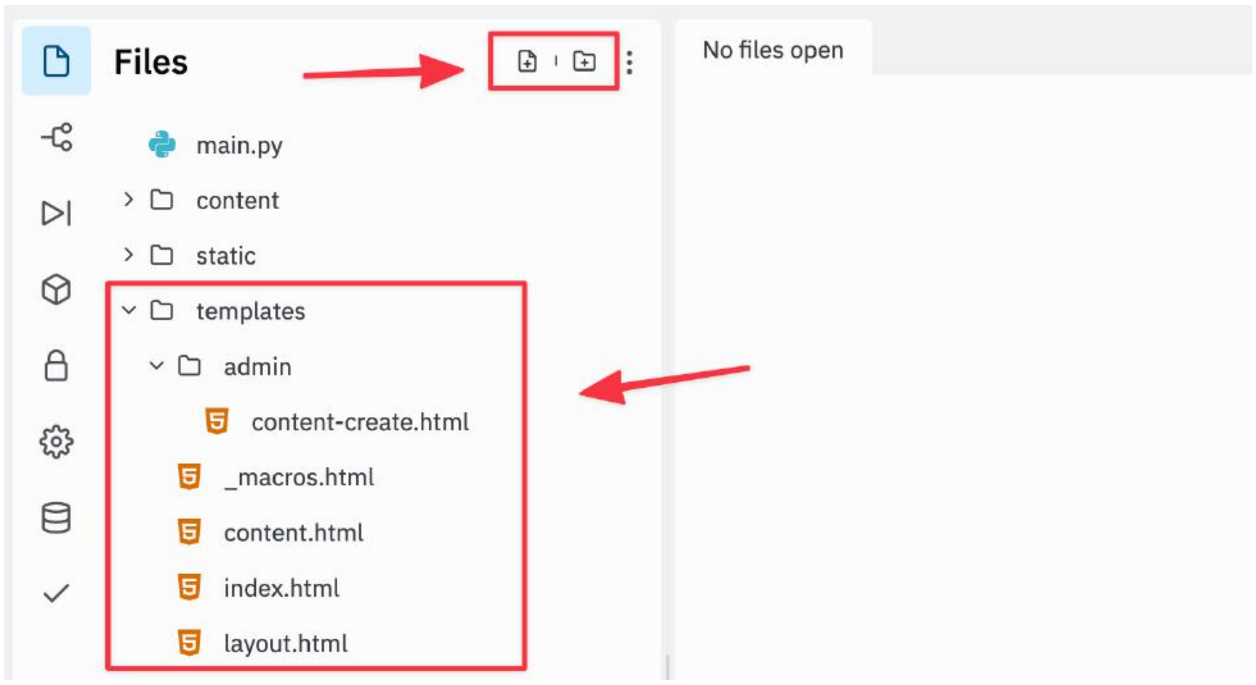
Creating the application frontend

We have most of our application back-end, so now let's create the front-end. We'll do this using HTML and Jinja[338], Flask's front-end templating language.

First, let's create the following HTML files in a new directory called templates:

[337]https://flask.palletsprojects.com/en/2.0.x/api/#flask.send_from_directory
[338]https://jinja.palletsprojects.com/en/3.0.x/templates/

```
templates/
    |__ admin/
    |     |__  content-create.html
    |__  _macros.html
    |__  content.html
    |__  index.html
    |__  layout.html
```



Folder structure

Once you've created these files, let's populate them, starting with `templates/layout.html`:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Books and Manuscripts</title>
    </head>
    <body>
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <ul class=flashes>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}

    {% if name != None %}
    <p>Logged in as {{ username }}</p>
    {% endif %}

    {% block body %}{% endblock %}
    </body>
</html>
```

We'll use this file as the base of all our pages, so we don't need to repeat the same HTML. It contains features we want on every page, such as flashed messages, and an indication of who's currently logged in. All subsequent pages will inject content into the body block:

```
{% block body %}{% endblock %}
```

Next, we need to populate another helper file, `templates/_macros.html`:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
```

This file defines the Jinja macro[339] `render_field`, which we'll use to provide our form fields with error-handling, provided by WTForms.

We'll use this macro in `templates/admin/content-create.html`, which we'll populate with the following code:

[339]https://jinja.palletsprojects.com/en/3.0.x/templates/#macros

```
{% extends "layout.html" %}
{% block body %}
    {% from "_macros.html" import render_field %}
    <h1>Upload content item</h1>
    <form action="/admin/content-create" method="post" enctype="multipart/form-data">
        {{ render_field(form.name) }}
        {{ render_field(form.description) }}
        {{ render_field(form.file) }}
        {{ render_field(form.image) }}
        {{ render_field(form.price) }}
        {{ form.csrf_token }}

        {{ form.submit }}
    </form>
{% endblock %}
```

Here, `{% extends "layout.html" %}` tells our templating engine to use layout.html as a base template, and `{% block body %} ... {% endblock %}` defines the code to place inside layout.html's body block.

Our `render_function` macro will be used to show our different form fields – some of these will be text input fields, while others will be file upload fields. Our form also has a hidden field specified by `{{ form.csrf_token }}`. This is a security feature WTForms provides to prevent cross-site request forgery[340] vulnerabilities.

Let's define our home page now, with a list of content items. Add the following code to `templates/index.html`:

```
{% extends "layout.html" %}
{% block body %}
    <h1>Marketplace</h1>
    <ul>
    {% for id, content in content.items() %}
        <li>
            <a href="/content/{{ id }}">{{ content.name }}</a>
            {% if content.paywalled %}
                {% if id in my_library %}
                    (PURCHASED)
                {% else %}
                    ({{ "${:,.2f}".format(content.price) }})
                {% endif %}
            {% endif %}
        </li>
    {% endfor %}
```

[340] https://owasp.org/www-community/attacks/csrf

```
    {% if admin %}
        <li><a href="/admin/content-create">NEW CONTENT...</a></li>
    {% endif %}
    </ul>

    {% if admin %}
        <h1>Admin functions</h1>
        <ul>
            <li><a href="/admin/db-flush">Flush database</a></li>
        </ul>
    {% endif %}
{% endblock %}
```

We display each piece of content in a list. If an item is paywalled, we show its price if the current user hasn't already purchased it, or "(PURCHASED)" if they have.

In addition, we use `{% if admin %}` blocks to include links to admin functionality, such as content creation and database flushing, that will only display when an admin is logged in.

The last page we need to create is `templates/content.html`, which will display information about individual PDFs:

```
{% extends "layout.html" %}
{% block body %}
<h1>{{ content[content_id].name }}</h1>
<img src='/{{ content[content_id].preview_image }}' style='max-width: 150px'>
<p>{{ content[content_id].description }}</p>
{% if content_id in my_library or not content[content_id].paywalled %}
    <a href="/content-file/{{ content_id }}">Download PDF</a>
{% else %}
    <form action="/checkout/{{ content_id }}" method="POST">
        <button type="submit" id="checkout-button">Buy {{ content[content_id].name }\
} for {{ "${:,.2f}".format(content[content_id].price) }}</button>
    </form>
{% endif %}
{% endblock %}
```

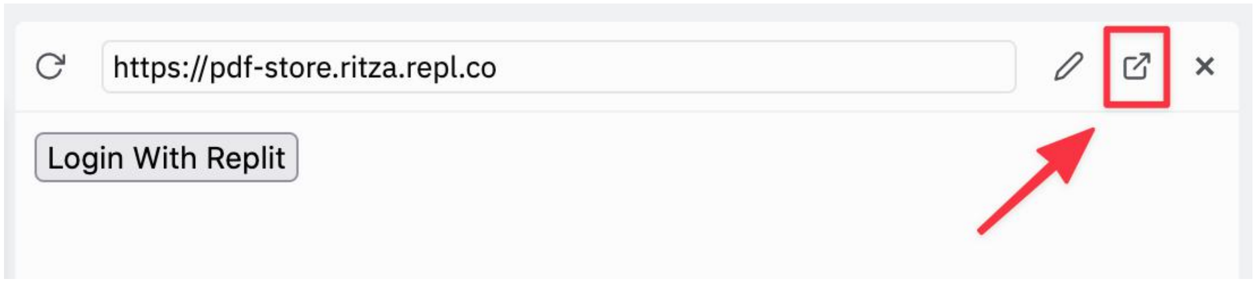
As with the home page, we display different parts of the page depending on whether the content is paywalled, and whether the current user owns it. If the user must purchase the PDF, we include a single-button form that posts to `/checkout/<content_id>`, an application route we'll create in the next section.

We've referred to a lot of different variables in our front-end templates. Flask's Jinja templating framework allows us to pass the variables we need into `render_template`, as we did when building the application backend. Our content creation page needed a form, and our content viewing pages needed an ID. In addition, we unpack the return value of a function named `context` to all of our rendered pages. Define this function now with our other helper functions in `main.py`, just below `owns_content`:

```
def context():
    if "content_library" in users.current.keys() and users.current["content_library"\
] is not None:
        my_library = users.current["content_library"]
    else:
        my_library = []

    return {
        "username": web.auth.name,
        "my_library": my_library,
        "admin": is_admin(web.auth.name),
        "content": db["content"]
    }
```

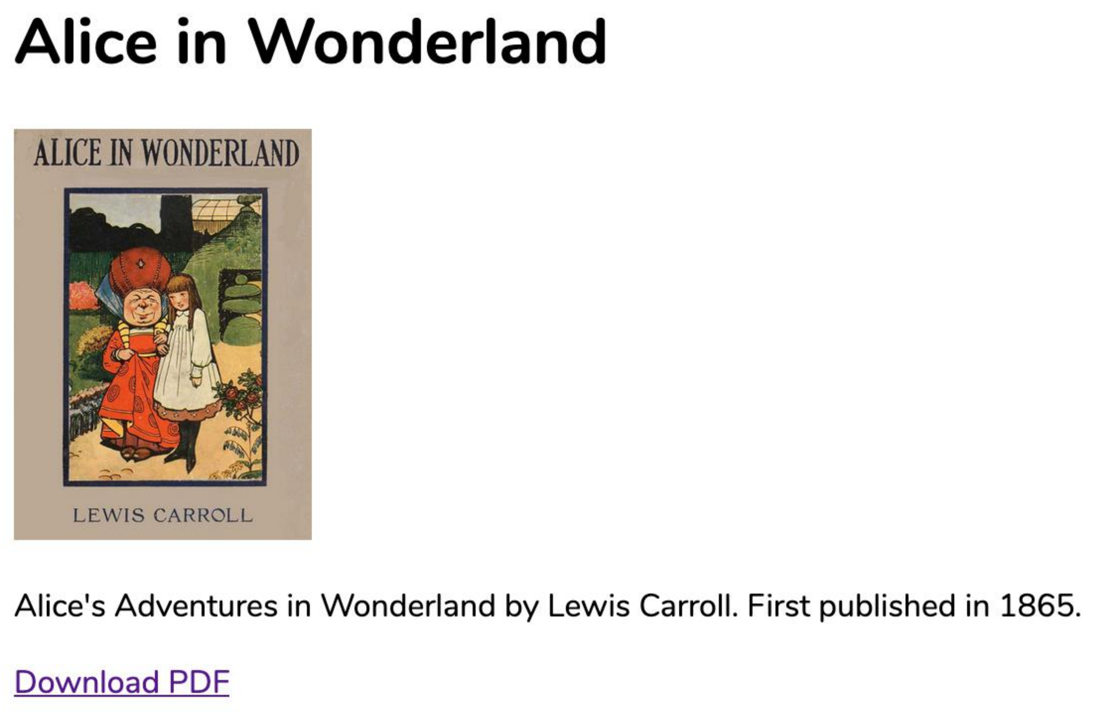
This will give every page most of the application's state, including the full content dictionary and the current user's library. If we find we need another piece of state later, we can add it to the `context` helper function, and it will be available to all our pages.

Run your repl now and add some content. For best results, open the site in a new tab, rather than using it in your repl's browser.



Open in new window

If you add free PDFs, you'll be able to download them, but you won't be able to purchase paywalled PDFs yet.


Alice in Wonderland

Alice's Adventures in Wonderland by Lewis Carroll. First published in 1865.

Download PDF

Free pdf download

Integrating with Stripe

Our application is fully functional for free PDFs. To have users pay for premium PDFs, we'll integrate Stripe Checkout[341]. This will save us the trouble and risk of developing our own payment gateway or storing users' card details.

To use Stripe Checkout, you will need an activated Stripe account. Create one now at https://stripe.com[342] if you haven't already.

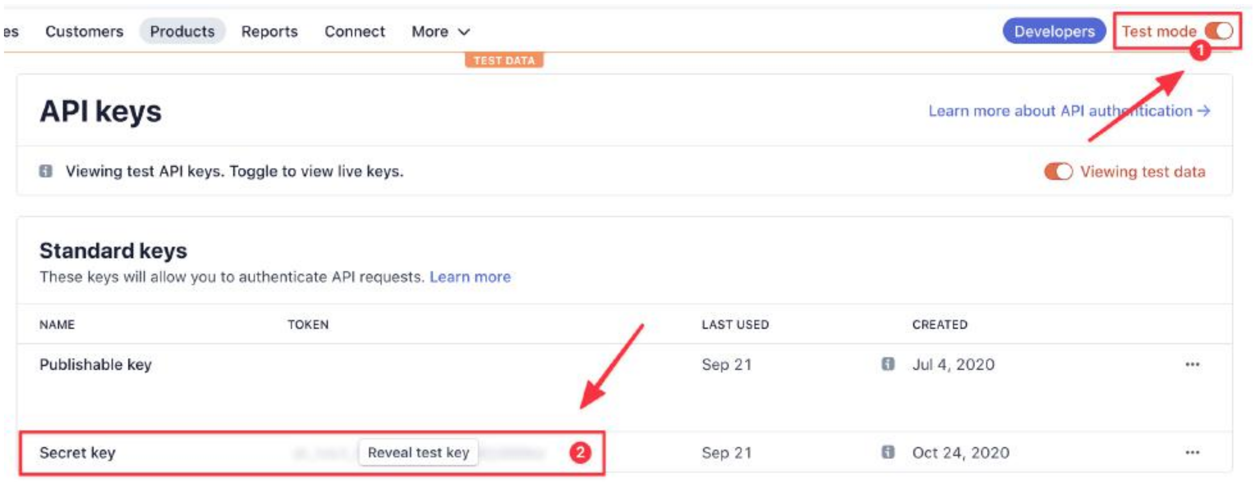
Once you've created a Stripe account, add the following code near the top of `main.py`, just below the `import` statements:

```python
# Stripe setup
stripe.api_key = os.environ["STRIPE_KEY"]

DOMAIN = "YOUR-REPL-URL-HERE"
```

You can find your Stripe API keys on this page of the developer dashboard[343]. Make sure that you're

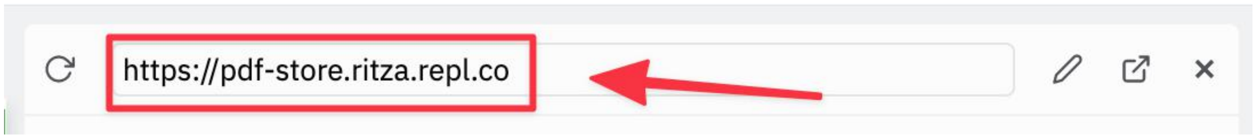
[341] https://stripe.com/en-gb-us/payments/checkout
[342] https://stripe.com/
[343] https://dashboard.stripe.com/test/apikeys

in test mode and copy the secret key to your clipboard. Then return to your repl and create an environment variable called `STRIPE_KEY` with the value you just copied from Stripe.



Stripe Key

You will also need to replace the value of `DOMAIN` with your repl's root URL. You can get this URL from the in-repl browser.



Repl URL

Stripe Checkout

Stripe provides detailed technical documentation and code snippets in a variety of languages, so setting up basic integration is largely a matter of copying and adapting these code snippets to our needs. We'll start by creating the `/checkout/<content_id>` route. This will create a new Stripe checkout session[344] and redirect the user to a Stripe payment page. Add the following code below your `content_file` function definition:

[344] https://stripe.com/docs/api/checkout/sessions

```python
# Stripe integration
@app.route("/checkout/<content_id>", methods=["POST"])
@web.authenticated
def checkout(content_id):

    # Proceed to checkout
    try:
        checkout_session = stripe.checkout.Session.create(
            line_items=[
                {
                    "price_data": {
                        "currency": "usd",
                        "product_data": {
                            "name": db["content"][content_id]["name"],
                            "images": [DOMAIN + "/" + db["content"][content_id]["pre\
view_image"]]
                        },
                        'unit_amount': int(db["content"][content_id]["price"]*100),
                    },
                    "quantity": 1
                },
            ],
            payment_method_types=[
              'card',
            ],
            mode='payment',
            success_url=DOMAIN + '/success?session_id={CHECKOUT_SESSION_ID}',
            cancel_url=DOMAIN + '/cancel'
        )
    except Exception as e:
        return str(e)

    # Record order
    order_id = checkout_session.id
    db["orders"][order_id] = {
        "content_id": content_id,
        "buyer": web.auth.name
    }

    return redirect(checkout_session.url, code=303)
```

This code is adapted from Stripe's sample integration Python code[345]. It initiates a checkout from the

[345]https://stripe.com/docs/checkout/integration-builder?server=python

pricing and product details we provide and redirects the user to Stripe's checkout website to pay. If payment is successful, it sends the user to a success_url on our site; otherwise, it sends to the user to a cancel_url. We'll define both of these shortly.

We've made two key changes to the sample code. First, we've included the details for our content item in line_items:

```
line_items=[
    {
        "price_data": {
            "currency": "usd",
            "product_data": {
                "name": db["content"][content_id]["name"],
                "images": [DOMAIN + "/" + db["content"][content_id]["preview_image"]]
            },
            'unit_amount': int(db["content"][content_id]["price"]*100),
        },
        "quantity": 1
    },
],
```

Rather than defining individual products on Stripe's side, we're programmatically constructing our products at checkout time. This saves us from having to add our PDF metadata in two places. We provide our product's name, and the full URL of its preview image, so both can be shown on the Stripe Checkout page. As Stripe expects prices in cents, we multiply the price from our database by 100 before converting it to an integer.

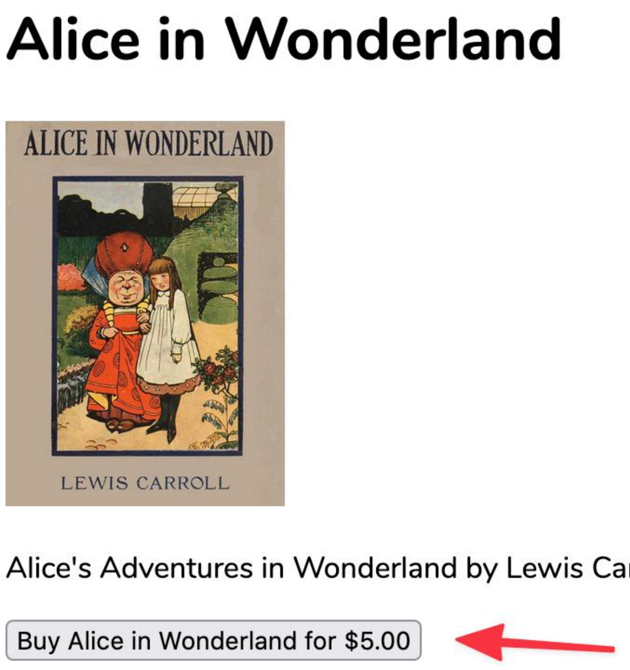
The second change we've made to the sample code is to record the order details in our database. We need to do this so that we can fulfill the order once it's paid for.

```
    # Record order
    order_id = checkout_session.id
    db["orders"][order_id] = {
        "content_id": content_id,
        "buyer": web.auth.name
    }
```

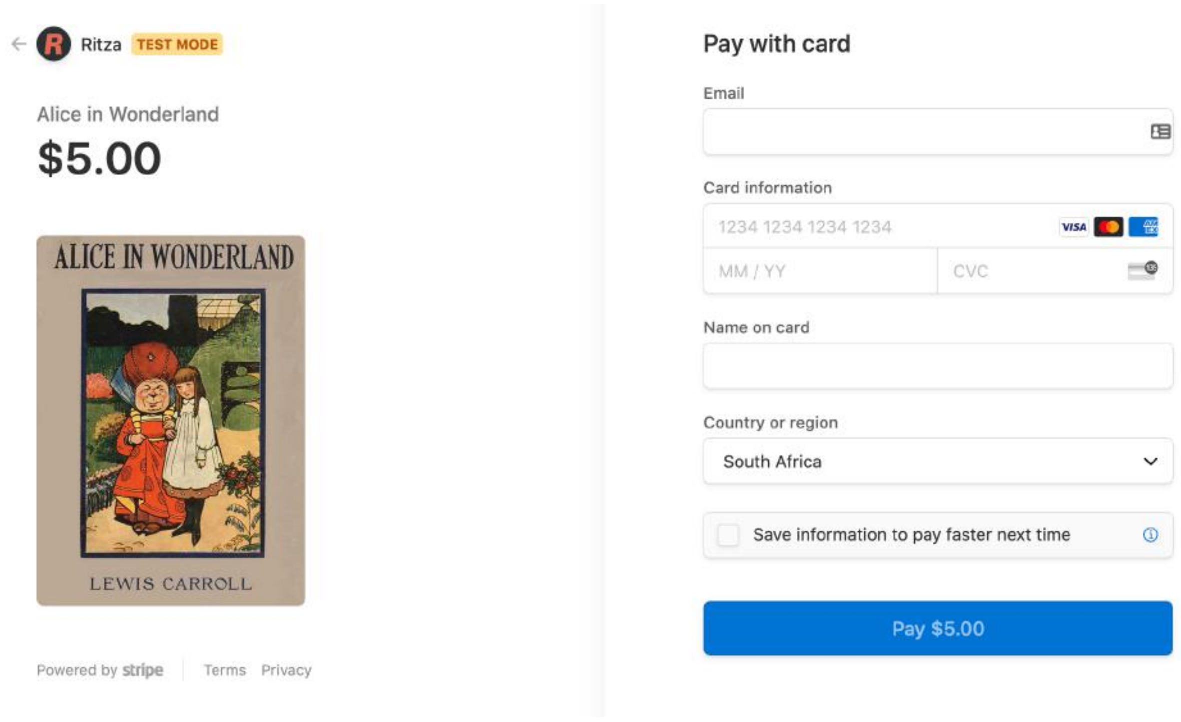
We reuse Stripe's Checkout Session[346] object's id as our order_id so that we can link the two later.

If you run your repl now, you should be able to reach the Stripe checkout page for any paywalled content you've added. Don't try to pay for anything yet though, as we still need to build order fulfillment.

[346]https://stripe.com/docs/api/checkout/sessions/object


Alice in Wonderland
ALICE IN WONDERLAND
LEWIS CARROLL
Alice's Adventures in Wonderland by Lewis Carroll. First published in 1865.
Buy Alice in Wonderland for $5.00

Paywall



Checkout page

Stripe fulfillment

As we're selling digital goods, we can integrate fulfillment directly into our application by adding purchased content to the buyer's library as soon as payment has been made. We'll do this with a function called `fulfill_order`, which you can add just below the `checkout` function definition.

```python
def fulfill_order(session):
    # Get order details
    content_id = db["orders"][session.id]["content_id"]
    buyer = db["orders"][session.id]["buyer"]

    # Add content to library
    if session.payment_status == "paid" and not owns_content(buyer, content_id):
        if users[buyer]["content_library"] is not None:
            users[buyer]["content_library"].append(content_id)
        else:
            users[buyer]["content_library"] = [content_id]
```

This function takes a Stripe Checkout Session object, retrieves the corresponding order from our database, and then adds the order's content to the buyer's library if a payment has been made, and the buyer does not already own the content.

We'll invoke this function from our `/success` route, which we'll define just below it.

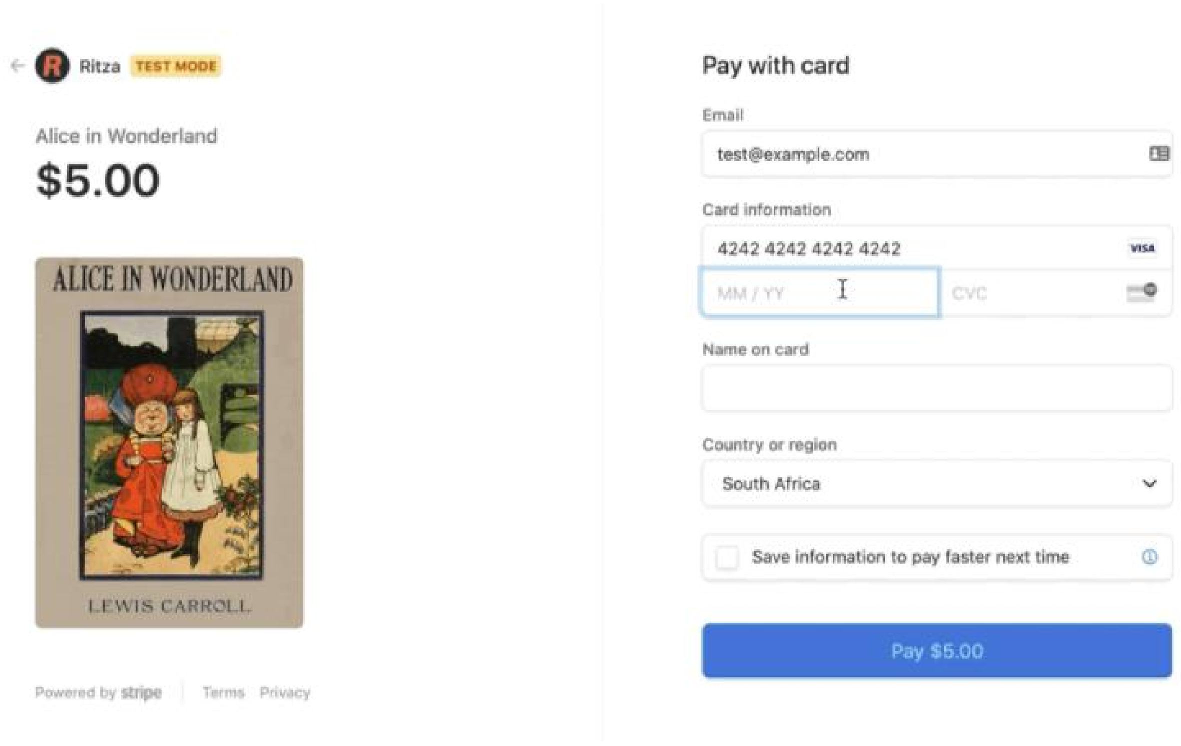
```python
@app.route('/success', methods=['GET'])
@web.authenticated
def success():

    # Get payment info from Stripe
    session = stripe.checkout.Session.retrieve(request.args.get('session_id'))

    # Abort if user is not buyer
    if web.auth.name != db["orders"][session.id]["buyer"]:
        return "Access denied."

    fulfill_order(session)

    return render_template_string(f'<html><body><h1>Thanks for your order, {web.auth\
.name}!</h1><p>Your purchase has been added to your <a href="/">library</a>.</p></bo\
dy></html>')
```

Here we retrieve the session details from the `session_id` GET parameter Stripe passed to our app, ensure that the current user is also the order buyer, and call `fulfill_order`. We then render a simple success page. You can replace this with a full Jinja template if you want to make it a bit fancier.

We also need to define the `/cancel` route, used if the payment fails. This one is quite simple:

```python
@app.route('/cancel', methods=['GET'])
@web.authenticated
def cancel():
    return render_template_string("<html><body><h1>Order canceled.</h1></body></html\
>")
```

If you run your repl now, you should be able to purchase content. You can find test credit card numbers on the Stripe integration testing[347] documentation page. You can use any future date as the expiry date and any CVV.



Email: test@example.com

Card information: 4242 4242 4242 4242

Country or region: South Africa

PDF purchased

click to open gif[348]

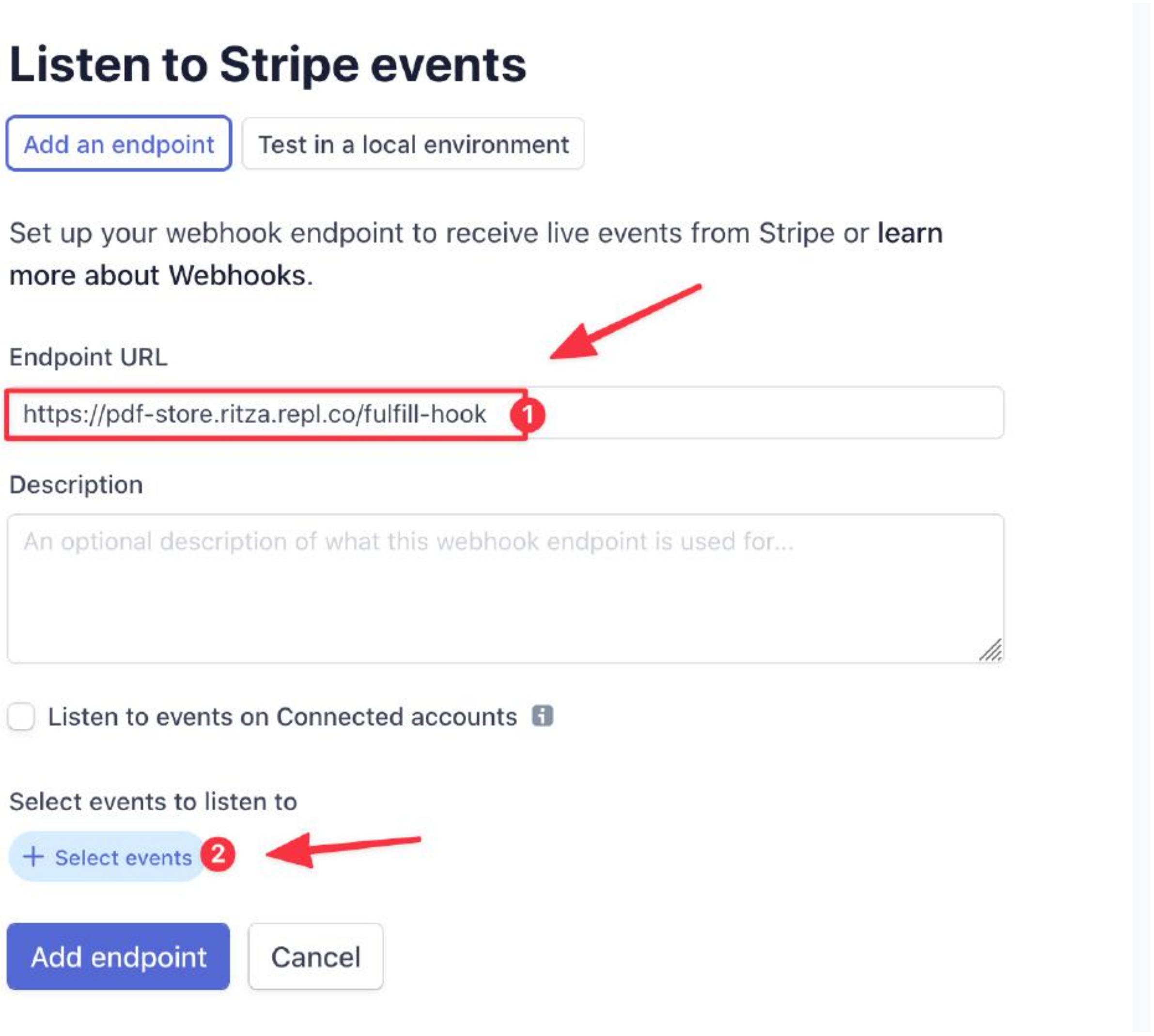
Webhooks

A potential problem with the way we're fulfilling orders is that a user might close the Stripe Checkout tab or lose internet connectivity after their payment has been confirmed, but before they're redirected to our `/success` route. If this happens, we'll have their money, but they won't have their PDF.

[347]https://stripe.com/docs/testing
[348]https://docs.replit.com/images/tutorials/29-paid-content-site/alice-purchased.gif

For this reason, Stripe provides an additional method for fulfilling orders, based on webhooks[349]. A webhook is an HTTP route intended to be used by machines rather than people. Much like we've created routes for our admins to upload PDFs, and our users to buy PDFs, we'll now create a route for Stripe's bots to notify our application of completed payments.

First, you'll need to create a webhook on your Stripe Dashboard. Visit the Webhooks[350] page and click **Add endpoint**. You should then see a page like this:



Listen to Stripe events

Set up your webhook endpoint to receive live events from Stripe or learn more about Webhooks.

Endpoint URL: https://pdf-store.ritza.repl.co/fulfill-hook

Description: An optional description of what this webhook endpoint is used for...

Listen to events on Connected accounts

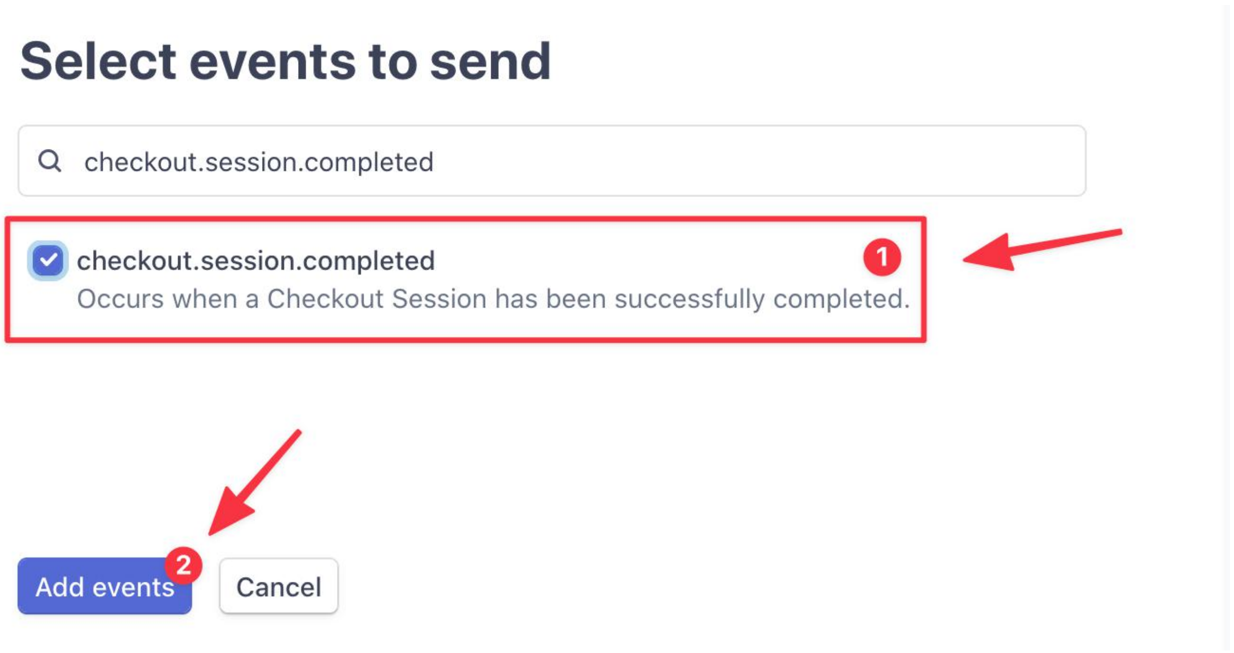
Select events to listen to

Add webhook

On this page, do the following:

1. For the **Endpoint URL** value, enter your repl's URL, followed by `/fulfill-hook`.
2. Select the `checkout.session.completed` event from **Select events to listen to**.

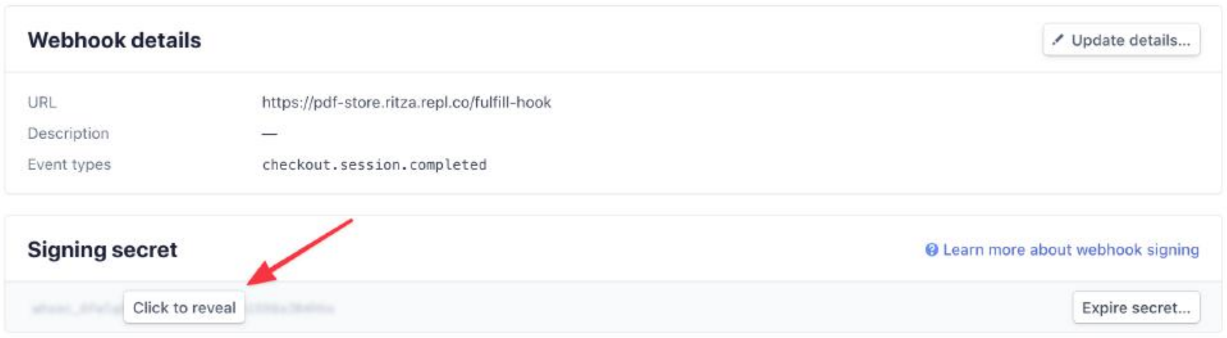
[349]https://en.wikipedia.org/wiki/Webhook
[350]https://dashboard.stripe.com/test/webhooks


Select events to send
checkout.session.completed
checkout.session.completed
Occurs when a Checkout Session has been successfully completed.
1
2
Add events
Cancel

Webhook event

3. Click **Add endpoint**.

Stripe should then redirect you to your new webhook's details page. From here you can see webhook details, logs and the signing secret. The signing secret is used to ensure that our webhook only accepts requests from Stripe – otherwise, anyone could call it with spoofed data and complete orders without paying. Reveal your webhook's signing secret and copy it to your clipboard, then return to your repl.



URL https://pdf-store.ritza.repl.co/fulfill-hook
Description —
Event types checkout.session.completed

Signing secret

We'll use another environment variable here. Add the following code below your `cancel` function definition:

```
endpoint_secret = os.environ['ENDPOINT_SECRET']
```

Then create an environment variable called `ENDPOINT_SECRET` with the value you just copied from Stripe.

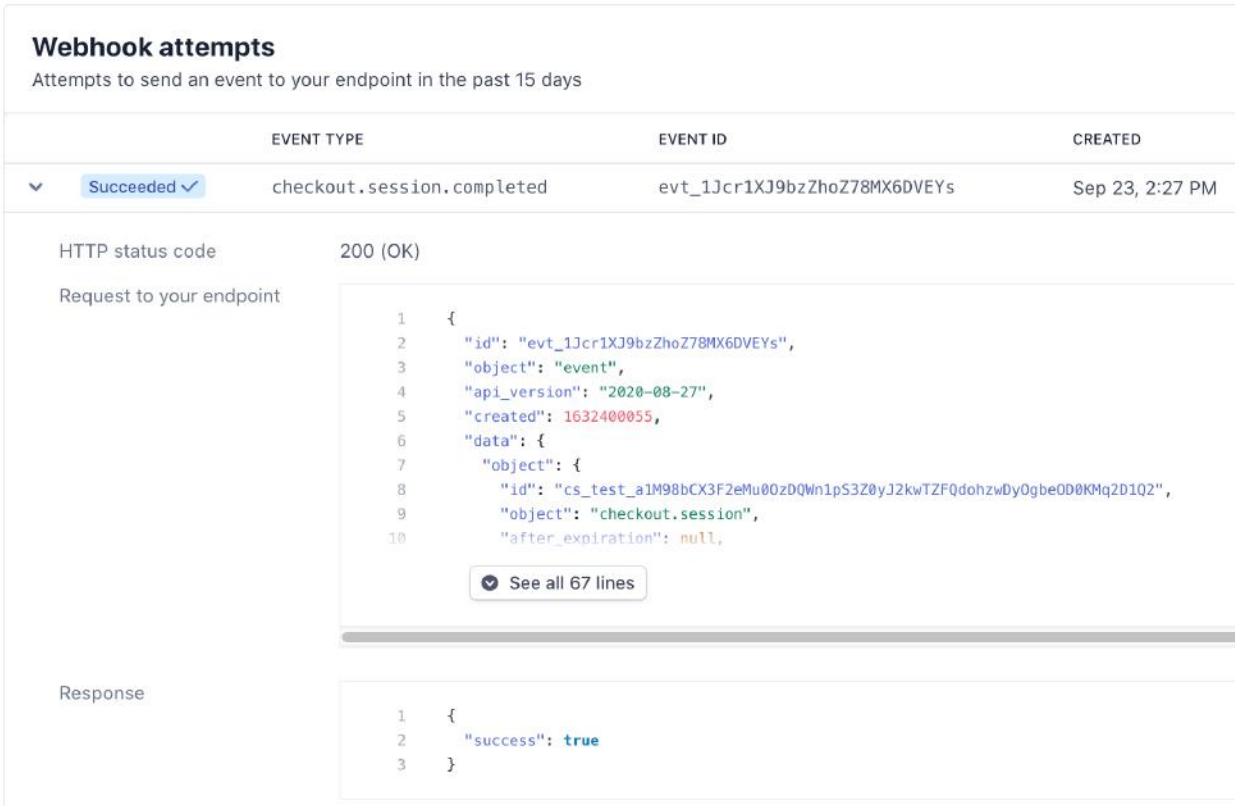
For our app's webhook code, we can once again tweak Stripe's sample code. We'll use this order fulfillment code[351] as a base. Add this code below your `endpoint_secret` assignment:

```python
@app.route('/fulfill-hook', methods=['POST'])
def fulfill_webhook():
    event = None
    payload = request.data
    sig_header = request.headers['STRIPE_SIGNATURE']

    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, endpoint_secret
        )
    except ValueError as e:
        # Invalid payload
        raise e
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
        raise e

    # Handle the event
    if event['type'] == 'checkout.session.completed':
        session = event['data']['object']

        # Fulfill the purchase...
        fulfill_order(session)
    else:
        print('Unhandled event type {}'.format(event['type']))

    return jsonify(success=True)
```

After ensuring that the request we've received comes from Stripe, we retrieve the Checkout Session object from Stripe's `checkout.session.completed` event and use it to call `fulfill_order`.

If you run your repl now, you should be able to purchase a PDF, close the checkout page after your payment is accepted but before being redirected, and still end up with the PDF in your library. You can also view webhook invocation logs on the Stripe Dashboard[352].

[351]https://stripe.com/docs/payments/checkout/fulfill-orders#fulfill
[352]https://dashboard.stripe.com/test/webhooks



Webhook attempts

Attempts to send an event to your endpoint in the past 15 days

	EVENT TYPE	EVENT ID	CREATED
Succeeded ✓	checkout.session.completed	evt_1Jcr1XJ9bzZhoZ78MX6DVEYs	Sep 23, 2:27 PM

HTTP status code 200 (OK)

Request to your endpoint

```
{
  "id": "evt_1Jcr1XJ9bzZhoZ78MX6DVEYs",
  "object": "event",
  "api_version": "2020-08-27",
  "created": 1632400055,
  "data": {
    "object": {
      "id": "cs_test_a1M98bCX3F2eMu0OzDQWn1pS3Z0yJ2kwTZFQdohzwDyOgbeOD0KMq2D1Q2",
      "object": "checkout.session",
      "after_expiration": null,
```

See all 67 lines

Response

```
{
  "success": true
}
```

Stripe webhook success

Where next?

We've built a functional if fairly basic storefront for digital goods. If you'd like to continue with this project, consider the following extensions:

- Improving the site's appearance with custom CSS.
- Branching out from PDFs to other files, such as audio podcasts, videos, or desktop software.
- Providing a subscription option that gives users access to all PDFs for a limited time.
- Converting the site into a peer-to-peer marketplace where users can all upload and purchase files from each other.

Next Tutorial

In the next tutorial, we're going to take a look at how to build an email news digest app with Nix, Python and Celery. To send emails in bulk the app will use the Mailgun API to avoid building an email server on top of the Nix app. By the end of the tutorial, you would have learned how to use Nix on Replit to set up a database, webserver, message broker and background task handlers.

Build an email news digest app with Nix, Python and Celery

In this tutorial, we'll build an application that sends regular emails to its users. Users will be able to subscribe to RSS[353] and Atom[354] feeds, and will receive a daily email with links to the newest stories in each one, at a specified time.

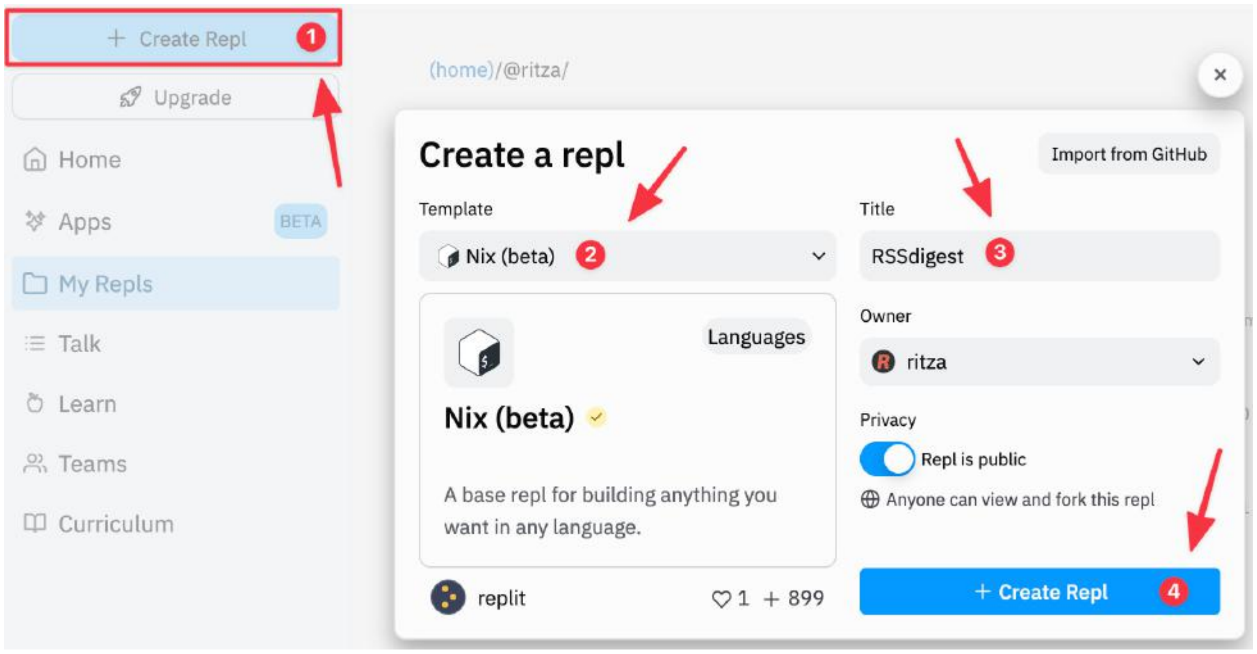
As this application will require a number of different components, we're going to build it using the power of Nix repls. By the end of this tutorial, you'll be able to:

- Use Nix on Replit to set up a database, webserver, message broker and background task handlers.
- Use Python Celery to schedule and run tasks in the background.
- Use Mailgun to send automated emails.
- Build a dynamic Python application with multiple discrete parts.

Getting started

To get started, sign in to Replit[355] or create an account[356] if you haven't already. Once logged in, create a Nix repl.

[353]https://en.wikipedia.org/wiki/RSS
[354]https://en.wikipedia.org/wiki/Atom_(Web_standard)
[355]https://replit.com
[356]https://replit.com/signup



Create a nix repl

Installing dependencies

We'll start by using Nix to install the packages and libraries we'll need to build our application. These are:

1. **Python 3.9**, the programming language we'll write our application in.
2. **Flask**, Python's most popular micro web framework, which we'll use to power our web application.
3. **MongoDB**, the NoSQL database we'll use to store persistent data for our application.
4. **PyMongo**, a library for working with MongoDB in Python.
5. **Celery**, a Python task queuing system. We'll use this to send regular emails to users.
6. **Redis**, a data store and message broker used by Celery to track tasks.
7. Python's Redis library.
8. Python's Requests library, which we'll use to interact with an external API to send emails.
9. Python's feedparser library, which we'll use to parse news feeds.
10. Python's dateutil library, which we'll use to parse timestamps in news feeds.

To install these dependencies, open `replit.nix` and edit it to include the following:

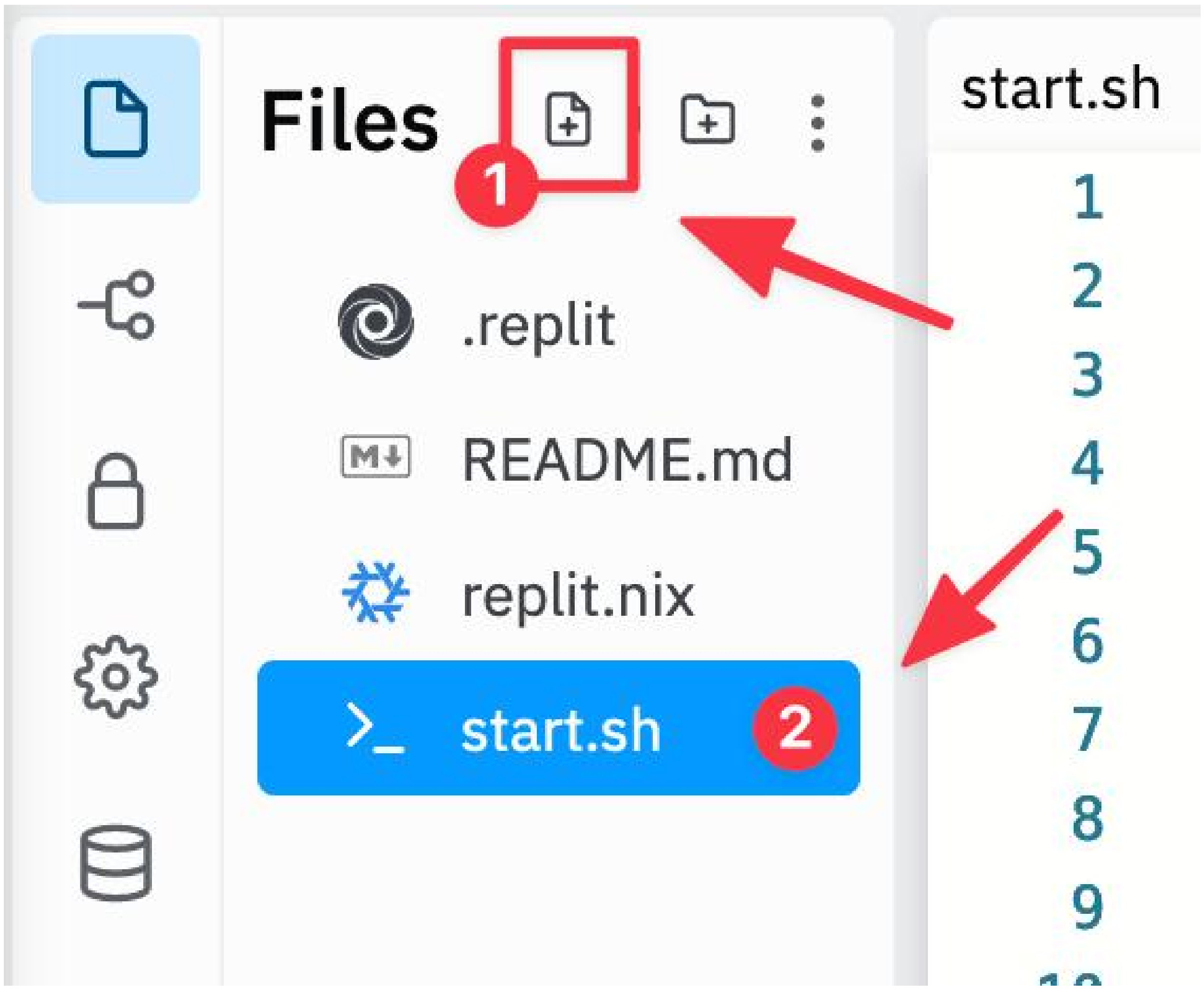
```
{ pkgs }: {
        deps = [
        pkgs.cowsay
        pkgs.python39
        pkgs.python39Packages.flask
        pkgs.mongodb
        pkgs.python39Packages.pymongo
        pkgs.python39Packages.celery
        pkgs.redis
        pkgs.python39Packages.redis
        pkgs.python39Packages.requests
        pkgs.python39Packages.feedparser
        pkgs.python39Packages.dateutil
        ];
}
```

Run your repl now to install all the packages. Once the Nix environment is finished loading, you should see a welcome message from `cowsay`.

Now edit your repl's `.replit` file to run a script called `start.sh`:

```
run = "sh start.sh"
```

Next we need to create `start.sh` in the repl's files tab:



Create Start.sh

And add the following bash code to `start.sh`:

```bash
#!/bin/sh

# Clean up
pkill mongo
pkill redis
pkill python
pkill start.sh
rm data/mongod.lock
mongod --dbpath data --repair

# Run Mongo with local paths
mongod --fork --bind_ip="127.0.0.1" --dbpath=./data --logpath=./log/mongod.log
```

```

# Run redis
redis-server --daemonize yes --bind 127.0.0.1
```

The first section of this script will kill all the running processes so they can be restarted. While it may not be strictly necessary to stop and restart MongoDB or Redis every time you run your repl, doing so means we can reconfigure them should we need to, and prevents us from having to check whether they're stopped or started, independent of our other code.

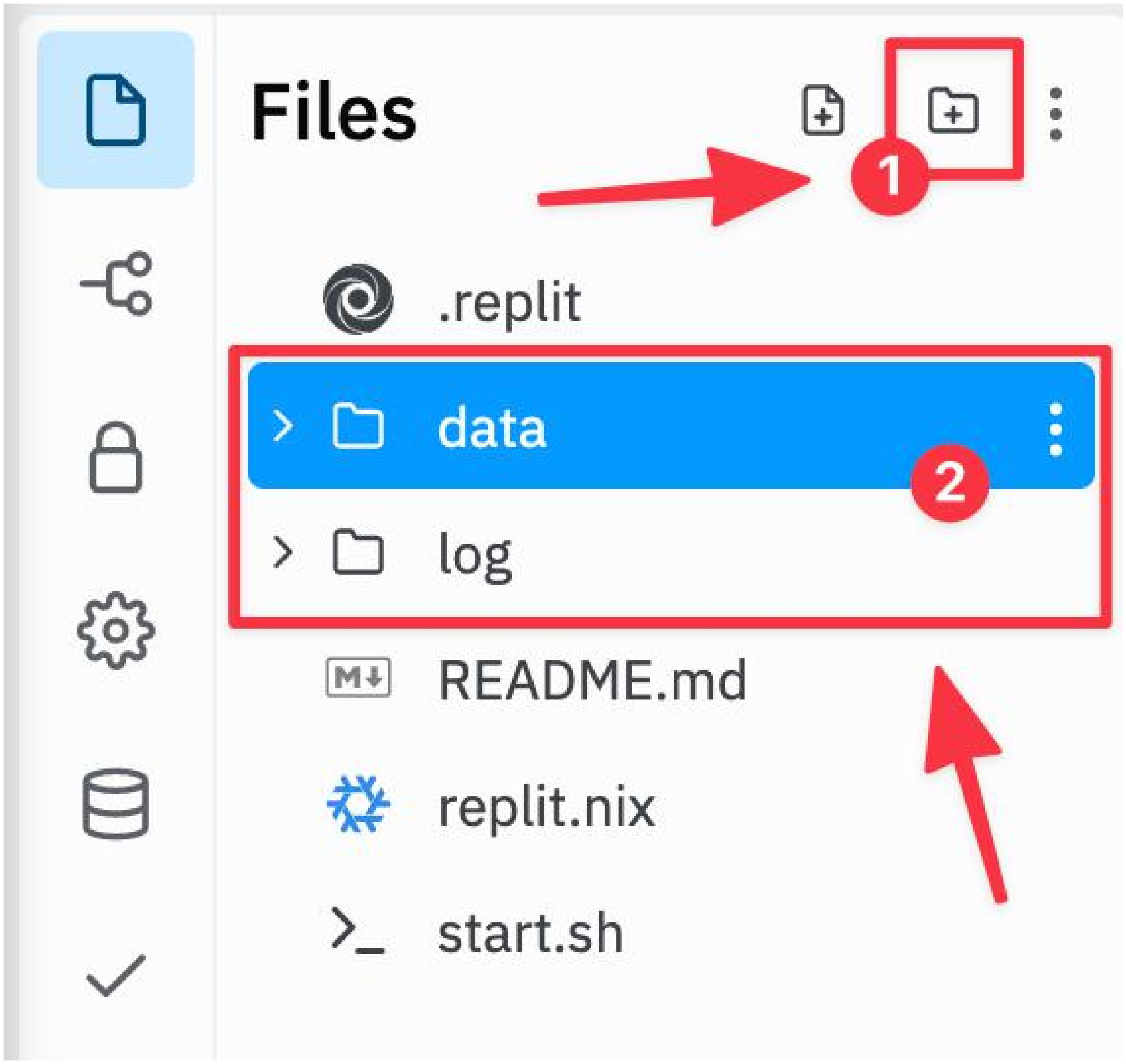
The second section of the script starts MongoDB with the following configuration options:

- `--fork`: This runs MongoDB in a background process, allowing the script to continue executing without shutting it down.
- `--bind_ip="127.0.0.1"`: Listen on the local loopback address only, preventing external access to our database.
- `--dbpath=./data` and `--logpath=./log/mongod.log`: Use local directories for storage. This is important for getting programs to work in Nix repls, as we discussed in our previous tutorial on building with Nix[357].

The third section starts Redis. We use the `--bind` flag to listen on the local loopback address only, similar to how we used it for MongoDB, and `--daemonize yes` runs it as a background process (similar to MongoDB's `--fork`).

Before we run our repl, we'll need to create our MongoDB data and logging directories, `data` and `log`. Create these directories now in your repl's filepane.

[357]https://docs.replit.com/tutorials/30-build-with-nix

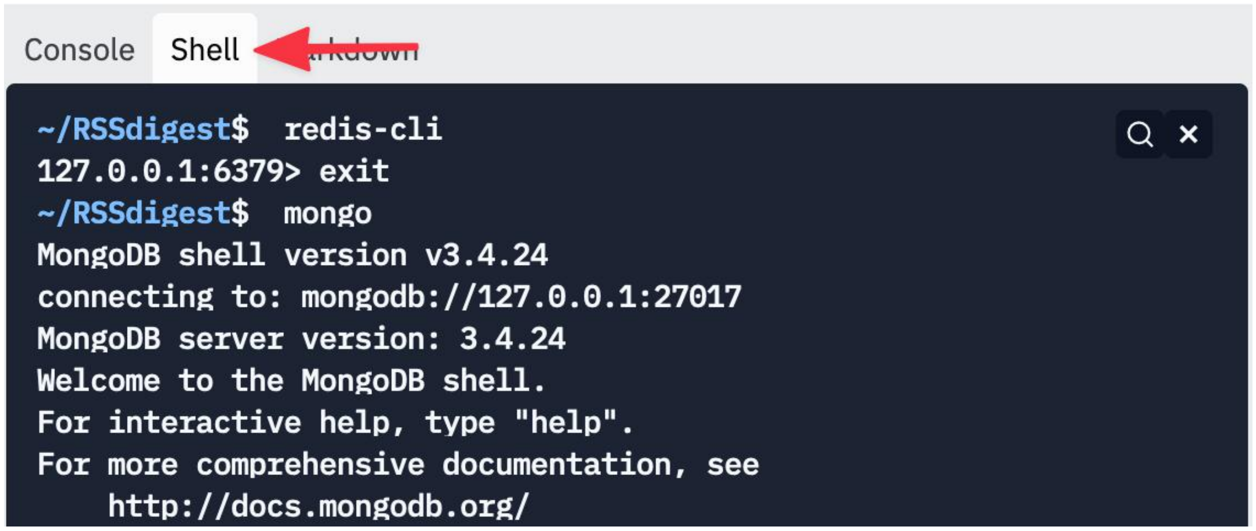


Create Directories

Once that's done, you can run your repl, and it will start MongoDB and Redis. You can interact with MongoDB by running `mongo` in your repl's shell, and with Redis by running `redis-cli`. If you're interested, you can find an introduction to these clients at the links below:

- Working with the `mongo` Shell[358]
- `redis-cli`, the Redis command line interface[359]

[358]https://docs.mongodb.com/v4.4/mongo/#working-with-the-mongo-shell
[359]https://redis.io/topics/rediscli



```
~/RSSdigest$  redis-cli
127.0.0.1:6379> exit
~/RSSdigest$  mongo
MongoDB shell version v3.4.24
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.24
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    http://docs.mongodb.org/
```

Running mongo and redis cli

These datastores will be empty for now.

Important note: Sometimes, when stopping and starting your repl, you may see the following error message:

```
ERROR: child process failed, exited with error number 100
```

This means that MongoDB has failed to start. If you see this, restart your repl, and MongoDB should start up successfully.

Scraping RSS and Atom feeds

We're going to build the feed scraper first. If you've completed any of our previous web-scraping tutorials, you might expect to do this by parsing raw XML with Beautiful Soup[360]. While this would be possible, we would need to account for a large number of differences in feed formats and other gotchas specific to parsing RSS and Atom feeds. Instead, we'll use the feedparser[361] library, which has already solved most of these problems.

Create a directory named `lib`, and inside that directory, a Python file named `scraper.py`. Add the following code to it:

[360]https://beautiful-soup-4.readthedocs.io/en/latest/
[361]https://pypi.org/project/feedparser/

```python
import feedparser, pytz, time
from datetime import datetime, timedelta
from dateutil import parser

def get_title(feed_url):
    pass

def get_items(feed_url, since=timedelta(days=1)):
    pass
```

Here we import the libraries we'll need for web scraping, XML parsing, and time handling. We also define two functions:

- `get_title`: This will return the name of the website, for a given feed track (e.g. "Hacker News" for https://news.ycombinator.com/rss).
- `get_items`: This will return the feed's items – depending on the feed, these can be articles, videos, podcast episodes, or other content. The `since` parameter will allow us to only fetch recent content, and we'll use one day as the default cutoff.

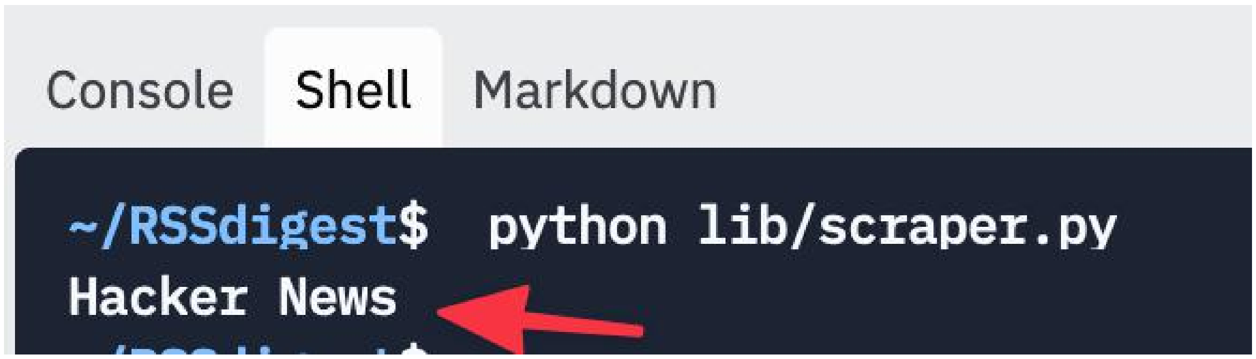
Edit the `get_title` function with the following:

```python
def get_title(feed_url):
    feed = feedparser.parse(feed_url)

    return feed["feed"]["title"]
```

Add the following line to the bottom of `scraper.py` to test it out:

```python
print(get_title("https://news.ycombinator.com/rss"))
```

Instead of rewriting our `start.sh` script to run this Python file, we can just run `python lib/scraper.py` in our repl's shell tab, as shown below. If it's working correctly, we should see "Hacker News" as the script's output.



```
~/RSSdigest$  python lib/scraper.py
Hacker News
```

Script Test

Now we need to write the second function. Add the following code to the `get_items` function definition:

```python
def get_items(feed_url, since=timedelta(days=1)):
    feed = feedparser.parse(feed_url)

    items = []
    for entry in feed.entries:
        title = entry.title
        link = entry.link
        if "published" in entry:
            published = parser.parse(entry.published)
        elif "pubDate" in entry:
            published = parser.parse(entry.pubDate)
```

Here we extract each item's title, link, and publishing timestamp. Atom feeds use the `published` element and RSS feeds use the `pubDate` element, so we look for both. We use `parser`[362] to convert the timestamp from a string to a `datetime` object. The `parse` function is able to convert a large number of different formats, which saves us from writing a lot of extra code.

We need to evaluate the age of the content and package it in a dictionary so we can return it from our function. Add the following code to the bottom of the `get_items` function:

```python
# evaluating content age
        if (since and published > (pytz.utc.localize(datetime.today()) - since)) or \
not since:
            item = {
                "title": title,
                "link": link,
                "published": published
            }
            items.append(item)

    return items
```

We get the current time with `datetime.today()`, convert it to the UTC timezone, and then subtract our `since` `timedelta` object. Because of the way we've constructed this `if` statement, if we pass in `since=None` when calling `get_items`, we'll get all feed items irrespective of their publish date.

Finally, we construct a dictionary of our item's data and add it to the `items` list, which we return at the bottom of the function, outside the `for` loop.

Add the following lines to the bottom of `scraper.py` and run the script in your repl's shell again. We use `time.sleep`[363] to avoid being rate-limited for fetching the same file twice in quick succession.

[362]https://dateutil.readthedocs.io/en/stable/parser.html
[363]https://docs.python.org/3/library/time.html#time.sleep

```
time.sleep(1)
print(get_items("https://news.ycombinator.com/rss"))
```

You should see a large number of results in your terminal. Play around with values of `since` and see what difference it makes.

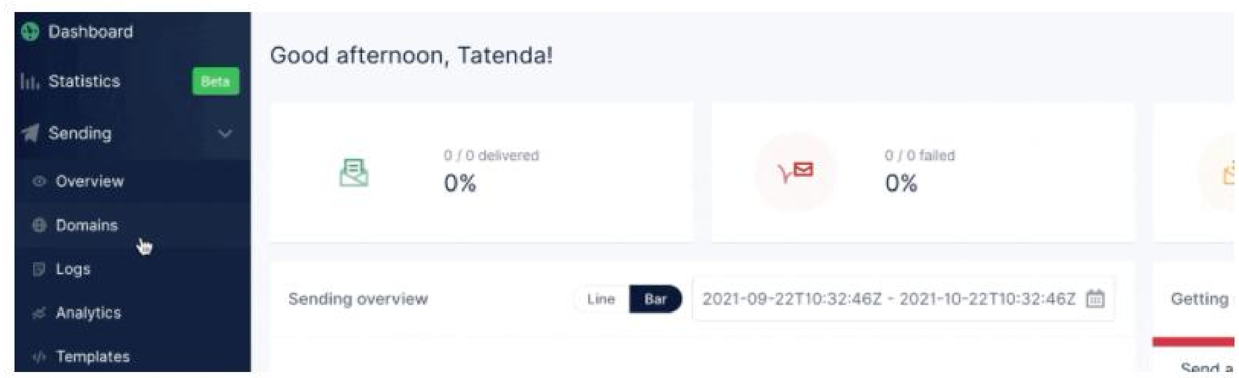
Once you're done, remove the `print` statements from the bottom of the file. We've now built our feed scraper, which we'll use as a library in our main application.

Setting up Mailgun

Now that we can retrieve content for our email digests, we need a way of sending emails. To avoid having to set up our own email server, we'll use the Mailgun[364] API to actually send emails. Sign up for a free account now, and verify your email and phone number.

Once your account is created and verified, you'll need an API key and domain from Mailgun.

To find your domain, navigate to **Sending → Domains**. You should see a single domain name, starting with "sandbox". Click on that and copy the full domain name (it looks like: `sandboxlongstringoflettersandnumb`

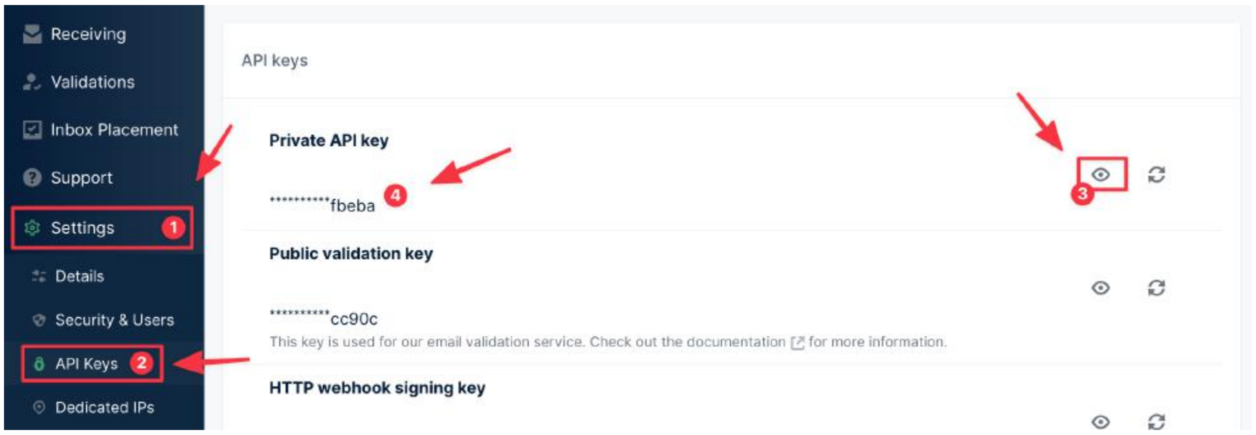


Mailgun domain

click to open gif[365]

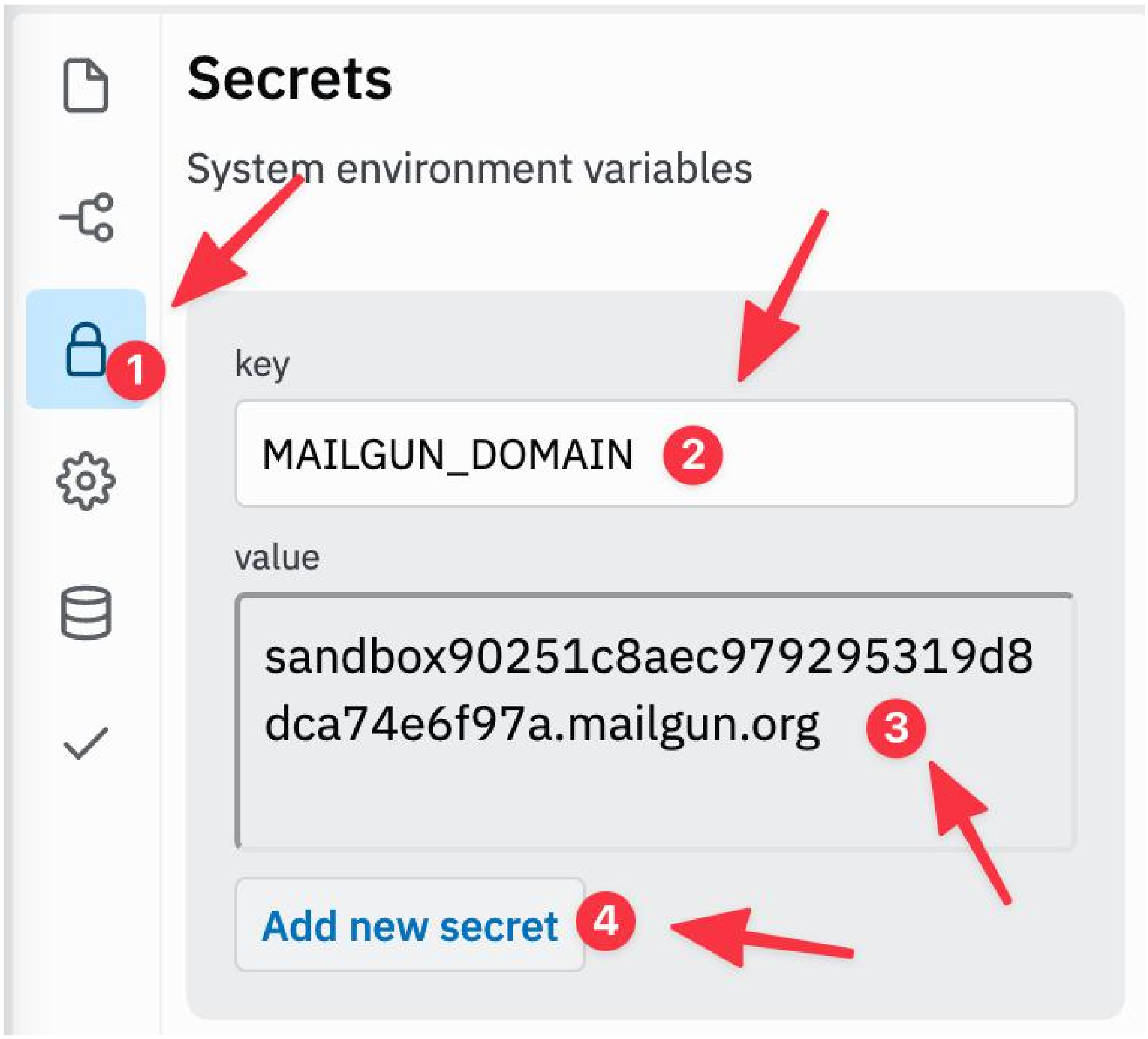
To find your API key, navigate to **Settings → API Keys**. Click on the view icon next to **Private API key** and copy the revealed string somewhere safe.

[364]https://www.mailgun.com/
[365]https://docs.replit.com/images/tutorials/31-news-digest-app/mailgun-domain.gif



Mailgun api key

Back in your repl, create two environment variables, `MAILGUN_DOMAIN` and `MAILGUN_APIKEY`, and provide the strings you copied from Mailgun as values for each.


Secrets
System environment variables
key
MAILGUN_DOMAIN
value
sandbox90251c8aec979295319d8dca74e6f97a.mailgun.org
Add new secret
1
2
3
4

Add Environment Variables

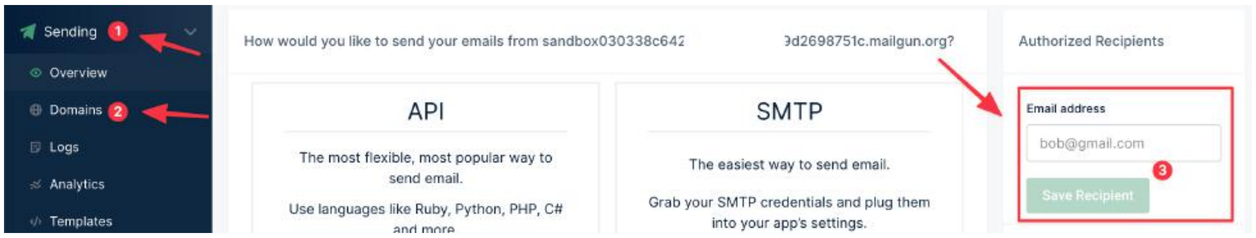
Run your repl now to set these environment variables. Then create a file named `lib/tasks.py`, and populate it with the code below.

```python
import requests, os

# Mailgun config
MAILGUN_APIKEY = os.environ["MAILGUN_APIKEY"]
MAILGUN_DOMAIN = os.environ["MAILGUN_DOMAIN"]

def send_test_email(to_address):
    res = requests.post(
        f"https://api.mailgun.net/v3/{MAILGUN_DOMAIN}/messages",
        auth=("api", MAILGUN_APIKEY),
        data={"from": f"News Digest <digest@{MAILGUN_DOMAIN}>",
              "to": [to_address],
              "subject": "Testing Mailgun",
              "text": "Hello world!"})

    print(res)

send_test_email("YOUR-EMAIL-ADDRESS-HERE")
```

Here we use Python Requests to interact with the Mailgun API[366]. Note the inclusion of our domain and API key.

To test that Mailgun is working, replace `YOUR-EMAIL-ADDRESS-HERE` with your email address, and then run `python lib/tasks.py` in your repl's shell. You should receive a test mail within a few minutes, but as we're using a free sandbox domain, it may end up in your spam folder.

Without further verification on Mailgun, we can only send up to 100 emails per hour, and a free account limits us to 5,000 emails per month. Additionally, Mailgun's sandbox domains can only be used to send emails to specific, whitelisted addresses. The address you created your account with will work, but if you want to send emails to other addresses, you'll have to add them to the domain's authorized recipients, which can be done from the page you got the full domain name from. Keep these limitations in mind as you build and test this application.


Sending
1
Overview
Domains
2
Logs
Analytics
Templates
How would you like to send your emails from sandbox030338c642 ɔd2698751c.mailgun.org?
API
The most flexible, most popular way to send email.
Use languages like Ruby, Python, PHP, C# and more
SMTP
The easiest way to send email.
Grab your SMTP credentials and plug them into your app's settings.
Authorized Recipients
Email address
bob@gmail.com
3
Save Recipient

Recipients

After you've received your test email, you can delete or comment out the function call in the final line of `lib/tasks.py`.

366https://documentation.mailgun.com/en/latest/api-sending.html

Interacting with MongoDB

As we will have two different components of our application interacting with our Mongo database – our email-sending code in `lib/tasks.py` and the web application code we will put in `main.py` – we're going to put our database connection code in another file, which can be imported by both. Create `lib/db.py` now and add the following code to it:

```python
import pymongo

def connect_to_db():
    client = pymongo.MongoClient()
    return client.digest
```

We will call `connect_to_db()` whenever we need to interact with the database. Because of how MongoDB works, a new database called "digest" will be created the first time we connect. Much of the benefit MongoDB provides over traditional SQL databases is that you don't have to define schemas before storing data.

Mongo databases are made up of *collections*, which contain *documents*. You can think of the collections as lists and the documents as dictionaries. When we read and write data to and from MongoDB, we will be working with lists of dictionaries.

Creating the web application

Now that we've got a working webscraper, email sender and database interface, it's time to start building our web application.

Create a file named `main.py` in your repl's filepane and add the following import code to it:

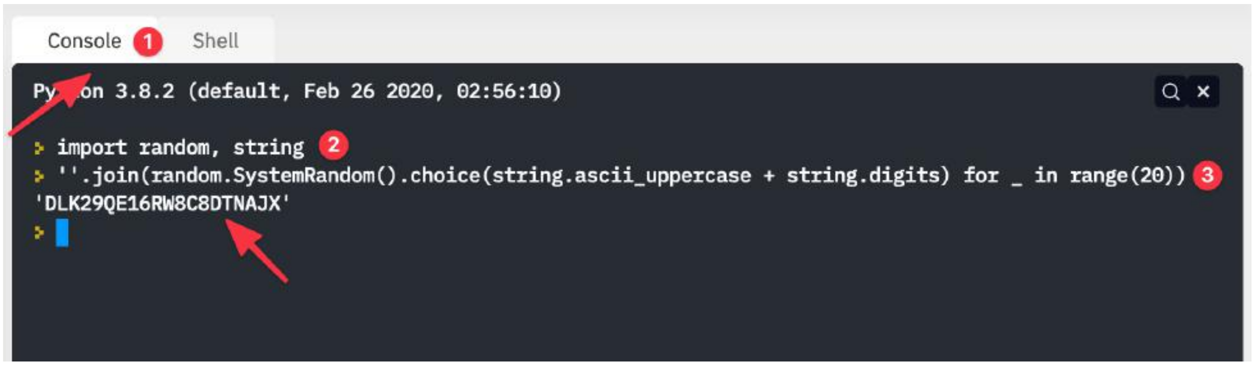
```python
from flask import Flask, request, render_template, session, flash, redirect, url_for
from functools import wraps
import os, pymongo, time

import lib.scraper as scraper
import lib.tasks as tasks
from lib.db import connect_to_db
```

We've imported everything we'll need from Flask and other Python modules, as well as our three local files from `lib`: `scraper.py`, `tasks.py` and `db.py`. Next, add the following code to initialize the application and connect to the database:

```
app = Flask(__name__)

app.config['SECRET_KEY'] = os.environ['SECRET_KEY']

db = connect_to_db()
```

Our secret key will be a long, random string, stored in an environment variable. You can generate one in your repl's Python console with the following two lines of code:

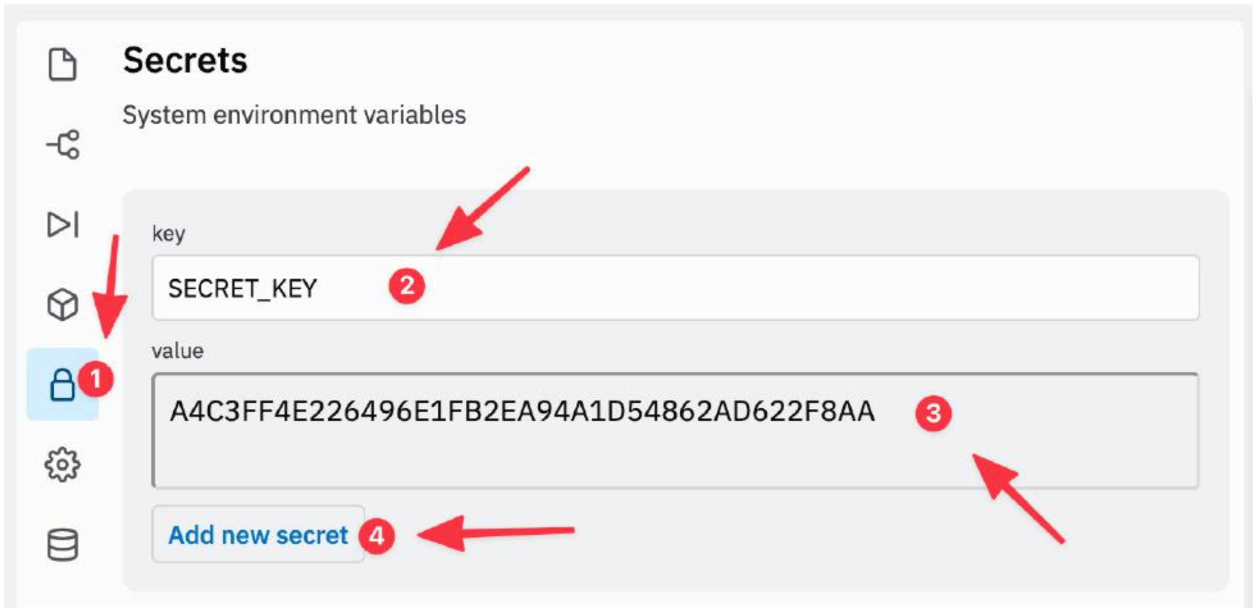
```
import random, string
''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ i\
n range(20))
```



```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> import random, string
> ''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digits) for _ in range(20))
'DLK29QE16RW8C8DTNAJX'
>
```

Random string

In your repl's "Secrets" tab, add a new key named `SECRET_KEY` and enter the random string you just generated as its value.


Secrets
System environment variables
key
SECRET_KEY
value
A4C3FF4E226496E1FB2EA94A1D54862AD622F8AA
Add new secret
1
2
3
4

Repl secret key

Next, we will create the `context` helper function. This function will provide the current user's data from our database to our application frontend. Add the following code to the bottom of `main.py`:

```python
def context():
    email = session["email"] if "email" in session else None

    cursor = db.subscriptions.find({ "email": email })
    subscriptions = [subscription for subscription in cursor]

    return {
        "user_email": email,
        "user_subscriptions": subscriptions
    }
```

When we build our user login, we will store the current user's email address in Flask's `session` object[367], which corresponds to a cookie[368] that will be cryptographically signed with the secret key we defined above. Without this, users would be able to impersonate each other by changing their cookie data.

We query our MongoDB database by calling `db.<name of collection>.find()`[369]. If we call `find()` without any arguments, all items in our collection will be returned. If we call `find()` with an argument, as we've done above, it will return results with keys and values that match our argument. The `find()` method returns a `Cursor`[370] object, which we can extract the results of our query from.

[367]https://flask.palletsprojects.com/en/2.0.x/quickstart/#sessions
[368]https://en.wikipedia.org/wiki/HTTP_cookie
[369]https://docs.mongodb.com/manual/reference/method/db.collection.find/
[370]https://pymongo.readthedocs.io/en/stable/api/pymongo/cursor.html

Next, we need to create an authentication function decorator[371], which will restrict parts of our application to logged-in users. Add the following code below the definition of the `context` function:

```python
# Authentication decorator
def authenticated(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):

        if "email" not in session:
            flash("Permission denied.", "warning")
            return redirect(url_for("index"))

        return f(*args, **kwargs)

    return decorated_function
```

The code in the second function may look a bit strange if you haven't written your own decorators before. Here's how it works: `authenticated` is the name of our decorator. You can think of decorators as functions that take other functions as arguments. (The two code snippets below are for illustration and not part of our program.) Therefore, if we write the following:

```python
@authenticated
def authenticated_function():
    return f"Hello logged-in user!"

authenticated_function()
```

It will be roughly equivalent to:

```python
def authenticated_function():
    return f"Hello logged-in user!"

authenticated(authenticated_function)
```

So whenever `authenticated_function` gets called, the code we've defined in `decorated_function` will execute before anything we define in `authenticated_function`. This means we don't have to include the same authentication checking code in every piece of authenticated functionality. As per the code, if a non-logged-in user attempts to access restricted functionality, our app will flash a warning message and redirect them to the home page.

Next, we'll add code to serve our home page and start our application:

[371] https://realpython.com/primer-on-python-decorators/

```python
# Routes
@app.route("/")
def index():
    return render_template("index.html", **context())

app.run(host='0.0.0.0', port=8080)
```

This code will serve a Jinja[372] template, which we will create now in a separate file. In your repl's filepane, create a directory named `templates`, and inside that directory, a file named `index.html`. Add the following code to `index.html`:

```html
<!DOCTYPE html>
<html>
    <head>
        <title>News Digest</title>
    </head>
    <body>
        {% with messages = get_flashed_messages() %}
            {% if messages %}
                <ul class=flashes>
                {% for message in messages %}
                <li>{{ message }}</li>
                {% endfor %}
                </ul>
            {% endif %}
        {% endwith %}

        {% if user_email == None %}
        <p>Please enter your email to sign up/log in:</p>
        <form action="/login" method="post">
            <input type="text" name="email">
            <input type="submit" value="Login">
        </form>
        {% else %}
        <p>Logged in as {{ user_email }}.</p>
        <h1>Subscriptions</h1>
        <ul>
        {% for subscription in user_subscriptions %}
            <li>
                <a href="{{ subscription.url }}">{{ subscription.title }}</a>
                <form action="/unsubscribe" method="post" style="display: inline">
```

[372] https://jinja.palletsprojects.com/en/3.0.x/templates/

```
                    <input type="hidden" name="feed_url" value="{{subscription.url}}\
">
                    <input type="submit" value="Unsubscribe">
                </form>
            </li>
        {% endfor %}
        </ul>

        <p>Add a new subscription:</p>
        <form action="/subscribe" method="post">
            <input type="text" name="feed_url">
            <input type="submit" value="Subscribe">
        </form>

        <p>Send digest to your email now:</p>
        <form action="/send-digest" method="post">
            <input type="submit" value="Send digest">
        </form>

        <p>Choose a time to send your daily digest (must be UTC):</p>
        <form action="/schedule-digest" method="post">
            <input type="time" name="digest_time">
            <input type="submit" value="Schedule digest">
        </form>
        {% endif %}

    </body>
</html>
```

As this will be our application's only page, it contains a lot of functionality. From top to bottom:

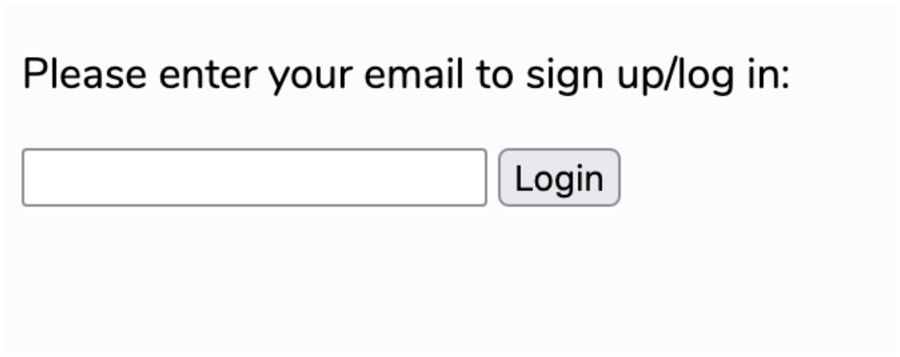
- We've included code to display flashed messages[373] at the top of the page. This allows us to show users the results of their actions without creating additional pages.
- If the current user is not logged in, we display a login form.
- If the current user is logged in, we display:
  * A list of their current subscriptions, with an unsubscribe button next to each one.
  * A form for adding new subscriptions.
  * A button to send an email digest immediately.
  * A form for sending email digests at a specific time each day.

To start our application when our repl runs, we must add an additional line to the bottom of `start.sh`:

[373]https://flask.palletsprojects.com/en/2.0.x/patterns/flashing/

```
# Run Flask app
python main.py
```

Once that's done, run your repl. You should see a login form.



Login Form

Adding user login

We will implement user login by sending a single-use login link to the email address provided in the login form. This provides a number of benefits:

- We can use the code we've already written for sending emails.
- We don't need to implement user registration separately.
- We can avoid worrying about user passwords.

To send login emails asynchronously, we'll set up a Celery task.

In `main.py`, add the following code for the `/login` route below the definition of `index`:

```python
# Login
@app.route("/login", methods=['POST'])
def login():
    email = request.form['email']
    tasks.send_login_email.delay(email)
    flash("Check your email for a magic login link!")

    return redirect(url_for("index"))
```

In this function, we get the user's email, and pass it to a function we will define in `lib/tasks.py`. As this function will be a Celery task[374] rather than a conventional function, we must call it with `.delay()`, a function in Celery's task-calling API[375].

Let's implement this task now. Open `lib/tasks.py` and modify it as follows:

[374] https://docs.celeryproject.org/en/stable/userguide/tasks.html
[375] https://docs.celeryproject.org/en/stable/userguide/calling.html

```python
import requests, os
import random, string # NEW IMPORTS
from celery import Celery # NEW IMPORT
from celery.schedules import crontab # NEW IMPORT
from datetime import datetime # NEW IMPORT

import lib.scraper as scraper # NEW IMPORT
from lib.db import connect_to_db # NEW IMPORT

# NEW LINE BELOW
REPL_URL = f"https://{os.environ['REPL_SLUG']}.{os.environ['REPL_OWNER']}.repl.co"

# NEW LINES BELOW
# Celery configuration
CELERY_BROKER_URL = "redis://127.0.0.1:6379/0"
CELERY_BACKEND_URL = "redis://127.0.0.1:6379/0"

celery = Celery("tasks", broker=CELERY_BROKER_URL, backed=CELERY_BACKEND_URL)

celery.conf.enable_utc = True

# Mailgun config
MAILGUN_APIKEY = os.environ["MAILGUN_APIKEY"]
MAILGUN_DOMAIN = os.environ["MAILGUN_DOMAIN"]

# NEW FUNCTION DECORATOR
@celery.task
def send_test_email(to_address):
    res = requests.post(
        f"https://api.mailgun.net/v3/{MAILGUN_DOMAIN}/messages",
        auth=("api", MAILGUN_APIKEY),
        data={"from": f"News Digest <digest@{MAILGUN_DOMAIN}>",
              "to": [to_address],
              "subject": "Testing Mailgun",
              "text": "Hello world!"})

    print(res)

# COMMENT OUT THE TESTING LINE
# send_test_email("YOUR-EMAIL-ADDRESS-HERE")
```

We’ve added the following:

- Additional imports for Celery and our other local files.
- A `REPL_URL` variable containing our repl's URL, which we construct using environment variables defined in every repl.
- Instantiation of a Celery object, configured to use Redis as a message broker and data backend[376], and the UTC timezone.
- A function decorator which converts our `send_test_email` function into a Celery task.

Next, we'll define a function to generate unique IDs for our login links. Add the following code below the `send_test_email` function definition:

```python
def generate_login_id():
    return ''.join(random.SystemRandom().choice(string.ascii_uppercase + string.digi\
ts) for _ in range(30))
```

This code is largely similar to the code we used to generate our secret key.

Next, we'll create the task we called in `main.py`: `send_login_email`. Add the following code below the definition of `generate_login_id`:

```python
@celery.task
def send_login_email(to_address):

    # Generate ID
    login_id = generate_login_id()

    # Set up email
    login_url = f"{REPL_URL}/confirm-login/{login_id}"

    text = f"""
    Click this link to log in:

    {login_url}
    """

    html = f"""
    <p>Click this link to log in:</p>

    <p><a href={login_url}>{login_url}</a></p>
    """

    # Send email
    res = requests.post(
```

[376] https://docs.celeryproject.org/en/stable/getting-started/backends-and-brokers/index.html

```
        f"https://api.mailgun.net/v3/{MAILGUN_DOMAIN}/messages",
        auth=("api", MAILGUN_APIKEY),
        data={"from": f"News Digest <digest@{MAILGUN_DOMAIN}>",
              "to": [to_address],
              "subject": "News Digest Login Link",
              "text": text,
              "html": html })

    # Add to user_sessions collection if email sent successfully
    if res.ok:
        db = connect_to_db()
        db.user_sessions.insert_one({"login_id": login_id, "email": to_address})

        print(f"Sent login email to {to_address}")
    else:
        print("Failed to send login email.")
```

This code will generate a login ID, construct an email containing a `/confirm-login` link containing that ID, and then send the email. If the email is sent successfully, it will add a document to our MongoDB containing the email address and login ID.

Now we can return to `main.py` and create the `/confirm-login` route. Add the following code below the `login` function definition:

```
@app.route("/confirm-login/<login_id>")
def confirm_login(login_id):
    login = db.user_sessions.find_one({"login_id": login_id})

    if login:
        session["email"] = login["email"]
        db.user_sessions.delete_one({"login_id": login_id}) # prevent reuse
    else:
        flash("Invalid or expired login link.")

    return redirect(url_for("index"))
```

When a user clicks the login link in their email, they will be directed to this route. If a matching login ID is found in the database, they will be logged in, and the login ID will be deleted so it can't be reused.

We've implemented all of the code we need for user login. The last thing we need to do to get it working is to configure our repl to start a Celery worker[377]. When we invoke a task with `.delay()`, this worker will execute the task.

[377]https://docs.celeryproject.org/en/stable/userguide/workers.html

In `start.sh`, add the following between the line that starts Redis and the line that starts our web application:

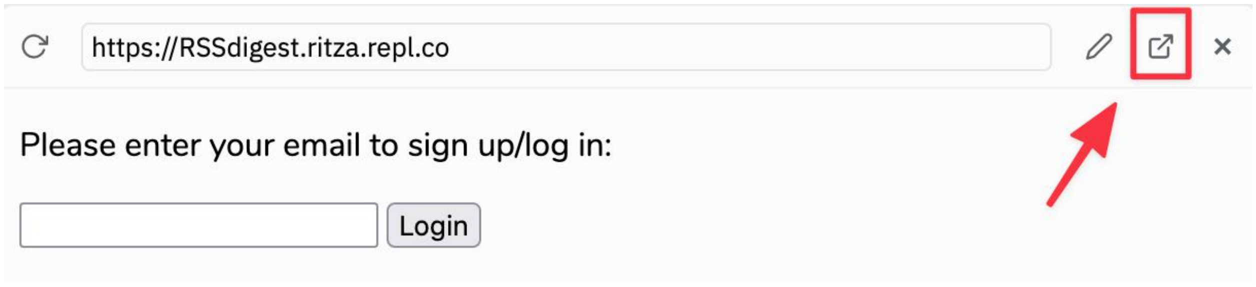
```
# Run Celery worker
celery -A lib.tasks.celery worker -P processes --loglevel=info &
```

This will start a Celery worker, configured with the following flags:

- `-A lib.tasks.celery`: This tells Celery to run tasks associated with the `celery` object in `tasks.py`.
- `-P processes`: This tells Celery to start new processes for individual tasks.
- `--loglevel=info`: This ensures we'll have detailed Celery logs to help us debug problems.

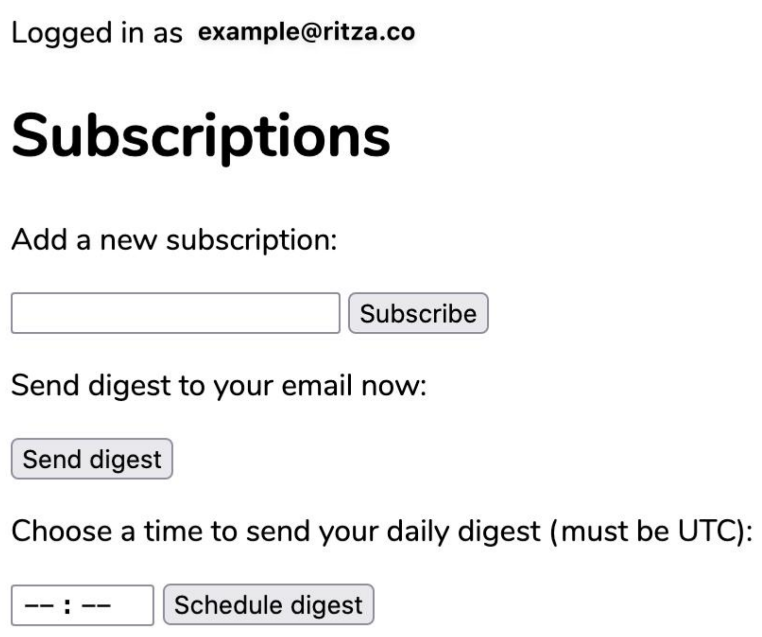
We use `&` to run the worker in the background – this is a part of Bash's syntax rather than a program-specific backgrounding flag like we used for MongoDB and Redis.

Run your repl now, and you should see the worker start up with the rest of our application's components. Once the web application is started, open it in a new tab. Then try logging in with your email address – remember to check your spam box for your login email.


https://RSSdigest.ritza.repl.co
Please enter your email to sign up/log in:
Login

Open in new window

If everything's working correctly, you should see a page like this after clicking your login link:


Logged in as example@ritza.co

Subscriptions

Add a new subscription:

Send digest to your email now:

Choose a time to send your daily digest (must be UTC):

Logged in view

Adding and removing subscriptions

Now that we can log in, let's add the routes that handle subscribing to and unsubscribing from news feeds. These routes will only be available to logged-in users, so we'll use our `authenticated` decorator on them. Add the following code below the `confirm_login` function definition in `main.py`:

```python
# Subscriptions
@authenticated
@app.route("/subscribe", methods=['POST'])
def subscribe(): # new feed
    feed_url = request.form["feed_url"]

    # Test feed
    try:
        items = scraper.get_items(feed_url, None)
    except Exception as e:
        print(e)
        flash("Invalid feed URL.")
        return redirect(url_for("index"))

    if items == []:
        flash("Invalid feed URL")
        return redirect(url_for("index"))

    # Get feed title
    time.sleep(1)
    feed_title = scraper.get_title(feed_url)
```

This code will validate feed URLs by attempting to fetch their contents. Note that we are passing `None` as the argument for `since` in `scraper.get_items` – this will fetch the whole feed, not just the last day's content. If it fails for any reason, or returns an empty list, an error message will be shown to the user and the subscription will not be added.

Once we're sure that the feed is valid, we sleep for one second and then fetch the title. The sleep is necessary to prevent rate-limiting by some websites.

Now that we've validated the feed and have its title, we can add it to our MongoDB. Add the following code to the bottom of the function:

```
    # Add subscription to Mongodb
    try:
        db.subscriptions.insert_one({"email": session["email"], "url": feed_url, "ti\
tle": feed_title})
    except pymongo.errors.DuplicateKeyError:
        flash("You're already subscribed to that feed.")
        return redirect(url_for("index"))
    except Exception:
        flash("An unknown error occured.")
        return redirect(url_for("index"))

    # Create unique index if it doesn't exist
    db.subscriptions.create_index([("email", 1), ("url", 1)], unique=True)

    flash("Feed added!")
    return redirect(url_for("index"))
```

Here, we populate a new document with our subscription details and insert it into our "subscriptions" collection. To prevent duplicate subscriptions, we use `create_index`[378] to create a unique compound index[379] on the "email" and "url" fields. As `create_index` will only create an index that doesn't already exist, we can safely call it on every invocation of this function.

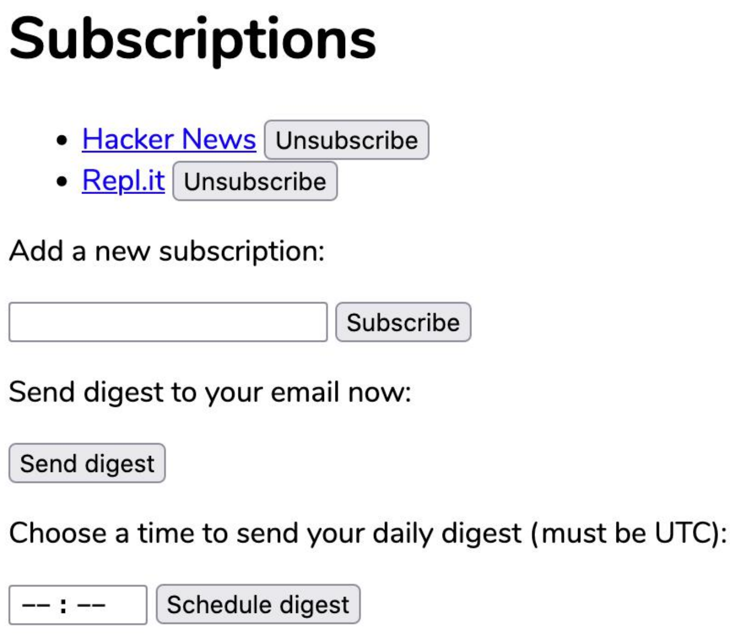
Next, we'll create the code for unsubscribing from feeds. Add the following function definition below the one above:

[378]https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.create_index
[379]https://docs.mongodb.com/manual/core/index-unique/

```
@authenticated
@app.route("/unsubscribe", methods=['POST'])
def unsubscribe(): # remove feed

    feed_url = request.form["feed_url"]
    deleted = db.subscriptions.delete_one({"email": session["email"], "url": feed_ur\
l})

    flash("Unsubscribed!")
    return redirect(url_for("index"))
```

Run your repl, and try subscribing and unsubscribing from some feeds. You can use the following URLs to test:

- Hacker News feed: https://news.ycombinator.com/rss
- /r/replit on Reddit feed: https://www.reddit.com/r/replit.rss



Subscriptions

Sending digests

Once you've added some subscriptions, we can implement the `/send-digest` route. Add the following code below the definition of `unsubscribe` in `main.py`:

```python
# Digest
@authenticated
@app.route("/send-digest", methods=['POST'])
def send_digest():

    tasks.send_digest_email.delay(session["email"])

    flash("Digest email sent! Check your inbox.")
    return redirect(url_for("index"))
```

Then, in `tasks.py`, add the following new Celery task:

```python
@celery.task
def send_digest_email(to_address):

    # Get subscriptions from Mongodb
    db = connect_to_db()
    cursor = db.subscriptions.find({"email": to_address})
    subscriptions = [subscription for subscription in cursor]

    # Scrape RSS feeds
    items = {}
    for subscription in subscriptions:
        items[subscription["title"]] = scraper.get_items(subscription["url"])
```

First, we connect to the MongoDB and find all subscriptions created by the user we're sending to. We then construct a dictionary of scraped items for each feed URL.

Once that's done, it's time to create the email content. Add the following code to the bottom of `send_digest_email` function:

```python
    # Build email digest
    today_date = datetime.today().strftime("%d %B %Y")

    html = f"<h1>Daily Digest for {today_date}</h1>"

    for site_title, feed_items in items.items():
        if not feed_items: # empty list
            continue

        section = f"<h2>{site_title}</h2>"
        section += "<ul>"

        for item in feed_items:
            section += f"<li><a href={item['link']}>{item['title']}</a></li>"

        section += "</ul>"
        html += section
```

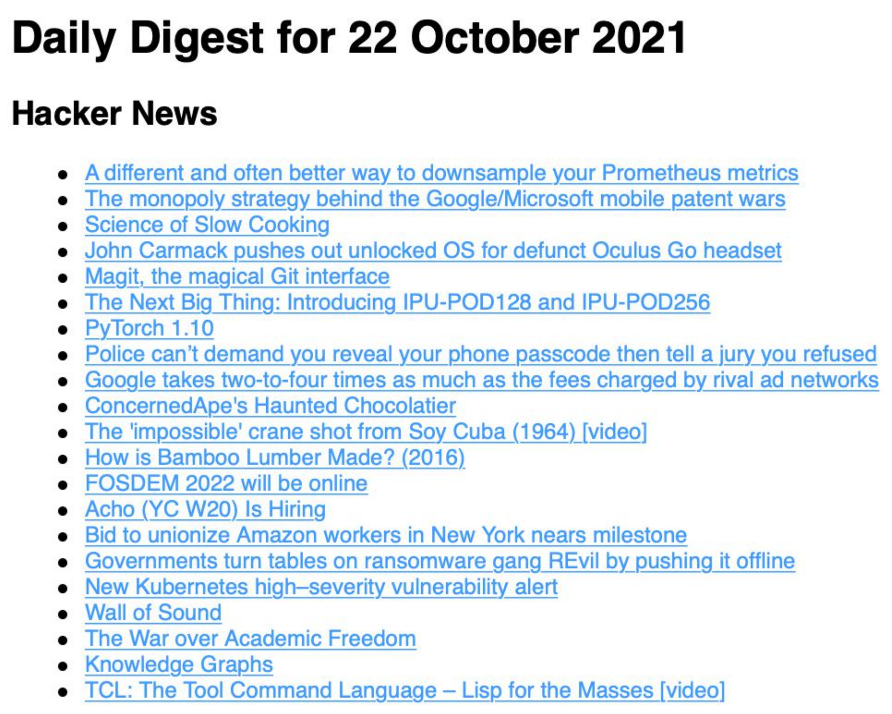
In this code, we construct an HTML email with a heading and bullet list of linked items for each feed. If any of our feeds have no items for the last day, we leave them out of the digest. We use `strftime`[380] to format today's date in a human-readable manner.

After that, we can send the email. Add the following code to the bottom of the function:

```
    # Send email
    res = requests.post(
        f"https://api.mailgun.net/v3/{MAILGUN_DOMAIN}/messages",
        auth=("api", MAILGUN_APIKEY),
        data={"from": f"News Digest <digest@{MAILGUN_DOMAIN}>",
              "to": [to_address],
              "subject": f"News Digest for {today_date}",
              "text": html,
              "html": html })

    if res.ok:
        print(f"Sent digest email to {to_address}")
    else:
        print("Failed to send digest email.")
```

Run your repl, and click on the **Send digest** button. You should receive an email digest with today's items from each of your subscriptions within a few minutes. Remember to check your spam!

[380]https://www.programiz.com/python-programming/datetime/strftime



Daily Digest for 22 October 2021

Hacker News

- A different and often better way to downsample your Prometheus metrics
- The monopoly strategy behind the Google/Microsoft mobile patent wars
- Science of Slow Cooking
- John Carmack pushes out unlocked OS for defunct Oculus Go headset
- Magit, the magical Git interface
- The Next Big Thing: Introducing IPU-POD128 and IPU-POD256
- PyTorch 1.10
- Police can't demand you reveal your phone passcode then tell a jury you refused
- Google takes two-to-four times as much as the fees charged by rival ad networks
- ConcernedApe's Haunted Chocolatier
- The 'impossible' crane shot from Soy Cuba (1964) [video]
- How is Bamboo Lumber Made? (2016)
- FOSDEM 2022 will be online
- Acho (YC W20) Is Hiring
- Bid to unionize Amazon workers in New York nears milestone
- Governments turn tables on ransomware gang REvil by pushing it offline
- New Kubernetes high–severity vulnerability alert
- Wall of Sound
- The War over Academic Freedom
- Knowledge Graphs
- TCL: The Tool Command Language – Lisp for the Masses [video]

Digest email

Scheduling digests

The last thing we need to implement is scheduled digests, to allow our application to send users a digest every day at a specified time.

In `main.py`, add the following code below the `send_digest` function definition:

```python
@authenticated
@app.route("/schedule-digest", methods=['POST'])
def schedule_digest():

    # Get time from form
    hour, minute = request.form["digest_time"].split(":")

    tasks.schedule_digest(session["email"], int(hour), int(minute))

    flash(f"Your digest will be sent daily at {hour}:{minute} UTC")
    return redirect(url_for("index"))
```

This function retrieves the requested digest time from the user and calls `tasks.schedule_digest`. As `schedule_digest` will be a regular function that schedules a task rather than a task itself, we can call it directly.

Celery supports scheduling tasks through its beat functionality[381]. This will require us to run an additional Celery process, which will be a beat rather than a worker.

By default, Celery does not support dynamic addition and alteration of scheduled tasks, which we need in order to allow users to set and change their digest schedules arbitrarily. So we'll need a custom scheduler[382] that supports this.

Many custom Celery scheduler packages are available on PyPI[383], but as of October 2021, none of these packages have been added to Nixpkgs. Therefore, we'll need to create a custom derivation for the scheduler we choose. Let's do that in `replit.nix` now. Open the file, and add the `let ... in` block below:

```
{ pkgs }:
let
    redisbeat = pkgs.python39Packages.buildPythonPackage rec {
        pname = "redisbeat";
        version = "1.2.4";

        src = pkgs.python39Packages.fetchPypi {
            inherit pname version;
            sha256 = "0b800c6c20168780442b575d583d82d83d7e9326831ffe35f763289ebcd8b4\
f6";
        };

        propagatedBuildInputs = with pkgs.python39Packages; [
            jsonpickle
            celery
            redis
        ];

        postPatch = ''
            sed -i "s/jsonpickle==1.2/jsonpickle/" setup.py
        '';
    };
in
{
        deps = [
                pkgs.python39
        pkgs.python39Packages.flask
        pkgs.python39Packages.celery
        pkgs.python39Packages.pymongo
```

[381] https://docs.celeryproject.org/en/stable/userguide/periodic-tasks.html#using-custom-scheduler-classes
[382] https://docs.celeryproject.org/en/stable/userguide/periodic-tasks.html#using-custom-scheduler-classes
[383] https://pypi.org/search/?q=celery+beat&o=

```
        pkgs.python39Packages.requests
        pkgs.python39Packages.redis
        pkgs.python39Packages.feedparser
        pkgs.python39Packages.dateutil
        pkgs.mongodb
        pkgs.redis
        redisbeat # <-- ALSO ADD THIS LINE
        ];
}
```

We've chosen to use `redisbeat`[384], as it is small, simple and uses Redis as a backend. We construct a custom derivation for it using the `buildPythonPackage` function, to which we pass the following information:

- The package's `name` and `version`.
- `src`: Where to find the package's source code (in this case, from PyPI, but we could also use GitHub, or a generic URL).
- `propagatedBuildInputs`: The package's dependencies (all of which are available from Nixpkgs).
- `postPatch`: Actions to take before installing the package. For this package, we remove the version specification for dependency `jsonpickle` in `setup.py`. This will force `redisbeat` to use the latest version of `jsonpickle`, which is available from Nixpkgs and, as a bonus, does not contain this critical vulnerability[385].

You can learn more about using Python with Nixpkgs in this section of the official documentation[386].

To actually install `redisbeat`, we must also add it to our `deps` list. Once you've done that, run your repl. Building custom Nix derivations like this one often takes some time, so you may have to wait a while before your repl finishes loading the Nix environment.

While we wait, let's import `redisbeat` in `lib/tasks.py` and create our `schedule_digest` function. Add the following code to the bottom of `lib/tasks.py`:

[384]https://github.com/liuliqiang/redisbeat
[385]https://nvd.nist.gov/vuln/detail/CVE-2020-22083
[386]https://github.com/NixOS/nixpkgs/blob/master/doc/languages-frameworks/python.section.md

```python
from redisbeat.scheduler import RedisScheduler

scheduler = RedisScheduler(app=celery)

def schedule_digest(email, hour, minute):

    scheduler.add(**{
        "name": "digest-" + email,
        "task": "lib.tasks.send_digest_email",
        "kwargs": {"to_address": email },
        "schedule": crontab(minute=minute, hour=hour)
    })
```

This code uses `redisbeat`'s `RedisScheduler` to schedule the execution of our `send_digest_email` task. Note that we've used the task's full path, with `lib` included: this is necessary when scheduling.

We've used Celery's crontab[387] schedule type, which is highly suited to managing tasks that run at a certain time each day.

If a task with the same name already exists in the schedule, `scheduler.add` will update it rather than adding a new task. This means our users can change their digest time at will.

Now that our code is in place, we can add a new Celery beat process to `start.sh`. Add the following line just after the line that starts the Celery worker:

```
celery -A lib.tasks.celery beat -S redisbeat.RedisScheduler --loglevel=debug &
```

Now run your repl. You can test this functionality out now by scheduling your digest about ten minutes in the future. If you want to receive regular digests, you will need to enable Always-on[388] in your repl. Also, remember that all times must be specified in the UTC timezone.

Where next?

We've built a useful multi-component application, but its functionality is fairly rudimentary. If you'd like to keep working on this project, here are some ideas for next steps:

- Set up a custom domain with Mailgun to help keep your digest emails out of spam.
- Feed scraper optimization. Currently, we fetch the whole feed twice when adding a new subscription and have to sleep to avoid rate-limiting. The scraper could be optimized to fetch feed contents only once.

[387] https://docs.celeryproject.org/en/stable/userguide/periodic-tasks.html#crontab-schedules
[388] https://docs.replit.com/hosting/enabling-always-on

- Intelligent digest caching. If multiple users subscribe to the same feed and schedule their digests for the same time, we will unnecessarily fetch the same content for each one.
- Multiple digests per user. Users could configure different digests with different contents at different times.
- Allow users to schedule digests in their local timezones.
- Styling of both website and email content with CSS.
- A production WSGI and web server to improve the web application's performance, like we used in our previous tutorial on building with Nix[389].

[389]https://docs.replit.com/tutorials/30-build-with-nix

Closing note

We have now come to the end of the series of tutorials. You have learnt the basics of the Repl.it IDE, worked with more advanced features and gone through a number of practical projects. This doesn't mean the end of fun projects, for you should now be equipped to tackle your own projects, which you can start from scratch or use the code from the tutorials as a basis.