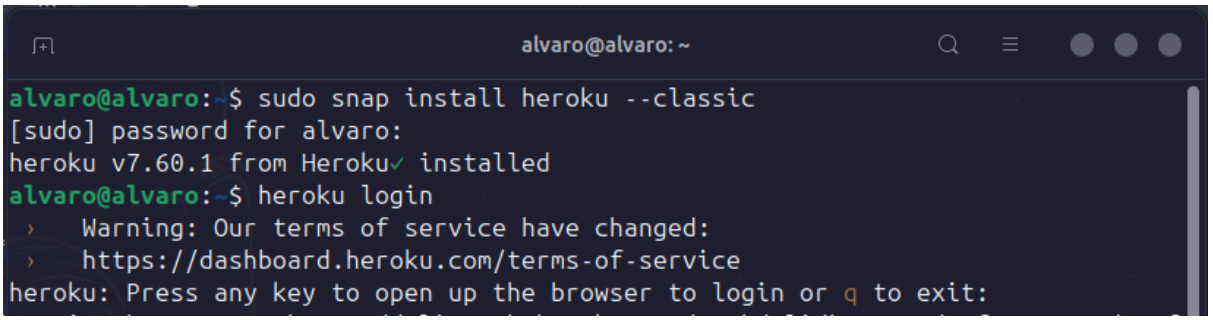
DESPLIEGUE EN LA NUBE HEROKU

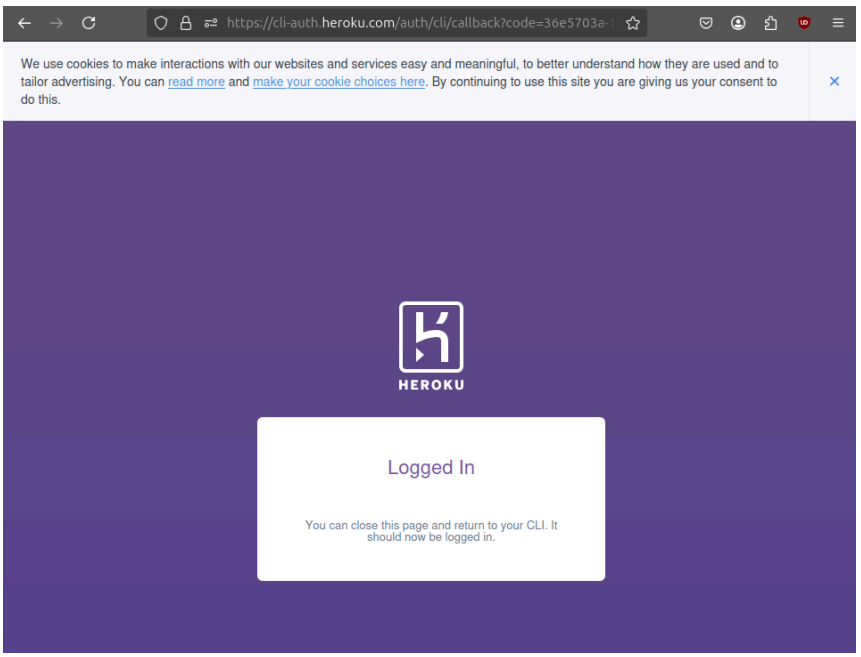
Heroku es un **proveedor** de tecnología en la nube **Paas**, es decir, Platform as a service. Durante el transcurso de esta práctica, **desarrollaremos** una aplicación web con **flask** y la **desplegaremos** en la nube utilizando la plataforma de **Heroku**.

Lo primero que debemos hacer es **abrirnos** una cuenta en Heroku e instalar la **CLI** del mismo que nos permitirá **administrar** y crear las aplicaciones Heroku.

$ sudo snap install heroku --classic

Y luego nos logueamos con la cuenta creada anteriormente.

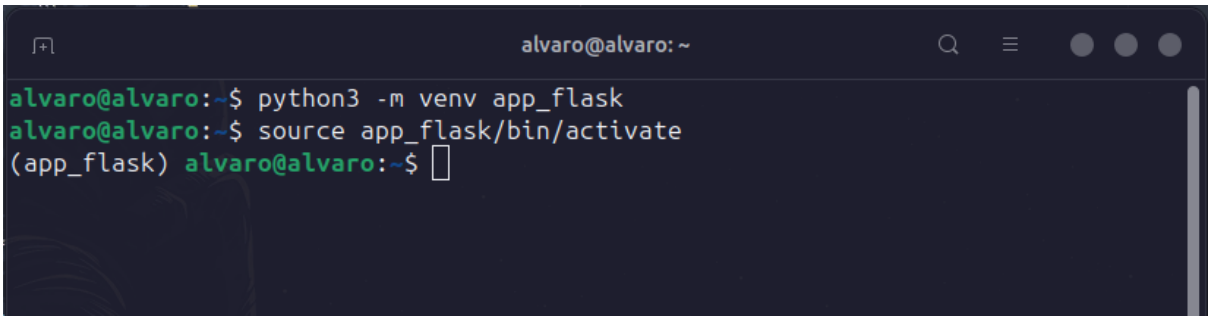




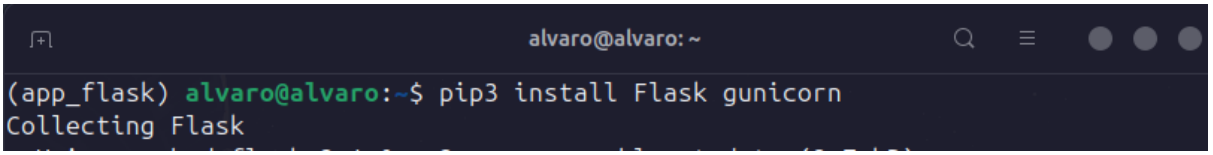
El siguiente paso a realizar es la creación de un **entorno virtual** de desarrollo para nuestra aplicación en **Python**. Utilizamos este tipo de herramientas para evitar **conflictos** de versiones y **aislar** el proyecto.

Para ello instalamos **venv**, la herramienta que nos ayudará a crear dichos entornos.
venv →Virtual environment

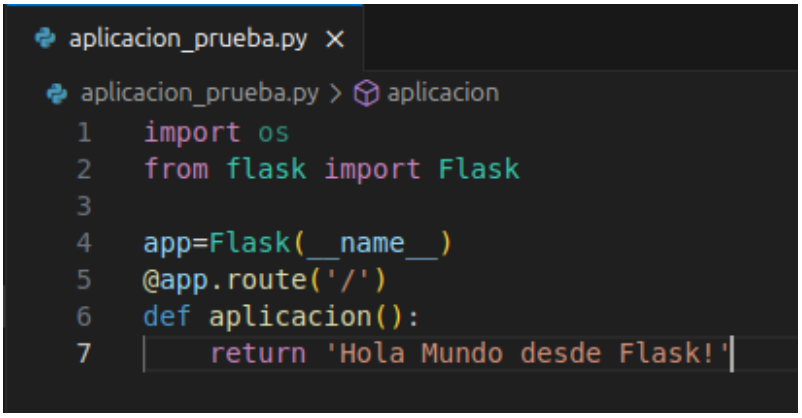
Para crear nuestro primer **entorno virtual** debemos ejecutar:
$ python3 -m venv nombre_del_entorno donde la opción **-m** indica que vamos a usar un **módulo**.
Y listo, ahora para **activarlo** ejecutaremos: $ source nombre_venv/bin/activate
Deberemos ver como ha cambiado el prompt de la CLI.



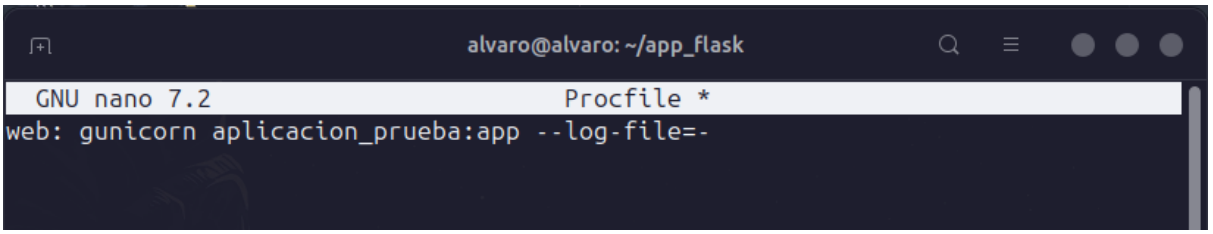
Ahora dentro de nuestro entorno podremos **instalar** cualquier paquete **pip** sin romper **ninguna dependencia**. Instalaremos flask y Gunicorn (Un servidor web).



Utilizando algún **IDE** o herramienta de texto, iniciaremos nuestra **aplicación** en **python** adaptada a **Flask.**



Hay que crear un fichero llamado **Procfile** para declarar qué **comando** debe ejecutarse para **arrancar** un web dyno (Un contenedor) de **Heroku**. Este fichero se localiza en la **raíz** del proyecto y **no** tendrá **extensión**.



Ya podemos **arrancar** los **procesos** definidos en el **Procfile** con $ heroku local

Lo siguiente que vamos a crear es un **documento** de texto llamado **requirements**, que vamos a localizarlo en la **raíz** del proyecto. Este fichero contendrá los **módulos** de **python necesarios** para que nuestra aplicación funcione. Para generarlo podremos utilizar un comando **pip** que lo hace **automáticamente**.

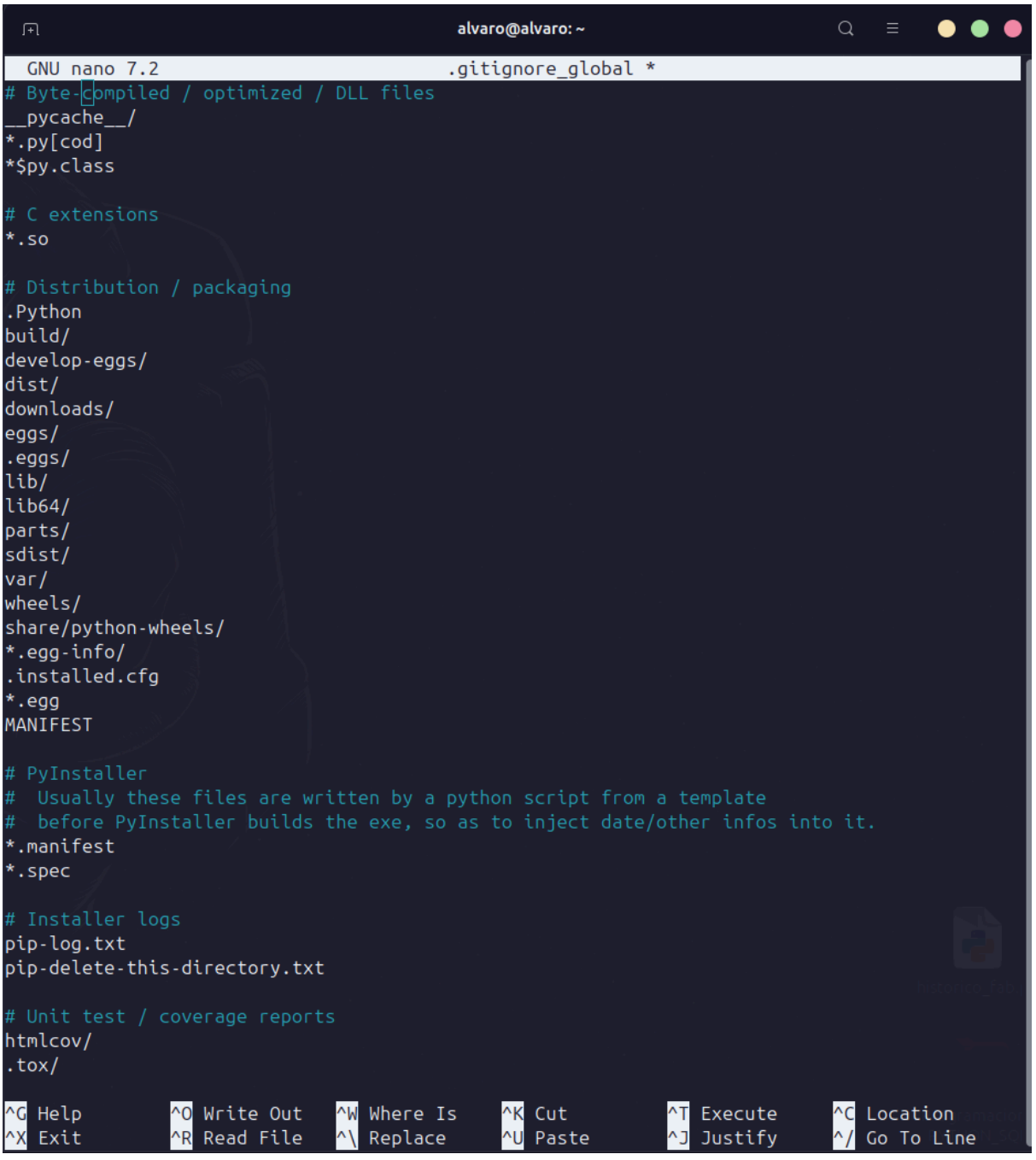
Una vez lista la base del **proyecto**, vamos a almacenarlo en un repositorio **Git**.
Antes de hacerlo, conviene configurar qué archivos **no** queremos que se incluyan en el **control de versiones**. Esto se puede hacer de dos formas:

- A nivel de proyecto, creando un archivo llamado **.gitignore** en el directorio **raíz**, donde se listan los archivos o carpetas que **Git** debe ignorar.

- A nivel global, definiendo reglas que se aplican a **todos los repositorios Git de tu sistema**, mediante un archivo de configuración global.

En dicho fichero copiaremos el contenido de un **archivo** que **Github** nos proporciona ya configurado para **python**

https://github.com/github/gitignore/blob/main/Python.gitignore


alvaro@alvaro: ~
GNU nano 7.2 .gitignore_global *
Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
C extensions
*.so
Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST
PyInstaller
Usually these files are written by a python script from a template
before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec
Installer logs
pip-log.txt
pip-delete-this-directory.txt
Unit test / coverage reports
htmlcov/
.tox/
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line

Una vez listo el fichero **gitignore_global**, vamos a convertir nuestro **directorio** con nuestra aplicación en un **repositorio git**. Para ello usaremos los siguientes comandos:

- git init → Lanzado **dentro** del directorio, para **inicializarlo** como repositorio git.

- git add . → Manda todos los archivos y directorios a un area de preparacion es decir, los prepara para un **commit** (el punto hace referencia a todos los archivos).

- git commit -m “Texto de ejemplo” → Realiza un **commit** con -m para añadir un mensaje o nombre que **identificara** al **commit** o versión. Este mensaje es **obligatorio**.

```
alvaro@alvaro: ~/app_flask
(app_flask) alvaro@alvaro:~/app_flask$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/alvaro/app_flask/.git/
```

```
(app_flask) alvaro@alvaro:~/app_flask$ git add .
(app_flask) alvaro@alvaro:~/app_flask$ git commit -m "Version 1.0"
[master (root-commit) 76e2ea2] Version 1.0
 1452 files changed, 263861 insertions(+)
 create mode 100644 Procfile
 create mode 100644 __pycache__/aplicacion_prueba.cpython-312.pyc
```

Ya tenemos todo preparado, ahora vamos a **desplegar** el proyecto en la **plataforma como servicio** de heroku, creando una **aplicación remota** mediante la **CLI** de Heroku:

```
alvaro@alvaro: ~/app_flask
(app_flask) alvaro@alvaro:~/app_flask$ heroku create rfab
 ›   Warning: heroku update available from 7.60.1 to 10.4.1.
Creating ⬢ rfab... done
https://rfab-3e2bcfaabfea.herokuapp.com/ | https://git.heroku.com/rfab.git
(app_flask) alvaro@alvaro:~/app_flask$
```

Y luego con **git push** realizamos el **despliegue** (**master** es la rama **principal** de nuestro **repositorio**)

```
(app_flask) alvaro@alvaro:~/app_flask$ git push heroku master
Enumerating objects: 1587, done.
Counting objects: 100% (1587/1587), done.
Delta compression using up to 12 threads
Compressing objects: 100% (1571/1571), done.
Writing objects: 100% (1587/1587), 6.20 MiB | 3.87 MiB/s, done.
Total 1587 (delta 73), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (73/73), done.
```

```
alvaro@alvaro: ~/app_flask
remote:        Downloading MarkupSafe-3.0.2-cp313-cp313-manylinux_2_17_x86_64.manylin
ux2014_x86_64.whl (23 kB)
remote:        Downloading packaging-24.2-py3-none-any.whl (65 kB)
remote:        Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
remote:        Installing collected packages: packaging, MarkupSafe, itsdangerous, cl
ick, blinker, Werkzeug, Jinja2, gunicorn, Flask
remote:        Successfully installed Flask-3.1.0 Jinja2-3.1.6 MarkupSafe-3.0.2 Werkz
eug-3.1.3 blinker-1.9.0 click-8.1.8 gunicorn-23.0.0 itsdangerous-2.2.0 packaging-24.2
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 24.1M
remote: -----> Launching...
remote:        Released v3
remote:        https://rfab-3e2bcfaabfea.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/rfab.git
 * [new branch]      master -> master
(app_flask) alvaro@alvaro:~/app_flask$
```

Vamos a profundizar un poco más sobre el término **dynos** y el archivo **Procfile** que hemos declarado antes. En el **Procfile** que hemos declarado anteriormente, se detallan qué **comandos** van a ser **ejecutados** por los **dynos** cuando arranca tu aplicación.. Un dyno es básicamente un **contenedor** ligero, similar a un docker. Es ahí donde se **ejecuta** tu aplicación.

¿Qué hace un dyno?

- Ejecuta procesos de tu app (por ejemplo, un servidor web).
- Puedes tener **varios dynos**: unos para atender peticiones web, otros para tareas en segundo plano, etc.
- Cada dyno tiene recursos **limitados** (RAM, CPU...), y pueden apagarse o reiniciarse dependiendo del plan.

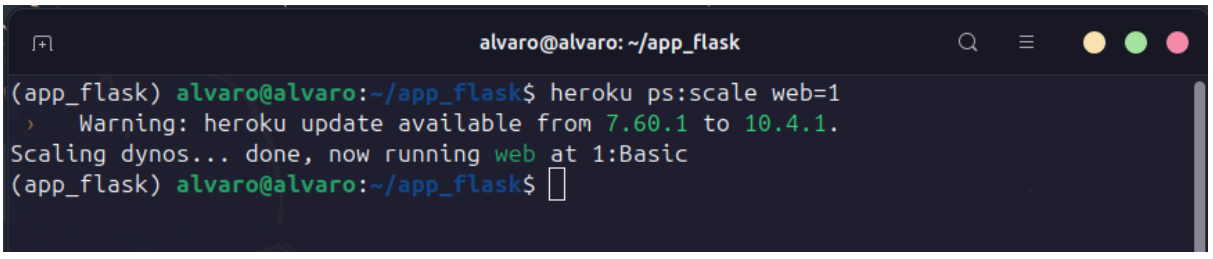
Ahora vamos a ver si tenemos un dyno para el tipo de proceso web con el comando

$ heroku ps:scale web=1

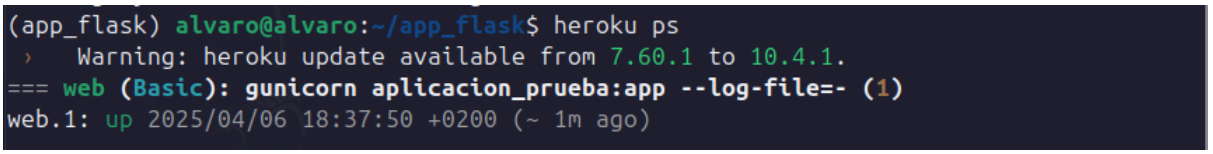
ps hace referencia a "**processes**" similar al comando docker ps

scale es la acción que quieres hacer: **escalar**, es decir, ajustar cuántos dynos están ejecutando un determinado tipo de proceso.

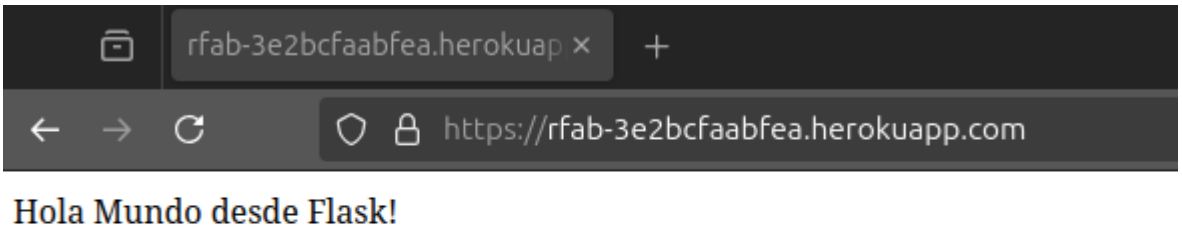
web=1 Le estás diciendo a Heroku: "quiero **1 dyno** ejecutando el proceso de tipo web.



Comprobamos el **estado** de los dynos de nuestra app.



Y por último **visitamos** nuestra aplicación web.



Vamos a jugar un poco más añadiendo una **función** algo más **interesante**: **fab.py.** He creado una función llamada **fab**, que voy a **importar** en la función **principal** de **Flask**. Esta última abrirá un fichero **CSV** localizado en la **raíz** del proyecto y le **aplicará** la función fab. Luego, el resultado se devolverá a la **web** utilizando el **protocolo HTTP**.

Fab.py

```
import sys

def fab(a):


   equipos = {}
   partes = {}
   for linea in a:
       linea = linea.strip().split(',')
       evento = linea[0]

       if evento[0] == 'Q':
           idcuar = int(evento[1])
           partes[idcuar] = {}

           for ideq in equipos:
               partes[idcuar][ideq] = 0

       elif evento == 'eq':
           ideq = int(linea[1])
           nomeq = linea[2]
           equipos[ideq] = nomeq

       elif evento == 'b':
           linea_canastas = linea[1].split('#')
           puntos = int(linea[2])
           ideq = int(linea_canastas[0])



           if partes[idcuar][ideq] == 0:
               partes[idcuar][ideq] = puntos
           else:
               partes[idcuar][ideq] += puntos


   return equipos, partes
```

APLICACIÓN FLASK

```python
import os
from flask import Flask
from fab3 import fab


app = Flask(__name__)

@app.route('/')
def aplicacion():
    b = ""
    try:
        with open('envivo.csv', 'r') as archivo:
            # Procesa el archivo si existe
            contenido = archivo.readlines()
            lista = fab(contenido)

            equipos = lista[0]
            puntaje = lista[1]


            for idcuar in puntaje:
                b += f"{idcuar}° Cuarto."

                for ideq, nomeq in equipos.items():
                    punt = puntaje[idcuar].get(ideq, "N/A")
                    b += f'{nomeq}: {punt}\n'

                b += '<br>' # Salto de linea en html

    except FileNotFoundError:
        return 'Error: El archivo envivo.csv no existe.'
    except Exception as e:
        return f'Ocurrió un error inesperado: {e}'

    return b

if __name__ == '__main__':
    app.run(debug=True)
```

En el código de la **aplicación Flask**, lo único que hago es **formatear** la salida y **proteger el código** con una estructura **try-except**, por si ocurre algún error al **procesar** el fichero. Al principio había utilizado el método .**read(),** pero esto devolvía una **cadena de texto** (string), no una **lista** de líneas como cuando usamos **.readlines()**. Esto me obligaba a dividir manualmente el contenido con .split('\n'), lo cual era más incómodo y menos limpio. Con .**readlines()** obtengo directamente una lista, lo que **facilita** mucho el tratamiento línea a línea del **archivo CSV**.

Salida en el navegador:

Por último me gustaría profundizar en la definición de Paas, los tipos de servicios que nos proporcionan y los distintos proveedores cloud que nos suministran este servicio.

¿Qué es PaaS? (Platform as a Service)

PaaS (Plataforma como Servicio) es un modelo de computación en la nube que proporciona a los desarrolladores una plataforma completa para construir, desplegar y gestionar aplicaciones sin tener que preocuparse por la infraestructura subyacente (servidores, almacenamiento, redes, etc.).

¿Qué incluye un servicio PaaS?

Entorno de desarrollo integrado (IDE)

Bases de datos

Middleware

Servicios de backend (como autenticación, gestión de APIs, colas de mensajes)

Herramientas de testing y despliegue continuo

Escalado automático y gestión de recursos

Proveedores cloud que ofrecen PaaS

1. Heroku

Muy popular para proyectos pequeños o medianos.

Soporta múltiples lenguajes (Python, Node.js, Ruby, Java, etc.).

Despliegue extremadamente fácil (con solo hacer git push).

Ideal para desarrolladores que buscan rapidez sin complicarse.

2. Google App Engine (GAE)

Parte de Google Cloud Platform.

Soporta varios lenguajes y escalado automático.

Integración nativa con otros servicios de Google (Cloud Functions, BigQuery, etc.).

Buena opción para aplicaciones que esperan tráfico variable.

3. Microsoft Azure App Service

Plataforma PaaS de Microsoft.

Compatible con .NET, Java, Node.js, PHP, Python, etc.

Incluye integración con Visual Studio, CI/CD con GitHub y Azure DevOps.

Ideal para empresas que ya usan herramientas Microsoft.

4. AWS Elastic Beanstalk

Solución PaaS de Amazon Web Services.

Permite desplegar aplicaciones web y servicios fácilmente.

Maneja automáticamente el aprovisionamiento, balanceo de carga, escalado y monitoreo.

Permite acceso al entorno subyacente (más control, pero también más responsabilidad).

5. Red Hat OpenShift

Basado en Kubernetes y enfocado a entornos empresariales.

Ofrece control total sobre los contenedores, con una capa PaaS encima.

Muy usado en empresas que necesitan despliegues híbridos (nube y on-premise).

BIBLIOGRAFÍA:

https://azure.microsoft.com/en-us/products/app-service/

https://aws.amazon.com/es/elasticbeanstalk/

https://devcenter.heroku.com/