Django y React 2024

Diego Saavedra

Aug 13, 2024

Table of contents

1 Bienvenido

¡Bienvenido al Curso Completo de Django y React!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso completo me llevará desde los fundamentos básicos de la programación hasta la construcción de aplicaciones prácticas utilizando los frameworks Django y la biblioteca de React.

A través de una combinación de teoría y ejercicios prácticos, me sumergiré en los conceptos esenciales del desarrollo web y avanzaré hacia la creación de proyectos del mundo real.

Desde la configuración del entorno de desarrollo hasta la construcción de una aplicación web de pila completa, este curso me proporcionará una comprensión sólida y experiencia práctica con Django y React.

1.2 ¿Para quién es este curso?

Este curso está diseñado para principiantes y aquellos con poca o ninguna experiencia en programación.

Ya sea que sea un estudiante curioso, un profesional que busca cambiar de carrera o simplemente alguien que quiere aprender desarrollo web, este curso es para usted. Desde adolescentes hasta adultos, todos son bienvenidos a participar y explorar el emocionante mundo del desarrollo web con Django y React.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en linea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

Esperamos que disfrute este emocionante viaje de aprendizaje y descubrimiento en el mundo del desarrollo web con Django y React!

Part I

Unidad 1: Introducción a Python

2 Git y GitHub

Figure 2.1: Git and Github

2.1 ¿Qué es Git y GitHub?

Git y GitHub son herramientas ampliamente utilizadas en el desarrollo de software para el control de versiones y la colaboración en proyectos.

Git es un sistema de control de versiones distribuido que permite realizar un seguimiento de los cambios en el código fuente durante el desarrollo de software. Fue creado por Linus Torvalds en 2005 y se utiliza mediante la línea de comandos o a través de interfaces gráficas de usuario.

GitHub, por otro lado, es una plataforma de alojamiento de repositorios Git en la nube. Proporciona un entorno colaborativo donde los desarrolladores pueden compartir y trabajar en proyectos de software de forma conjunta. Además, ofrece características adicionales como seguimiento de problemas, solicitudes de extracción y despliegue continuo.

En este tutorial, aprenderás los conceptos básicos de Git y GitHub, así como su uso en un proyecto de software real.

2.2 ¿Quiénes utilizan Git?

Figure 2.2: Git

Es ampliamente utilizado por desarrolladores de software en todo el mundo, desde estudiantes hasta grandes empresas tecnológicas. Es una herramienta fundamental para el desarrollo colaborativo y la gestión de proyectos de software.

2.3 ¿Cómo se utiliza Git?

```
commit e072c20b5577c37af7c4fb274b6b53d15dd336ae
Author: Julio Xavier <julioxavierr@live.com>
Date:   Fri Aug 19 16:17:10 2016 -0300

    Commit with error

commit a497c0c03657549e7d4c5ba1b23ffce5faaf46b8
Author: Julio Xavier <julioxavierr@live.com>
Date:   Mon Jan 11 10:51:42 2016 -0200

    Adding common html code in a form

commit 9fa7605ad1837aa44dfb9c711dc8bd60cabc7c5d
Author: Julio Xavier <julioxavierr@live.com>
Date:   Sun Jan 10 22:29:52 2016 -0200

    Pages to show 'details' + Editing Clients
```

Figure 2.3: Git en Terminal

Se utiliza mediante la **línea de comandos** o a través de **interfaces gráficas** de usuario. Proporciona comandos para realizar operaciones como:

1. Inicializar un repositorio,
2. Realizar cambios,
3. Revisar historial,
4. Fusionar ramas,
5. Entre otros.

2.4 ¿Para qué sirve Git?

Figure 2.4: Seguimiento de Cambios con Git

Sirve para realizar un seguimiento de los cambios en el código fuente, coordinar el trabajo entre varios desarrolladores, revertir cambios no deseados y mantener un historial completo de todas las modificaciones realizadas en un proyecto.

2.5 ¿Por qué utilizar Git?

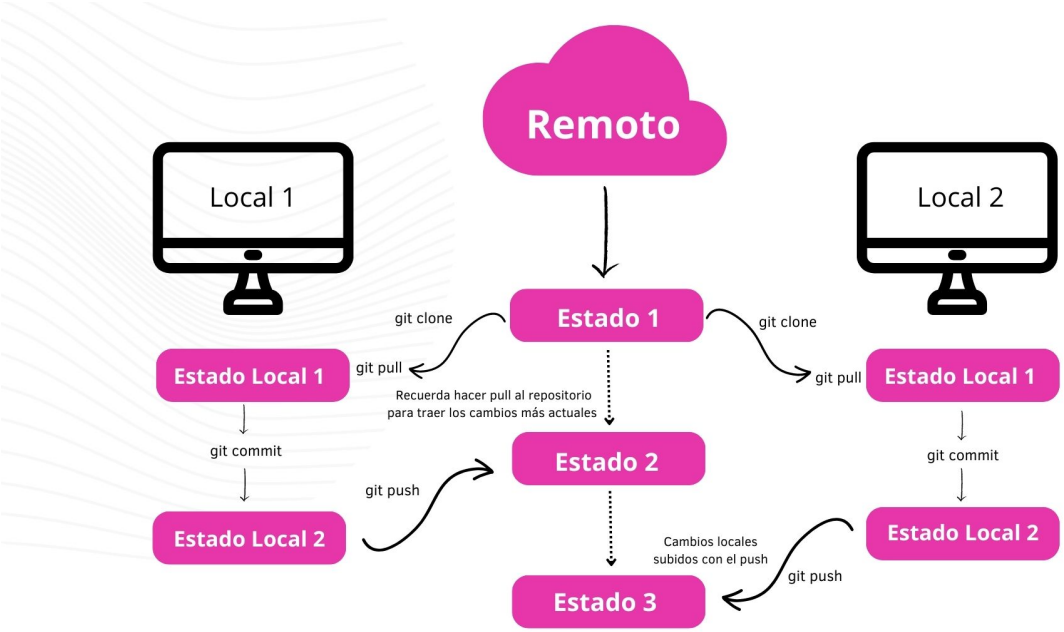


Figure 2.5: Ventajas de Git

Ofrece varias ventajas, como:

- La capacidad de trabajar de forma distribuida
- La gestión eficiente de ramas para desarrollar nuevas funcionalidades
- Corregir errores sin afectar la rama principal
- La posibilidad de colaborar de forma efectiva con otros desarrolladores.

2.6 ¿Dónde puedo utilizar Git?

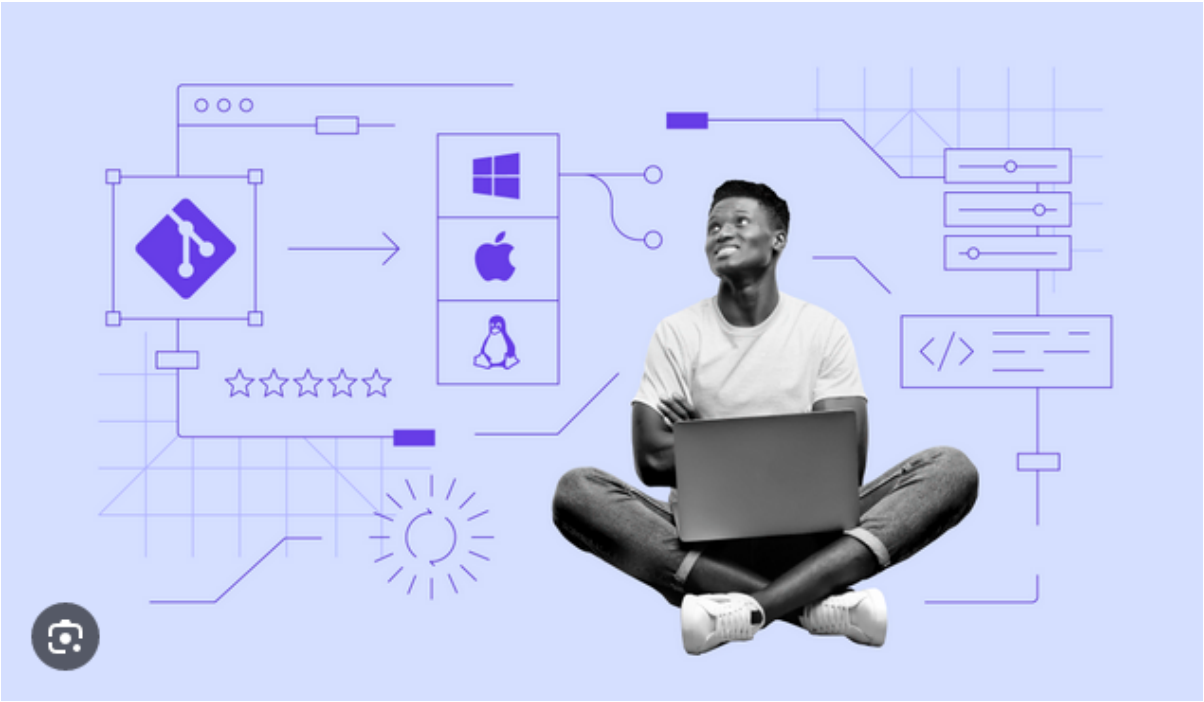


Figure 2.6: Git en Diferentes Sistemas Operativos

Puede ser utilizado en cualquier sistema operativo, incluyendo Windows, macOS y Linux. Además, es compatible con una amplia variedad de plataformas de alojamiento de repositorios, siendo GitHub una de las más populares.

2.7 Pasos Básicos

> **Tip**
>
> Es recomendable tomar en cuenta una herramienta para la edición de código, como Visual Studio Code, Sublime Text o Atom, para trabajar con Git y GitHub de manera eficiente.

2.8 Instalación de Visual Studio Code

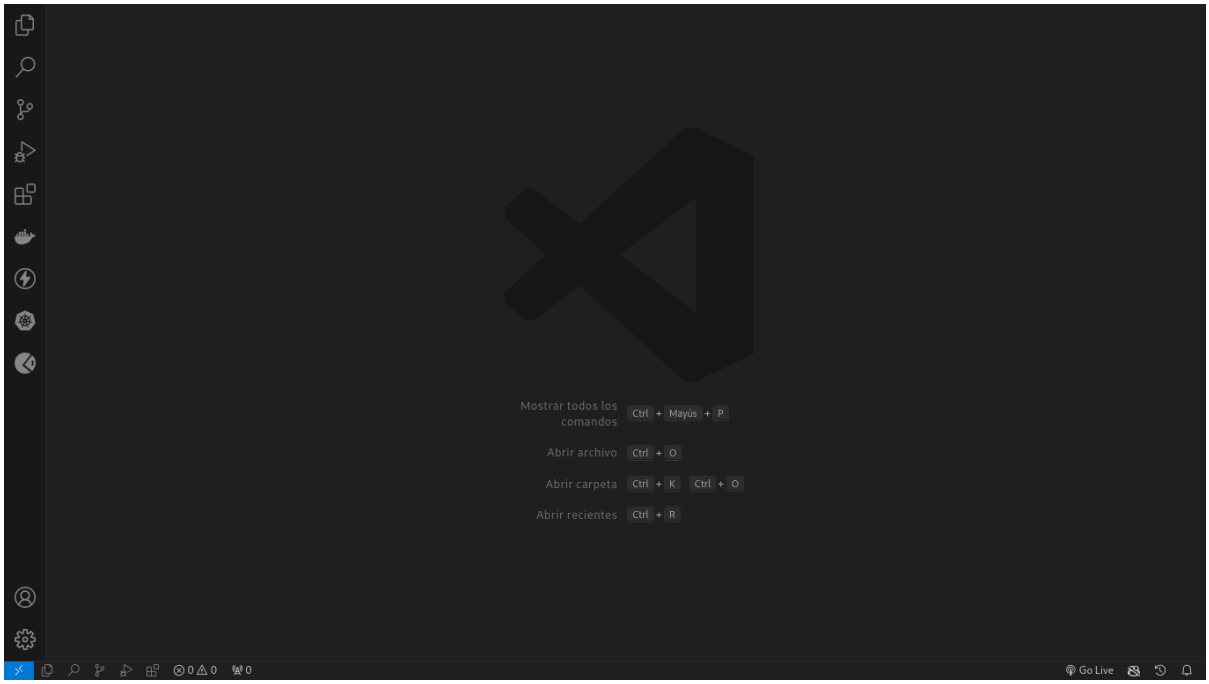


Figure 2.7: Visual Studio Code

Si aún no tienes Visual Studio Code instalado, puedes descargarlo desde https://code.visualstudio.com/download. Es una herramienta gratuita y de código abierto que proporciona una interfaz amigable para trabajar con Git y GitHub.

A continuación se presentan los pasos básicos para utilizar Git y GitHub en un proyecto de software.

2.8.1 Descarga e Instalación de Git

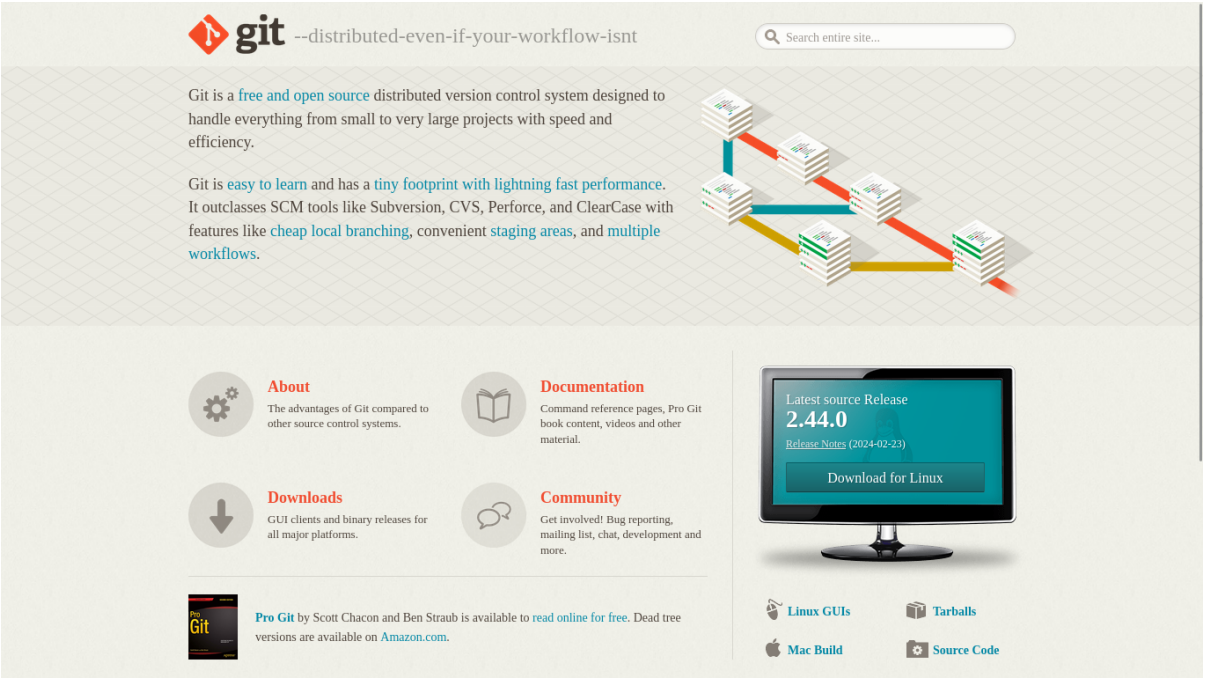


Figure 2.8: Git

1. Visita el sitio web oficial de Git en https://git-scm.com/downloads.
2. Descarga el instalador adecuado para tu sistema operativo y sigue las instrucciones de instalación.

2.8.2 Configuración

Figure 2.9: Configuración de Git

Una vez instalado Git, es necesario configurar tu nombre de usuario y dirección de correo electrónico. Esto se puede hacer mediante los siguientes comandos:

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu@email.com"
```

2.8.3 Creación de un Repositorio "helloWorld" en Python

- Crea una nueva carpeta para tu proyecto y ábrela en Visual Studio Code.
- Crea un archivo Python llamado **hello_world.py** y escribe el siguiente código:

```
def welcome_message():
    name = input("Ingrese su nombre: ")
    print("Bienvenio,", name, "al curso de Django y React!")

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

- Guarda el archivo y abre una terminal en Visual Studio Code.
- Inicializa un repositorio Git en la carpeta de tu proyecto con el siguiente comando:

```
git init
```

- Añade el archivo al área de preparación con:

```
git add hello_world.py
```

- Realiza un commit de los cambios con un mensaje descriptivo:

```
git commit -m "Añadir archivo hello_world.py"
```

2.8.4 Comandos Básicos de Git

- **git init:** Inicializa un nuevo repositorio Git.
- **git add :** Añade un archivo al área de preparación.
- **git commit -m “”**: Realiza un commit de los cambios con un mensaje descriptivo.
- **git push:** Sube los cambios al repositorio remoto.
- **git pull:** Descarga cambios del repositorio remoto.
- **git branch:** Lista las ramas disponibles.
- **git checkout :** Cambia a una rama específica.
- **git merge :** Fusiona una rama con la rama actual.
- **git reset :** Descarta los cambios en un archivo.
- **git diff:** Muestra las diferencias entre versiones.

2.8.5 Estados en Git

- **Local:** Representa los cambios que realizas en tu repositorio local antes de hacer un commit. Estos cambios están únicamente en tu máquina.
- **Staging:** Indica los cambios que has añadido al área de preparación con el comando `git add`. Estos cambios están listos para ser confirmados en el próximo commit.
- **Commit:** Son los cambios que has confirmado en tu repositorio local con el comando `git commit`. Estos cambios se han guardado de manera permanente en tu repositorio local.
- **Server:** Son los cambios que has subido al repositorio remoto con el comando `git push`. Estos cambios están disponibles para otros colaboradores del proyecto.

---

3 Tutorial: Moviendo Cambios entre Estados en Git

3.1 Introducción

En este tutorial, aprenderemos a utilizar Git para gestionar cambios en nuestro proyecto y moverlos entre diferentes estados. Utilizaremos un ejemplo práctico para comprender mejor estos conceptos.

```
def welcome_message():
    name = input("Ingrese su nombre: ")
    print("Bienvenio,", name, "al curso de Django y React!")

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

3.2 Sección 1: Modificar Archivos en el Repositorio

En esta sección, aprenderemos cómo realizar cambios en nuestros archivos y reflejarlos en Git.

3.3 Mover Cambios de Local a Staging:

1. Abre el archivo **hello_world.py** en Visual Studio Code.
2. Modifica el mensaje de bienvenida a “Bienvenido” en lugar de “Bienvenio”.
3. Guarda los cambios y abre una terminal en Visual Studio Code.

Hemos corregido un error en nuestro archivo y queremos reflejarlo en Git.

```
def welcome_message():
    name = input("Ingrese su nombre: ")
    print("Bienvenido,", name, "al curso de Django y React!")

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

3.4 Agregar Cambios de Local a Staging:

```
git add hello_world.py
```

Hemos añadido los cambios al área de preparación y están listos para ser confirmados en el próximo commit.

3.5 Sección 2: Confirmar Cambios en un Commit

En esta sección, aprenderemos cómo confirmar los cambios en un commit y guardarlos de manera permanente en nuestro repositorio.

3.6 Mover Cambios de Staging a Commit:

```
git commit -m "Corregir mensaje de bienvenida"
```

Hemos confirmado los cambios en un commit con un mensaje descriptivo.

3.7 Sección 3: Creación y Fusión de Ramas

En esta sección, aprenderemos cómo crear y fusionar ramas en Git para desarrollar nuevas funcionalidades de forma aislada.

3.8 Crear una Nueva Rama:

```
git branch feature
```

Hemos creado una nueva rama llamada "feature" para desarrollar una nueva funcionalidad.

3.9 Implementar Funcionalidades en la Rama:

1. Abre el archivo **hello_world.py** en Visual Studio Code.
2. Añade una nueva función para mostrar un mensaje de despedida.
3. Guarda los cambios y abre una terminal en Visual Studio Code.
4. Añade los cambios al área de preparación y confírmalos en un commit.
5. Cambia a la rama principal con `git checkout main`.

3.10 Fusionar Ramas con la Rama Principal:

```
git merge feature
```

Hemos fusionado la rama “feature” con la rama principal y añadido la nueva funcionalidad al proyecto.

3.11 Sección 4: Revertir Cambios en un Archivo

En esta sección, aprenderemos cómo revertir cambios en un archivo y deshacerlos en Git.

3.12 Revertir Cambios en un Archivo:

```
git reset hello_world.py
```

Hemos revertido los cambios en el archivo **hello_world.py** y deshecho las modificaciones realizadas.

3.13 Conclusión

En este tutorial, hemos aprendido a gestionar cambios en nuestro proyecto y moverlos entre diferentes estados en Git. Estos conceptos son fundamentales para trabajar de forma eficiente en proyectos de software y colaborar con otros desarrolladores.

4 Asignación

Hello World!

Este proyecto de ejemplo está escrito en Python y se prueba con pytest.

La Asignación

Las pruebas están fallando en este momento porque el método no está devolviendo la cadena correcta. Corrige el código del archivo **hello.py** para que las pruebas sean exitosas, debe devolver la cadena correcta **"Hello World!"**x

El comando de ejecución del test es:

```
pytest test_hello.py
```

¡Mucha suerte!

5 GitHub Classroom

Figure 5.1: Github Classroom

GitHub Classroom es una herramienta poderosa que facilita la gestión de tareas y asignaciones en GitHub, especialmente diseñada para entornos educativos.

5.1 ¿Qué es GitHub Classroom?

Figure 5.2: Github Classroom Windows

GitHub Classroom es una extensión de GitHub que permite a los profesores crear y gestionar asignaciones utilizando repositorios de GitHub. Proporciona una forma organizada y eficiente de distribuir tareas a los estudiantes, recopilar y revisar su trabajo, y proporcionar retroalimentación.

5.1.1 Funcionalidades Principales

Creación de Asignaciones: Los profesores pueden crear tareas y asignaciones directamente desde GitHub Classroom, proporcionando instrucciones detalladas y estableciendo

criterios de evaluación.

Distribución Automatizada: Una vez que se crea una asignación, GitHub Classroom genera automáticamente repositorios privados para cada estudiante o equipo, basándose en una plantilla predefinida.

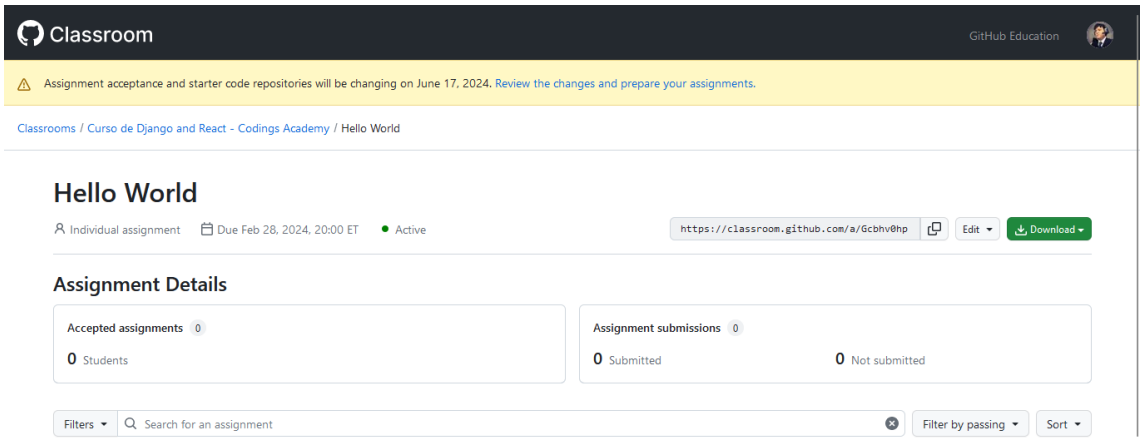
Seguimiento de Progreso: Los profesores pueden realizar un seguimiento del progreso de los estudiantes y revisar sus contribuciones a través de solicitudes de extracción (pull requests) y comentarios en el código.

Revisión y Retroalimentación: Los estudiantes envían sus trabajos a través de solicitudes de extracción, lo que permite a los profesores revisar y proporcionar retroalimentación específica sobre su código.

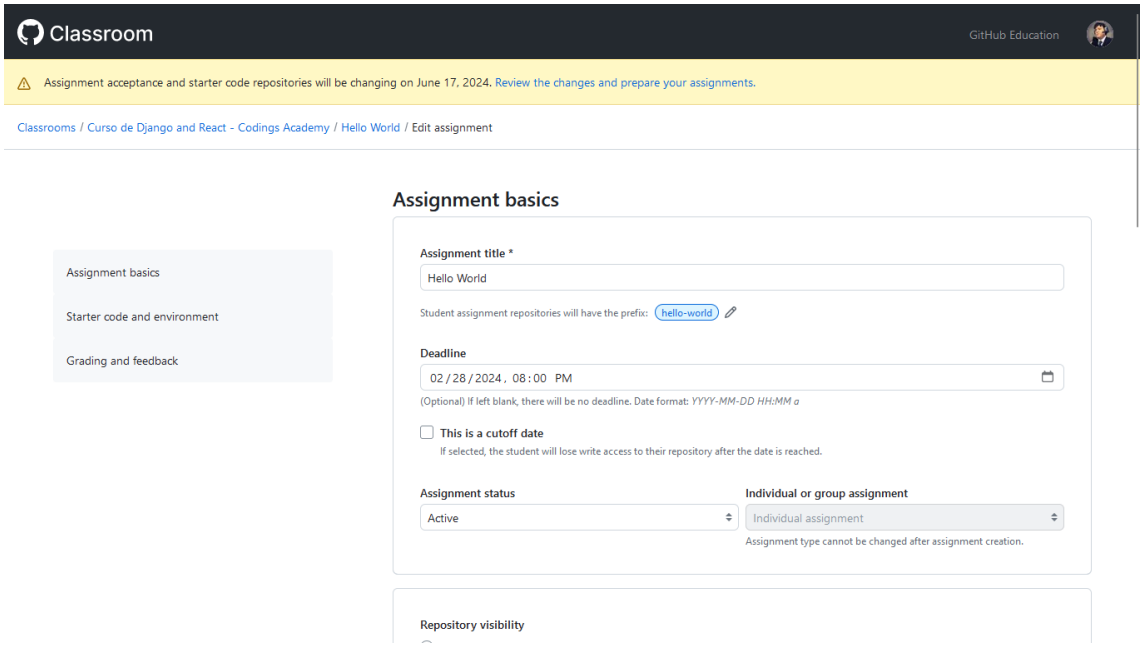
5.2 Ejemplo Práctico

5.2.1 Creación de una Asignación en GitHub Classroom

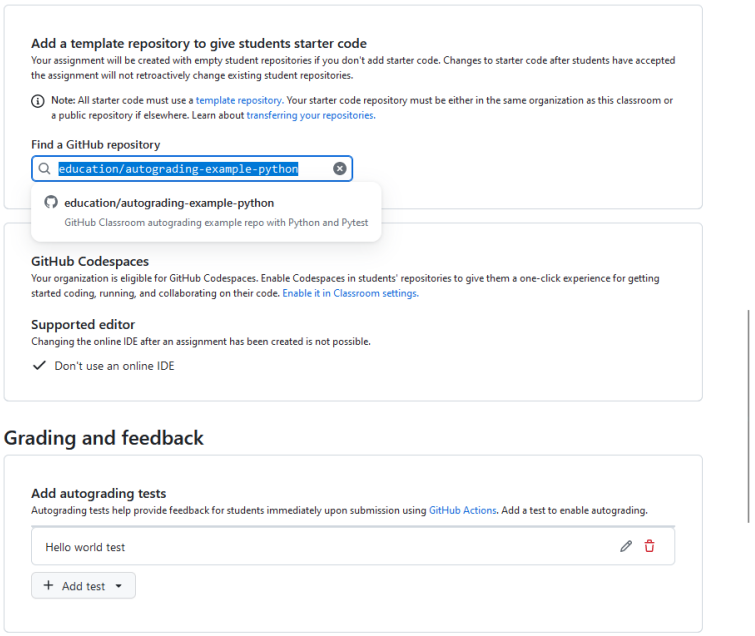
Iniciar Sesión: Ingresa a GitHub Classroom con tu cuenta de GitHub y selecciona la opción para crear una nueva asignación.



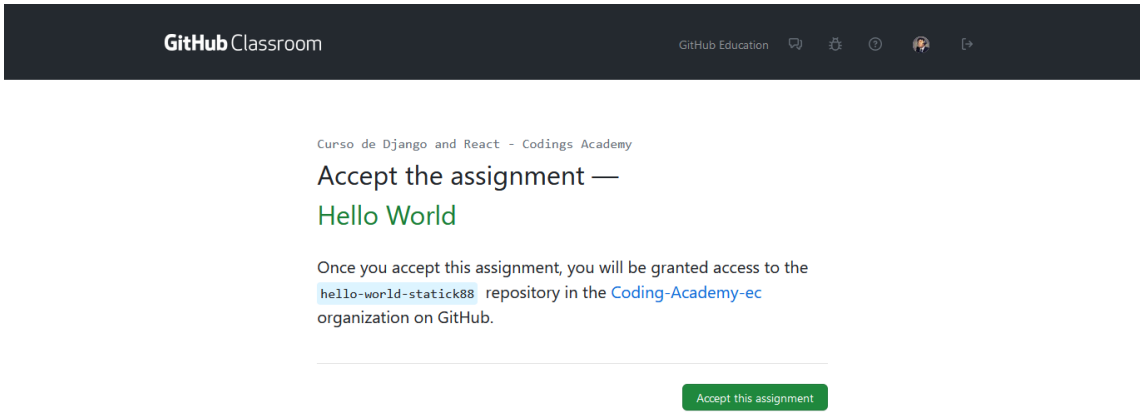
Definir la Tarea: Proporciona instrucciones claras y detalladas sobre la tarea, incluyendo cualquier código base o recursos necesarios. Establece los criterios de evaluación para guiar a los estudiantes.



Configurar la Plantilla: Selecciona una plantilla de repositorio existente o crea una nueva plantilla que servirá como base para los repositorios de los estudiantes.

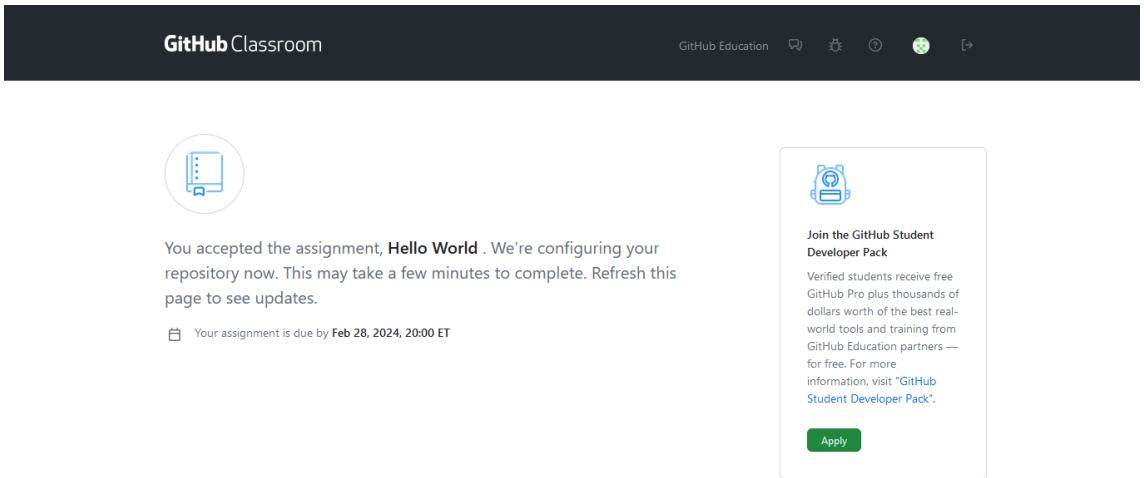


Distribuir la Asignación: Una vez configurada la asignación, comparte el enlace generado con tus estudiantes para que puedan acceder a sus repositorios privados.

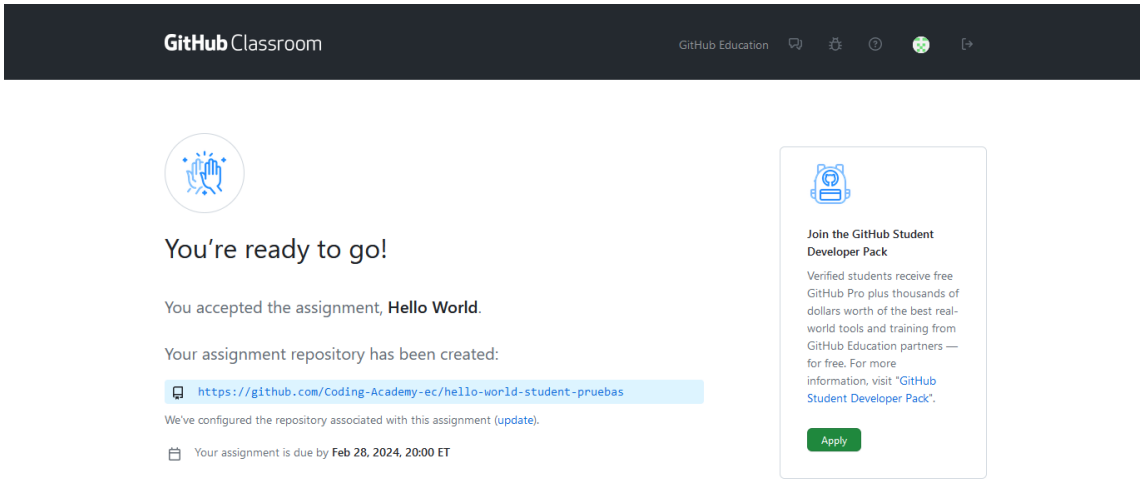


5.3 Trabajo de los Estudiantes

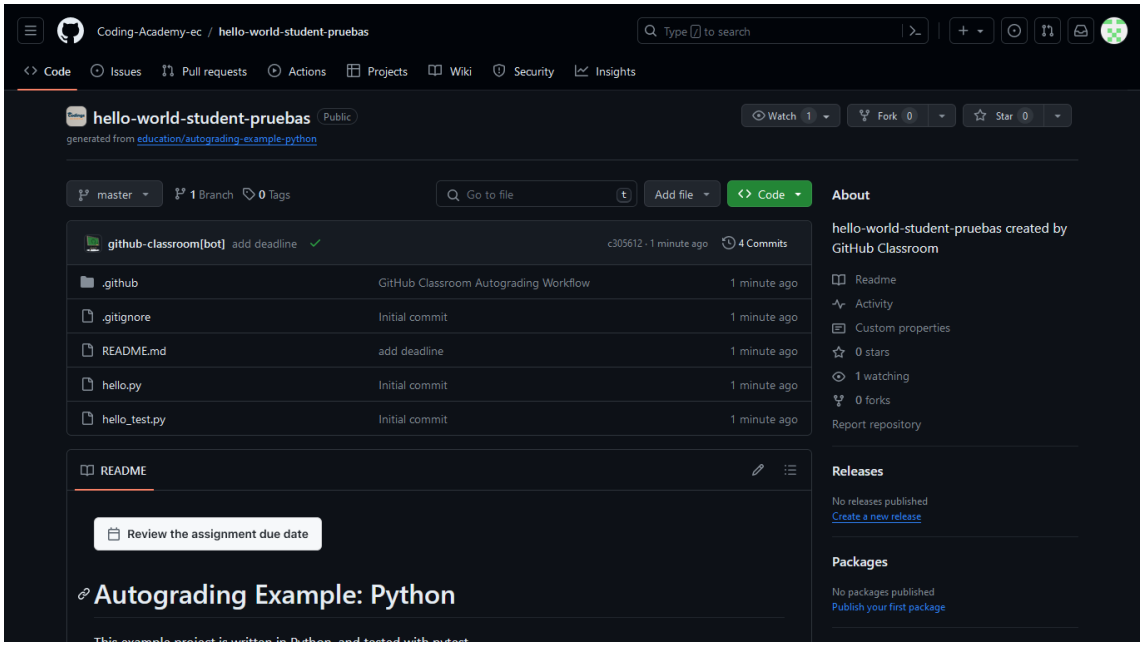
Aceptar la Asignación: Los estudiantes reciben el enlace de la asignación y aceptan la tarea, lo que les permite crear un repositorio privado basado en la plantilla proporcionada.



Actualizar el Navegador: Los estudiantes actualizan su navegador para ver el nuevo repositorio creado en su cuenta de GitHub.



Clonar el Repositorio: Los estudiantes clonan el repositorio asignado en su computadora local utilizando el enlace proporcionado.



Utilizar el comando git clone: Aplique el comando git clone para clonar el repositorio en su computadora local.

```
git clone <enlace-del-repositorio>
```

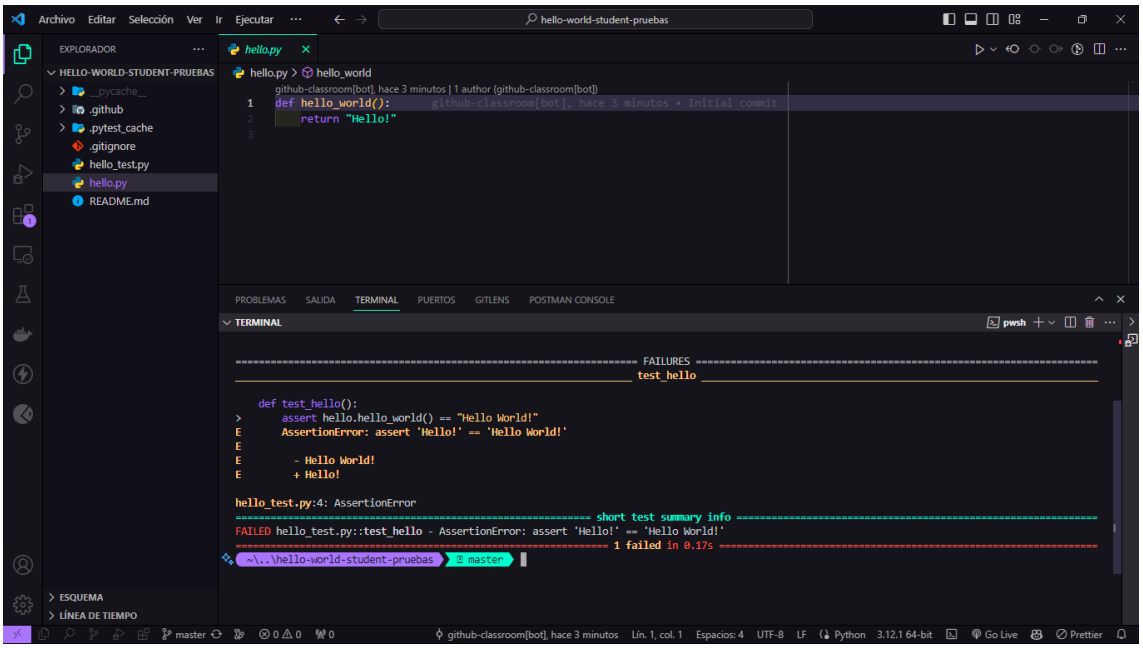
```
Desktop :: pwsh
~\Desktop git clone https://github.com/Coding-Academy-ec/hello-world-student-pruebas.git
Cloning into 'hello-world-student-pruebas'...
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 19 (delta 4), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (19/19), 4.69 KiB | 1.17 MiB/s, done.
Resolving deltas: 100% (4/4), done.
```

Desarrollar la Tarea: Los estudiantes trabajan en la tarea, realizando los cambios necesarios y realizando commits de manera regular para mantener un historial de su trabajo.

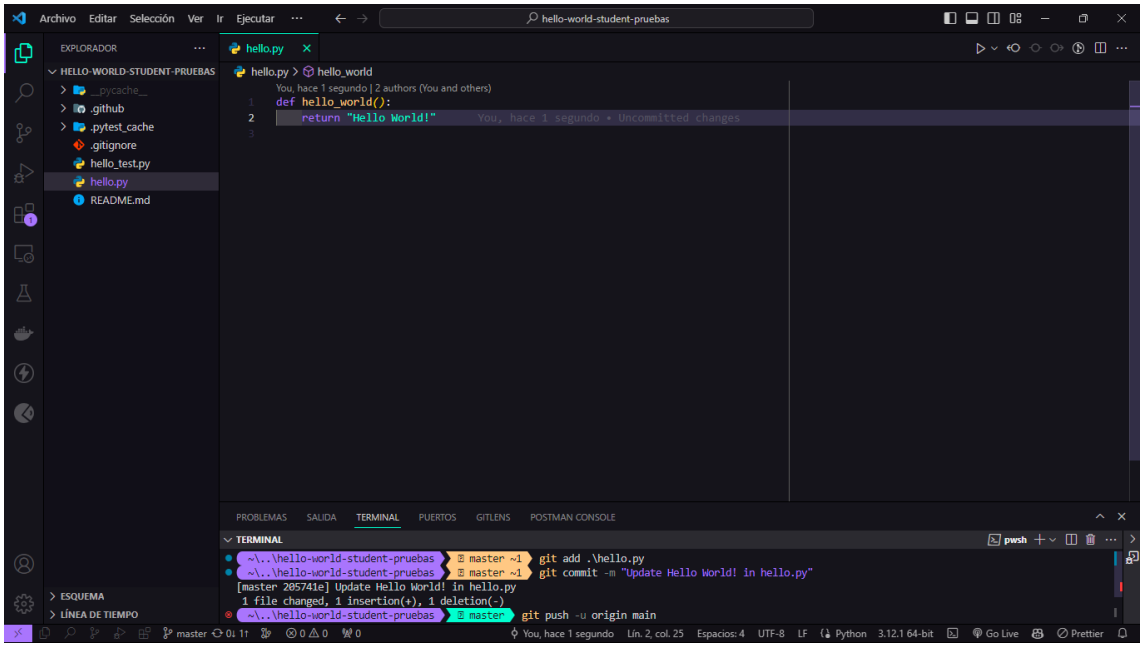
> **Tip**
>
> Puedes probar el test incorporado con el comando `pytest` en la terminal, para verificar que el código cumple con los requerimientos

```
pytest
```

Una vez desarrollado el código de acuerdo a la asignación en local deberían pasar el o los test

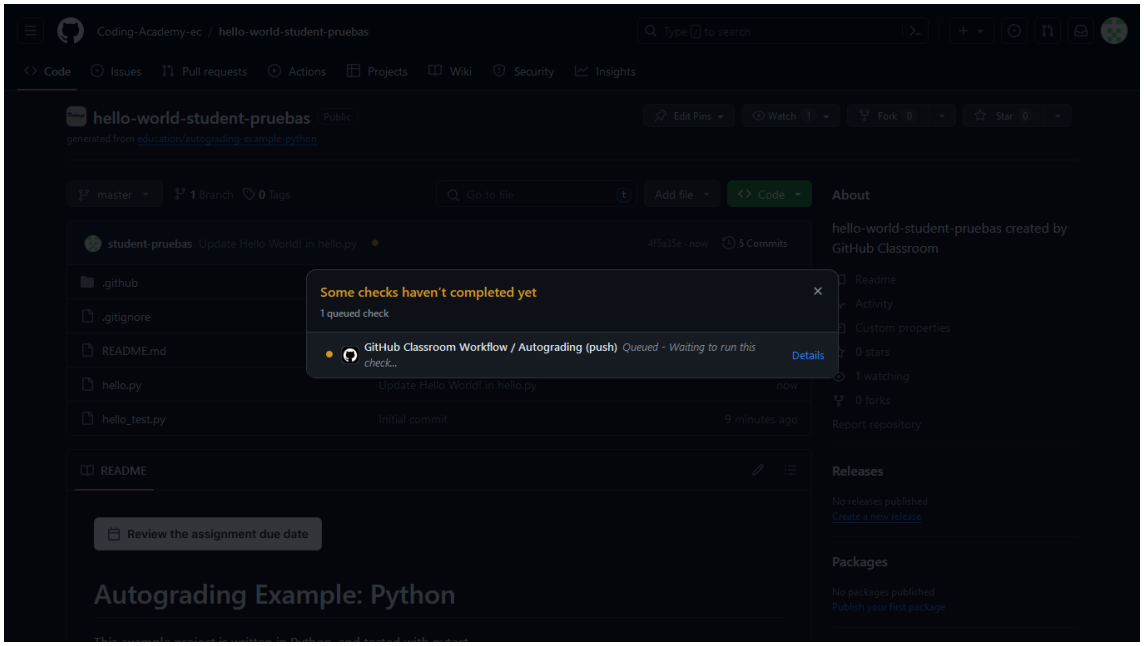


Enviar la Solicitud de Extracción: Una vez completada la tarea, los estudiantes envían una solicitud de extracción desde su rama hacia la rama principal del repositorio, solicitando la revisión del profesor.

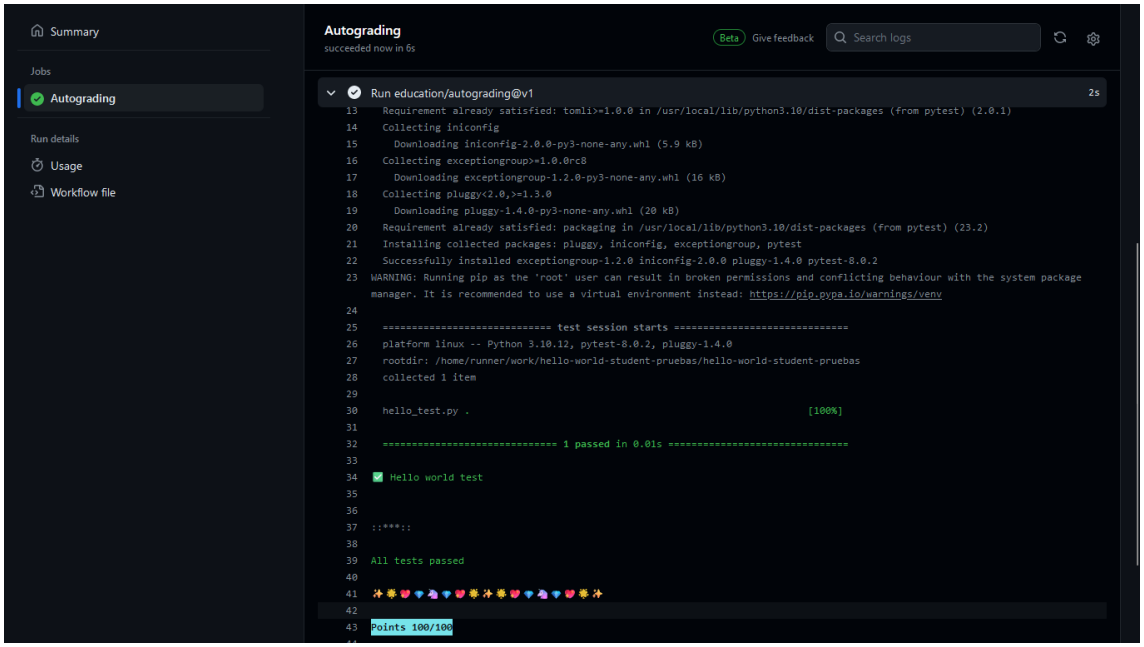


Una vez realizado el `push` se envía al respositorio principal y se ejecutan los test en Github

> **Tip**
>
> Se recomienda hacer las pruebas en local antes de enviar los cambios al respositorio en Github

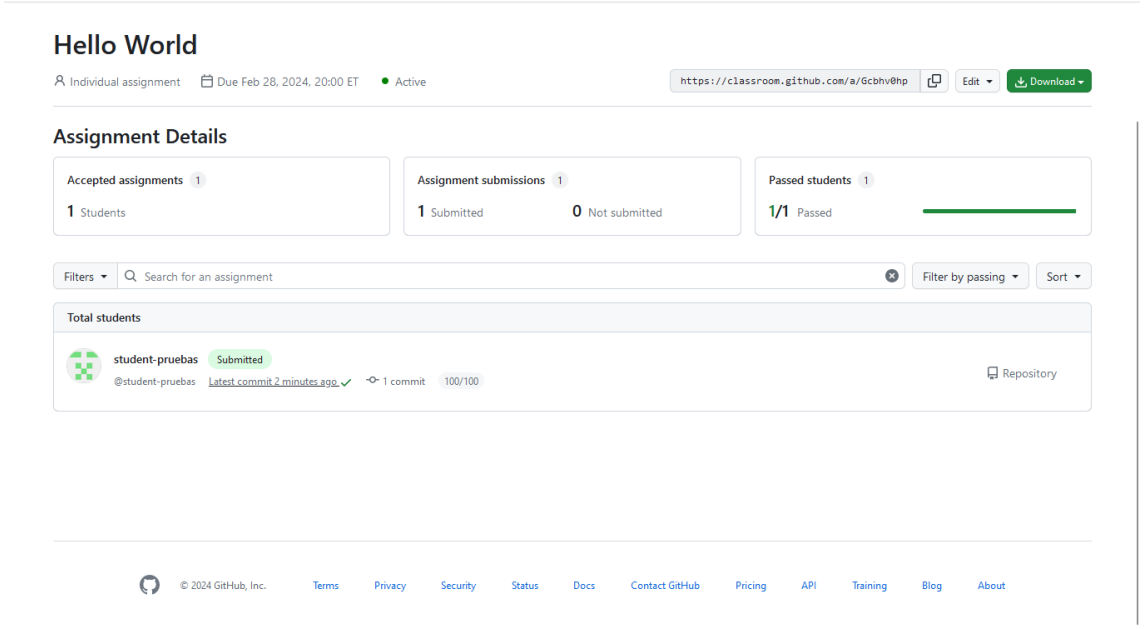


Este Action lo que hace es evaluar los cambios realizados



> **Tip**
>
> Se recomienda hacer las pruebas en local antes de enviar los cambios al respositorio en Github

Revisión y Retroalimentación: Los profesores revisan las solicitudes de extracción, proporcionan comentarios sobre el código y evalúan el trabajo de los estudiantes según los criterios establecidos.



Tip

GitHub Classroom ofrece una manera eficiente y organizada de administrar tareas y asignaciones en entornos educativos, fomentando la colaboración, el aprendizaje y la retroalimentación efectiva entre profesores y estudiantes.

6 Docker

Figure 6.1: Docker

Docker es una plataforma de código abierto que permite a los desarrolladores empaquetar, distribuir y ejecutar aplicaciones en contenedores. Los contenedores son entornos ligeros y portátiles que incluyen todo lo necesario para ejecutar una aplicación de forma consistente en cualquier entorno.

7 Conceptos Básicos de Docker

7.1 Imagen

```
docker pull python:3.9-slim
```

Una imagen de Docker es un paquete de software ligero y portátil que incluye todo lo necesario para ejecutar una aplicación, incluidos el código, las bibliotecas y las dependencias. Las imágenes se utilizan como plantillas para crear contenedores.

7.2 Contenedor

```
docker run -d -p 5000:5000 myapp
```

Un contenedor de Docker es una instancia en tiempo de ejecución de una imagen de Docker. Los contenedores son entornos aislados que ejecutan aplicaciones de forma independiente y comparten recursos del sistema operativo subyacente. Cada contenedor está aislado del entorno de host y otros contenedores, lo que garantiza la consistencia y la portabilidad de las aplicaciones.

7.3 Dockerfile

```
# Dockerfile
# Define la imagen base
FROM python:3.9-slim

# Instala las dependencias necesarias
RUN apt-get update && apt-get install -y \
    build-essential \
    libpq-dev \
    libffi-dev \
    && rm -rf /var/lib/apt/lists/*

# Establece el directorio de trabajo
WORKDIR /app
```

```
# Copia los archivos de la aplicación al contenedor
COPY . .

# Instala las dependencias de Python
RUN pip install --no-cache-dir -r requirements.txt

# Establece el comando por defecto para ejecutar la aplicación
CMD ["python", "app.py"]
```

Un Dockerfile es un archivo de texto que contiene instrucciones para construir una imagen de Docker. Especifica qué software se instalará en la imagen y cómo configurar el entorno de ejecución. Los Dockerfiles permiten a los desarrolladores definir de manera reproducible el entorno de ejecución de sus aplicaciones y automatizar el proceso de construcción de imágenes.

7.4 Docker Compose

```
# docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      FLASK_ENV: development
```

Docker Compose es una herramienta que permite definir y ejecutar aplicaciones Docker multi-contenedor. Permite gestionar la configuración de varios contenedores como un solo servicio, lo que facilita el despliegue y la gestión de aplicaciones complejas que constan de múltiples componentes.

8 Uso de Docker

8.1 Definir un Dockerfile

```
# Dockerfile
# Define la imagen base
FROM python:3.9-slim

# Instala las dependencias necesarias
RUN apt-get update && apt-get install -y \
    build-essential \
    libpq-dev \
    libffi-dev \
    && rm -rf /var/lib/apt/lists/*

# Establece el directorio de trabajo
WORKDIR /app

# Copia los archivos de la aplicación al contenedor
COPY . .

# Instala las dependencias de Python
RUN pip install --no-cache-dir -r requirements.txt

# Establece el comando por defecto para ejecutar la aplicación
CMD ["python", "app.py"]
```

Para utilizar Docker, primero se crea un Dockerfile que especifica cómo construir la imagen de Docker, incluidas las dependencias y la configuración del entorno. El Dockerfile define las capas de la imagen y las instrucciones para configurar el entorno de ejecución de la aplicación.

8.2 Construir la Imagen

```
docker build -t myapp .
```

Una vez que se tiene el Dockerfile, se utiliza el comando docker build para construir la imagen de Docker a partir del Dockerfile. Este comando lee las instrucciones del Dockerfile

y crea una imagen en función de esas instrucciones. La imagen resultante se puede utilizar para crear y ejecutar contenedores.

8.3 Ejecutar un Contenedor

```
docker run -d -p 5000:5000 myapp
```

Después de construir la imagen, se ejecuta un contenedor utilizando el comando docker run, especificando la imagen que se utilizará y cualquier configuración adicional necesaria, como puertos expuestos, variables de entorno y volúmenes montados. El contenedor se ejecuta en un entorno aislado y se puede acceder a través de la red local o de Internet, según la configuración.

8.4 Gestionar Contenedores

```
docker ps
docker stop <container_id>
docker rm <container_id>
```

Docker proporciona varios comandos para gestionar contenedores, como docker ps para ver contenedores en ejecución, docker stop para detener un contenedor y docker rm para eliminar un contenedor. Estos comandos permiten a los usuarios administrar y controlar el ciclo de vida de los contenedores de manera eficiente.

8.5 Docker Compose

```
# docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      FLASK_ENV: development
```

Para aplicaciones más complejas que requieren múltiples contenedores, se utiliza Docker Compose para definir y gestionar la configuración de los contenedores en un archivo

YAML. Luego, se utiliza el comando docker-compose para gestionar los servicios definidos en el archivo YAML, lo que simplifica el despliegue y la gestión de aplicaciones multi-contenedor.

Part II

Unidad 2: Python Básico

9 Hola Mundo en Python

10 Introdución

En este tutorial aprenderemos los conceptos básicos necesarios para configurar nuestro entorno de desarrollo y escribir código en Python. Comenzaremos con la instalación de Python en Windows y luego veremos cómo escribir y ejecutar nuestro primer programa en Python utilizando Visual Studio Code como nuestro editor de texto.

10.0.1 Paso 1: Instalación de Python

Para poder escribir y ejecutar programas en Python, primero necesitamos instalar Python en nuestra computadora. Python es un lenguaje de programación de alto nivel que es ampliamente utilizado en el desarrollo de aplicaciones web, desarrollo de software, análisis de datos, inteligencia artificial, etc.

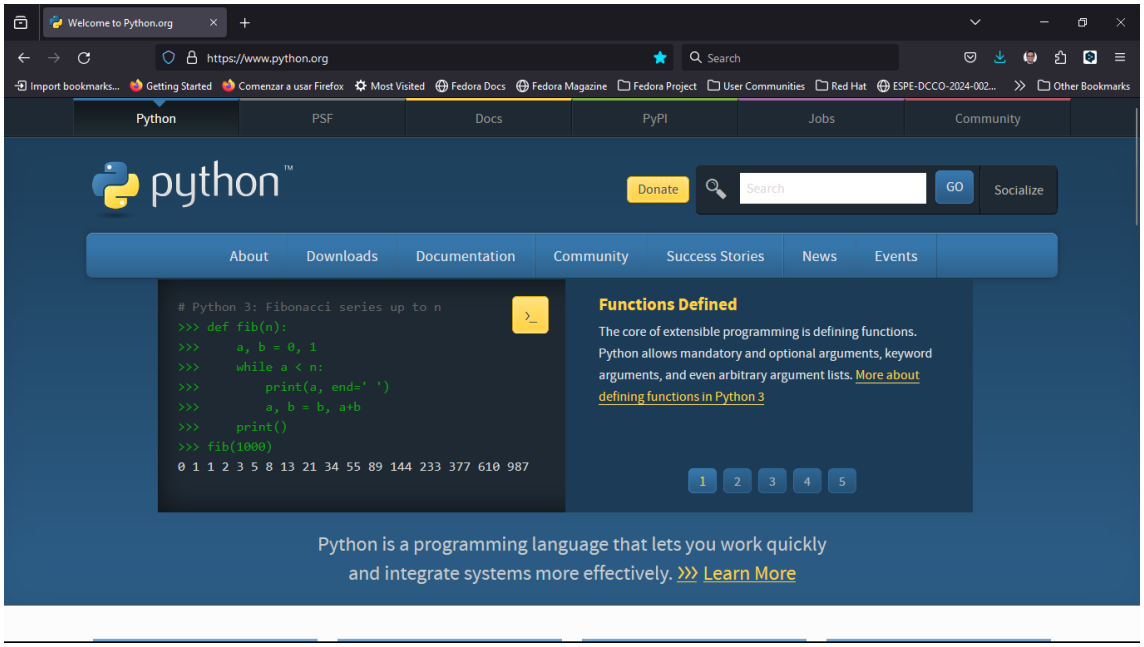
> **Note**
>
> Python se puede instalar en Windows, Mac y Linux. En este tutorial, veremos cómo instalar Python en Windows.

10.0.2 Paso 2: Instalación de Python en Windows

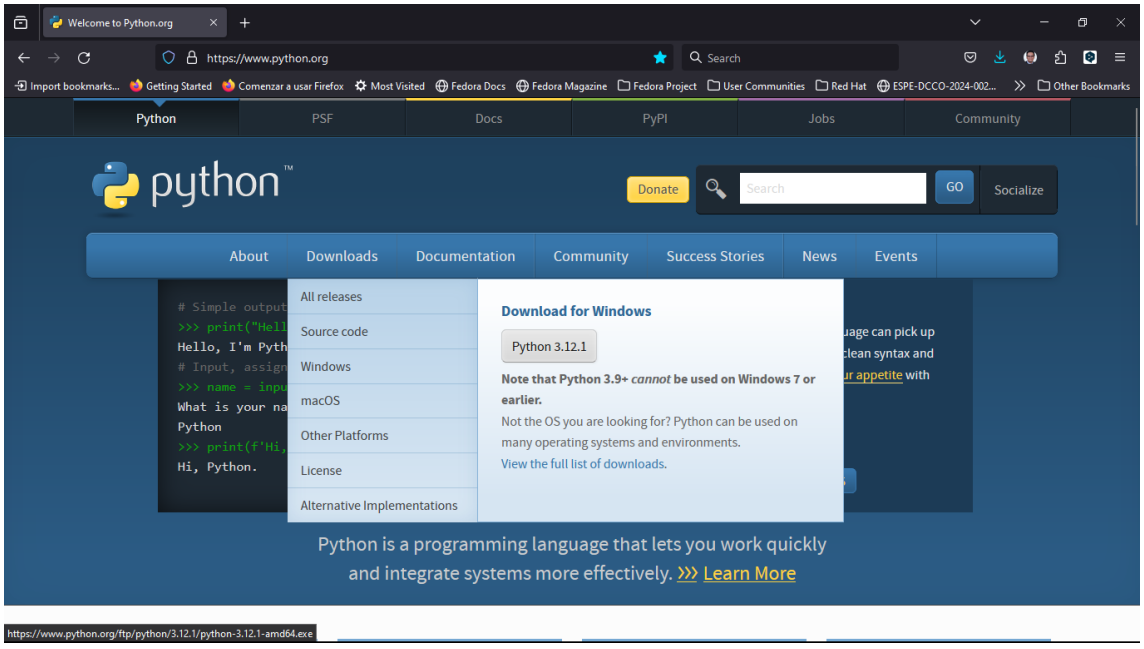
1. **Descargar Python**

Para instalar Python en Windows, primero necesitamos descargar el instalador de Python desde el sitio web oficial de Python. Para hacer esto, abra su navegador web y vaya a la página de descargas de Python en el siguiente enlace:

https://www.python.org/downloads/



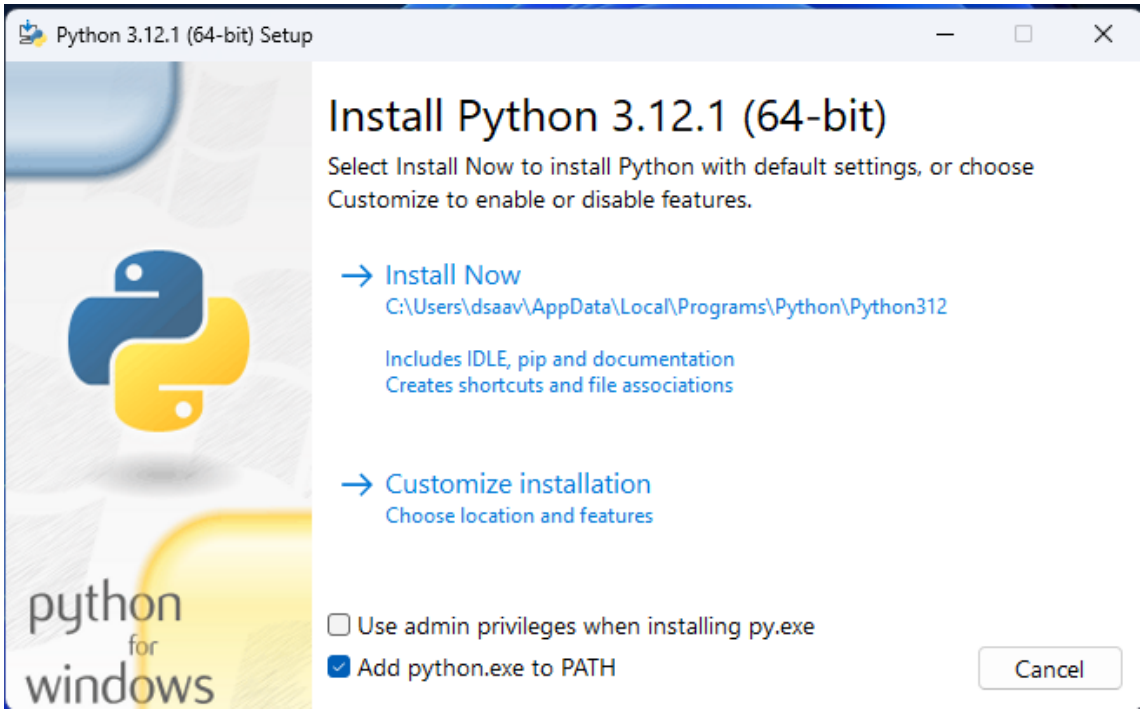
En la página de descargas, haga clic en el botón de descarga para la última versión de Python. En el momento de escribir este tutorial, la última versión de Python es 3.12.1



Excelente, ahora que hemos descargado el instalador de Python, podemos continuar con la instalación de Python en Windows.

2. **Instalar Python**

Una vez que el instalador de Python se haya descargado, haga doble clic en el archivo de instalación para iniciar el proceso de instalación. Asegúrese de marcar la casilla que dice "Add Python 3.12 to PATH" antes de hacer clic en el botón "Install Now".



Ahora que hemos instalado Python en nuestra computadora, podemos continuar con la configuración de nuestro entorno de desarrollo para escribir y ejecutar programas en Python.

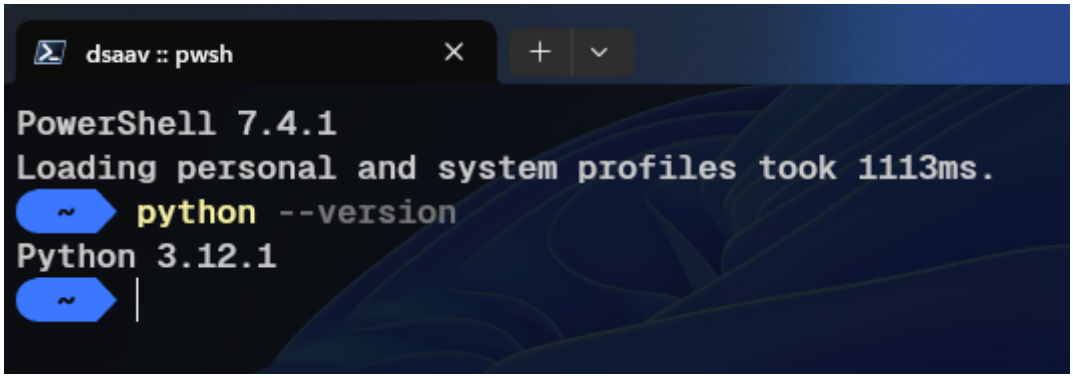
3. **Comprobar que tenemos instalado Python**

Para comprobar que Python se ha instalado correctamente en nuestra computadora, abra una ventana de comandos y escriba el siguiente comando:

```
python --version
```

4. **Impresion de la versión de Python**

Este comando imprimirá la versión de Python que está instalada en su computadora. En mi caso, la versión de Python es 3.12.1.



10.0.3 Paso 3: Crear nuestro primer “Hola Mundo” en Python .

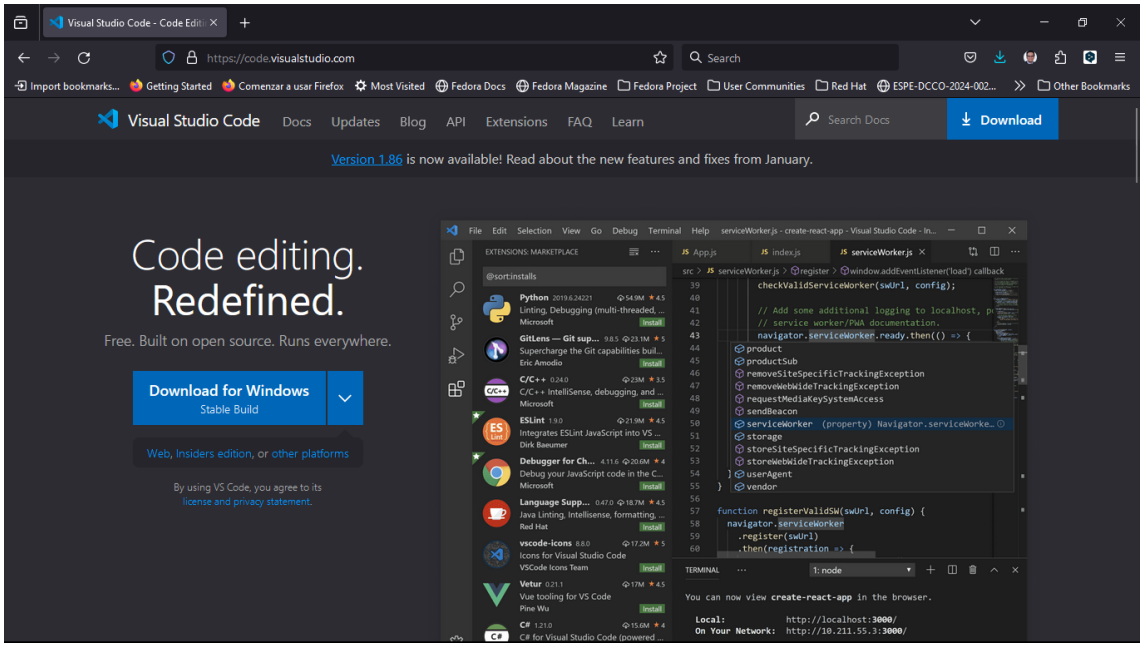
Ahora que hemos instalado Python en nuestra computadora, podemos comenzar a escribir y ejecutar programas en Python. Para hacer esto, necesitamos un editor de texto para escribir nuestro código y un intérprete de Python para ejecutar nuestro código.

En este tutorial, usaremos Visual Studio Code como nuestro editor de texto y el intérprete de Python que instalamos en el paso anterior.

1. **Instalar Visual Studio Code**

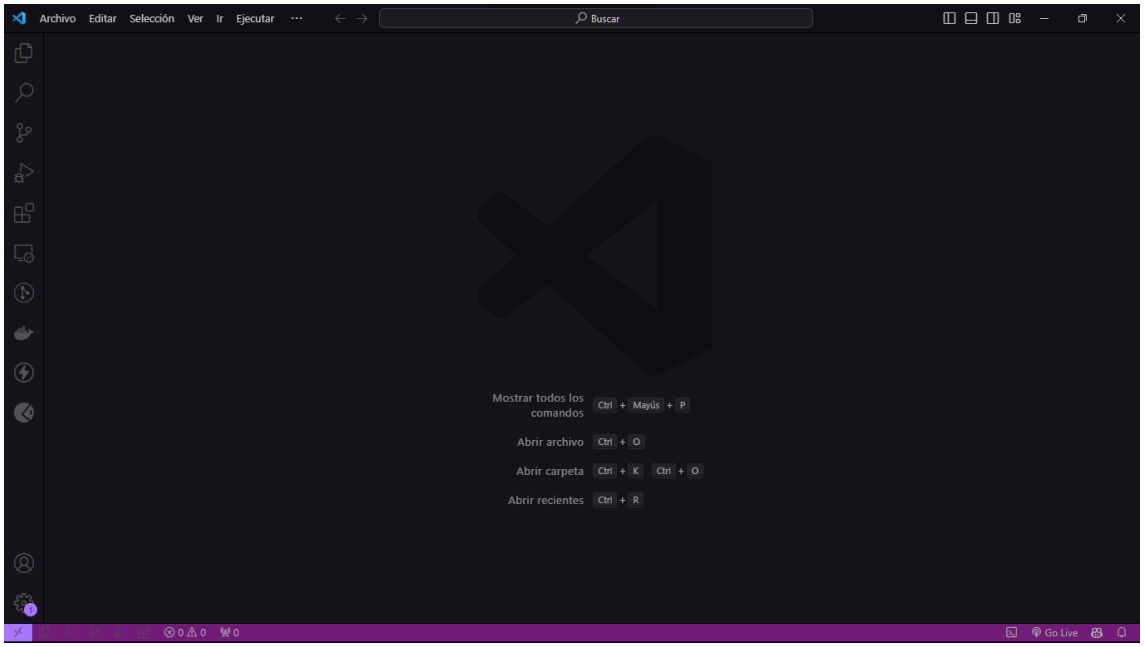
Para instalar Visual Studio Code, vaya al sitio web oficial de Visual Studio Code en el siguiente enlace:

https://code.visualstudio.com/



En la página de descargas, haga clic en el botón de descarga para su sistema operativo. En el momento de escribir este tutorial, la última versión de Visual Studio Code es 1.85.2.

Una vez que el instalador de Visual Studio Code se haya descargado, haga doble clic en el archivo de instalación para iniciar el proceso de instalación. Siga las instrucciones en pantalla para completar la instalación.

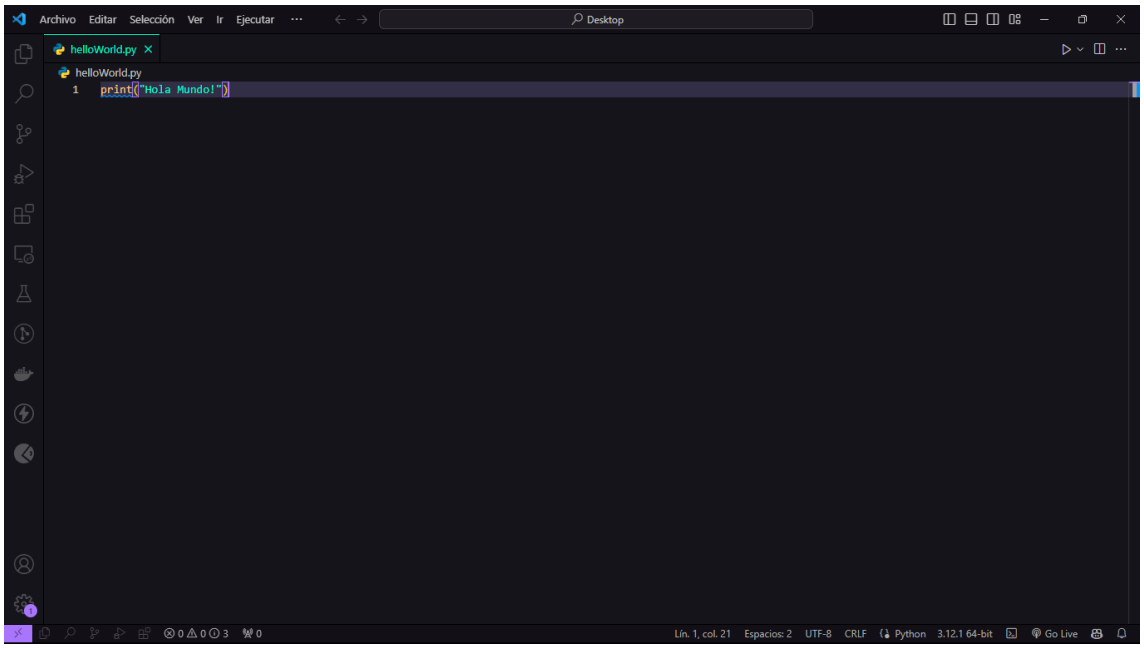


2. **Crear un nuevo archivo de Python**

Para crear un nuevo archivo de Python en Visual Studio Code, abra Visual Studio Code y haga clic en el botón “New File” en la barra de herramientas. Luego, escriba el siguiente código en el archivo:

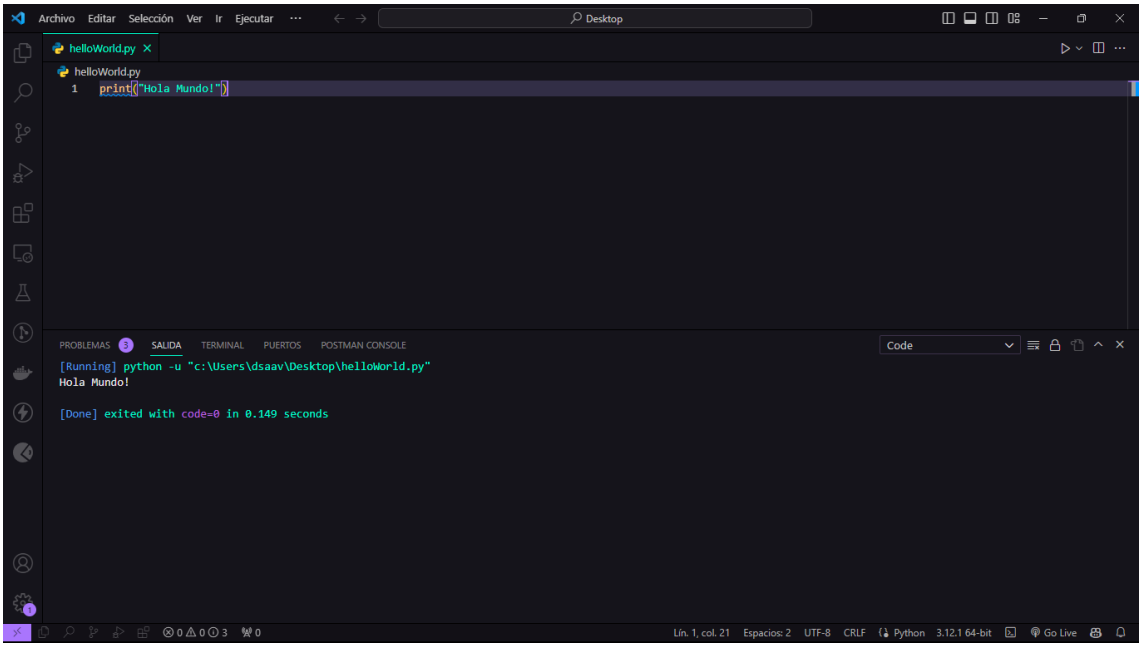
```
print("Hola Mundo")
```

3. Este código imprimirá “Hola Mundo” en la consola.



4. Ejecutar el programa

Para ejecutar el programa, haga clic en el botón “Run” en la barra de herramientas. Esto ejecutará el programa y mostrará “Hola Mundo” en la consola.



¡Felicidades!

Acabas de escribir y ejecutar tu primer programa en Python. Ahora que has configurado tu entorno de desarrollo y has escrito tu primer programa en Python, puedes comenzar a explorar el lenguaje de programación Python y aprender a escribir programas más complejos.

11 Sintaxis Básica

Figure 11.1: Python

```
if 5 > 2:
  print("Cinco es mayor que dos")                                    ①
```

① En el ejemplo anterior, la instrucción **print** está indentada, es decir, tiene un espacio al principio de la línea. Esto es necesario para que el código funcione correctamente.

Además de la indentación, en python se utilizan los dos puntos **:** para indicar que se va a escribir un bloque de código.

```
if 5 > 2:                                                            ①
  print("Cinco es mayor que dos")
```

① En este caso, el **:** indica que se va a escribir un bloque de código, el bloque de código que se ejecutará si la condición es verdadera.

12 Comentarios

Los comentarios en python se escriben con el símbolo #.

```
# Este es un comentario
print("Hola Mundo")                                                    ①
```

① En este caso, el comentario está al final de la línea de código.

13 Variables y Tipos de Datos

> **Tip**
>
> Existe la forma de declarar variables de python con su tipo de dato
> Ejemplo:

```
x: int = 5 ①
y: str = "Hola Mundo" ②
```

① En este caso, la variable **x** es de tipo entero.
② En este caso, la variable **y** es de tipo cadena de texto.

En python, las variables se definen de la siguiente manera:

```
x = 5 ①
y = "Hola Mundo" ②
```

① En este caso, la variable **x** es de tipo entero.
② En este caso, la variable **y** es de tipo cadena de texto.

14 Tipos de Datos

En python, los tipos de datos más comunes son:

- **int**: Entero

Ejemplo:

```
x = 5 ①
```

① La variable **x** es de tipo entero.

- **float**: Flotante

Ejemplo:

```
y = 5.0 ①
```

① La variable **y** es de tipo flotante.

- **str**: Cadena de texto

Ejemplo:

```
z = "Hola Mundo" ①
```

- **bool**: Booleano

Ejemplo:

```
a = True ①
```

① La variable **a** es de tipo booleano.

- **list**: Lista

Ejemplo:

```
b = [1, 2, 3] ①
```

- **tuple**: Tupla

Ejemplo:

```
c = (1, 2, 3)                                                    ①
```

- **dict**: Diccionario

Ejemplo:

```
d = {"nombre": "Juan", "edad": 25}                               ①
```

- **set**: Conjunto

Ejemplo:

```
e = {1, 2, 3}                                                    ①
```

① La variable **e** es de tipo conjunto.

- **None**: Nulo

Ejemplo:

```
f = None                                                         ①
```

① La variable **f** es de tipo **None**.

15 Operadores

En python, los operadores más comunes son:

- +: Suma

Ejemplo:

```
x = 5
y = 2
z = x + y                                                    ①
```

① La variable **z** es igual a la suma de **x** y **y**.

- **-**: Resta

Ejemplo:

```
x = 5
y = 2
a = x - y                                                    ①
```

① La variable **a** es igual a la resta de **x** y **y**.

- **“*“**: Multiplicación

Ejemplo:

```
x = 5
y = 2
b = x * y                                                    ①
```

① La variable **b** es igual a la multiplicación de **x** y **y**.

- /: División

Ejemplo:

```
x = 5
y = 2
c = x / y                                                    ①
```

① La variable **c** es igual a la división de **x** y **y**.

- //: División entera

Ejemplo:

```
x = 5
y = 2
d = x // y                                                    (1)
```

(1) La variable **d** es igual a la división entera de **x** y **y**.

- %: Módulo

Ejemplo:

```
x = 5
y = 2
e = x % y                                                     (1)
```

(1) La variable **e** es igual al módulo de **x** y **y**.

- “”**: Potencia

Ejemplo:

```
x = 5
y = 2
f = x ** y                                                    (1)
```

(1) La variable **f** es igual a la potencia de **x** y **y**.

- ==: Igualdad

Ejemplo:

```
x = 5
y = 2
g = x == y                                                    (1)
```

(1) La variable **g** es igual a la comparación de igualdad entre **x** y **y**.

- **!=**: Diferente

Ejemplo:

```
x = 5
y = 2
h = x != y                                                    (1)
```

(1) La variable **h** es igual a la comparación de diferencia entre **x** y **y**.

- >: Mayor que

Ejemplo:

```
x = 5
y = 2
i = x > y                                                    (1)
```

- <: Menor que

Ejemplo:

```
x = 5
y = 2
j = x < y                                                    (1)
```

(1) La variable **j** es igual a la comparación de menor que entre **x** y **y**.

- >=: Mayor

Ejemplo:

```
x = 5
y = 2
k = x >= y                                                   (1)
```

(1) La variable **k** es igual a la comparación de mayor o igual que entre **x** y **y**.

- <=: Menor

Ejemplo:

```
x = 5
y = 2
l = x <= y                                                   (1)
```

(1) La variable **l** es igual a la comparación de menor o igual que entre **x** y **y**.

- **and**: Y

Ejemplo:

```
x = 5
y = 2
m = x and y                                                  (1)
```

- **or**: O

Ejemplo:

```
x = 5
y = 2
n = x or y                                                   (1)
```

① La variable **n** es igual a la comparación lógica **or** entre **x** y **y**.

- **not**: Negación

Ejemplo:

```
x = 5
o = not x                                                    ①
```

① La variable **o** es igual a la negación de **x**.

16 Estructura de Control

En python, las estructuras de control más comunes son:

- **if**: Si

Ejemplo:

```
if 5 > 2:                                    ①
  print("Cinco es mayor que dos")            ②
```

① En este caso, se evalúa si 5 es mayor que 2.
② Si la condición es verdadera, se imprime el mensaje "Cinco es mayor que dos".

- **elif**: Si no

Ejemplo:

```
x = 5
y = 2
if x > y:                                    ①
  print("X es mayor que Y")                  ②
elif x < y:                                  ③
  print("X es menor que Y")                  ④
```

① En este caso, se evalúa si **x** es mayor que **y**.
② Si la condición es verdadera, se imprime el mensaje "X es mayor que Y".
③ Si la condición anterior es falsa, se evalúa si **x** es menor que **y**.
④ Si la condición es verdadera, se imprime el mensaje "X es menor que Y".

- **else**: Si no

Ejemplo:

```
x = 5
y = 2
if x > y:                                    ①
  print("X es mayor que Y")                  ②
elif x < y:                                  ③
  print("X es menor que Y")                  ④
else:                                        ⑤
  print("X es igual a Y")                    ⑥
```

① En este caso, se evalúa si **x** es mayor que **y**.

② Si la condición es verdadera, se imprime el mensaje “X es mayor que Y”.
③ Si la condición anterior es falsa, se evalúa si **x** es menor que **y**.
④ Si la condición es verdadera, se imprime el mensaje “X es menor que Y”.
⑤ Si las condiciones anteriores son falsas, se ejecuta el bloque de código del **else**.
⑥ En este caso, se imprime el mensaje “X es igual a Y”.

- **for**: Para

Ejemplo:

```
for x in range(5):                ①
  print(x)                        ②
```

① En este caso, se recorre un rango de 0 a 5.
② En cada iteración, se imprime el valor de **x**.

- **while**: Mientras

Ejemplo:

```
x = 0
while x < 5:                      ①
  print(x)                        ②
  x += 1                          ③
```

① En este caso, se evalúa si **x** es menor que 5.
② En cada iteración, se imprime el valor de **x**.
③ En cada iteración, se incrementa el valor de **x**.

- **break**: Romper

Ejemplo:

```
x = 0
while x < 5:                      ①
  print(x)                        ②
  x += 1                          ③
  if x == 3:                      ④
    break                         ⑤
```

① En este caso, se evalúa si **x** es menor que 5.
② En cada iteración, se imprime el valor de **x**.
③ En cada iteración, se incrementa el valor de **x**.
④ En cada iteración, se evalúa si **x** es igual a 3.
⑤ Si la condición es verdadera, se rompe el ciclo.

- **continue**: Continuar

Ejemplo:

```
x = 0
while x < 5:                ①
  x += 1                    ②
  if x == 3:                ③
    continue                ④
  print(x)                  ⑤
```

① En este caso, se evalúa si **x** es menor que 5.
② En cada iteración, se incrementa el valor de **x**.
③ En cada iteración, se evalúa si **x** es igual a 3.
④ Si la condición es verdadera, se continúa con la siguiente iteración.
⑤ En cada iteración, se imprime el valor de **x**.

- **pass**: Pasar

Ejemplo:

```
x = 0
if x == 0:                  ①
  pass                      ②
```

① En este caso, se evalúa si **x** es igual a 0.
② Si la condición es verdadera, no se hace nada.

- **return**: Retornar

Ejemplo:

```
def suma(x, y):             ①
  return x + y              ②
```

① En este caso, se define una función llamada **suma** que recibe dos parámetros **x** y **y**.
② La función retorna la suma de **x** y **y**.

- **def**: Definir

Ejemplo:

```
def suma(x, y):             ①
  return x + y              ②
```

① En este caso, se define una función llamada **suma** que recibe dos parámetros **x** y **y**.
② La función retorna la suma de **x** y **y**.

- **try**: Intentar

Ejemplo:

```
try:                                          ①
  print(x)                                    ②
except:                                       ③
  print("Ocurrió un error")                   ④
```

① En este caso, se intenta ejecutar el bloque de código.
② Si ocurre un error, se ejecuta el bloque de código del **except**.
③ En este caso, se imprime el mensaje "Ocurrió un error".
④ Si no ocurre un error, se continúa con la ejecución del código.

- **except**: Excepto

Ejemplo:

```
try:                                          ①
  print(x)                                    ②
except:                                       ③
  print("Ocurrió un error")                   ④
```

① En este caso, se intenta ejecutar el bloque de código.
② Si ocurre un error, se ejecuta el bloque de código del **except**.
③ En este caso, se imprime el mensaje "Ocurrió un error".
④ Si no ocurre un error, se continúa con la ejecución del código.

- **finally**: Finalmente

Ejemplo:

```
try:                                          ①
  print(x)                                    ②
except:                                       ③
  print("Ocurrió un error")                   ④
finally:                                      ⑤
  print("Finalizó la ejecución")              ⑥
```

① En este caso, se intenta ejecutar el bloque de código.
② Si ocurre un error, se ejecuta el bloque de código del `except`.
③ En este caso, se imprime el mensaje "Ocurrió un error".
④ Si no ocurre un error, se continúa con la ejecución del código.
⑤ Al finalizar la ejecución del bloque de código, se ejecuta el bloque de código del `finally`.
⑥ En este caso, se imprime el mensaje "Finalizó la ejecución".

- **raise**: Lanzar

Ejemplo:

```
if x < 0:                                     ①
  raise Exception("El número no puede ser negativo")  ②
```

① En este caso, se evalúa si `x` es menor que 0.

② Si la condición es verdadera, se lanza una excepción con el mensaje "El número no puede ser negativo".

- **assert**: Afirmar

Ejemplo:

```
assert x > 0, "El número no puede ser negativo"  ①
```

① En este caso, se evalúa si **x** es mayor que 0.

- **import**: Importar

Ejemplo:

```
import math  ①
```

① En este caso, se importa el módulo **math**.

- **from**: Desde

Ejemplo:

```
from math import sqrt  ①
```

① En este caso, se importa la función **sqrt** del módulo **math**.

- **as**: Como

Ejemplo:

```
import math as m  ①
```

① En este caso, se importa el módulo **math** como **m**.

17 Funciones

En python, las funciones se definen de la siguiente manera:

```
def suma(x, y):                                                    ①
  return x + y                                                     ②
```

① En este caso, se define una función llamada **suma** que recibe dos parámetros **x** y **y**.
② La función retorna la suma de **x** y **y**.

18 Llamada a Funciones

En python, las funciones se llaman de la siguiente manera:

```
z = suma(5, 2)                                                    ①
```

① En este caso, se llama a la función **suma** con los argumentos **5** y **2**.

19 Parámetros y Argumentos

En python, los parámetros y argumentos se definen de la siguiente manera:

```
def suma(x, y):                                                    ①
  return x + y                                                     ②

z = suma(5, 2)                                                     ③
```

① En este caso, se define una función llamada **suma** que recibe dos parámetros **x** y **y**.
② La función retorna la suma de **x** y **y**.
③ En este caso, se llama a la función **suma** con los argumentos **5** y **2**.

20 Retorno

En python, el retorno se realiza de la siguiente manera:

```
def suma(x, y):                                                    ①
  return x + y                                                     ②

z = suma(5, 2)                                                     ③
```

① En este caso, se define una función llamada **suma** que recibe dos parámetros **x** y **y**.
② La función retorna la suma de **x** y **y**.
③ En este caso, se llama a la función **suma** con los argumentos **5** y **2**.

21 Ejemplo

El programa **Calculadora de propinas** es un ejemplo práctico que permite calcular la propina a dejar en un restaurante.

El funcionamiento del programa es el siguiente:

1. El usuario ingresa el monto total de la cuenta del restaurante.
2. Luego, el usuario selecciona el porcentaje de propina que desea dejar. Por ejemplo, puede elegir un 10%, 15% o 20%.
3. El programa calcula la cantidad de propina a partir del monto total de la cuenta y el porcentaje seleccionado.
4. Finalmente, el programa muestra al usuario el monto total de la cuenta, la cantidad de propina a dejar y el monto total a pagar (suma del monto total de la cuenta y la propina).

Este programa es útil para aquellos que deseen calcular rápidamente la cantidad de propina a dejar en un restaurante, sin tener que hacer los cálculos manualmente. Además, puede ser una buena práctica para familiarizarse con el uso de variables y el control de flujo en la programación.

Ver Código

```
# Ejemplo práctico: Calculadora de propinas

def calcular_propina(total, porcentaje):                                    ①
    propina = total * (porcentaje / 100)                                    ②
    return propina                                                          ③

def calcular_total_con_propina(total, porcentaje):
    propina = calcular_propina(total, porcentaje)
    total_con_propina = total + propina
    return total_con_propina

def main():                                                                 ④
    print("Bienvenido a la calculadora de propinas")                        ⑤
    total = float(input("Ingrese el total de la cuenta: "))                 ⑥
    porcentaje = float(input("Ingrese el porcentaje de propina que desea dejar: ")) ⑦

    propina = calcular_propina(total, porcentaje)                           ⑧
    total_con_propina = calcular_total_con_propina(total, porcentaje)       ⑨

    print(f"La propina a dejar es: {propina}")                              ⑩
    print(f"El total con propina es: {total_con_propina}")                  ⑪
```

```
if __name__ == "__main__":                                    (12)
    main()                                                    (13)
```

(1) En este caso, se define una función llamada **calcular_propina** que recibe dos parámetros **total** y **porcentaje**.
(2) La función calcula la propina a partir del total y el porcentaje.
(3) La función retorna la propina.
(4) En este caso, se define una función llamada **main**.
(5) Se imprime un mensaje de bienvenida.
(6) Se solicita al usuario que ingrese el total de la cuenta.
(7) Se solicita al usuario que ingrese el porcentaje de propina que desea dejar.
(8) Se calcula la propina a partir del total y el porcentaje.
(9) Se calcula el total con propina.
(10) Se imprime la propina a dejar.
(11) Se imprime el total con propina.

22 Asignación

A continuación se sugiere realizar los siguientes ejercicios para practicar la sintaxis y estructura de Python.

Ejercicios Python Básico

22.1 Objetivo

El objetivo de este repositorio es proporcionar una serie de ejercicios de Python básico para que los principiantes en Python puedan practicar y adquirir experiencia en la sintaxis y estructura de Python.

22.2 ¿Qué debes hacer?

Debes Completar cada uno de los ejercicios propuetos a continuación cada uno en su respectivo archivo, el objetivo es adquirir práctica en la sintaxis y estructura de Python. Ejercicios

- **Imprimir Nombre:** Un programa simple que imprime tu nombre en la pantalla.
- **Suma de los Números del 1 al 10:** Un programa que calcula la suma de los números del 1 al 10.
- **Datos Personales:** Un programa que almacena tu edad, nombre y estatura en variables y las imprime en pantalla.
- **Par o Impar:** Un programa que determina si un número ingresado por el usuario es par o impar.
- **Área de un Círculo:** Una función que calcula el área de un círculo dado su radio.
- **Suma de Dos Números:** Una función que recibe dos números como argumentos y devuelve su suma.
- **Área de un Círculo con Parámetro:** Modificación de la función de área de un círculo para recibir el radio como parámetro.
- **Conversión de Temperatura:** Un programa que convierte grados Celsius a Fahrenheit y viceversa.

22.3 Pruebas

Cada ejercicio tiene su archivo de prueba en el que se utilizan las aserciones de pytest para verificar su correcto funcionamiento. Si por ejemplo quiero aplicar el test del primer ejercicio debo completar el primer ejercicio y comentar los demás, luego ejecutar el comando

pytest test_1.py para verificar que el programa funciona correctamente, luego continuar con cada uno de ellos e ir aplicando los test, hasta que al final todos los test pasen y completar la tarea

22.4 Ejecución

Para ejecutar cada programa, simplemente ejecute el archivo **programa.py**. Los archivos de prueba se pueden ejecutar con el comando **pytest**.

Part III

Unidad 3: Python Intermedio

23 Listas

Las listas son un tipo de dato que nos permite almacenar diferentes valores, en una sola variable.

- Las listas **son mutables**, es decir, podemos modificar su contenido.

Ejemplo:

```
mi_lista = [1, 2, 3, 4, 5]
```

Ejercicio:

Crear una lista con los números del 1 al 10, y mostrarla en pantalla.

Solución

```
  mi_lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  print(mi_lista)
```

24 Tuplas

Las tuplas son un tipo de dato que nos permite almacenar diferentes valores, en una sola variable.

- Las tuplas **son inmutables**, es decir, no podemos modificar su contenido.

Ejemplo:

```
mi_tupla = (1, 2, 3, 4, 5)
```

Ejercicio:

Crear una tupla con los números del 1 al 10, y mostrarla en pantalla.

Solución

```
  mi_tupla = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  print(mi_tupla)
```

25 Manipulación de Listas y Tuplas

Para modificar una **lista** se puede realizar diferentes operaciones, como agregar, eliminar, modificar, y acceder a los elementos de la lista o tupla.

Ejemplo:

```
mi_lista = [1, 2, 3, 4, 5]          ①
mi_lista.append(6)                  ②
mi_lista.remove(3)                  ③
mi_lista[0] = 10                    ④
print(mi_lista)                     ⑤
```

① Creamos una lista con los números del 1 al 5.
② Agregamos el número 6 al final de la lista.
③ Eliminamos el número 3 de la lista.
④ Modificamos el primer elemento de la lista por el número 10.
⑤ Mostramos la lista en pantalla.

Ejercicio:

Crear una lista con los números del 1 al 10, y mostrarla en pantalla. Luego, agregar el número 11 al final de la lista, y mostrarla en pantalla. Finalmente, eliminar el número 5 de la lista, y mostrarla en pantalla.

Solución

```
mi_lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(mi_lista)
mi_lista.append(11)
print(mi_lista)
mi_lista.remove(5)
print(mi_lista)
```

> **Tip**
>
> La caracteristica principal de las tuplas es que son inmutables, por lo que no se pueden modificar.

26 Funciones ingegradas para Listas y Tuplas

Python cuenta con funciones integradas que nos permiten realizar diferentes operaciones con listas y tuplas.

Ejemplo:

```
mi_lista = [1, 2, 3, 4, 5]
mi_tupla = (1, 2, 3, 4, 5)

print(len(mi_lista))                                                    ①
print(len(mi_tupla))                                                    ②
print(max(mi_lista))                                                    ③
print(max(mi_tupla))                                                    ④
print(min(mi_lista))                                                    ⑤
print(min(mi_tupla))                                                    ⑥
print(sum(mi_lista))                                                    ⑦
```

① Mostramos la longitud de la lista.
② Mostramos la longitud de la tupla.
③ Mostramos el número mayor de la lista.
④ Mostramos el número mayor de la tupla.
⑤ Mostramos el número menor de la lista.
⑥ Mostramos el número menor de la tupla.
⑦ Mostramos la suma de los elementos de la lista.

Ejercicio:

Crear una lista con los números del 1 al 10, y mostrar la longitud de la lista. Luego, mostrar el número mayor y menor de la lista, y finalmente mostrar la suma de los elementos de la lista.

Solución

```
mi_lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(len(mi_lista))
print(max(mi_lista))
print(min(mi_lista))
print(sum(mi_lista))
```

27 Listas Anidadas

Las listas anidadas son listas que contienen otras listas.

Ejemplo:

```
mi_lista = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(mi_lista)
```

Ejercicio:

Crear una lista anidada con los números del 1 al 9, y mostrarla en pantalla.

Solución

```
  mi_lista = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
  print(mi_lista)
```

28 Listas y Tuplas como Argumentos de Funciones

Las listas y tuplas pueden ser utilizadas como argumentos de funciones.

Ejemplo:

```
def mostrar_lista(lista):
    for elemento in lista:
        print(elemento)

mi_lista = [1, 2, 3, 4, 5]
mostrar_lista(mi_lista)
```

Ejercicio:

Crear una función que reciba una lista como argumento, y muestre en pantalla los elementos de la lista.

Solución

```
  def mostrar_lista(lista):
      for elemento in lista:
          print(elemento)

  mi_lista = [1, 2, 3, 4, 5]
  mostrar_lista(mi_lista)
```

29 Listas y Tuplas como Retorno de Funciones

Las listas y tuplas pueden ser utilizadas como retorno de funciones.

Ejemplo:

```
def crear_lista():
    return [1, 2, 3, 4, 5]

mi_lista = crear_lista()
print(mi_lista)
```

Ejercicio:

Crear una función que retorne una lista con los números del 1 al 10, y mostrarla en pantalla.

Solución

```
  def crear_lista():
      return [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  mi_lista = crear_lista()
  print(mi_lista)
```

30 Asignación

https://classroom.github.com/a/207M40z1

Este repositorio contiene una asignación para el curso de programación en Python. La asignación es sobre la manipulación de listas y tuplas en Python.

30.1 Descripción de la Asignación

El archivo ejercicio.py contiene un script que pide al usuario que ingrese una lista de compras. El usuario debe ingresar los elementos de la lista separados por comas. El script luego imprime la lista de compras.

Además, el script contiene una función llamada convertir_lista_a_tupla() que está destinada a convertir la lista de compras en una tupla. Esta función aún no está completa.

30.2 Tarea Pendiente:

- Completar la función convertir_lista_a_tupla() para que convierta la lista de compras en una tupla.

30.3 Cómo Ejecutar el Código

Para ejecutar el test puedes utilizar el siguiente comando en tu terminal:

```
pytest -s
```

Pytest es una biblioteca que facilita la escritura de pruebas en Python. El parámetro -s se utiliza para mostrar la salida de la prueba en la terminal.

30.4 Ejemplo de salida:

```
$ pytest -s
============================================= test session starts ==================
platform linux -- Python 3.8.10, pytest-6.2.4, pluggy-0.13.1
rootdir: /home/user/Documentos/Python/Asignacion_Lista_Compras
collected 1 item

test_ejercicio.py Lista de Compras: [manzanas, peras, plátanos, uvas]

============================================== 1 passed in 0.01s ===================
```

> **Tip**
>
> Se sugiere que practique la sección **Ejercicios Python - Nivel 3** para reforzar los conocimientos adquiridos.

31 Diccionarios

Los diccionarios son una estructura de datos que nos permite almacenar información en pares clave-valor. La clave es un identificador único que nos permite acceder al valor asociado a ella. Los diccionarios son mutables, es decir, podemos modificar su contenido.

Para crear un diccionario, utilizamos llaves **{}** y separamos cada par clave-valor con dos puntos **:**. Las claves y los valores pueden ser de cualquier tipo de dato.

Ejemplo:

```
mi_diccionario = {
    "nombre": "Juan",
    "edad": 25,
    "ciudad": "Bogotá"
}
```

Ejercicio:

- Crear un diccionario con las siguientes claves y valores:
  - "nombre": "Juan"
  - "edad": 25
  - "ciudad": "Bogotá"

Respuesta

```
mi_diccionario = {
    "nombre": "Juan",
    "edad": 25,
    "ciudad": "Bogotá"
}
```

Para acceder a un valor de un diccionario, utilizamos la clave correspondiente entre corchetes [].

Ejemplo:

```
print(mi_diccionario["nombre"])  # Juan

nombre = mi_diccionario["nombre"]
print(nombre)  # Juan
```

Ejercicio:

- Imprimir el valor de la clave “edad” del diccionario **mi_diccionario**.
- Guardar el valor de la clave “ciudad” del diccionario **mi_diccionario** en una variable llamada **ciudad**.
- Imprimir la variable **ciudad**.
- Imprimir el valor de la clave “nombre” del diccionario **mi_diccionario**.
- Guardar el valor de la clave “edad” del diccionario **mi_diccionario** en una variable llamada **edad**.

Respuesta

```
print(mi_diccionario["edad"])  # 25

ciudad = mi_diccionario["ciudad"]

print(ciudad)  # Bogotá

print(mi_diccionario["nombre"])  # Juan

edad = mi_diccionario["edad"]

print(edad)  # 25
```

Para modificar un valor de un diccionario, utilizamos la clave correspondiente entre corchetes [] y le asignamos el nuevo valor.

Ejemplo:

```
mi_diccionario["edad"] = 30
print(mi_diccionario)  # {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Bogotá'}
```

Ejercicio:

- Modificar el valor de la clave “edad” del diccionario **mi_diccionario** a 30.

Respuesta

```
mi_diccionario["edad"] = 30
print(mi_diccionario)  # {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Bogotá'}
```

Para agregar un nuevo par clave-valor a un diccionario, utilizamos la clave correspondiente entre corchetes [] y le asignamos el nuevo valor.

Ejemplo:

```
mi_diccionario["profesion"] = "Ingeniero"
print(mi_diccionario)  # {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Bogotá', 'profesion':
```

Ejercicio:

- Agregar un nuevo par clave-valor al diccionario **mi_diccionario** con la clave "profesion" y el valor "Ingeniero".
- Imprimir el diccionario **mi_diccionario**.

Respuesta

```
mi_diccionario["profesion"] = "Ingeniero"
print(mi_diccionario)  # {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Bogotá', 'profesion':
```

Respuesta

```
mi_diccionario["profesion"] = "Ingeniero"
print(mi_diccionario)  # {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Bogotá', 'profesion':
```

Para eliminar un par clave-valor de un diccionario, utilizamos la palabra reservada **del** seguida de la clave correspondiente entre corchetes [].

Ejemplo:

```
del mi_diccionario["edad"]
print(mi_diccionario)  # {'nombre': 'Juan', 'ciudad': 'Bogotá', 'profesion': 'Ingeniero'}
```

Ejercicio:

- Eliminar la clave "edad" del diccionario **mi_diccionario**.

Respuesta

```
del mi_diccionario["edad"]
print(mi_diccionario)  # {'nombre': 'Juan', 'ciudad': 'Bogotá', 'profesion': 'Ingeniero'}
```

Para recorrer un diccionario, podemos utilizar un ciclo **for** que recorra las claves del diccionario.

Ejemplo:

```
for clave in mi_diccionario:
    print(clave)

# nombre
# ciudad
# profesion
```

Ejercicio:

- Recorrer el diccionario **mi_diccionario** e imprimir las claves.

Respuesta

```
for clave in mi_diccionario:
    print(clave)

# nombre
# ciudad
# profesion
```

Para recorrer un diccionario y obtener tanto las claves como los valores, podemos utilizar el método **items()**.

Ejemplo:

```
for clave, valor in mi_diccionario.items():
    print(clave, valor)

# nombre Juan
# ciudad Bogotá
# profesion Ingeniero
```

Ejercicio:

- Recorrer el diccionario **mi_diccionario** e imprimir las claves y los valores.
- Imprimir el valor de la clave "nombre" del diccionario **mi_diccionario**.
- Imprimir el valor de la clave "ciudad" del diccionario **mi_diccionario**.
- Imprimir el valor de la clave "profesion" del diccionario **mi_diccionario**.

Respuesta

```
for clave, valor in mi_diccionario.items():
    print(clave, valor)

# nombre Juan
# ciudad Bogotá
# profesion Ingeniero

print(mi_diccionario["nombre"])  # Juan
print(mi_diccionario["ciudad"])  # Bogotá
print(mi_diccionario["profesion"])  # Ingeniero
```

Para verificar si una clave se encuentra en un diccionario, podemos utilizar la palabra reservada **in**.

Ejemplo:

```
if "nombre" in mi_diccionario:
    print("La clave 'nombre' se encuentra en el diccionario")

if "apellido" not in mi_diccionario:
    print("La clave 'apellido' no se encuentra en el diccionario")
```

Ejercicio:

- Verificar si la clave “nombre” se encuentra en el diccionario **mi_diccionario**.
- Verificar si la clave “apellido” no se encuentra en el diccionario **mi_diccionario**.
- Verificar si la clave “ciudad” se encuentra en el diccionario **mi_diccionario**.
- Verificar si la clave “profesion” no se encuentra en el diccionario **mi_diccionario**.
- Verificar si la clave “edad” se encuentra en el diccionario **mi_diccionario**.
- Verificar si la clave “telefono” no se encuentra en el diccionario **mi_diccionario**.

Respuesta

```
if "nombre" in mi_diccionario:
    print("La clave 'nombre' se encuentra en el diccionario")

if "apellido" not in mi_diccionario:
    print("La clave 'apellido' no se encuentra en el diccionario")

if "ciudad" in mi_diccionario:
    print("La clave 'ciudad' se encuentra en el diccionario")

if "profesion" not in mi_diccionario:
    print("La clave 'profesion' no se encuentra en el diccionario")

if "edad" in mi_diccionario:
    print("La clave 'edad' se encuentra en el diccionario")
else:
    print("La clave 'edad' no se encuentra en el diccionario")

if "telefono" not in mi_diccionario:
    print("La clave 'telefono' no se encuentra en el diccionario")
else:
    print("La clave 'telefono' se encuentra en el diccionario")
```

32 Conjuntos

Los conjuntos son una estructura de datos que nos permite almacenar elementos únicos. Los conjuntos son mutables, es decir, podemos modificar su contenido.

Para crear un conjunto, utilizamos llaves **{}** y separamos cada elemento con comas **,**.

Ejemplo:

```
mi_conjunto = {1, 2, 3, 4, 5}
```

Ejercicio:

- Crear un conjunto con los siguientes elementos: 1, 2, 3, 4, 5, 6

Respuesta

```
mi_conjunto = {1, 2, 3, 4, 5, 6}
```

Para agregar un elemento a un conjunto, utilizamos el método **add()**.

Ejemplo:

```
mi_conjunto.add(7)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7}
```

Ejercicio:

- Agregar el número 8 al conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.
- Agregar el número 9 al conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.
- Agregar el número 10 al conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.
- Agregar el número 11 al conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.
- Agregar el número 12 al conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.

Respuesta

```
mi_conjunto.add(8)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8}

mi_conjunto.add(9)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8, 9}

mi_conjunto.add(10)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

mi_conjunto.add(11)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

mi_conjunto.add(12)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

Para eliminar un elemento de un conjunto, utilizamos el método **remove()**.

Ejemplo:

```
mi_conjunto.remove(7)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6}
```

Ejercicio:

- Eliminar el número 12 del conjunto **mi_conjunto**.
- Imprimir el conjunto **mi_conjunto**.

Respuesta

```
mi_conjunto.remove(12)
print(mi_conjunto)  # {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Para recorrer un conjunto, podemos utilizar un ciclo **for**.

Ejemplo:

```
for elemento in mi_conjunto:
    print(elemento)

# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

```
# 10
# 11
```

Ejercicio:

- Recorrer el conjunto mi_conjunto e imprimir los elementos.

Respuesta

```
for elemento in mi_conjunto:
    print(elemento)
```

Para verificar si un elemento se encuentra en un conjunto, podemos utilizar la palabra reservada **in**.

Ejemplo:

```
if 1 in mi_conjunto:
    print("El número 1 se encuentra en el conjunto")

if 7 not in mi_conjunto:
    print("El número 7 no se encuentra en el conjunto")

if 10 in mi_conjunto:
    print("El número 10 se encuentra en el conjunto")

if 15 not in mi_conjunto:
    print("El número 15 no se encuentra en el conjunto")

if 20 in mi_conjunto:
    print("El número 20 se encuentra en el conjunto")

if 25 not in mi_conjunto:
    print("El número 25 no se encuentra en el conjunto")
```

Ejercicio:

- Verificar si el número 30 se encuentra en el conjunto mi_conjunto.
- Verificar si el número 35 no se encuentra en el conjunto mi_conjunto.
- Verificar si el número 40 se encuentra en el conjunto mi_conjunto.

Respuesta

```
if 30 in mi_conjunto:
    print("El número 30 se encuentra en el conjunto")
else:
    print("El número 30 no se encuentra en el conjunto")
```

```
if 35 not in mi_conjunto:
    print("El número 35 no se encuentra en el conjunto")
else:
    print("El número 35 se encuentra en el conjunto")

if 40 in mi_conjunto:
    print("El número 40 se encuentra en el conjunto")
else:
    print("El número 40 no se encuentra en el conjunto")
```

33 Operaciones con Diccionarios y Conjuntos

Para realizar operaciones con diccionarios y conjuntos, podemos utilizar los métodos y funciones que nos proporciona Python.

Ejercicio:

- Crear un diccionario con las siguientes claves y valores:
  - "nombre": "Diego"
  - "edad": 36
  - "ciudad": "Quito"
  - "profesion": "Ingeniero"
  - "telefono": "1234567890"
  - "email": "dsaavedra88@gmail.com"
  - "direccion": "Calle 123 # 45-67"
  - "pais": "Ecuador"
  - Crear un conjunto con los siguientes elementos:
    - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

Respuesta

```
mi_diccionario = {
    "nombre": "Diego",
    "edad": 36,
    "ciudad": "Quito",
    "profesion": "Ingeniero",
    "telefono": "1234567890",
    "email": "
    "direccion": "Calle 123 # 45-67",
    "pais": "Ecuador"
}

mi_conjunto = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Otra operación que podemos realizar con diccionarios y conjuntos es la unión. Para unir dos diccionarios, utilizamos el método **update()**. Para unir dos conjuntos, utilizamos el método **union()**.

Ejercicio:

- Crear un diccionario con las siguientes claves y valores:

- "apellido": "Saavedra"
- "genero": "Masculino"
- "estado_civil": "Soltero"
- "hijos": 0
- "mascotas": 1
- Crear un conjunto con los siguientes elementos:
  * 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22

Respuesta

```
mi_diccionario.update({
    "apellido": "Saavedra",
    "genero": "Masculino",
    "estado_civil": "Soltero",
    "hijos": 0,
    "mascotas": 1
})

mi_conjunto.union({12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22})
```