

# **Bootcamp Desarrollo Web FullStack**

Diego Saavedra

Nov 7, 2024

# Table of contents

<b>1 Bienvenido</b>	<b>8</b>
1.1 ¿De qué trata este Bootcamp? . . . . .	8
1.2 ¿Para quién es este bootcamp? . . . . .	8
1.3 ¿Qué aprenderás? . . . . .	8
1.4 ¿Cómo contribuir? . . . . .	8
<b>I Unidad 0: Introducción a Git y GitHub</b>	<b>10</b>
<b>2 Git y GitHub</b>	<b>11</b>
2.1 ¿Qué es Git y GitHub? . . . . .	11
2.2 ¿Quiénes utilizan Git? . . . . .	12
2.3 ¿Cómo se utiliza Git? . . . . .	12
2.4 ¿Para qué sirve Git? . . . . .	13
2.5 ¿Por qué utilizar Git? . . . . .	14
2.6 ¿Dónde puedo utilizar Git? . . . . .	15
2.7 Pasos Básicos . . . . .	15
2.8 Instalación de Visual Studio Code . . . . .	16
2.8.1 Descarga e Instalación de Git . . . . .	17
2.8.2 Configuración . . . . .	18
2.8.3 Creación de un Repositorio “helloWorld” en Python . . . . .	18
2.8.4 Comandos Básicos de Git . . . . .	19
2.8.5 Estados en Git . . . . .	19
<b>3 Tutorial: Moviendo Cambios entre Estados en Git</b>	<b>20</b>
3.1 Introducción . . . . .	20
3.2 Sección 1: Modificar Archivos en el Repositorio . . . . .	20
3.3 Mover Cambios de Local a Staging: . . . . .	20
3.4 Agregar Cambios de Local a Staging: . . . . .	21
3.5 Sección 2: Confirmar Cambios en un Commit . . . . .	21
3.6 Mover Cambios de Staging a Commit: . . . . .	21
3.7 Sección 3: Creación y Fusión de Ramas . . . . .	21
3.8 Crear una Nueva Rama: . . . . .	21
3.9 Implementar Funcionalidades en la Rama: . . . . .	21
3.10 Fusionar Ramas con la Rama Principal: . . . . .	22
3.11 Sección 4: Revertir Cambios en un Archivo . . . . .	22
3.12 Revertir Cambios en un Archivo: . . . . .	22
3.13 Conclusión . . . . .	22
<b>4 Asignación</b>	<b>23</b>

<b>5 GitHub Classroom</b>	<b>24</b>
5.1 ¿Qué es GitHub Classroom? . . . . .	24
5.1.1 Funcionalidades Principales . . . . .	24
5.2 Ejemplo Práctico . . . . .	25
5.2.1 Creación de una Asignación en GitHub Classroom . . . . .	25
5.3 Trabajo de los Estudiantes . . . . .	27
<b>II Unidad 1: Introducción e Instalaciones Necesarias</b>	<b>33</b>
<b>6 Introducción e Instalaciones Necesarias.</b>	<b>34</b>
6.1 Introducción General a la Programación . . . . .	35
6.2 Instalación de Python . . . . .	37
6.3 Uso de REPL, PEP 8 y Zen de Python . . . . .	40
6.3.1 REPL . . . . .	40
<b>7 Pep 8</b>	<b>41</b>
<b>8 Zen de python.</b>	<b>42</b>
8.1 Entornos de Desarrollo . . . . .	43
8.2 5 Consejos para mejorar la lógica de programación. . . . .	45
8.3 Conclusiones . . . . .	46
<b>9 Introducción a la Programación con Python</b>	<b>47</b>
9.1 ¿Qué es la programación? . . . . .	47
9.2 ¿Qué es Python? . . . . .	47
9.3 ¿Por qué aprender Python? . . . . .	47
9.4 ¿Qué aprenderemos en este bootcamp? . . . . .	48
9.5 Identación en Python . . . . .	48
9.6 Comentarios en python . . . . .	48
9.7 Variables y Variables Múltiples . . . . .	48
9.8 Concatenación de Cadenas . . . . .	49
<b>10 Actividad</b>	<b>50</b>
10.1 instrucciones . . . . .	50
<b>11 Conclusión</b>	<b>51</b>
<b>12 Tipos de Datos</b>	<b>52</b>
12.1 String y Números. . . . .	52
12.1.1 String . . . . .	52
12.1.2 Números . . . . .	53
12.2 Listas y Tuplas. . . . .	53
12.2.1 Listas . . . . .	53
12.2.2 Tuplas . . . . .	53
12.3 Diccionarios y Booleanos. . . . .	54
12.3.1 Diccionarios . . . . .	54
12.3.2 Booleanos . . . . .	54
12.4 Range . . . . .	54

<b>13 Actividad</b>	<b>55</b>
13.1 Instrucciones . . . . .	55
<b>14 Conclusiones</b>	<b>56</b>
<b>15 Control de Flujo</b>	<b>57</b>
15.1 If y Condicionales . . . . .	57
15.2 If, elif y else . . . . .	58
15.3 And y Or . . . . .	59
15.4 While loop . . . . .	59
15.5 While, break y continue . . . . .	60
15.6 For loop . . . . .	60
<b>16 Actividad</b>	<b>61</b>
16.1 instrucciones . . . . .	61
<b>17 Conclusiones</b>	<b>62</b>
<b>18 Funciones y recursividad.</b>	<b>63</b>
18.1 Introducción a Funciones . . . . .	63
18.2 Parámetros y Argumentos . . . . .	64
18.3 Retorno de valores . . . . .	64
18.4 Recursividad . . . . .	65
<b>19 Actividad</b>	<b>66</b>
19.1 Instrucciones . . . . .	66
<b>20 Conclusiones</b>	<b>67</b>
<b>III Unidad 2: Programación Orientada a Objetos</b>	<b>68</b>
<b>21 Programacion Orientada a Objetos.</b>	<b>69</b>
21.1 Objetos y Clases . . . . .	70
21.2 Atributos . . . . .	70
21.3 ¿Qué es self? . . . . .	70
21.4 Métodos . . . . .	71
21.5 Self, Eliminar Propiedades y Objetos . . . . .	71
21.6 Eliminar Propiedades y Objetos . . . . .	72
21.7 Herencia, Polimorfismo y Encapsulación . . . . .	72
21.7.1 Herencia . . . . .	72
21.7.2 Polimorfismo . . . . .	73
21.7.3 Encapsulación . . . . .	74
21.8 Actividad . . . . .	75
<b>22 Conclusiones</b>	<b>79</b>

<b>IV Unidad 3: Módulos y Paquetes</b>	<b>80</b>
<b>23 Módulos</b>	<b>81</b>
23.1 Introducción a Módulos . . . . .	81
23.2 Creando Módulos Personalizados . . . . .	82
23.3 Usando Módulos en un Archivo Principal . . . . .	82
23.4 Importando y Renombrando Módulos . . . . .	82
23.5 Importando Funciones Específicas de un Módulo . . . . .	83
23.6 Usando Módulos Externos con pip . . . . .	83
23.7 Instalando un módulo con pip . . . . .	83
23.8 Usando el módulo instalado . . . . .	83
23.9 Instalando otro módulo . . . . .	83
23.10 Usando el módulo emojis . . . . .	84
<b>24 Actividad Práctica</b>	<b>85</b>
<b>25 Conclusión</b>	<b>87</b>
<b>V Unidad 4: Docker</b>	<b>88</b>
<b>26 Docker</b>	<b>89</b>
26.1 Ejemplos: . . . . .	90
26.2 Comandos básicos de Docker: . . . . .	91
26.3 Atajos y Comandos Adicionales: . . . . .	92
26.4 Práctica: . . . . .	93
<b>27 Conclusiones</b>	<b>94</b>
<b>28 Dockerfile y Docker Compose</b>	<b>95</b>
28.1 Introducción . . . . .	95
28.1.1 Dockerfile . . . . .	95
28.1.2 Docker Compose . . . . .	95
28.2 Ejemplos: . . . . .	95
28.2.1 server.js . . . . .	96
28.2.2 Dockerfile . . . . .	96
28.2.3 docker-compose.yml . . . . .	97
28.3 Práctica: . . . . .	98
<b>29 Conclusión</b>	<b>99</b>
<b>30 DevContainers</b>	<b>100</b>
30.1 ¿Qué son los DevContainers? . . . . .	100
30.2 Instalación y Uso . . . . .	100
30.3 Ejemplos: . . . . .	101
30.4 Práctica . . . . .	104
30.5 Conclusiones . . . . .	105

<b>VI Proyectos</b>	<b>106</b>
<b>31 Laboratorio: Construcción de un Juego de Ahorcado en Python</b>	<b>107</b>
31.1 Objetivos del Laboratorio . . . . .	107
31.2 Prerrequisitos . . . . .	107
31.3 Paso 1: Crear la Estructura Inicial del Proyecto . . . . .	108
31.3.1 Crear un archivo de Python: . . . . .	108
31.4 Paso 2: Definir las Etapas del Ahorcado en ASCII . . . . .	108
31.4.1 Crear la lista AHORCADO_DIBUJO: . . . . .	108
31.4.2 Prueba del dibujo: . . . . .	109
31.5 Paso 3: Crear la Función para Mostrar el Dibujo del Ahorcado . . . . .	109
31.5.1 Definir la función mostrar_ahorcado: . . . . .	109
31.5.2 Prueba de la función: . . . . .	110
31.6 Paso 4: Crear Funciones para el Flujo del Juego . . . . .	110
31.6.1 Función para Seleccionar Palabra Aleatoria: . . . . .	110
31.6.2 Función para Mostrar el Estado Actual: . . . . .	110
31.6.3 Función para Manejar el Intento del Jugador: . . . . .	111
31.7 Paso 5: Crear la Función Principal del Juego . . . . .	111
31.7.1 Configurar el Juego: . . . . .	111
31.7.2 Ciclo del Juego: . . . . .	111
31.8 Paso 6: Crear Función de Resultado Final con Emojis . . . . .	112
31.8.1 Definir mostrar_resultado: . . . . .	112
31.9 Paso 7: Ejecutar el Juego . . . . .	112
31.9.1 Ejecutar el Juego: . . . . .	112
31.9.2 Prueba Final: . . . . .	113
31.10 Paso 8: Mejoras Opcionales . . . . .	113
31.10.1 Añadir Validación de Entradas: Controla que el jugador solo introduzca letras válidas. . . . .	113
<b>32 Conclusión</b>	<b>114</b>
<b>33 Que aprendimos</b>	<b>118</b>
<b>34 Gestor de Tareas con Prioridades</b>	<b>119</b>
34.1 Módulos del Proyecto . . . . .	119
34.1.1 Módulo de tareas . . . . .	119
34.2 Funciones Clave . . . . .	119
34.2.1 Desarrollo . . . . .	120
<b>35 Extra</b>	<b>122</b>
<b>36 Conclusión</b>	<b>123</b>
<b>37 Reto</b>	<b>124</b>
<b>38 Simulador de Tienda Online</b>	<b>125</b>
38.1 Módulos del Proyecto . . . . .	125
38.1.1 Módulo de Productos . . . . .	125
38.1.2 Módulo de Carrito . . . . .	125
38.1.3 Módulo de Cliente . . . . .	126

38.1.4	Módulo de Pedido . . . . .	126
<b>39 Desarrollo</b>		<b>127</b>
39.1	Productos . . . . .	127
39.2	Carrito . . . . .	128
39.3	Clientes . . . . .	128
39.4	Pedidos . . . . .	129
<b>40 Prueba del Simulador de Tienda Online</b>		<b>130</b>
<b>41 Extra</b>		<b>131</b>
<b>42 Conclusión</b>		<b>132</b>
<b>43 Sistema Universitario</b>		<b>133</b>
43.1	Objetivos . . . . .	134
43.2	Instrucciones. . . . .	134
43.3	Desarrollo . . . . .	134
<b>44 Conclusión</b>		<b>138</b>

# 1 Bienvenido

¡Bienvenido al Bootcamp de Desarrollo Web Fullstack

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

## 1.1 ¿De qué trata este Bootcamp?

Este bootcamp está diseñado para enseñarle a desarrollar aplicaciones web modernas utilizando Django, Flask y React.

## 1.2 ¿Para quién es este bootcamp?

Este bootcamp es para cualquier persona interesada en aprender a desarrollar aplicaciones web modernas.

## 1.3 ¿Qué aprenderás?

Aprenderás algunos lenguajes de programación como Python, JavaScript y TypeScript, así como algunos de los frameworks y bibliotecas más populares como Django, FastAPI y React.

## 1.4 ¿Cómo contribuir?

Valoramos su contribución a este bootcamp. 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, FastAPI y React!

## **Part I**

# **Unidad 0: Introducción a Git y GitHub**

## 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 9fa7605ad1837aa44dfb9c711dc8bd60cab7c5d
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?

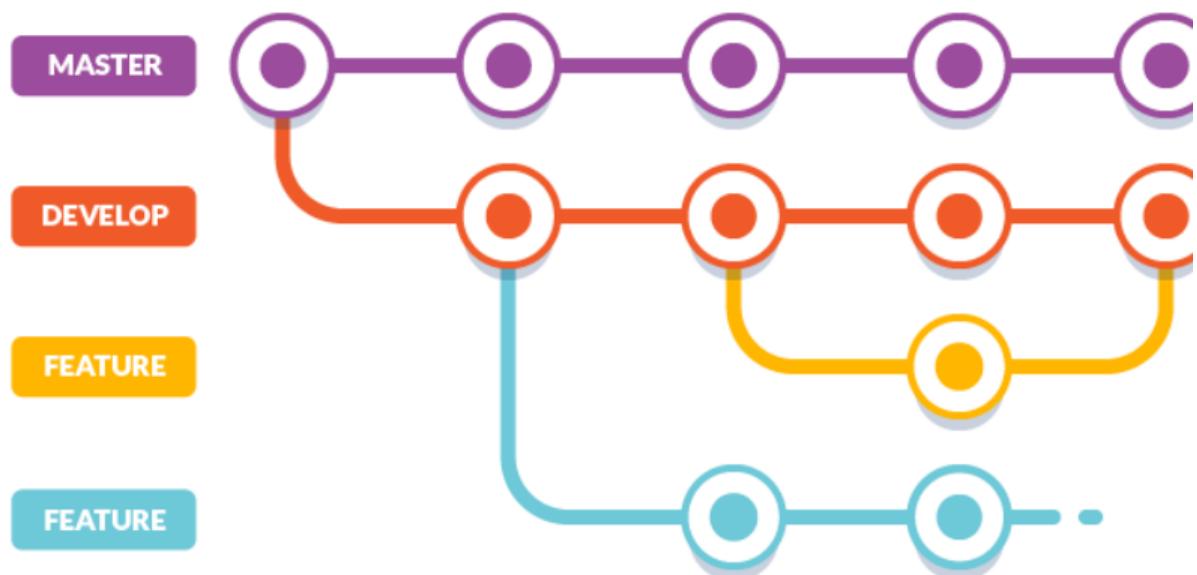


Figure 2.4: Seguimiento de Cambios con Git

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?

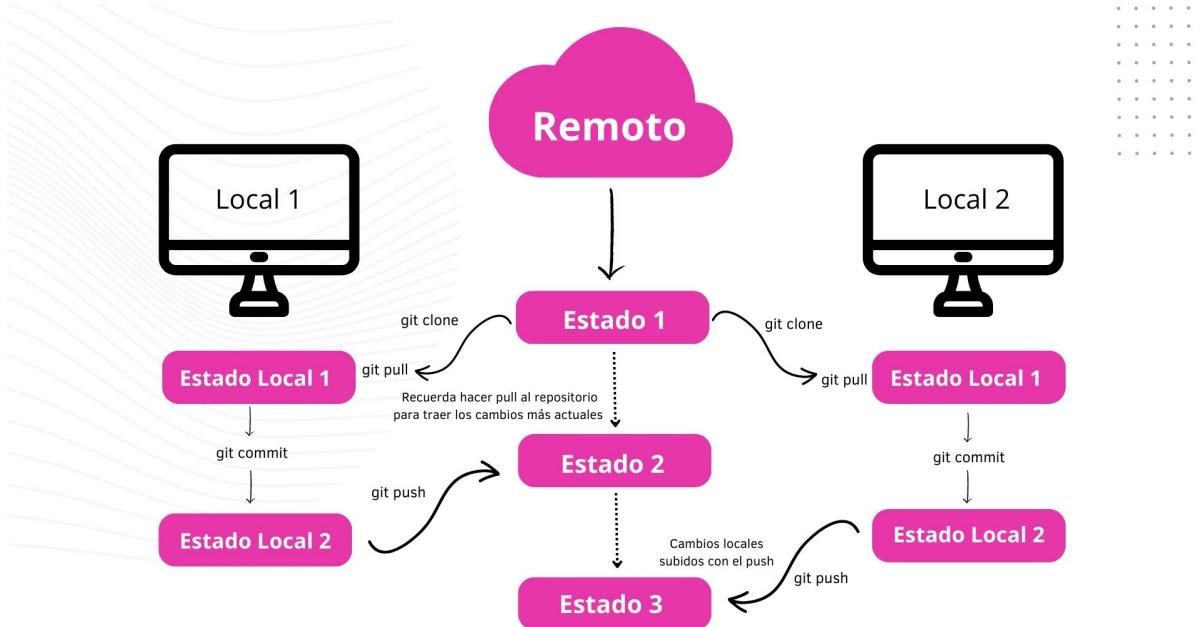


Figure 2.5: Ventajas de Git

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?

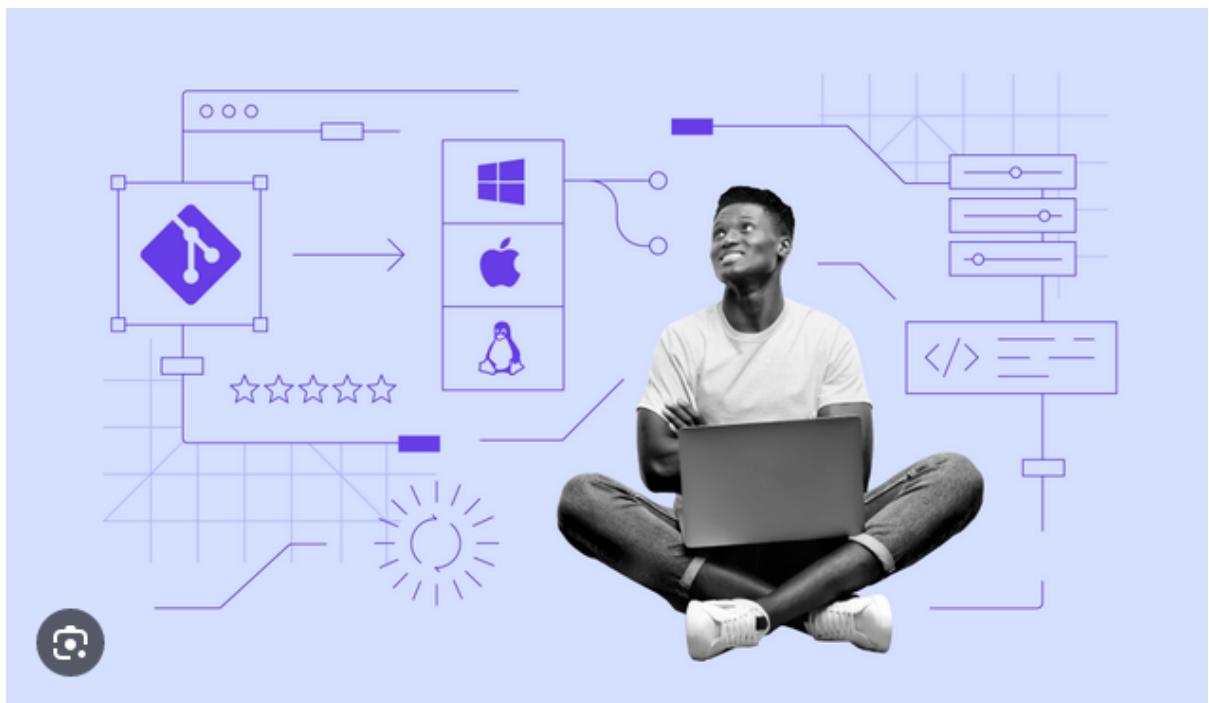


Figure 2.6: Git en Diferentes Sistemas Operativos

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

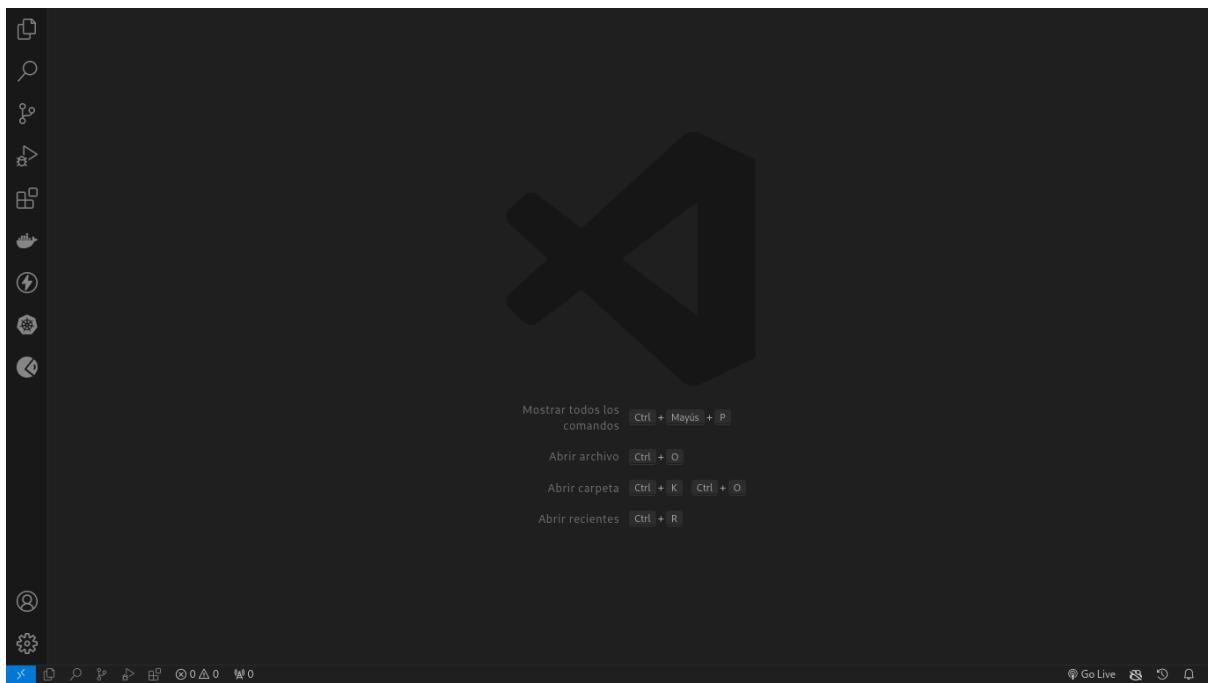


Figure 2.7: Visual Studio Code

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

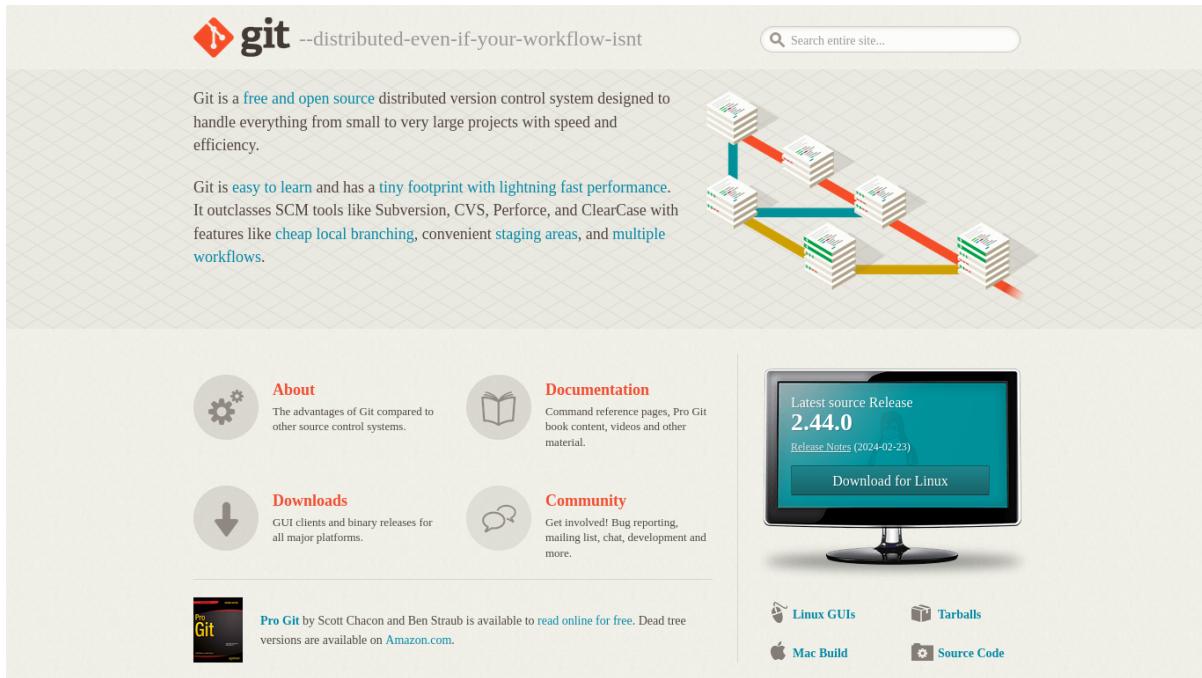


Figure 2.8: Git

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 ábrelo 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("Bienvenido,", 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!**”<sup>x</sup>

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.

The screenshot shows the GitHub Classroom interface. At the top, there's a banner with a warning about changes in assignment acceptance and starter code repositories. Below the banner, the navigation bar includes 'Classrooms / Curso de Django and React - Codings Academy / Hello World'. The main section displays an assignment titled 'Hello World'. It shows it's an individual assignment due on Feb 28, 2024, at 20:00 ET, and is currently active. There are links for the assignment URL (<https://classroom.github.com/a/Gcbhv0hp>), edit, and download. Below this, the 'Assignment Details' section shows 0 accepted assignments, 0 students, 0 assignment submissions, 0 submitted, and 0 not submitted. There are filters, a search bar, and sorting options at the bottom of this section.

**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.

The screenshot shows the GitHub Classroom interface for creating a new assignment. The main area is titled "Assignment basics". It includes fields for "Assignment title" (set to "Hello World"), "Deadline" (set to "02/28/2024, 08:00 PM"), and "Assignment status" (set to "Active"). The "Individual or group assignment" dropdown is set to "Individual assignment". A note states: "Assignment type cannot be changed after assignment creation." On the left sidebar, there are three tabs: "Assignment basics" (selected), "Starter code and environment", and "Grading and feedback". A yellow banner at the top says: "⚠ Assignment acceptance and starter code repositories will be changing on June 17, 2024. Review the changes and prepare your assignments."

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

The screenshot shows the continuation of the assignment configuration. It includes sections for "Starter code and environment" and "Grading and feedback". In the "Starter code and environment" section, there's a "Find a GitHub repository" search bar containing "education/autograding-example-python". Below it, there's a note about GitHub Codespaces and a "Supported editor" section with a checked option "Don't use an online IDE". In the "Grading and feedback" section, there's a "Add autograding tests" section with a "Hello world test" entry and a "+ Add test" button.

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

The screenshot shows the GitHub Classroom interface for a course titled "Curso de Django and React - Codings Academy". At the top, there's a navigation bar with links for "GitHub Education", "Help", "Settings", "Profile", and "Logout". Below the navigation, the course title is displayed. The main content area is titled "Accept the assignment — Hello World". It explains that accepting the assignment grants access to a repository named "hello-world-statick88" in the "Coding-Academy-ec" organization. A prominent green button labeled "Accept this assignment" is centered at the bottom of this section.

Curso de Django and React - Codings Academy

## Accept the assignment — Hello World

Once you accept this assignment, you will be granted access to the [hello-world-statick88](#) repository in the [Coding-Academy-ec](#) organization on GitHub.

[Accept this assignment](#)

## 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.

The screenshot shows the GitHub Classroom interface after the assignment has been accepted. The main message is "You accepted the assignment, Hello World. We're configuring your repository now. This may take a few minutes to complete. Refresh this page to see updates." Below this, a note states "Your assignment is due by Feb 28, 2024, 20:00 ET". To the right, there's a callout box with a student icon and the text "Join the GitHub Student Developer Pack". It explains that verified students receive free GitHub Pro plus thousands of dollars worth of real-world tools and training from GitHub Education partners. An "Apply" button is located at the bottom of this callout.

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

The screenshot shows the GitHub Classroom interface. At the top, there's a banner with the GitHub Classroom logo and navigation links for GitHub Education, notifications, help, and user profile. Below the banner, a large circular icon with a graduation cap and hands is displayed, followed by the message "You're ready to go!". A sub-section below it says "You accepted the assignment, Hello World." and "Your assignment repository has been created:". It includes a link to the repository (<https://github.com/Coding-Academy-ec/hello-world-student-pruebas>), a note about the repository being configured, and a due date of "Feb 28, 2024, 20:00 ET". To the right, there's a call-to-action for the "GitHub Student Developer Pack" with a "Join" button. The main content area shows a list of files in the repository, including .github, .gitignore, README.md, hello.py, and hello\_test.py. The README section contains a button to "Review the assignment due date". On the right side, there are sections for "About", "Releases", and "Packages".

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

This screenshot shows a GitHub repository page for "hello-world-student-pruebas". The repository is public and was generated from "education/autograding-example-python". It has 1 branch and 0 tags. The master branch contains several files: ".github-classroom[bot] add deadline", ".github", ".gitignore", "README.md", "hello.py", and "hello\_test.py". The "About" section indicates the repository was created by GitHub Classroom. The "Releases" section shows no releases published, with a link to "Create a new release". The "Packages" section shows no packages published, with a link to "Publish your first package".

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.

```

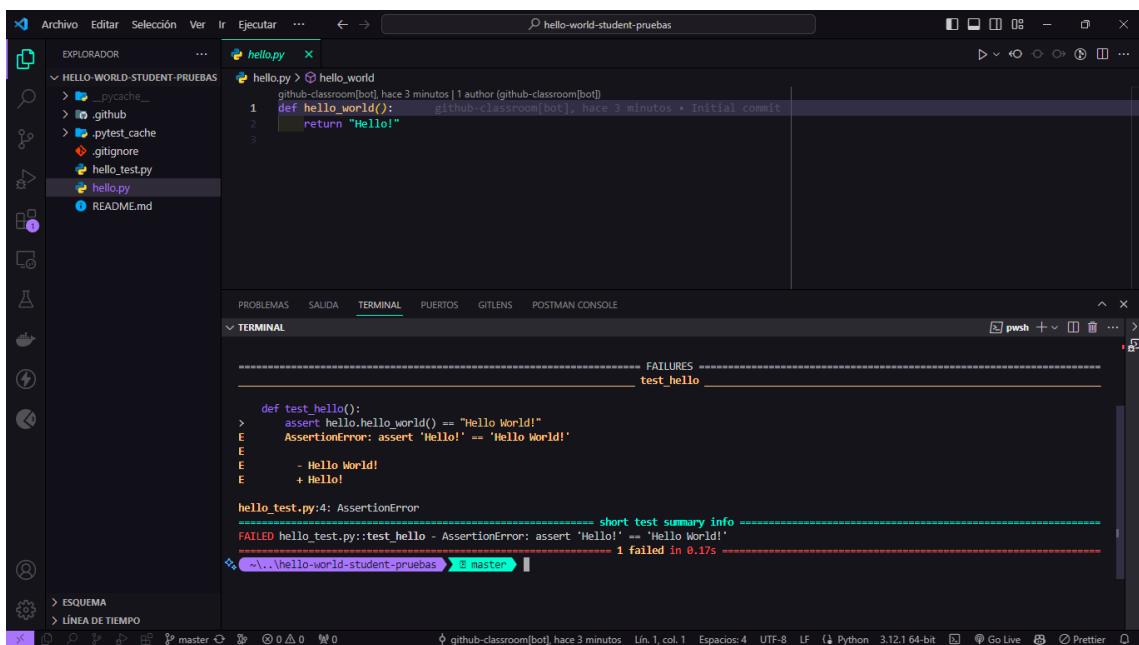
**Desarrollar la Tarea:** Los estudiantes trabajan en la tarea, realizando los cambios necesarios y realizando commits de manera regular para mantener un historial de su trabajo.

### 💡 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.

```

hello.py > hello.world
You hace 1 segundo | 2 authors (You and others)
1 def hello_world():
2     return "Hello World!" You, hace 1 segundo * Uncommitted changes
3

PROBLEMAS SALIDA TERMINAL PUERTOS GITLENS POSTMAN CONSOLE
TERMINAL
~\.\hello-world-student-pruebas > master ~1 git add .\hello.py
~\.\hello-world-student-pruebas > master ~1 git commit -m "Update Hello World! in hello.py"
[master 285741e] Update Hello World! in hello.py
1 file changed, 1 insertion(+), 1 deletion(-)
~\.\hello-world-student-pruebas > master git push -u origin main
You, hace 1 segundo Lin. 2, col. 25 Espacios: 4 UTF-8 LF Python 3.12.1 64-bit Go Live Prettier

```

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

### 💡 Tip

Se recomienda hacer las pruebas en local antes de enviar los cambios al repositorio en Github

**About**

hello-world-student-pruebas created by GitHub Classroom

- Readme
- Activity
- Custom properties
- 0 stars
- 1 watching
- 0 forks
- Report repository

**Releases**

No releases published [Create a new release](#)

**Packages**

No packages published [Publish your first package](#)

Este Action lo que hace es evaluar los cambios realizados

Se recomienda hacer las pruebas en local antes de enviar los cambios al repositorio 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.

# Hello World

Individual assignment Due Feb 28, 2024, 20:00 ET Active

<https://classroom.github.com/a/6cbhv0hp> [Edit](#) [Download](#)

## Assignment Details

Accepted assignments 1	Assignment submissions 1	Passed students 1
1 Students	1 Submitted 0 Not submitted	1/1 Passed <div style="width: 100%; background-color: #2e8b57; height: 10px; margin-top: 5px;"></div>

Filters [Search for an assignment](#) [Filter by passing](#) [Sort](#)

Total students
 <b>student-pruebas</b> <span>Submitted</span> @student-pruebas <a href="#">Latest commit 2 minutes ago</a> ✓  1 commit  100/100 <a href="#">Repository</a>



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.

## **Part II**

# **Unidad 1: Introducción e Instalaciones Necesarias**

## 6 Introducción e Instalaciones Necesarias.

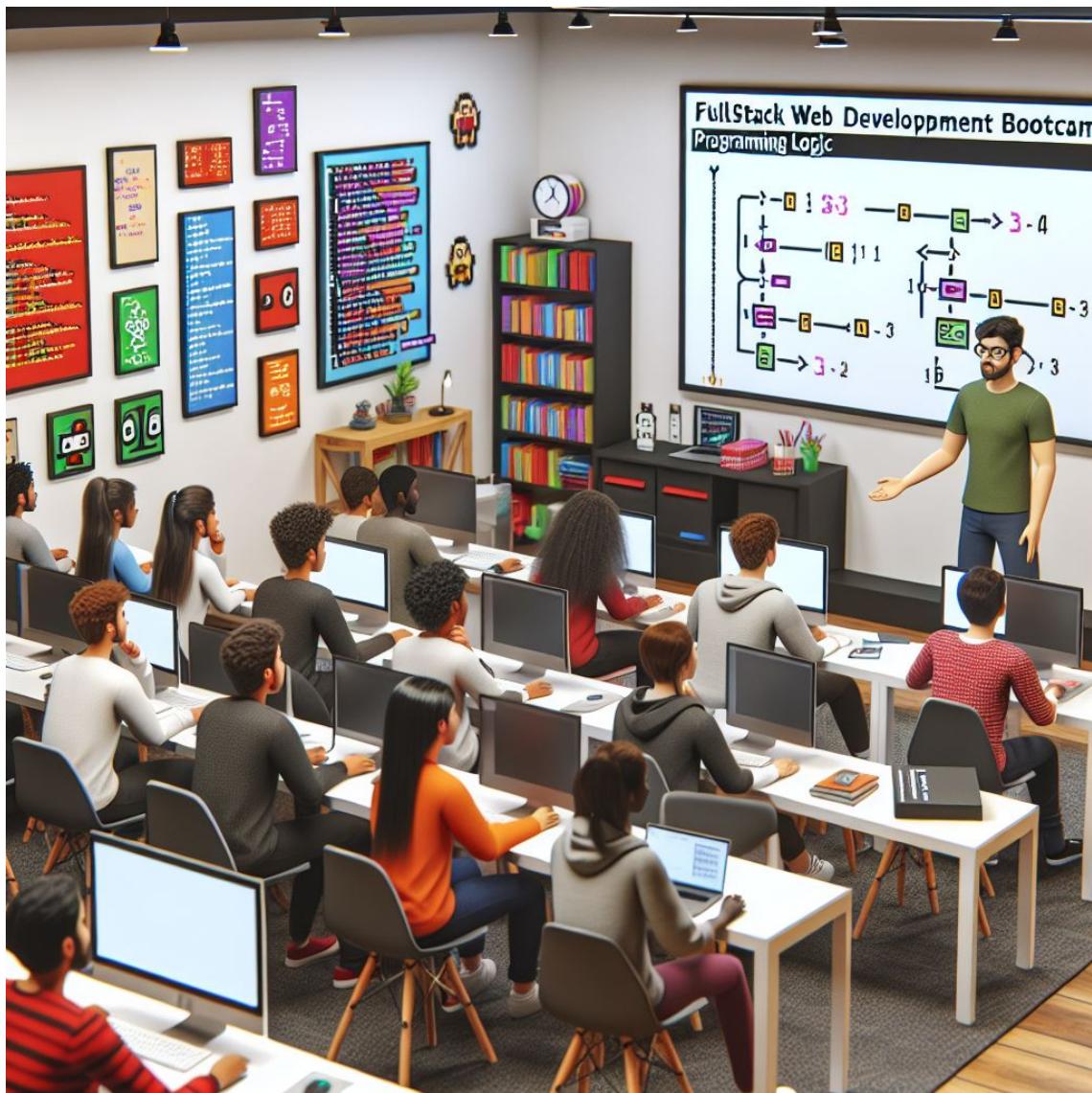


Figure 6.1: Lógica de la Programación

En este Bootcamp aprenderemos las bases y fundamentos necesarios del desarrollo web fullstack, esto es desde el frontend hasta el backend.

Para ello, utilizaremos Python como lenguaje de programación principal, y Django y FastAPI como frameworks para el desarrollo de aplicaciones web.

Por otra parte esta tambien el frontend, donde utilizaremos HTML, CSS y JavaScript para el desarrollo de interfaces de usuario, aprenderemos acerca de Node.js y React.js para el desarrollo de aplicaciones web del lado del cliente.

Sin embargo antes de empezar con el desarrollo web, es necesario tener una base sólida en programación, por lo que en este primer módulo aprenderemos acerca de Python, un lenguaje de programación de alto nivel, interpretado y orientado a objetos.

Por otra parte es necesario saber que cualquier lenguaje de programación no es suficiente para poder desarrollar sistemas que permitan resolver problemas del diario vivir, es necesario tener un entorno de desarrollo adecuado, por lo que en este módulo también aprenderemos acerca de los entornos de desarrollo que podemos utilizar para programar en Python.

En este módulo aprenderemos acerca de los siguientes temas:

- Introducción General a la Programación
- Instalación de Python
- Uso de REPL, PEP 8 y Zen de Python
- Entornos de Desarrollo

## 6.1 Introducción General a la Programación

Si más preámbulos, empecemos con la introducción general a la programación.

Es el proceso de diseñar e implementar un programa de computadora, es decir, un conjunto de instrucciones que le dicen a una computadora qué hacer.

Es una habilidad muy valiosa en el mundo actual, ya que la mayoría de las tareas que realizamos a diario involucran el uso de computadoras y software.

Nos permite automatizar tareas, resolver problemas de manera eficiente y crear aplicaciones y sistemas que nos ayudan en nuestra vida diaria.

En este módulo aprenderemos los fundamentos de la programación utilizando Python, un lenguaje de programación de alto nivel, interpretado y orientado a objetos.

Antes de introducirnos en el aprendizaje del lenguaje de programación, es importante conocer que debemos desarrollar la **lógica de la programación**, es decir, la habilidad de pensar de manera lógica y estructurada para resolver problemas de manera eficiente.

Analicemos el siguiente problema para entender la importancia de la lógica de programación:

- **Problema:** Supongamos que queremos escribir un programa que imprima los números del 1 al 10.

¿Cómo resolverías este problema?

Una posible solución sería escribir un programa que imprima los números del 1 al 10 de manera secuencial.

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

En el ejemplo anterior, hemos resuelto el problema de imprimir los números del 1 al 10 de manera secuencial. Sin embargo, esta solución no es escalable, ya que si quisieramos imprimir los números del 1 al 1000, tendríamos que escribir 1000 instrucciones de impresión.

Una solución más eficiente sería utilizar un bucle para imprimir los números del 1 al 10 de manera automática.

```
for i in range(1, 11):
    print(i)
```

En el ejemplo anterior, hemos utilizado un bucle **for** para imprimir los números del 1 al 10 de manera automática. Esta solución es más eficiente y escalable, ya que podemos cambiar el rango del bucle para imprimir los números del 1 al 1000 sin tener que modificar el código.

- **Problema:** Supongamos que queremos escribir un programa que imprima un saludo personalizado.

¿Cómo resolverías este problema?

Una posible solución sería escribir un programa que solicite al usuario su nombre y luego imprima un saludo personalizado.

```
name = input("Ingrese su nombre: ")
print("Hola, " + name + "!")
```

En el ejemplo anterior, hemos resuelto el problema de imprimir un saludo personalizado solicitando al usuario su nombre. Esta solución es interactiva y personalizada, ya que el saludo se adapta al nombre del usuario.

En resumen, la lógica de programación es la habilidad de pensar de manera lógica y estructurada para resolver problemas de manera eficiente. Es fundamental para desarrollar programas y sistemas que nos ayuden en nuestra vida diaria.

A continuación te ofresco algunas páginas que puedes revisar por tu cuenta y que te permitirán practicar el desarrollo de la lógica de programación:

- [HackerRank](#)
- [LeetCode](#)
- [Retodo de Programación](#)
- [Geeks for Geeks](#)

## 6.2 Instalación de Python



Figure 6.2: Python

Para instalar Python en tu computadora, sigue los siguientes pasos:

1. Ve al sitio web oficial de Python en <https://www.python.org/>.

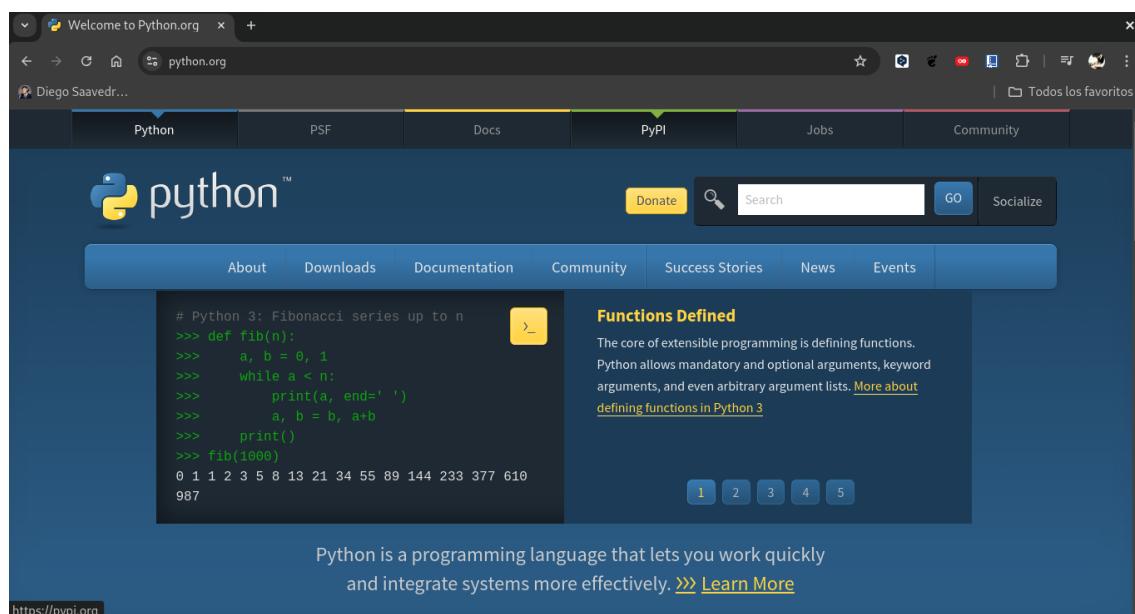


Figure 6.3: Python

2. Haz clic en el botón de descarga de Python.

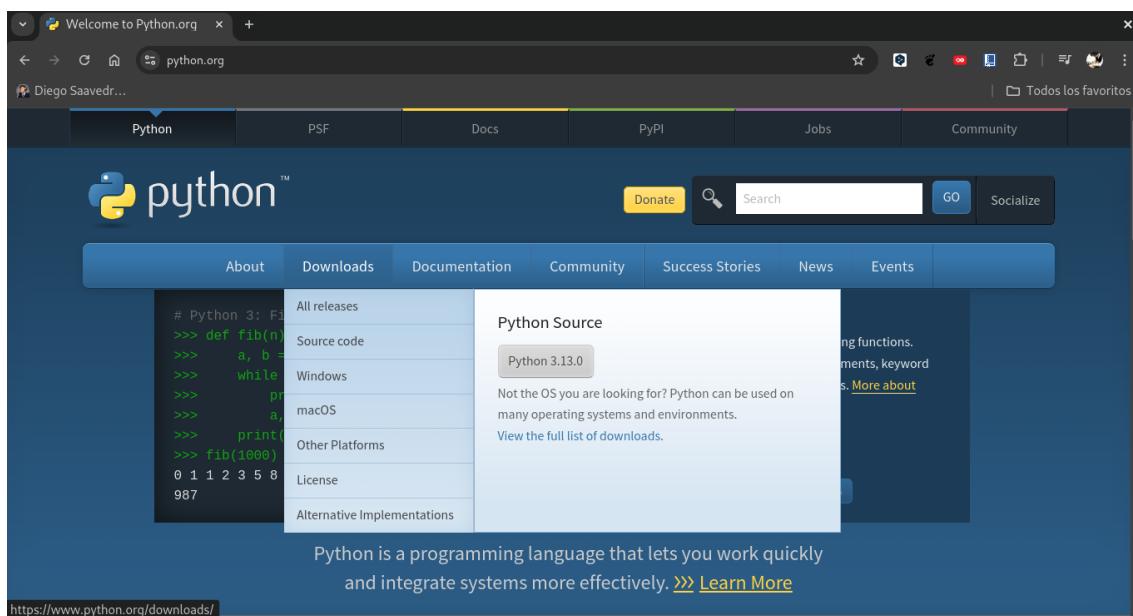


Figure 6.4: Python

3. Selecciona la versión de Python que deseas instalar (recomendamos la versión más reciente).
4. Descarga el instalador de Python para tu sistema operativo (Windows, macOS o Linux).

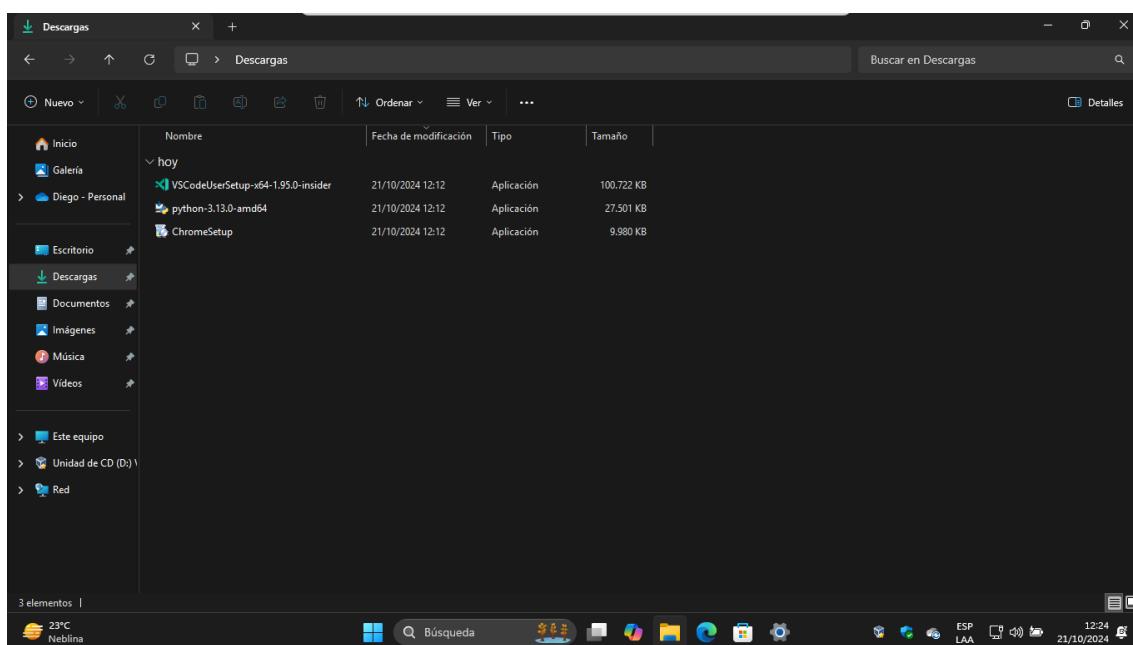


Figure 6.5: Python

5. Ejecuta el instalador de Python y sigue las instrucciones en pantalla para completar la instalación.

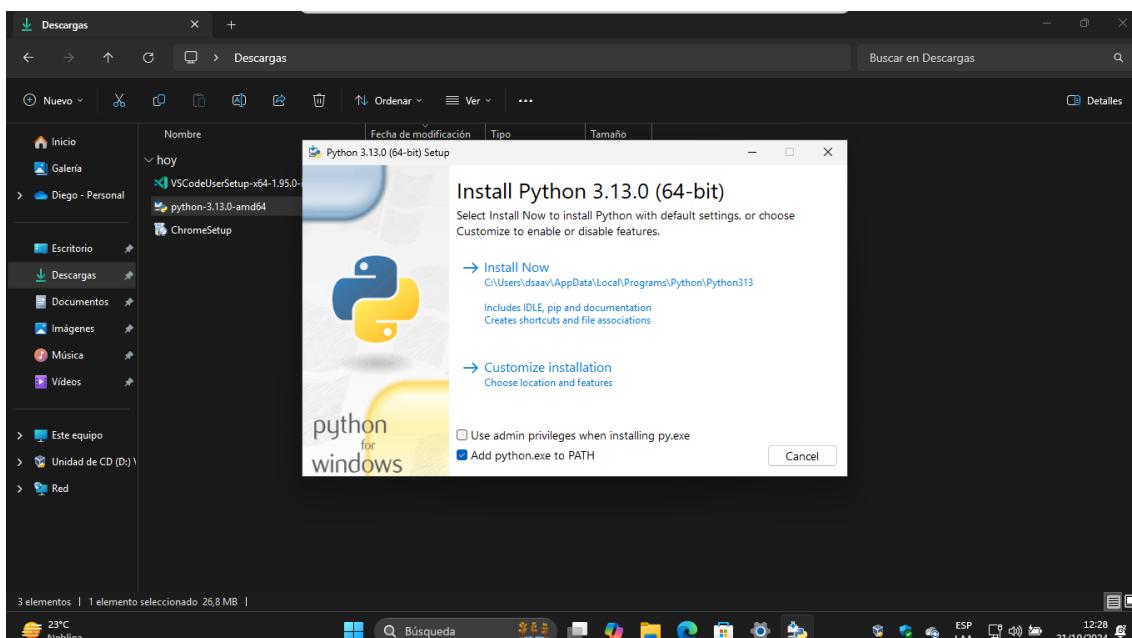


Figure 6.6: Python

Una vez que hayas instalado Python en tu computadora, puedes verificar que la instalación se haya realizado correctamente abriendo una terminal y ejecutando el siguiente comando:

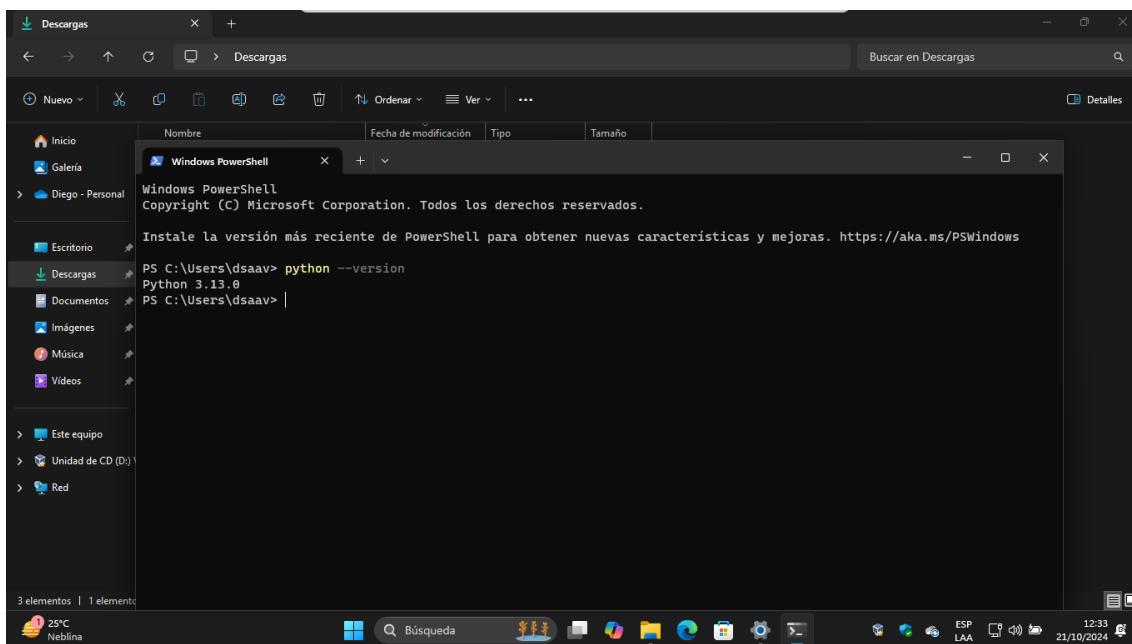


Figure 6.7: Python

```
python --version
```

Si la instalación se realizó correctamente, verás la versión de Python instalada en tu

computadora.

## 6.3 Uso de REPL, PEP 8 y Zen de Python

En esta sección, aprenderemos acerca de REPL, PEP 8 y Zen de Python.

### 6.3.1 REPL

REPL (Read-Eval-Print Loop) es un entorno interactivo que permite escribir y ejecutar código de Python de manera interactiva. Es una excelente herramienta para probar y experimentar con el lenguaje de programación.

Para abrir el REPL de Python, abre una terminal y ejecuta el siguiente comando:

```
python
```

Una vez que hayas abierto el REPL de Python, puedes escribir y ejecutar código de Python de manera interactiva. Por ejemplo, puedes escribir una expresión matemática y ver el resultado:

```
>>> 2 + 2
>>> 4
>>> 3 * 4
>>> 12
>>> 10 / 2
>>> 5.0
>>> 2 ** 3
>>> 8
>>> "Hola, Mundo!"
>>> 'Hola, Mundo!'
>>> "Hola, " + "Mundo!"
>>> 'Hola, ' * 3
>>> 'Hola, Hola, Hola, '
>>> print("Hola, Mundo!")
>>> Hola, Mundo!
```

## 7 Pep 8

PEP 8 (Python Enhancement Proposal 8) es una guía de estilo para escribir código de Python de manera clara y legible. Es una excelente referencia para seguir buenas prácticas de codificación y mantener un código limpio y ordenado.

Algunas recomendaciones de PEP 8 son:

- Utiliza sangrías de 4 espacios para indentar el código.
- Utiliza líneas en blanco para separar funciones y clases.
- Utiliza nombres descriptivos para las variables y funciones.
- Utiliza comentarios para explicar el código y hacerlo más legible.
- Utiliza espacios alrededor de los operadores y después de las comas.
- Utiliza comillas simples o dobles de manera consistente para las cadenas de texto.
- Utiliza la función `print()` para imprimir en la consola.

## 8 Zen de python.

El Zen de Python es una colección de 19 aforismos que resumen los principios de diseño y filosofía de Python. Fueron escritos por Tim Peters, uno de los desarrolladores originales de Python, y se pueden ver en cualquier instalación de Python utilizando el siguiente comando:

```
import this
```

Algunos de los aforismos más conocidos del Zen de Python son:

- Hermoso es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- La legibilidad cuenta.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los errores nunca deberían pasar en silencio.
- A menos que sean silenciados.
- En la cara de la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que seas holandés.

En el ejemplo anterior, hemos utilizado el REPL de Python para ejecutar expresiones matemáticas y cadenas de texto. Es una excelente manera de probar y experimentar con el lenguaje de programación.

## 8.1 Entornos de Desarrollo

Un entorno de desarrollo es un conjunto de herramientas que nos permiten escribir, depurar y ejecutar código de manera eficiente. Es fundamental para desarrollar programas y sistemas de manera efectiva.

Existen varios entornos de desarrollo que podemos utilizar para programar en Python. Algunos de los más populares son:

- **IDLE**: Es el entorno de desarrollo integrado (IDE) oficial de Python. Viene incluido con la instalación de Python y es una excelente opción para programar en Python.

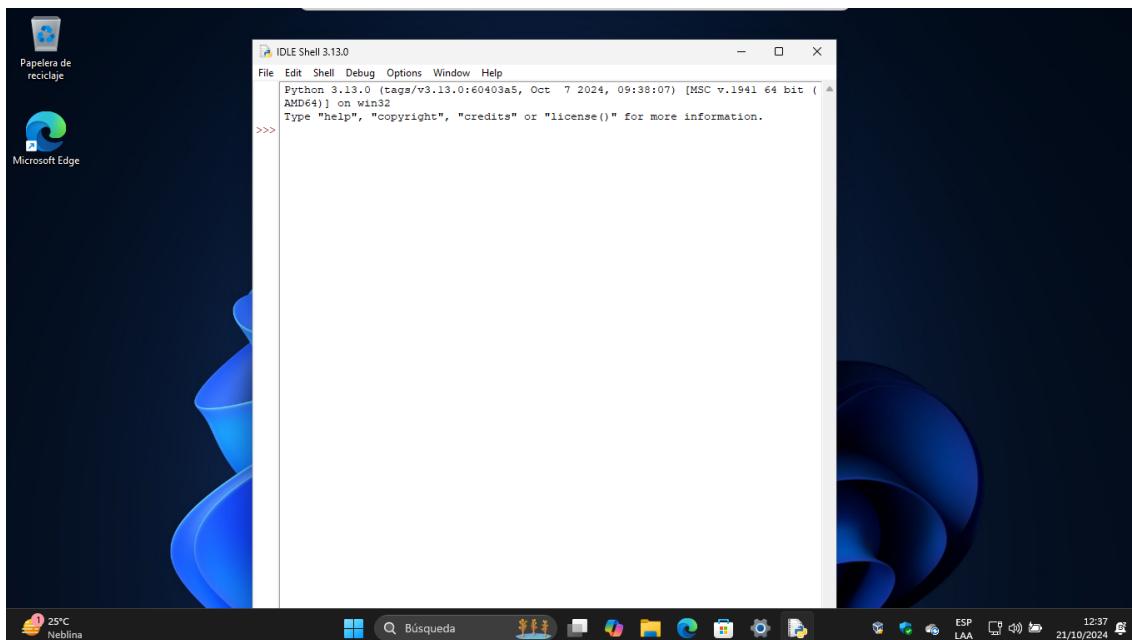


Figure 8.1: IDLE

- **PyCharm**: Es un IDE de Python desarrollado por JetBrains. Es una excelente opción para programar en Python, ya que ofrece muchas características y herramientas útiles.

```

1 # This is a sample Python script.
2
3 # Press Mayús+F10 to execute it or replace it with your code.
4 # Press Double Shift to search everywhere for classes, files, tool windows, actions, and settings.
5
6
7 def print_hi(name):
8     # Use a breakpoint in the code line below to debug your script.
9     print(f'Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.
10
11
12 # Press the green button in the gutter to run the script.
13 if __name__ == '__main__':
14     print_hi('PyCharm')
15
16 # See PyCharm help at https://www.jetbrains.com/help/pycharm/
17

```

Figure 8.2: PyCharm

- **Visual Studio Code:** Es un editor de código desarrollado por Microsoft. Es una excelente opción para programar en Python, ya que ofrece muchas extensiones y herramientas útiles.

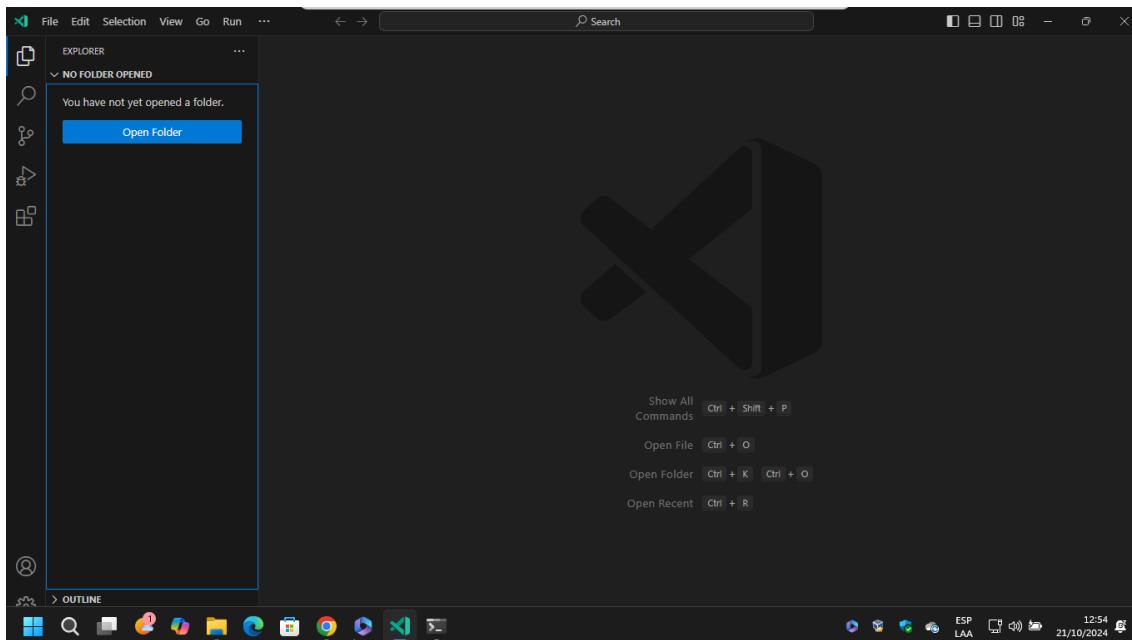


Figure 8.3: Visual Studio Code

- **Jupyter Notebook:** Es una aplicación web que nos permite crear y compartir documentos interactivos que contienen código de Python, visualizaciones y texto explicativo.

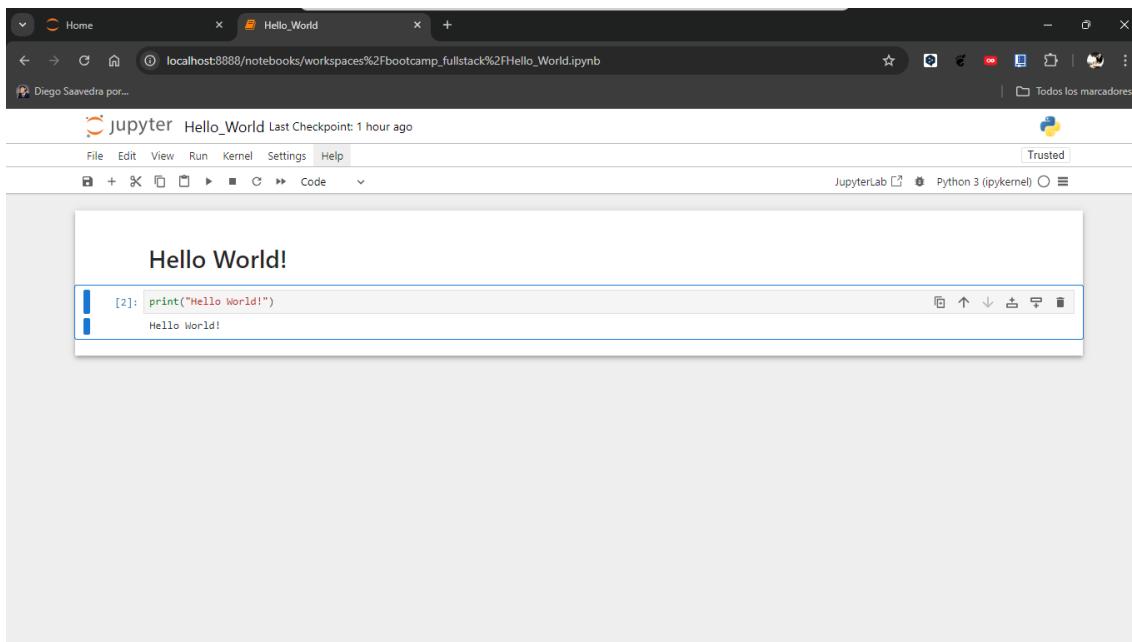


Figure 8.4: Jupyter Notebook

En este bootcam utilizaremos **Visual Studio Code** como editor de código para programar en Python. Sin embargo, te recomiendo que explores otros entornos de desarrollo y elijas el que mejor se adapte a tus necesidades y preferencias.

## 8.2 5 Consejos para mejorar la lógica de programación.

1. **Practica regularmente:** La práctica es fundamental para mejorar la lógica de programación. Dedica tiempo a resolver problemas de programación y desafíos lógicos de manera regular.
2. **Descompón el problema:** Divide los problemas complejos en problemas más pequeños y manejables. Esto te ayudará a abordar el problema de manera más efectiva y eficiente.
3. **Utiliza pseudocódigo:** Antes de escribir código, utiliza pseudocódigo para planificar y diseñar tu solución. Esto te ayudará a visualizar el problema y encontrar una solución más clara.
4. **Comenta tu código:** Utiliza comentarios para explicar tu código y hacerlo más legible. Esto te ayudará a entender tu código y a identificar posibles errores.
5. **Colabora con otros:** Trabaja en equipo con otros programadores para resolver problemas de programación. La colaboración te permitirá aprender de otros y mejorar tus habilidades de programación.

¡Espero que estos consejos te sean útiles para mejorar tu lógica de programación!

## **8.3 Conclusiones**

En este módulo hemos aprendido acerca de la introducción general a la programación, la instalación de Python, el uso de REPL, PEP 8 y Zen de Python, y los entornos de desarrollo que podemos utilizar para programar en Python.

# **9 Introducción a la Programación con Python**



Figure 9.1: Python

## **9.1 ¿Qué es la programación?**

La programación es el proceso de diseñar e implementar un programa de computadora. Un programa es un conjunto de instrucciones que le dice a la computadora qué hacer. Estas instrucciones pueden ser escritas en diferentes lenguajes de programación, como Python, Java, C++, entre otros.

## **9.2 ¿Qué es Python?**

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos. Fue creado por Guido van Rossum en 1991 y es uno de los lenguajes de programación más populares en la actualidad. Python es conocido por su sintaxis simple y legible, lo que lo hace ideal para principiantes en programación.

## **9.3 ¿Por qué aprender Python?**

Python es un lenguaje de programación versátil que se puede utilizar para una amplia variedad de aplicaciones, como desarrollo web, análisis de datos, inteligencia artificial, entre otros. Además, Python es fácil de aprender y de usar, lo que lo convierte en una excelente opción para aquellos que quieren iniciarse en la programación.

## 9.4 ¿Qué aprenderemos en este bootcamp?

En este bootcamp aprenderemos los conceptos básicos de programación con Python, incluyendo variables, tipos de datos, operadores, estructuras de control, funciones, entre otros. Al final del bootcamp, tendrás los conocimientos necesarios para crear tus propios programas en Python y continuar tu aprendizaje en programación.

¡Vamos a empezar!

## 9.5 Identación en Python

Python utiliza la identación para definir bloques de código. La identación es el espacio en blanco al principio de una línea de código y se utiliza para indicar que una línea de código pertenece a un bloque de código. Por ejemplo, en el siguiente código, la línea `print("Hola, mundo!")` está identada con cuatro espacios, lo que indica que pertenece al bloque de código del `if`.

```
if True:  
    print("Hola, mundo!")
```

En el código anterior, la línea `print("Hola, mundo!")` se ejecutará si la condición del `if` es verdadera. Si la línea no estuviera identada, no se ejecutaría dentro del bloque de código del `if`.

## 9.6 Comentarios en python

Los comentarios son líneas de texto que se utilizan para explicar el código y hacerlo más legible. En Python, los comentarios se crean utilizando el símbolo `#`. Todo lo que sigue al símbolo `#` en una línea se considera un comentario y no se ejecuta como código.

```
# Este es un comentario  
print("Hola, mundo!") # Este es otro comentario
```

En el código anterior, la línea `print("Hola, mundo!")` se ejecutará, pero los comentarios no se ejecutarán.

## 9.7 Variables y Variables Múltiples

Una variable es un contenedor que se utiliza para almacenar datos en un programa. En Python, una variable se crea asignando un valor a un nombre de variable. Por ejemplo, en el siguiente código, la variable `nombre` se crea y se le asigna el valor `"Juan"`.

```
nombre = "Juan"  
print(nombre)
```

En el código anterior, la variable **nombre** se imprime en la consola y se muestra el valor “Juan”.

En Python, también se pueden crear múltiples variables en una sola línea. Por ejemplo, en el siguiente código, se crean tres variables **a**, **b** y **c** y se les asignan los valores **1**, **2** y **3** respectivamente.

```
a, b, c = 1, 2, 3  
print(a, b, c)
```

En el código anterior, las variables **a**, **b** y **c** se imprimen en la consola y se muestran los valores **1**, **2** y **3** respectivamente.

## 9.8 Concatenación de Cadenas

La concatenación de cadenas es la unión de dos o más cadenas en una sola cadena. En Python, se puede concatenar cadenas utilizando el operador **+**. Por ejemplo, en el siguiente código, se concatenan las cadenas “Hola” y “mundo” en una sola cadena.

```
saludo = "Hola" + "mundo"  
print(saludo)
```

En el código anterior, la variable **saludo** se imprime en la consola y se muestra la cadena “Hola mundo”.

Algunos ejemplos adicionales de concatenación de cadenas son:

```
nombre = "Juan"  
apellido = "Pérez"  
nombre_completo = nombre + " " + apellido  
print(nombre_completo)
```

En el código anterior, la variable **nombre\_completo** se imprime en la consola y se muestra la cadena “Juan Pérez”.

```
edad = 30  
mensaje = "Tengo " + str(edad) + " años"  
print(mensaje)
```

En el código anterior, la variable **mensaje** se imprime en la consola y se muestra la cadena “Tengo 30 años”.

# 10 Actividad

## 10.1 instrucciones

1. Crea una variable llamada **nombre** y asígnale tu nombre.
2. Crea una variable llamada **edad** y asígnale tu edad.
3. Crea una variable llamada **ciudad** y asígnale tu ciudad de origen.
4. Imprime en la consola un mensaje que contenga tu nombre, edad y ciudad de origen utilizando la concatenación de cadenas.
5. Crea una variable llamada **mensaje** y asígnale el siguiente mensaje: “Hola, mi nombre es [nombre], tengo [edad] años y soy de [ciudad].”
6. Imprime en la consola el mensaje utilizando la variable **mensaje**.

Pistas

- Para concatenar cadenas en Python, utiliza el operador **+**.
  - Para convertir un número entero en una cadena, utiliza la función **str()**.

# 11 Conclusión

En este módulo, aprendimos los conceptos básicos de programación con Python, incluyendo variables, identación, comentarios y concatenación de cadenas. Estos conceptos son fundamentales para comprender y escribir programas en Python. En los módulos siguientes, profundizaremos en otros aspectos de la programación con Python, como tipos de datos, operadores, estructuras de control, funciones, entre otros. ¡Sigue adelante!

# 12 Tipos de Datos



Figure 12.1: Python

Los tipos de Datos en Python son la forma en que Python clasifica y almacena los datos. Los tipos de datos más comunes en Python son:

- Números
- Cadenas
- Listas
- Tuplas
- Diccionarios
- Booleanos
- Rango

En esta actividad, aprenderás sobre los diferentes tipos de datos en Python y cómo se utilizan.

## 12.1 String y Números.

Los String y los Números son dos de los tipos de datos más comunes en Python. Los String son secuencias de caracteres, como letras, números y símbolos, que se utilizan para representar texto. Los Números, por otro lado, son valores numéricos, como enteros y decimales, que se utilizan para realizar cálculos matemáticos.

### 12.1.1 String

Los String en Python se crean utilizando comillas simples ' ' o comillas dobles " ". Por ejemplo:

```
nombre = "Juan"  
apellido = 'Pérez'
```

En el código anterior, se crean dos variables, **nombre** y **apellido**, que contienen los String “Juan” y “Pérez” respectivamente.

### 12.1.2 Números

Los Números en Python pueden ser enteros o decimales. Los enteros son números enteros, como **1**, **2**, **3**, mientras que los decimales son números con decimales, como **1.5**, **2.75**, **3.14**. Por ejemplo:

```
entero = 10
decimal = 3.14
```

En el código anterior, se crean dos variables, **entero** y **decimal**, que contienen los números **10** y **3.14** respectivamente.

## 12.2 Listas y Tuplas.

Las listas y las tuplas son dos tipos de datos en Python que se utilizan para almacenar colecciones de elementos. Las listas son colecciones ordenadas y modificables de elementos, mientras que las tuplas son colecciones ordenadas e inmutables de elementos.

### 12.2.1 Listas

Las listas en Python se crean utilizando corchetes [ ] y pueden contener cualquier tipo de datos, como números, String, listas, tuplas, diccionarios, etc. Por ejemplo:

```
numeros = [1, 2, 3, 4, 5]
nombres = ["Juan", "María", "Pedro"]
```

En el código anterior, se crean dos listas, **numeros** y **nombres**, que contienen los números **1, 2, 3, 4, 5** y los nombres “Juan”, “María”, “Pedro” respectivamente.

### 12.2.2 Tuplas

Las tuplas en Python se crean utilizando paréntesis ( ) y pueden contener cualquier tipo de datos, como números, String, listas, tuplas, diccionarios, etc. Por ejemplo:

```
coordenadas = (10, 20)
colores = ("rojo", "verde", "azul")
```

En el código anterior, se crean dos tuplas, **coordenadas** y **colores**, que contienen las coordenadas **(10, 20)** y los colores “rojo”, “verde”, “azul” respectivamente.

## 12.3 Diccionarios y Booleanos.

Los diccionarios y los booleanos son dos tipos de datos en Python que se utilizan para almacenar información y tomar decisiones.

### 12.3.1 Diccionarios

Los diccionarios en Python se crean utilizando llaves {} y contienen pares de claves y valores. Por ejemplo:

```
persona = {"nombre": "Juan", "edad": 30, "ciudad": "Bogotá"}
```

En el código anterior, se crea un diccionario **persona** que contiene las claves “**nombre**”, “**edad**” y “**ciudad**” con los valores “**Juan**”, **30** y “**Bogotá**” respectivamente.

### 12.3.2 Booleanos

Los booleanos en Python son valores lógicos que pueden ser **True** o **False**. Se utilizan para tomar decisiones en un programa. Por ejemplo:

```
es_mayor_de_edad = True  
es_estudiante = False
```

En el código anterior, se crean dos variables booleanas, **es\_mayor\_de\_edad** y **es\_estudiante**, que contienen los valores **True** y **False** respectivamente.

## 12.4 Range

El tipo de datos **range** en Python se utiliza para generar una secuencia de números. Se crea utilizando la función **range()** y puede contener hasta tres argumentos: **start**, **stop** y **step**. Por ejemplo:

```
numeros = range(1, 10, 2)
```

En el código anterior, se crea un objeto **range** llamado **numeros** que contiene los números **1, 3, 5, 7, 9**.

# 13 Actividad

## 13.1 Instrucciones

1. Crea una lista llamada **numeros** que contenga los números del **1** al **10**.
2. Crea una tupla llamada **colores** que contenga los colores “rojo”, “verde” y “azul”.
3. Crea un diccionario llamado **persona** que contenga las claves “nombre”, “edad” y “ciudad” con los valores “Juan”, **30** y “Bogotá” respectivamente.
4. Crea una variable booleana llamada **es\_mayor\_de\_edad** y asígnale el valor **True**.
5. Imprime en la consola las variables **numeros**, **colores**, **persona** y **es\_mayor\_de\_edad**.
6. ¿Qué tipo de datos es la variable **numeros**? ¿Y la variable **colores**? ¿Y la variable **persona**? ¿Y la variable **es\_mayor\_de\_edad**?
7. ¿Qué tipo de datos es la variable **numeros[0]**? ¿Y la variable **colores[1]**? ¿Y la variable **persona[“nombre”]**? ¿Y la variable **es\_mayor\_de\_edad**?
8. ¿Qué tipo de datos es la variable **numeros[0:5]**? ¿Y la variable **colores[1:]**? ¿Y la variable **persona.keys()**? ¿Y la variable **es\_mayor\_de\_edad**?
9. ¿Qué tipo de datos es la variable **range(1, 10, 2)**? ¿Y la variable **range(10)**? ¿Y la variable **range(1, 10)**? ¿Y la variable **range(1, 10, 1)**?
10. ¿Qué tipo de datos es la variable **range(1, 10, 2)[0]**? ¿Y la variable **range(10)[0]**? ¿Y la variable **range(1, 10)[0]**? ¿Y la variable **range(1, 10, 1)[0]**?

Posibles soluciones

1. La variable **numeros** es una lista.
2. La variable **colores** es una tupla.
3. La variable **persona** es un diccionario.
4. La variable **es\_mayor\_de\_edad** es un booleano.
5. La variable **numeros[0]** es un número.
6. La variable **colores[1]** es un String.
7. La variable **persona[“nombre”]** es un String.
8. La variable **numeros[0:5]** es una lista.
9. La variable **range(1, 10, 2)** es un objeto **range**.
10. La variable **range(1, 10, 2)[0]** es un número.

## 14 Conclusiones

En esta actividad, aprendiste sobre los diferentes tipos de datos en Python, como los String, los Números, las Listas, las Tuplas, los Diccionarios, los Booleanos y el tipo de datos **range**.

También aprendiste cómo crear y utilizar estos tipos de datos en Python. Ahora estás listo para utilizar estos conocimientos en tus propios programas y proyectos.

¡Sigue practicando y mejorando tus habilidades de programación en Python!

# 15 Control de Flujo



Figure 15.1: Python

El control de flujo en Python se refiere a la forma en que se ejecutan las instrucciones en un programa. Python proporciona varias estructuras de control de flujo que permiten tomar decisiones, repetir tareas y ejecutar instrucciones en función de ciertas condiciones.

La sintaxis de las estructuras de control de flujo en Python se basa en la indentación, lo que significa que las instrucciones dentro de un bloque de código deben estar indentadas con la misma cantidad de espacios o tabulaciones. Esto hace que el código sea más legible y fácil de entender.

En esta sección, aprenderemos sobre las siguientes estructuras de control de flujo en Python:

- If y Condicionales
- If, elif y else
- And y Or
- While loop
- While, break y continue
- For loop

## 15.1 If y Condicionales

Para entender el concepto de If y Condicionales en Python, primero debemos comprender qué es una condición. Una condición es una expresión que se evalúa como verdadera o falsa. En Python, las condiciones se utilizan para tomar decisiones en un programa.

La estructura básica de un If en Python es la siguiente:

```
if condicion:  
    # Bloque de código si la condición es verdadera
```

En el código anterior, si la condición es verdadera, se ejecutará el bloque de código dentro del If. Si la condición es falsa, el bloque de código no se ejecutará.

Por ejemplo:

```
edad = 18

if edad >= 18:
    print("Eres mayor de edad")
```

En el código anterior, si la variable **edad** es mayor o igual a 18, se imprimirá en la consola el mensaje “Eres mayor de edad”.

## 15.2 If, elif y else

Además del If, Python también proporciona las palabras clave **elif** y **else** para tomar decisiones más complejas en un programa. La estructura básica de un If, elif y else en Python es la siguiente:

```
if condicion1:
    # Bloque de código si la condicion1 es verdadera
elif condicion2:
    # Bloque de código si la condicion2 es verdadera
else:
    # Bloque de código si ninguna de las condiciones anteriores es verdadera
```

En el código anterior, si la **condicion1** es verdadera, se ejecutará el bloque de código dentro del If. Si la **condicion1** es falsa y la **condicion2** es verdadera, se ejecutará el bloque de código dentro del **elif**. Si ninguna de las condiciones anteriores es verdadera, se ejecutará el bloque de código dentro del **else**.

Por ejemplo:

```
edad = 18

if edad < 18:
    print("Eres menor de edad")
elif edad == 18:
    print("Tienes 18 años")
else:
    print("Eres mayor de edad")
```

En el código anterior, si la variable **edad** es menor que 18, se imprimirá en la consola el mensaje “Eres menor de edad”. Si la variable **edad** es igual a 18, se imprimirá en la consola el mensaje “Tienes 18 años”. Si ninguna de las condiciones anteriores es verdadera, se imprimirá en la consola el mensaje “Eres mayor de edad”.

## 15.3 And y Or

Para entender el concepto de And y Or en Python, primero debemos comprender cómo funcionan los operadores lógicos. Los operadores lógicos se utilizan para combinar o modificar condiciones en una expresión lógica.

En Python, los operadores lógicos más comunes son **and** y **or**. El operador **and** devuelve **True** si ambas condiciones son verdaderas. El operador **or** devuelve **True** si al menos una de las condiciones es verdadera.

Por ejemplo:

```
edad = 18

if edad >= 18 and edad <= 30:
    print("Tienes entre 18 y 30 años")
```

En el código anterior, si la variable **edad** es mayor o igual a 18 y menor o igual a 30, se imprimirá en la consola el mensaje “Tienes entre 18 y 30 años”.

## 15.4 While loop

Para entender el concepto de While loop en Python, primero debemos comprender qué es un bucle. Un bucle es una estructura de control que se utiliza para repetir una secuencia de instrucciones varias veces. En Python, el bucle **while** se utiliza para repetir un bloque de código mientras una condición sea verdadera.

La estructura básica de un While loop en Python es la siguiente:

```
while condicion:
    # Bloque de código que se repetirá mientras la condición sea verdadera
```

En el código anterior, si la condición es verdadera, se ejecutará el bloque de código dentro del While loop. El bloque de código se repetirá hasta que la condición sea falsa.

Por ejemplo:

```
contador = 0

while contador < 5:
    print(contador)
    contador += 1
```

En el código anterior, se crea una variable **contador** con el valor **0**. Luego, se ejecuta un While loop que imprime el valor del **contador** y luego incrementa el **contador** en **1** en cada iteración. El bucle se repetirá hasta que el **contador** sea mayor o igual a **5**.

## 15.5 While, break y continue

Para entender el concepto de While, break y continue en Python, primero debemos comprender cómo funcionan las palabras clave **break** y **continue** en un bucle **while**.

La palabra clave **break** se utiliza para salir de un bucle **while** antes de que la condición sea falsa. La palabra clave **continue** se utiliza para saltar a la siguiente iteración del bucle **while** sin ejecutar el resto del bloque de código.

Por ejemplo:

```
contador = 0

while contador < 5:
    if contador == 3:
        break
    print(contador)
    contador += 1
```

En el código anterior, se crea una variable **contador** con el valor **0**. Luego, se ejecuta un While loop que imprime el valor del **contador** y luego incrementa el **contador** en **1** en cada iteración. Si el **contador** es igual a **3**, se ejecuta la palabra clave **break** y se sale del bucle.

## 15.6 For loop

Para entender el concepto de For loop en Python, primero debemos comprender cómo funciona un bucle **for**. Un bucle **for** se utiliza para iterar sobre una secuencia de elementos, como una lista, una tupla, un diccionario, etc.

La estructura básica de un For loop en Python es la siguiente:

```
for elemento in secuencia:
    # Bloque de código que se repetirá para cada elemento en la secuencia
```

En el código anterior, el bucle **for** recorre cada elemento en la secuencia y ejecuta el bloque de código para cada elemento.

Por ejemplo:

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

for numero in numeros:
    print(numero)
```

En el código anterior, se crea una lista **numeros** con los números del **1** al **5**. Luego, se ejecuta un For loop que imprime cada número en la lista.

# 16 Actividad

## 16.1 instrucciones

1. Crea una lista llamada **numeros** que contenga los números del **1** al **10**.
2. Crea una tupla llamada **colores** que contenga los colores “rojo”, “verde” y “azul”.
3. Crea un diccionario llamado **persona** que contenga las claves “nombre”, “edad” y “ciudad” con los valores “Diego”, 36 y “Quito” respectivamente.
4. Crea una variable booleana llamada **es\_mayor\_de\_edad** y asígnale el valor **True**.
5. Imprime en la consola las variables **numeros**, **colores**, **persona** y **es\_mayor\_de\_edad**.
6. ¿Qué tipo de datos es la variable **numeros**? ¿Y la variable **colores**? ¿Y la variable **persona**? ¿Y la variable **es\_mayor\_de\_edad**?
7. ¿Qué tipo de datos es la variable **numeros[0]**? ¿Y la variable **colores[1]**? ¿Y la variable **persona[“nombre”]**? ¿Y la variable **es\_mayor\_de\_edad**?
8. ¿Qué tipo de datos es la variable **numeros[0:5]**? ¿Y la variable **colores[1:]**? ¿Y la variable **persona.keys()**? ¿Y la variable **es\_mayor\_de\_edad**?
9. ¿Qué tipo de datos es la variable **range(1, 10, 2)**? ¿Y la variable **range(10)**? ¿Y la variable **range(1, 10)**? ¿Y la variable **range(1, 10, 1)**?
10. ¿Qué tipo de datos es la variable **range(1, 10, 2)[0]**? ¿Y la variable **range(10)[0]**? ¿Y la variable **range(1, 10)[0]**? ¿Y la variable **range(1, 10, 1)[0]**?

Posibles soluciones

1. La variable **numeros** es una lista.
2. La variable **colores** es una tupla.
3. La variable **persona** es un diccionario.
4. La variable **es\_mayor\_de\_edad** es un booleano.
5. La variable **numeros[0]** es un número.
6. La variable **colores[1]** es un String.
7. La variable **persona[“nombre”]** es un String.
8. La variable **numeros[0:5]** es una lista.
9. La variable **range(1, 10, 2)** es un objeto **range**.
10. La variable **range(1, 10, 2)[0]** es un número.

## 17 Conclusiones

En esta sección, aprendimos sobre las estructuras de control de flujo en Python, como If, elif, else, And, Or, While loop y For loop. Estas estructuras nos permiten tomar decisiones, repetir tareas y ejecutar instrucciones en función de ciertas condiciones en un programa. Es importante comprender cómo funcionan estas estructuras para poder escribir código más eficiente y legible en Python.

# 18 Funciones y recursividad.



Figure 18.1: Python

Las funciones son bloques de código reutilizables que realizan una tarea específica. En Python, las funciones se definen utilizando la palabra clave **def** seguida del nombre de la función y los parámetros entre paréntesis. Por ejemplo:

```
def saludar():
    print("Hola, ¿cómo estás?")
```

En el código anterior, se define una función llamada **saludar** que imprime en la consola el mensaje “Hola, ¿cómo estás?”. Para llamar a una función en Python, simplemente se escribe el nombre de la función seguido de paréntesis. Por ejemplo:

```
saludar()
```

En el código anterior, se llama a la función **saludar** y se imprime en la consola el mensaje “Hola, ¿cómo estás?”.

## 18.1 Introducción a Funciones

Para entender de mejor forma cómo funcionan las funciones en Python, vamos a crear una función que reciba dos números como parámetros y devuelva la suma de los mismos. Por ejemplo:

```
def sumar(a, b):
    return a + b
```

En el código anterior, se define una función llamada **sumar** que recibe dos parámetros **a** y **b** y devuelve la suma de los mismos. Para llamar a la función **sumar** y obtener el resultado, se puede hacer de la siguiente manera:

```
resultado = sumar(5, 3)
print(resultado)
```

En el código anterior, se llama a la función **sumar** con los números **5** y **3** como parámetros y se imprime en la consola el resultado **8**.

## 18.2 Parámetros y Argumentos

En Python, los parámetros son las variables que se definen en la declaración de la función, mientras que los argumentos son los valores que se pasan a la función cuando se llama. Por ejemplo:

```
def saludar(nombre):
    print("Hola, " + nombre + "!")
```

En el código anterior, la función **saludar** tiene un parámetro llamado **nombre**. Para llamar a la función **saludar** con un argumento, se puede hacer de la siguiente manera:

```
saludar("Juan")
```

En el código anterior, se llama a la función **saludar** con el argumento “**Juan**” y se imprime en la consola el mensaje “Hola, Juan!”.

## 18.3 Retorno de valores

En Python, las funciones pueden devolver valores utilizando la palabra clave **return** seguida del valor que se desea devolver. Por ejemplo:

```
def sumar(a, b):
    return a + b
```

En el código anterior, la función **sumar** devuelve la suma de los números **a** y **b**. Para obtener el valor devuelto por la función, se puede asignar a una variable y luego imprimir en la consola. Por ejemplo:

```
resultado = sumar(5, 3)
print(resultado)
```

En el código anterior, se llama a la función **sumar** con los números **5** y **3** como parámetros, se asigna el resultado a la variable **resultado** y se imprime en la consola el valor **8**.

## 18.4 Recursividad

La recursividad es un concepto en programación en el que una función se llama a sí misma para resolver un problema más pequeño. Por ejemplo, la función factorial se puede definir de forma recursiva de la siguiente manera:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

En el código anterior, la función **factorial** calcula el factorial de un número **n** de forma recursiva. Para llamar a la función **factorial** y obtener el resultado, se puede hacer de la siguiente manera:

```
resultado = factorial(5)
print(resultado)
```

En el código anterior, se llama a la función **factorial** con el número **5** como parámetro y se imprime en la consola el resultado **120**.

Otro ejemplo de recursividad es la función Fibonacci, que calcula el enésimo término de la secuencia de Fibonacci de forma recursiva. Por ejemplo:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

En el código anterior, la función **fibonacci** calcula el enésimo término de la secuencia de Fibonacci de forma recursiva. Para llamar a la función **fibonacci** y obtener el resultado, se puede hacer de la siguiente manera:

```
resultado = fibonacci(10)
print(resultado)
```

En el código anterior, se llama a la función **fibonacci** con el número **10** como parámetro y se imprime en la consola el resultado **55**.

# 19 Actividad

## 19.1 Instrucciones

1. Crea una función llamada **saludar** que reciba un parámetro **nombre** y devuelva un saludo personalizado. Por ejemplo, si el nombre es “**Juan**”, la función debe devolver el mensaje “**Hola, Juan!**”.
2. Crea una función llamada **calcular\_promedio** que reciba una lista de números como parámetro y devuelva el promedio de los mismos. Por ejemplo, si la lista es **[1, 2, 3, 4, 5]**, la función debe devolver **3.0**.
3. Crea una función llamada **es\_par** que reciba un número como parámetro y devuelva **True** si el número es par y **False** si no lo es.
4. Crea una función llamada **calcular\_factorial** que reciba un número como parámetro y devuelva el factorial del mismo. Por ejemplo, si el número es **5**, la función debe devolver **120**.
5. Crea una función llamada **calcular\_fibonacci** que reciba un número como parámetro y devuelva el enésimo término de la secuencia de Fibonacci. Por ejemplo, si el número es **10**, la función debe devolver **55**.
6. Llama a cada una de las funciones creadas con valores de ejemplo y muestra los resultados en la consola.

### Pistas

- Para definir una función en Python, utiliza la palabra clave **def** seguida del nombre de la función y los parámetros entre paréntesis.
  - Para devolver un valor en una función, utiliza la palabra clave **return** seguida del valor que deseas devolver.
  - Para llamar a una función en Python, simplemente escribe el nombre de la función seguido de paréntesis y los argumentos si es necesario.

## 20 Conclusiones

Las funciones y la recursividad son conceptos fundamentales en programación que nos permiten escribir código más modular, reutilizable y eficiente. Al entender cómo funcionan las funciones y cómo se pueden llamar de forma recursiva, podemos resolver una amplia variedad de problemas de programación de manera más sencilla y elegante. Además, las funciones nos permiten encapsular la lógica de nuestro código y separar las diferentes tareas en bloques más pequeños y manejables. En resumen, las funciones y la recursividad son herramientas poderosas que nos ayudan a escribir un código más limpio, organizado y fácil de mantener.

## **Part III**

# **Unidad 2: Programación Orientada a Objetos**

# 21 Programacion Orientada a Objetos.



Figure 21.1: Python

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza objetos y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, encapsulación, polimorfismo y abstracción.

Su sintaxis es más clara y sencilla de entender que otros paradigmas de programación. Al permitirnos modelar entidades del mundo real de forma más directa.

Ejemplo:

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def acelerar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está acelerando")

    def frenar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está frenando")

    def __str__(self):
        return f"Coche {self.marca} {self.modelo} de color {self.color}"
```

En el código anterior se define una clase **Coche** con tres atributos **marca**, **modelo** y **color**. Además, se definen tres métodos **acelerar**, **frenar** y **str**. El método **str** es un método especial que se llama cuando se convierte un objeto a una cadena de texto.

Para crear un objeto de la clase **Coche** se hace de la siguiente manera:

```
coche = Coche("Toyota", "Corolla", "Rojo")
print(coche)
coche.acelerar()
coche.frenar()
```

En el código anterior se crea un objeto **coche** de la clase **Coche** con los atributos **Toyota**, **Corolla** y **Rojo**. Luego se imprime el objeto **coche** y se llama a los métodos **acelerar** y **frenar**.

## 21.1 Objetos y Clases

Los objetos son instancias de una clase. Una clase es una plantilla para crear objetos. Los objetos tienen atributos y métodos.

## 21.2 Atributos

Los atributos son variables que pertenecen a un objeto. Los atributos pueden ser de cualquier tipo de datos.

Ejemplo:

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
```

En el código anterior se definen tres atributos **marca**, **modelo** y **color**.

## 21.3 ¿Qué es self?

Self es una palabra reservada en Python que se refiere al objeto actual. Se utiliza para acceder a los atributos y métodos de un objeto.

En el ejemplo anterior, **self.marca**, **self.modelo** y **self.color** se refieren a los atributos de un objeto.

Ejemplo:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años")
```

En el ejemplo anterior se define una clase **Persona** con dos atributos **nombre** y **edad**. Además, se define un método **saludar** que imprime un mensaje con los atributos **nombre** y **edad**.

## 21.4 Métodos

Los métodos son funciones que pertenecen a un objeto. Los métodos pueden acceder a los atributos de un objeto.

Ejemplo:

```
class Coche:
    def acelerar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está acelerando")

    def frenar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está frenando")
```

En el código anterior se definen dos métodos **acelerar** y **frenar**.

## 21.5 Self, Eliminar Propiedades y Objetos

El primer parámetro de un método es **self**. **Self** es una referencia al objeto actual. Se utiliza para acceder a los atributos y métodos de un objeto.

Ejemplo:

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def acelerar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está acelerando")

    def frenar(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está frenando")

    def __del__(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} ha sido eliminado")

coche = Coche("Toyota", "Corolla", "Rojo")
print(coche)
coche.acelerar()
coche.frenar()
del coche
```

En el código anterior se define un método especial **del** que se llama cuando un objeto es eliminado. Luego se crea un objeto **coche** de la clase **Coche** y se elimina el objeto **coche**.

Por otra parte la palabra reservada **self** se utiliza para acceder a los atributos y métodos de un objeto.

Tambien se está creando una instancia de la clase **Coche** y se está eliminando el objeto **coche**.

## 21.6 Eliminar Propiedades y Objetos

Para eliminar Propiedades y Objetos se utiliza la palabra reservada **del**.

Como observamos en el código anterior la propiedad **del** se utiliza para eliminar un objeto.

Ejemplo:

```
class Persona:  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
    def __del__(self):  
        print(f"La persona {self.nombre} ha sido eliminada")  
  
persona = Persona("Juan Perez", 30)  
print(persona)  
del persona
```

En el código anterior se define un método especial **del** que se llama cuando un objeto es eliminado. Luego se crea un objeto **persona** de la clase **Persona** y se elimina el objeto **persona**. Al final obtendremos un mensaje como este:

```
La persona Juan Perez ha sido eliminada
```

## 21.7 Herencia, Polimorfismo y Encapsulación

### 21.7.1 Herencia

La herencia es una característica de la POO que permite crear una nueva clase a partir de una clase existente. La nueva clase hereda los atributos y métodos de la clase existente.

Ejemplo:

```

class Animal:
    def __init__(self, nombre):
        self.nombre = nombre
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        print(f"{self.nombre} dice guau")

class Gato(Animal):
    def hablar(self):
        print(f"{self.nombre} dice miau")

animal = Perro("Firulais")
animal2 = Gato("Garfield")

```

En el código anterior se define una clase **Animal** con un método **hablar**. Luego se definen dos clases **Perro** y **Gato** que heredan de la clase **Animal** y sobrescriben el método **hablar**.

### 21.7.2 Polimorfismo

El polimorfismo es una característica de la POO que permite que un objeto se computadora de diferentes maneras dependiendo del contexto.

Ejemplo:

```

class Deporte:
    def jugar(self):
        pass

class Futbol(Deporte):
    def jugar(self):
        print(f"Jugando futbol")

class Baloncesto(Deporte):
    def jugar(self):
        print(f"Jugando baloncesto")

class Tenis(Deporte):
    def jugar(self):
        print(f"Jugando tenis")

deporte = Futbol()
deporte.jugar()

```

```

deporte1 = Baloncesto()
deporte1.jugar()

deporte2 = Tenis()
deporte2.jugar()

```

En el ejemplo anterior se define una clase **Deporte** con un método **jugar**. Luego se definen tres clases **Futbol**, **Baloncesto** y **Tenis** que heredan de la clase **Deporte** y sobrescriben el método **jugar**. Aunque los tres objetos son de la clase **Deporte**, se comportan de manera diferente.

### 21.7.3 Encapsulación

La encapsulación es una característica de la POO que permite ocultar los detalles de implementación de un objeto. Los atributos y métodos de un objeto pueden ser públicos, protegidos o privados.

Ejemplo:

```

class CuentaBancaria:
    def __init__(self, nombre, saldo):
        self.nombre = nombre
        self.__saldo = saldo # El saldo es privado

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Fondos insuficientes")

    def obtener_saldo(self):
        return self.__saldo # Método para acceder al saldo

    def __str__(self):
        return f"Cuenta Bancaria de {self.nombre} con saldo {self.__saldo}"

# Creación de instancias de cuentas bancarias
cuenta1 = CuentaBancaria("Juan Perez", 1000)
cuenta2 = CuentaBancaria("Maria Lopez", 2000)
cuenta3 = CuentaBancaria("Pedro Ramirez", 3000)

# Operaciones en las cuentas
cuenta1.depositar(500)

```

```

cuenta1.retirar(200)
print(cuenta1.nombre)
print(cuenta1.obtener_saldo()) # Acceso al saldo a través de un método

print(cuenta2.nombre)
cuenta2.depositar(500)
cuenta2.retirar(200)
print(cuenta2.obtener_saldo())

print(cuenta3.nombre)
cuenta3.depositar(1000)
cuenta3.retirar(500)
print(cuenta3.obtener_saldo())

```

La encapsulación es un principio fundamental en la programación orientada a objetos que permite proteger los datos de un objeto. En Python, se logra utilizando variables privadas y métodos de acceso para controlar cómo se accede y modifica la información dentro de una clase.

En el ejemplo de CuentaBancaria, el atributo **saldo** es privado (**indicado por el prefijo \_\_**) y no puede ser accedido directamente desde fuera de la clase. Esto significa que no se puede escribir cuenta1.\_\_saldo para leer o modificar el saldo.

Para interactuar con el saldo de manera segura, la clase proporciona métodos públicos como depositar y retirar, que permiten modificar el saldo solo bajo condiciones controladas. En este caso, se agregó un método obtener\_saldo para acceder al saldo de manera segura. Este enfoque evita que se altere el saldo de forma indebida y permite implementar lógica adicional, como verificar si hay fondos suficientes antes de retirar una cantidad.

Este ejemplo demuestra cómo la encapsulación ayuda a proteger y controlar el acceso a los datos de un objeto, asegurando que su estado interno se gestione correctamente.

## 21.8 Actividad

1. Crear una clase **Persona** con los atributos **nombre**, **edad** y **sexo**.
2. Crear una clase **Estudiante** que herede de la clase **Persona** con los atributos **carnet** y **carrera**.
3. Crear una clase **Profesor** que herede de la clase **Persona** con los atributos **codigo** y **especialidad**.
4. Crear una clase **Curso** con los atributos **nombre**, **codigo** y **profesor**.
5. Crear una clase **Universidad** con los atributos **nombre** y **cursos**.
6. Crear un objeto **universidad** de la clase **Universidad** con el nombre **Universidad de El Salvador** y los siguientes cursos:
  - **Curso 1:** Nombre: **Matematicas**, Código: **MAT101**, Profesor: **Juan Perez**

- **Curso 2:** Nombre: **Fisica**, Codigo: **FIS101**, Profesor: **Maria Lopez**
- **Curso 3:** Nombre: **Quimica**, Codigo: **QUI101**, Profesor: **Pedro Ramirez**

7. Imprimir el objeto **universidad**.
8. Crear un objeto **estudiante** de la clase **Estudiante** con los siguientes atributos:
  - Nombre: **Carlos Perez**
  - Edad: **20**
  - Sexo: **Masculino**
  - Carnet: **202010101**
  - Carrera: **Ingenieria en Sistemas Informaticos**
9. Imprimir el objeto **estudiante**.
10. Crear un objeto **profesor** de la clase **Profesor** con los siguientes atributos:
  - Nombre: **Juan Perez**
  - Edad: **30**
  - Sexo: **Masculino**
  - Codigo: **202020202**
  - Especialidad: **Matematicas**
11. Imprimir el objeto **profesor**.
12. Crear un objeto **curso** de la clase **Curso** con los siguientes atributos:
  - Nombre: **Matematicas**
  - Codigo: **MAT101**
  - Profesor: **Juan Perez**
13. Imprimir el objeto **curso**.
14. Agregar el objeto **curso** al objeto **universidad**.
15. Imprimir el objeto **universidad**.
16. Crear un objeto **curso** de la clase **Curso** con los siguientes atributos:
  - Nombre: **Fisica**
  - Codigo: **FIS101**
  - Profesor: **Maria Lopez**

Respuesta

```

class Persona:
    def __init__(self, nombre, edad, sexo):
        self.nombre = nombre
        self.edad = edad
        self.sexo = sexo

class Estudiante(Persona):
    def __init__(self, nombre, edad, sexo, carnet, carrera):
        super().__init__(nombre, edad, sexo)
        self.carnet = carnet
        self.carrera = carrera

class Profesor(Persona):
    def __init__(self, nombre, edad, sexo, codigo, especialidad):
        super().__init__(nombre, edad, sexo)
        self.codigo = codigo
        self.especialidad = especialidad

class Curso:
    def __init__(self, nombre, codigo, profesor):
        self.nombre = nombre
        self.codigo = codigo
        self.profesor = profesor

class Universidad:
    def __init__(self, nombre):
        self.nombre = nombre
        self.cursos = []

universidad = Universidad("Universidad de las Fuerzas Armadas ESPE")
curso1 = Curso("Matematicas", "MAT101", "Juan Perez")
curso2 = Curso("Fisica", "FIS101", "Maria Lopez")
curso3 = Curso("Quimica", "QUI101", "Pedro Ramirez")
universidad.cursos.append(curso1)
universidad.cursos.append(curso2)
universidad.cursos.append(curso3)
print(universidad)

estudiante = Estudiante("Carlos Perez", 20, "Masculino", "202010101", "Ingenieria en Sist
print(estudiante)

profesor = Profesor("Juan Perez", 30, "Masculino", "202020202", "Matematicas")
print(profesor)

curso = Curso("Matematicas", "MAT101", "Juan Perez")
print(curso)

curso = Curso("Fisica", "FIS101", "Maria Lopez")

```

```
universidad.cursos.append(curso)
print(universidad)
```

## **22 Conclusiones**

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza objetos y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, encapsulación, polimorfismo y abstracción.

## **Part IV**

# **Unidad 3: Módulos y Paquetes**

# 23 Módulos



Figure 23.1: Python

## 23.1 Introducción a Módulos

En Python, un módulo es un archivo que contiene código, generalmente funciones, clases, y variables, que puedes importar y reutilizar en diferentes partes de tu programa. Esto te ayuda a dividir tu código en partes organizadas y reutilizables, haciendo que el desarrollo sea más eficiente y limpio.

Por ejemplo, imagina que quieras crear un módulo para realizar saludos y otro para despedidas:

Ejemplo 1: Módulo de saludo

```
# modulo_saludo.py
def saludar():
    print("Hola Mundo")
```

Ejemplo 2: Módulo de despedida

```
# modulo_despedida.py
def despedir():
    print("Adiós Mundo")
```

Estos módulos contienen funciones que realizan acciones específicas: uno saluda y el otro se despide. Ahora veremos cómo crear y utilizar módulos en Python.

## 23.2 Creando Módulos Personalizados

Para crear un módulo en Python, solo necesitas crear un archivo .py y definir en él las funciones o clases que deseas usar. A continuación, veamos cómo crear módulos más complejos.

Ejemplo: Módulo de saludo con nombre

```
# modulo_saludar.py

def saludar(nombre):
    print(f"Hola, {nombre}!")
```

Este módulo acepta un argumento nombre, permitiéndote personalizar el saludo.

## 23.3 Usando Módulos en un Archivo Principal

Para utilizar los módulos que has creado, necesitas importarlos en un archivo principal. Aquí, importamos ambos módulos anteriores y ejecutamos las funciones:

```
# main.py
import modulo_saludar
import modulo_despedida

if __name__ == "__main__":
    modulo_saludar.saludar("Juan")
    modulo_despedida.despedir()
```

En este ejemplo, importamos los módulos modulo\_saludar y modulo\_despedida y usamos las funciones saludar y despedir.

## 23.4 Importando y Renombrando Módulos

A veces, renombrar un módulo en el momento de importarlo hace el código más claro. Esto se logra con la palabra clave as:

```
# main.py
import modulo_saludar as saludo
import modulo_despedida as despedida

saludo.saludar("Ana")
despedida.despedir()
```

Esto permite usar nombres cortos y descriptivos en el código.

## 23.5 Importando Funciones Específicas de un Módulo

Si solo necesitas una función de un módulo, puedes importarla directamente:

```
from modulo_saludar import saludar  
saludar("Carlos")
```

Aquí importamos únicamente la función saludar de modulo\_saludar, sin necesidad de importar el módulo completo.

## 23.6 Usando Módulos Externos con pip

Además de tus propios módulos, Python permite instalar y utilizar módulos externos usando pip, el gestor de paquetes de Python. Veamos un ejemplo con numpy, un módulo popular para trabajar con arreglos numéricos.

## 23.7 Instalando un módulo con pip

```
pip install numpy
```

## 23.8 Usando el módulo instalado

```
import numpy as np  
  
a = np.array([1, 2, 3, 4, 5])  
print(a)
```

Este ejemplo muestra cómo instalar y utilizar numpy para crear un arreglo.

## 23.9 Instalando otro módulo

Además de numpy, Python tiene muchos módulos útiles para diferentes tareas. Por ejemplo, emojis es un módulo que te permite imprimir emojis en la consola.

```
pip install emojis
```

## 23.10 Usando el módulo emojis

```
import emojis  
print(emojis.encode(":smile:"))
```

Este ejemplo muestra cómo instalar y utilizar el módulo emojis para imprimir emojis en la consola.

## 24 Actividad Práctica

Sigue estos pasos para practicar la creación y uso de módulos en Python.

1. Crear un módulo modulo\_calculadora.py que contenga las funciones sumar, restar, multiplicar, y dividir:

Ver solución

```
# modulo_calculadora.py

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

def dividir(a, b):
    return a / b
```

Crear un archivo main.py que importe el módulo modulo\_calculadora y utilice sus funciones:

```
# main.py
import modulo_calculadora

print(modulo_calculadora.sumar(10, 5))
print(modulo_calculadora.restar(10, 5))
print(modulo_calculadora.multiplicar(10, 5))
print(modulo_calculadora.dividir(10, 5))
```

2. Instalar numpy y crear un archivo main\_numpy.py que lo use para crear un arreglo:

Ver solución

```
# main_numpy.py
import numpy as np

a = np.array([1, 2, 3, 4, 5])
print(a)
```

Crear un archivo main\_pandas.py que utilice pandas para crear un DataFrame y lo imprima:

```
# main_pandas.py
import pandas as pd

data = {'Nombre': ['Juan', 'Ana', 'Luis'], 'Edad': [23, 30, 27]}
df = pd.DataFrame(data)
print(df)
```

3. Crear un archivo main\_matplotlib.py para graficar una función:

Ver solución

```
# main_matplotlib.py
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```

4. Instalar el módulo emojis y crear un archivo main\_emojis.py que imprima emojis en la consola:

Ver solución

```
# main_emojis.py
import emojis

print(emojis.encode(":smile:"))
print(emojis.encode(":heart:"))
print(emojis.encode(":rocket:"))
```

## **25 Conclusión**

Los módulos en Python son una herramienta poderosa para estructurar y reutilizar código. Con módulos, puedes dividir tu código en archivos independientes y organizados, lo cual facilita el desarrollo de aplicaciones escalables y mantenibles.

## **Part V**

# **Unidad 4: Docker**

## 26 Docker

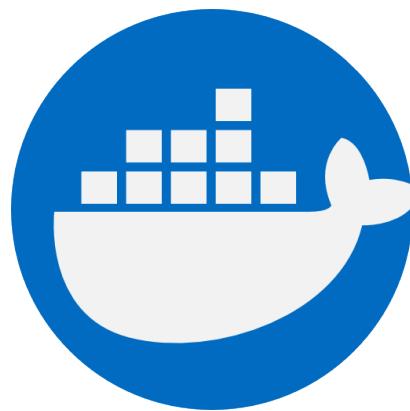


Figure 26.1: Docker

Docker es una plataforma que permite desarrollar, enviar y ejecutar aplicaciones en contenedores. Un **contenedor** es una instancia ejecutable de una **imagen**, que es una especie de **plantilla** que contiene todo lo necesario para ejecutar una aplicación.

Haciendo una analogía con los contenedores de transporte, una **imagen** sería el **contenedor** en sí, y el **contenedor** sería la **carga** que se transporta.

Docker resuelve un problema principal en el desarrollo de software: la **portabilidad**. Al empaquetar una aplicación y sus dependencias en un contenedor, se garantiza que la aplicación se ejecute de manera **consistente** en diferentes entornos.

En esta lección, aprenderemos a crear y ejecutar contenedores Docker, y a utilizarlos para ejecutar aplicaciones de manera aislada y portátil.

Con docker se acaba la frase típica de los desarrolladores **En mi máquina funciona**. Con Docker, puedes estar seguro de que tu aplicación funcionará de la misma manera en cualquier entorno.



*Docker*

Una imagen Docker es una plantilla inmutable que contiene un conjunto de instrucciones para crear un contenedor Docker. Las imágenes son portátiles y pueden ser compartidas, almacenadas y actualizadas.

#### 💡 Tip

Las imágenes Docker son inmutables, lo que significa que no se pueden modificar una vez creadas. Si se realizan cambios en una imagen, se debe crear una nueva versión de la imagen.



## Container

Un contenedor Docker es una instancia ejecutable de una imagen Docker. Se ejecuta de manera aislada y contiene todo lo necesario para ejecutar la aplicación, incluyendo el código, las dependencias, el entorno de ejecución, las bibliotecas y los archivos de configuración.

#### 💡 Tip

Un contenedor aisla la aplicación de su entorno, lo que garantiza que la aplicación se ejecute de manera consistente en diferentes entornos.

## 26.1 Ejemplos:

Descargar una imagen:

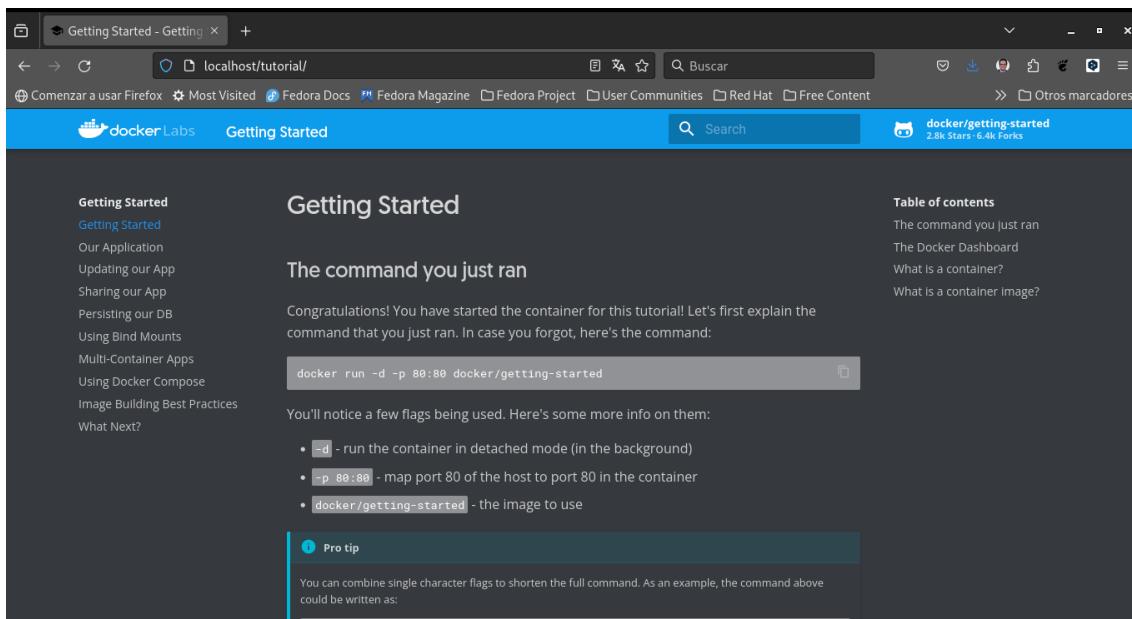
```
docker pull docker/getting-started
```

```
statick main U:11 ?:? ~/workspaces/curso_docker/book docker pull docker/getting-started
Using default tag: latest
latest: Pulling from docker/getting-started
Digest: sha256:d79336f4812b6547a53e735480dde67f8f8f7071b414fb9297609ffb989abc1
Status: Image is up to date for docker/getting-started:latest
docker.io/docker/getting-started:latest
```

Este comando descarga la imagen **getting-started** desde el registro público de Docker.

Correr un contenedor en el puerto 80:

```
docker run -d -p 80:80 docker/getting-started
```



Este comando ejecuta un contenedor desenlazado en segundo plano (-d) y mapea el puerto 80 de la máquina host al puerto 80 del contenedor (-p 80:80).

Descargar una imagen desde un registro.

#### 💡 Tip

El comando -p se utiliza para mapear los puertos de la máquina host al contenedor, muchas personas consideran que significa “puerto”. Sin embargo en realidad significa “publicar” o “publicar puerto”.

## 26.2 Comandos básicos de Docker:

Descargar una imagen desde un registro.

```
docker pull <IMAGE_NAME:TAG>
```

Listar las imágenes descargadas.

```
docker images
```

Listar contenedores en ejecución.

```
docker ps
```

Listar todos los contenedores, incluyendo los detenidos.

```
docker ps -a
```

Ejecutar un contenedor a partir de una imagen.

```
docker run -d -p <HOST_PORT>:<CONTAINER_PORT> <IMAGE_NAME:TAG>
```

Detener un contenedor en ejecución.

```
docker stop <CONTAINER_ID>
```

Iniciar un contenedor detenido.

```
docker start <CONTAINER_ID>
```

Eliminar un contenedor.

```
docker rm <CONTAINER_ID>
```

Eliminar una imagen.

```
docker rmi <IMAGE_NAME:TAG>
```

## 26.3 Atajos y Comandos Adicionales:

Ejecutar comandos dentro de un contenedor en ejecución.

```
docker inspect <CONTAINER_ID or IMAGE_NAME:TAG>
```

Ver los logs de un contenedor.

```
docker logs <CONTAINER_ID>
```

Utilizar Docker Compose para gestionar aplicaciones multi-contenedor.

```
docker-compose up -d
```

## 26.4 Práctica:

- Descarga la imagen de Nginx desde el registro público.
- Crea y ejecuta un contenedor de Nginx en el puerto 8080.
- Detén y elimina el contenedor creado
- Utiliza los comandos para detener y eliminar un contenedor.

Resolución de la Actividad Práctica

1. Abre tu terminal o línea de comandos.
2. Descarga la imagen de Nginx desde el registro público de Docker:

```
docker pull nginx
```

3. Crea y ejecuta un contenedor de Nginx en el puerto 8080:

```
docker run -d -p 8080:80 nginx
```

Elige un puerto en tu máquina local (por ejemplo, 8080) para mapearlo al puerto 80 del contenedor.

4. Verifica que el contenedor esté en ejecución:

```
docker ps
```

5. Si el contenedor está en ejecución, deténlo utilizando el siguiente comando:

```
docker stop <CONTAINER_ID>
```

Reemplaza `<CONTAINER_ID>` con el ID real del contenedor que obtuviste en el paso anterior.

6. Elimina el contenedor detenido:

```
docker rm <CONTAINER_ID>
```

Reemplaza `<CONTAINER_ID>` con el ID real del contenedor.



Tip

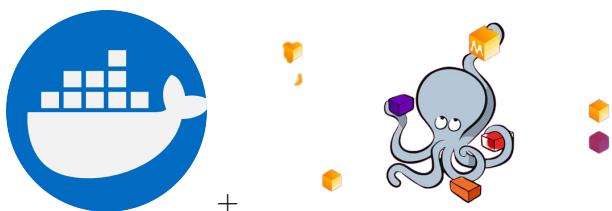
Combina los comandos `docker ps`, `docker stop`, y `docker rm` para gestionar contenedores eficientemente.

¡Practica estos pasos para familiarizarte con el ciclo de vida de los contenedores Docker!

## 27 Conclusiones

En esta lección aprendimos sobre la creación y uso de módulos en Python, así como la creación y ejecución de contenedores Docker. Los módulos son archivos que contienen funciones y variables que pueden ser reutilizadas en otros programas. Los contenedores Docker son instancias ejecutables de imágenes que contienen todo lo necesario para ejecutar una aplicación.

# 28 Dockerfile y Docker Compose



## 28.1 Introducción

Dockerfile y Docker Compose son herramientas esenciales para la construcción y gestión de aplicaciones Docker. Un Dockerfile es un archivo de texto que define cómo se construirá una imagen Docker, mientras que Docker Compose es una herramienta para definir y gestionar aplicaciones Docker con múltiples contenedores. En esta lección, aprenderemos cómo usar Dockerfile y Docker Compose para personalizar imágenes Docker y orquestar servicios en un entorno multi-contenedor.

A continuación veremos algunos conceptos básicos sobre Dockerfile y Docker Compose.

### 28.1.1 Dockerfile

Un Dockerfile es un archivo de texto que contiene una serie de instrucciones para construir una imagen Docker. Estas instrucciones incluyen la configuración del sistema operativo base, la instalación de paquetes y dependencias, la configuración de variables de entorno y la definición de comandos para ejecutar la aplicación.

### 28.1.2 Docker Compose

Docker Compose es una herramienta para definir y gestionar aplicaciones Docker con múltiples contenedores. Permite definir servicios, redes y volúmenes en un archivo YAML y orquestar la ejecución de los contenedores en un entorno de desarrollo o producción.

## 28.2 Ejemplos:

En este ejemplo vamos a dockerizar una aplicación nodejs con un servidor sencillo en express.

Empezamos por el código de nuestra aplicación:

Para ello creamos un nuevo proyecto nodejs con el siguiente comando:

```
npm init -y
```

Instalamos el paquete express con el siguiente comando:

```
npm install express
```

Creamos los siguientes archivos:

- server.js
- package.json
- Dockerfile
- docker-compose.yml

### 28.2.1 server.js

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

### 28.2.2 Dockerfile

```
# Use the official Node.js 14 image
FROM node:14

# Set the working directory in the container
WORKDIR /app

# Copy the dependencies file to the working directory
COPY package.json .

# Install dependencies
RUN npm install

# Copy the app code to the working directory
```

```
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Serve the app
CMD ["node", "server.js"]
```

### 28.2.3 docker-compose.yml

```
services:
  myapp:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    volumes:
      - .:/app
```

En este ejemplo, el Dockerfile define una imagen Docker para una aplicación Node.js. El archivo docker-compose.yml define un servicio llamado myapp que utiliza el Dockerfile.nodejs para construir la imagen y expone el puerto 3000 para acceder a la aplicación.

#### 💡 Tip

El puerto del lado izquierdo de los 2 puntos en el archivo docker-compose.yml es el puerto en el host, mientras que el puerto del lado derecho es el puerto en el contenedor.

Para probar nuestro ejemplo, ejecutamos el siguiente comando:

```
docker-compose up -d
```

Esto construirá la imagen Docker y ejecutará el contenedor en segundo plano. Podemos acceder a la aplicación en <http://localhost:3000>.

Para verificar que el contenedor está en ejecución, ejecutamos el siguiente comando:

```
docker ps
```

Podemos utilizar una aplicación como Thunder Client o Postman para enviar una solicitud HTTP a la aplicación y ver la respuesta.

Para detener y eliminar el contenedor, ejecutamos el siguiente comando:

```
docker-compose down
```

💡 Tip

Recuerda: La imagen que se crea a partir del Dockerfile se almacena en el caché local de Docker. Si realizas cambios en el Dockerfile y deseas reconstruir la imagen, puedes usar el comando

```
docker-compose up --build
```

## 28.3 Práctica:

- Crea un Dockerfile para una aplicación Python simple.
- Configura un archivo docker-compose.yml para ejecutar la aplicación.

Resolución de la Actividad Práctica

Ejemplo de aplicación Python simple:

```
# app.py
print("Hello, World!")
```

Ejemplo de Dockerfile:

```
FROM python:3.12
WORKDIR /app
COPY . .
CMD ["python", "app.py"]
```

Ejemplo de docker-compose.yml:

```
services:
  myapp:
    build:
      context: .
      dockerfile: Dockerfile.python
    image: my-python-app
```

## 29 Conclusión

En esta lección, aprendimos cómo usar Dockerfile y Docker Compose para construir y gestionar aplicaciones Docker. Con Dockerfile, podemos personalizar imágenes Docker para nuestras aplicaciones, mientras que Docker Compose nos permite definir y orquestar servicios en un entorno multi-contenedor. Estas herramientas son esenciales para el desarrollo y despliegue de aplicaciones en contenedores Docker.

# 30 DevContainers

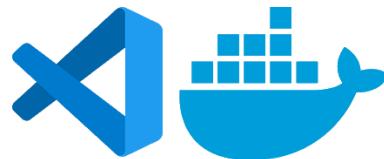


Figure 30.1: DevContainers

## 30.1 ¿Qué son los DevContainers?

Los DevContainers son entornos de desarrollo basados en contenedores Docker que permiten a los desarrolladores crear, compartir y ejecutar aplicaciones en un entorno aislado y portátil. Los DevContainers proporcionan un entorno de desarrollo consistente y reproducible, lo que garantiza que las aplicaciones se ejecuten de la misma manera en diferentes entornos.

Los DevContainers son una herramienta poderosa para el desarrollo de software, ya que permiten a los desarrolladores trabajar en un entorno aislado y preconfigurado, sin tener que preocuparse por la configuración del sistema operativo, las dependencias de software o las bibliotecas de terceros.

## 30.2 Instalación y Uso

Para utilizar DevContainers, es necesario tener instalado Docker en el sistema. Una vez instalado Docker, se puede instalar una extensión de DevContainers en el editor de código favorito, como Visual Studio Code, y utilizarla para crear, compartir y ejecutar DevContainers.

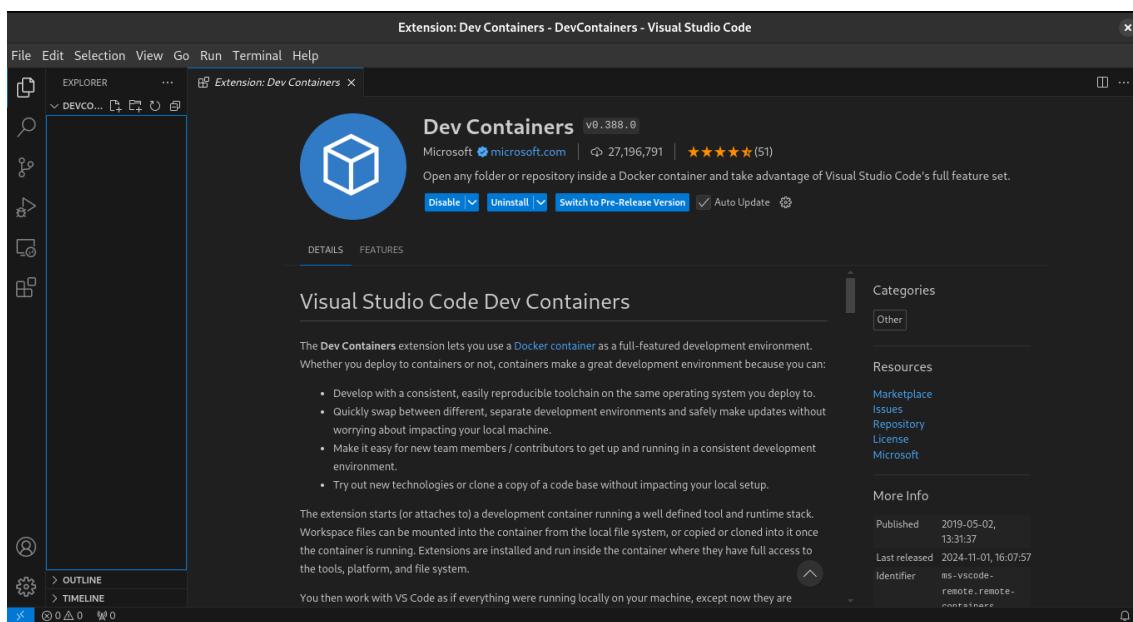


Figure 30.2: DevContainer en Visual Studio Code

### 30.3 Ejemplos:

En la parte inferior izquierda de Visual Studio Code existe un botón que hace referencia a los **DevContainers**, al hacer clic en este botón se abrirá un menú con las opciones para crear, abrir o configurar un DevContainer.

En este punto damos clic en **New DevContainer** y seleccionamos la opción **Python 3**. Esto creará un archivo **.devcontainer** con la configuración necesaria para ejecutar la aplicación en un contenedor Docker.

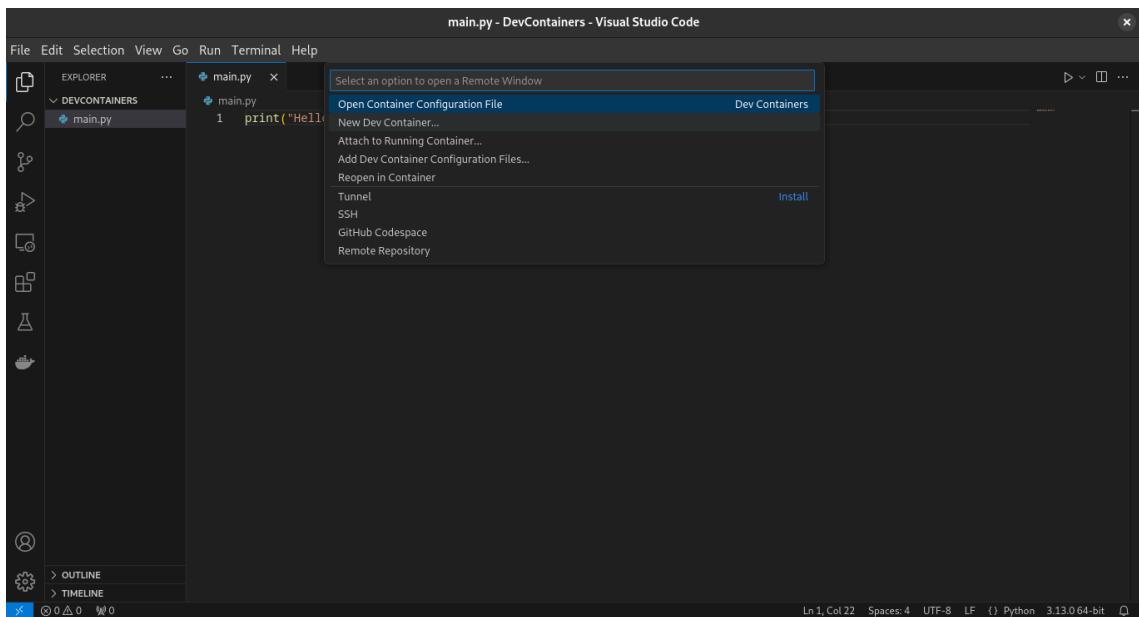


Figure 30.3: New DevContainer

En la imagen anterior podemos observar el menú de DevContainer, en esta sección es posible seleccionar **New DevContainer**. Al seleccionar esta opción se desplegará un menú con las opciones de configuración de DevContainer.

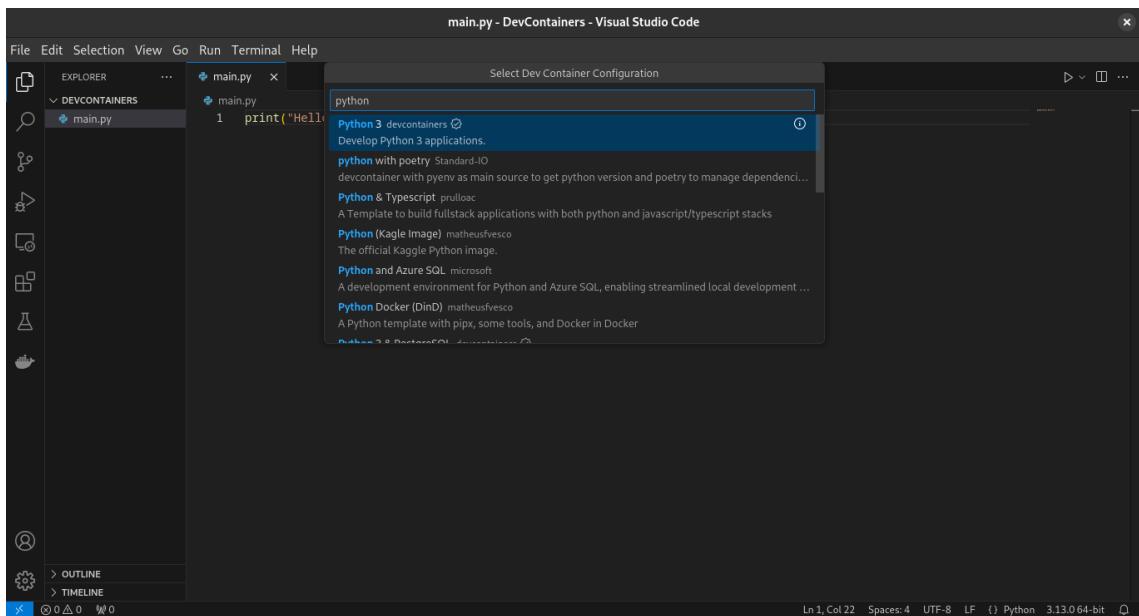


Figure 30.4: Python 3 DevContainer

En la imagen anterior se describe la búsqueda de diferentes plantillas, en este caso seleccionamos **Python 3**. Al seleccionar esta opción se creará un archivo **.devcontainer** con la configuración necesaria para ejecutar la aplicación en un contenedor Docker.

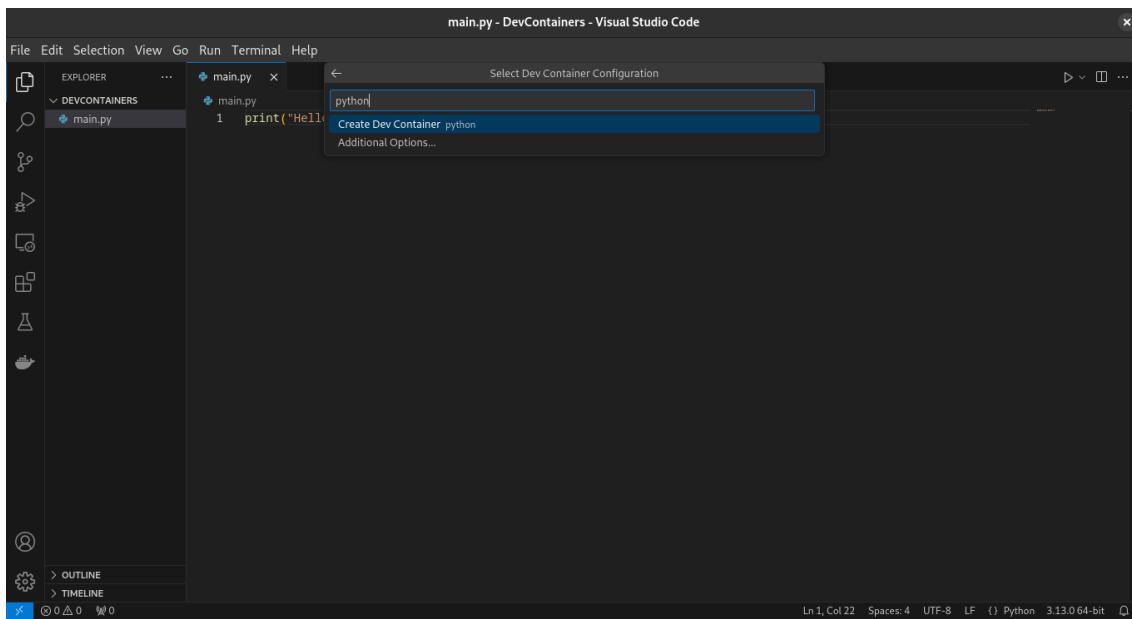


Figure 30.5: Create DevContainer

Finalmente seleccionamos la opción **Create DevContainers** para crear el archivo **.devcontainer** con la configuración necesaria para ejecutar la aplicación en un contenedor Docker.

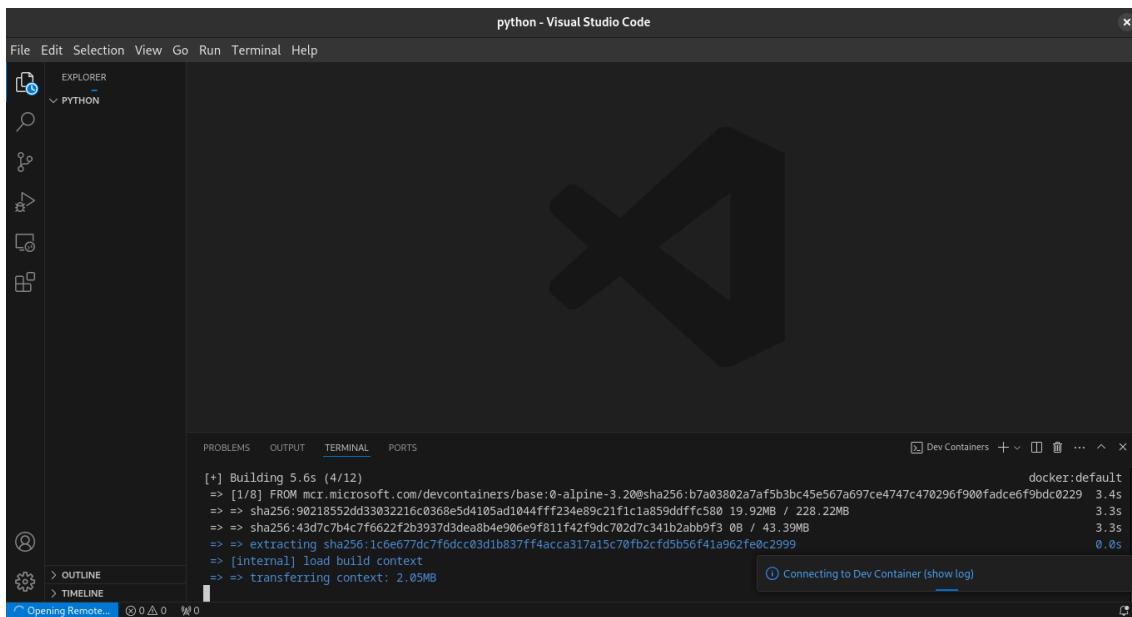


Figure 30.6: Create DevContainer

Ahora solo resta esperar como se observa en la imagen anterior la creación del **DevContainer**. Una vez finalizado el proceso, se abrirá una nueva ventana con el archivo **main.py** en el editor de código y se mostrará un mensaje en la parte inferior derecha indicando que se está construyendo el contenedor.

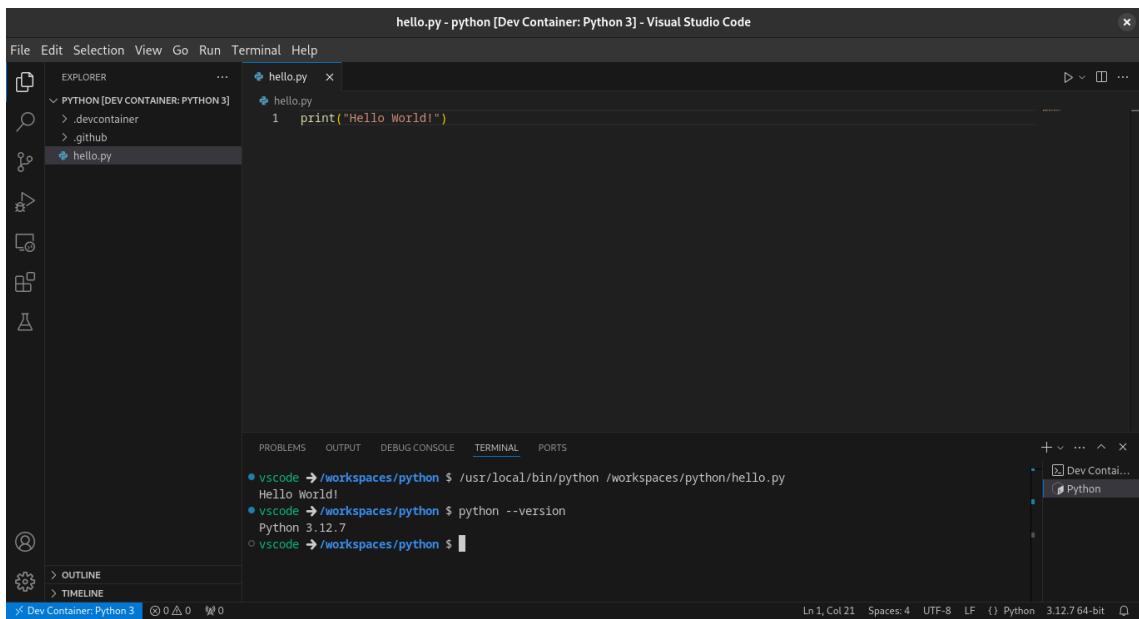


Figure 30.7: Python in DevContainer

Creamos una aplicación Hola Mundo en Python para ser ejecutada en un DevContainer:

Crear un archivo **main.py** con el siguiente código:

```
# main.py
print("Hola, Mundo!")
```

Una vez creado el **DevContainer** se mostrará un mensaje en la parte inferior derecha indicando que se está construyendo el contenedor. En este punto se puede ejecutar la aplicación en el contenedor Docker haciendo clic en el botón **Run** en la parte superior derecha.

Puedes verificar que la versión de python en el terminal del DevContainer creado es diferente a la del Sistema Operativo en el que te encuentres y la instalación global del sistema.

## 30.4 Práctica

- Crear un nuevo DevContainer con una plantilla en Python.
- Crear un archivo **main.py** con un código sencillo en Python.
- Ejecutar la aplicación en el DevContainer.

## **30.5 Conclusiones**

Los DevContainers son una herramienta poderosa para el desarrollo de software, ya que permiten a los desarrolladores trabajar en un entorno aislado y preconfigurado, sin tener que preocuparse por la configuración del sistema operativo, las dependencias de software o las bibliotecas de terceros. Los DevContainers proporcionan un entorno de desarrollo consistente y reproducible, lo que garantiza que las aplicaciones se ejecuten de la misma manera en diferentes entornos.

# **Part VI**

# **Proyectos**

# 31 Laboratorio: Construcción de un Juego de Ahorcado en Python

The screenshot shows a terminal window with two tabs: 'ahorcado.py' and 'bash'. The 'ahorcado.py' tab displays the source code of a Python hangman game. The 'bash' tab shows the terminal session where the game is being played. The user has typed 'i' and 'g', both of which are correct guesses. The game output shows the gallows diagram and the message '¡Felicitaciones, ganaste! 😊'.

```
Neo-tree
└── ejercicios
    └── ahorcado
        ├── ahorcado.py
        ├── 1_intro.py
        ├── 2_conversion.py
        ├── 3_pares.py
        ├── 4_notas.py
        └── intermedio
            └── 1_hidden_item

Terminal
ahorcado.py | bash
15         break
14     else:
13         print("Incorrecto.")
12         intentos_fallidos += 1
11     else:
10         mostrar_ahorcado(intentos_fallidos)
9         mostrar_resultado(False)
8
7     def mostrar_resultado(ganador):
6         if ganador:
5             print("¡Felicitaciones, ganaste! 😊")
4         else:
3             print("Lo siento, perdiste. 😞")
2
1     if __name__ == "__main__":
106     jugar_ahorcado()

15 Introduce una letra: i
14 ¡Correcto!
13
12   |
11   |
10   o
9   /|\ 
8   |
7   /
6
5 programacion
4 Introduce una letra: g
3 ¡Correcto!
2 ¡Felicitaciones, ganaste! 😊
1 253 statick at fedora in -/.../ejercicios/ahorcado on main #11 x

TERMINAL ➜ main 3:/bin/bash
```

Figure 31.1: Ahorcado

## 31.1 Objetivos del Laboratorio

1. Desarrollar un juego de Ahorcado usando funciones en Python.
2. Usar estructuras de datos como listas y cadenas de texto.
3. Implementar lógica condicional y bucles para manejar el flujo del juego.
4. Mostrar mensajes finales (con emojis) según el resultado del juego.

## 31.2 Prerrequisitos

- **Conocimiento básico de Python:** funciones, listas, cadenas, condicionales y bucles.
- Instalación de Python 3 en tu equipo.

## 31.3 Paso 1: Crear la Estructura Inicial del Proyecto

### 31.3.1 Crear un archivo de Python:

Abre tu editor de texto o IDE favorito (se recomienda utilizar Vscode) y crea un nuevo archivo llamado **ahorcado.py**.

**Definir el objetivo del proyecto en el archivo:**

Añade un comentario en la primera línea que describa el propósito del proyecto:

```
# Juego de Ahorcado en Python
```

## 31.4 Paso 2: Definir las Etapas del Ahorcado en ASCII

### 31.4.1 Crear la lista AHORCADO\_DIBUJO:

Define las etapas progresivas del dibujo del ahorcado usando una lista de cadenas en ASCII.

Cada elemento de la lista representa una etapa del juego.

```
AHORCADO_DIBUJO = [
    """
    |
    |
    |
    |
    """,
    """
    |
    |
    |
    |
    0
    |
    """
,
    """
    |
    |
    |
    |
    0
    /
    """
,
    """
    |
    |
    |
    |
    0
    /|\\
    |
    """
,
```

```
"""
| |
0
/|\ \
|
/
"""
,
"""
| |
0
/|\ \
|
/ \\
"""
]

```

### 31.4.2 Prueba del dibujo:

Prueba imprimiendo cada elemento de la lista para asegurarte de que el dibujo es correcto.

```
print(len(AHORCADO_DIBUJO))
for etapa in AHORCADO_DIBUJO:
    print(etapa)
```



Tip

**Nota:** Puedes ejecutar el código en tu terminal o en un entorno de Python para verificar que el dibujo se imprime correctamente.



Tip

No olvides utilizar la función **print()** para mostrar los elementos de la lista en la consola. Y los comentarios para poder identificar cada etapa del dibujo.

## 31.5 Paso 3: Crear la Función para Mostrar el Dibujo del Ahorcado

### 31.5.1 Definir la función **mostrar\_ahorcado**:

Esta función tomará el número de intentos fallidos como argumento e imprimirá la etapa correspondiente del ahorcado.

```
def mostrar_ahorcado(intentos_fallidos):
    print(AHORCADO_DIBUJO[intentos_fallidos])
```

### 31.5.2 Prueba de la función:

Llama a **mostrar\_ahorcado** varias veces con diferentes valores para verificar que cada etapa se muestra correctamente.

## 31.6 Paso 4: Crear Funciones para el Flujo del Juego

### 31.6.1 Función para Seleccionar Palabra Aleatoria:

Define una lista de palabras para que el juego seleccione aleatoriamente una de ellas.

Usa la biblioteca **random** para elegir una palabra al azar.

```
import random

def seleccionar_palabra():
    palabras = ["python", "programacion", "juego", "ahorcado", "computadora"]
    return random.choice(palabras)
```

En el código anterior, la función **seleccionar\_palabra** devuelve una palabra aleatoria de la lista de palabras. También aparece el método `choice` de `random` que selecciona una palabra aleatoria de la lista.

### 31.6.2 Función para Mostrar el Estado Actual:

Esta función mostrará el progreso actual del jugador, mostrando las letras adivinadas y guiones bajos `_` para letras no adivinadas.

```
def mostrar_progreso(palabra, letras_adivinadas):
    progreso = [letra if letra in letras_adivinadas else '_' for letra in palabra]
    print(" ".join(progreso))
```

El código anterior crea una lista de letras adivinadas y guiones bajos para las letras no adivinadas. Luego, une los elementos de la lista en una cadena con un espacio entre cada letra.

Este proceso se conoce como **list comprehension** y es una forma concisa de crear listas en Python.

Para ampliar la información sobre list comprehension, puedes consultar la documentación oficial de Python en el siguiente enlace: [List Comprehensions](#)

### 31.6.3 Función para Manejar el Intento del Jugador:

Define una función que reciba una letra y verifique si está en la palabra.

```
def intentar_letra(palabra, letra, letras_adivinadas):
    if letra in palabra:
        letras_adivinadas.add(letra)
        return True
    return False
```

En el código anterior, la función **intentar\_letra** verifica si la letra está en la palabra y la agrega a la colección de letras adivinadas. Devuelve True si la letra está en la palabra y False si no lo está.

## 31.7 Paso 5: Crear la Función Principal del Juego

### 31.7.1 Configurar el Juego:

Define la función **jugar\_ahorcado()** que controlará el flujo completo del juego.

Establece la palabra a adivinar, el número de intentos, y una colección para almacenar las letras adivinadas.

```
def jugar_ahorcado():
    palabra = seleccionar_palabra()
    letras_adivinadas = set()
    intentos_fallidos = 0
    max_intentos = len(AHORCADO_DIBUJO) - 1
```

En el código anterior, la función **jugar\_ahorcado** selecciona una palabra aleatoria, inicializa una colección de letras adivinadas, y establece el número máximo de intentos.

### 31.7.2 Ciclo del Juego:

Crea un bucle while que continúe mientras el jugador tenga intentos restantes y no haya adivinado la palabra completa.

```
while intentos_fallidos < max_intentos:
    mostrar_ahorcado(intentos_fallidos)
    mostrar_progreso(palabra, letras_adivinadas)

    letra = input("Introduce una letra: ").lower()

    if letra in letras_adivinadas:
        print("Ya intentaste esa letra.")
        continue
```

```

if intentar_letra(palabra, letra, letras_adivinadas):
    print("¡Correcto!")
    if all(l in letras_adivinadas for l in palabra):
        mostrar_resultado(True)
        break
    else:
        print("Incorrecto.")
        intentos_fallidos += 1
else:
    mostrar_ahorcado(intentos_fallidos)
    mostrar_resultado(False)

```

En el código anterior, el bucle while muestra el dibujo actual del ahorcado, el progreso del jugador y solicita una letra al jugador.

## 31.8 Paso 6: Crear Función de Resultado Final con Emojis

### 31.8.1 Definir `mostrar_resultado`:

Esta función mostrará un mensaje final con un emoji dependiendo de si el jugador gana o pierde.

```

def mostrar_resultado(ganador):
    if ganador:
        print("¡Felicitaciones, ganaste! ")
    else:
        print("Lo siento, perdiste. ")

```

En el código anterior, la función `mostrar_resultado` imprime un mensaje de felicitación si el jugador gana y un mensaje de consuelo si pierde.

## 31.9 Paso 7: Ejecutar el Juego

### 31.9.1 Ejecutar el Juego:

Agrega una condición para ejecutar el juego cuando el archivo sea ejecutado directamente.

```

if __name__ == "__main__":
    jugar_ahorcado()

```

En el código anterior, la condición `if name == "main":` verifica si el archivo se ejecuta directamente y llama a la función `jugar_ahorcado` en ese caso.



Tip

**Nota:** Puedes ejecutar el juego en tu terminal o en un entorno de Python para jugar al Ahorcado.

### 31.9.2 Prueba Final:

Ejecuta **ahorcado.py** y juega una partida completa. Verifica que los mensajes y el flujo del juego sean los correctos.

```
python ahorcado.py
```

## 31.10 Paso 8: Mejoras Opcionales

### 31.10.1 Añadir Validación de Entradas: Controla que el jugador solo introduzca letras válidas.

- **Agregar Dificultad:** Permite al jugador elegir entre palabras cortas, medias y largas.

## 32 Conclusión

Con este laboratorio, has creado un juego de Ahorcado en Python que:

- Utiliza funciones para modular el código
  - mostrar\_ahorcado,
  - seleccionar\_palabra,
  - mostrar\_progreso,
  - intentar\_letra,
  - jugar\_ahorcado,
  - mostrar\_resultado

Si separas las funciones en un archivo aparte, puedes importarlas en el archivo principal.

Ejemplo:

Los archivos que son necesarios crear deben estar dentro del directorio funciones.

```
funciones/
    __init__.py
    funciones.py
ahorcado.py
```

El código del archivo **funciones.py** debe ser el siguiente:

```
AHORCADO_DIBUJO = [
    """
    |
    |
    |
    |
    """,
    """
    |
    |
    0
    |
    """
,
    """
    |
    |
    0
    /
    |
```

```

    """
    """
    |
    |
    0
    /|\\
    |
    """
    """
    |
    |
    0
    /|\\
    |
    /
"""
"""
|
|
0
/|\\
|
/
"""
"""

]

def mostrar_ahorcado(intentos_fallidos):
    print(AHORCADO_DIBUJO[intentos_fallidos])

import random

def seleccionar_palabra():
    palabras = ["python", "programacion", "juego", "ahorcado", "computadora"]
    return random.choice(palabras)

def mostrar_progreso(palabra, letras_adivinadas):
    progreso = [letra if letra in letras_adivinadas else '_' for letra in palabra]
    print(" ".join(progreso))

def intentar_letra(palabra, letra, letras_adivinadas):
    if letra in palabra:
        letras_adivinadas.add(letra)
        return True
    return False

def jugar_ahorcado():
    palabra = seleccionar_palabra()
    letras_adivinadas = set()

```

```

intentos_fallidos = 0
max_intentos = len(AHORCADO_DIBUJO) - 1

while intentos_fallidos < max_intentos:
    mostrar_ahorcado(intentos_fallidos)
    mostrar_progreso(palabra, letras_adivinadas)

    letra = input("Introduce una letra: ").lower()

    if letra in letras_adivinadas:
        print("Ya intentaste esa letra.")
        continue

    if intentar_letra(palabra, letra, letras_adivinadas):
        print("¡Correcto!")
        if all(l in letras_adivinadas for l in palabra):
            mostrar_resultado(True)
            break
    else:
        print("Incorrecto.")
        intentos_fallidos += 1
else:
    mostrar_ahorcado(intentos_fallidos)
    mostrar_resultado(False)

def mostrar_resultado(ganador):
    if ganador:
        print("¡Felicitaciones, ganaste! ")
    else:
        print("Lo siento, perdiste. ")

if __name__ == "__main__":
    jugar_ahorcado()

```

El archivo principal **ahorcado.py** debe tener el siguiente código:

```

from funciones import mostrar_ahorcado
from funciones import seleccionar_palabra
from funciones import mostrar_progreso
from funciones import intentar_letra
from funciones import jugar_ahorcado

if __name__ == "__main__":
    jugar_ahorcado()

```

 Tip

**Nota:** Puedes personalizar el juego añadiendo más palabras, emojis, o mensajes según tus preferencias.

- **Personalizar Mensajes:** Cambia los mensajes de victoria y derrota para hacerlos más divertidos.
- **Agregar Sonidos:** Añade sonidos o efectos de sonido al juego para mejorar la experiencia del jugador.
- **Diseño Gráfico:** Crea un diseño gráfico más elaborado para el ahorcado y las letras adivinadas.
- **Más Palabras:** Añade más palabras al juego para aumentar la variedad y dificultad.

## 33 Que aprendimos

- **Funciones en Python:** Cómo definir y llamar funciones en Python.
- **Listas y Cadenas de Texto:** Cómo trabajar con listas y cadenas de texto en Python.
- **Lógica Condicional y Bucles:** Cómo usar lógica condicional y bucles para controlar el flujo del programa.
- **List Comprehensions:** Cómo usar list comprehensions para crear listas de forma concisa.
- **Importar Módulos:** Cómo importar funciones de otros archivos en Python.

¡Espero que hayas disfrutado este laboratorio y te animes a personalizar el juego de Ahorcado con tus propias ideas! ¡Felicitaciones por completar el laboratorio!

# 34 Gestor de Tareas con Prioridades

The screenshot shows a terminal window with three tabs. The left tab displays a file tree for a project named 'tareas' located at '~/workspaces/practicas/tareas'. The middle tab contains Python code in 'test.py':

```
tareas.py      __init__.py      bash      test.py
1 # test.py
2
3 from tareas import Tarea
4
5 tareal = Tarea("Hacer la compra", "Comprar leche, pan y fruta", "2022-12-31", "alta")
6
7 print(tareal)
```

The right tab shows the output of running 'test.py' with 'python':

```
statick at fedora in ~/.../practicas/tareas
1 python -m tareas.py
2 /usr/bin/python: Error while finding module specification for 'tareas.py' (ModuleNotFoundError: __path__ attribute not found on 'tareas' while trying to find 'tareas.py'). Try using 'tareas' instead of 'tareas.py' as the module name.
3 statick at fedora in ~/.../practicas/tareas
4 python -m .
5 /usr/bin/python: Relative module names not supported
6 statick at fedora in ~/.../practicas/tareas
7 python test.py
8 Hacer la compra - alta - 2022-12-31
9 statick at fedora in ~/.../practicas/tareas
10
11
```

At the bottom, the terminal prompt shows 'TERMINAL 39:/bin/bash'.

Figure 34.1: Gestor de Tareas

Una aplicación interactiva que permite organizar tus tareas de manera eficiente, asignando prioridades y estableciendo fechas límite.

## 34.1 Módulos del Proyecto

### 34.1.1 Módulo de tareas

- Crear una nueva tarea con título, descripción, fecha límite y prioridad.
- Marcar tareas como completadas o en progreso .
- Organizar las tareas en orden de prioridad o por fecha límite .

## 34.2 Funciones Clave

- Prioriza tus tareas con un sistema de prioridades: baja, media y alta .

### 34.2.1 Desarrollo

Creamos la siguiente estructura de carpetas para organizar nuestro proyecto:

```
proyecto_modulos/
    tareas/
        __init__.py
        tareas.py
```

En el archivo **tareas.py** definimos las clases y funciones necesarias para gestionar las tareas.

```
# tareas.py

class Tarea:
    def __init__(self, titulo, descripcion, fecha_limite, prioridad):
        self.titulo = titulo
        self.descripcion = descripcion
        self.fecha_limite = fecha_limite
        self.prioridad = prioridad
        self.completada = False

    def marcar_completada(self):
        self.completada = True

    def marcar_en_progreso(self):
        self.completada = False

    def __str__(self):
        return f"{self.titulo} - {self.prioridad} - {self.fecha_limite}"
```

En el archivo **init.py** definimos las funciones principales para interactuar con las tareas.

```
# __init__.py

from tareas import Tarea

def crear_tarea(titulo, descripcion, fecha_limite, prioridad):
    return Tarea(titulo, descripcion, fecha_limite, prioridad)

def marcar_completada(tarea):
    tarea.marcar_completada()

def marcar_en_progreso(tarea):
    tarea.marcar_en_progreso()
```

Con esta estructura básica, podemos empezar a desarrollar la funcionalidad de nuestro gestor de tareas. En los siguientes módulos, ampliaremos las capacidades de nuestra aplicación y añadiremos nuevas funcionalidades.

Para poder probar nuestro código, podemos crear un script de prueba en la misma carpeta:

```
# test.py

from tareas import Tarea

tarea1 = Tarea("Hacer la compra", "Comprar leche, pan y fruta", "2022-12-31", "alta")

print(tarea1)
```

Al ejecutar el script de prueba, deberíamos ver la información de la tarea creada.

```
$ python test.py
Hacer la compra - alta - 2022-12-31
```

## 35 Extra

- Añadir la funcionalidad de editar y eliminar tareas.

```
def editar_tarea(tarea, titulo=None, descripcion=None, fecha_límite=None, prioridad=None):
    if titulo:
        tarea.titulo = titulo
    if descripcion:
        tarea.descripcion = descripcion
    if fecha_límite:
        tarea.fecha_límite = fecha_límite
    if prioridad:
        tarea.prioridad = prioridad
```

- Implementar un sistema de notificaciones para recordar las fechas límite de las tareas.

```
import datetime

def notificar_tareas(tareas):
    hoy = datetime.date.today()
    for tarea in tareas:
        if tarea.fecha_límite == hoy:
            print(f"¡Recuerda! La tarea '{tarea.titulo}' vence hoy.")
```

- Crear una interfaz gráfica para una mejor experiencia de usuario.

```
import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text="Gestor de Tareas")
label.pack()

root.mainloop()
```

## **36 Conclusión**

Con estos módulos básicos, hemos sentado las bases para desarrollar un gestor de tareas con prioridades. A medida que añadamos más funcionalidades y módulos, nuestra aplicación se volverá más completa y útil para organizar nuestras tareas diarias.

## **37 Reto**

- Implementar un sistema de categorías para organizar las tareas por proyectos o áreas de interés.

# 38 Simulador de Tienda Online

The screenshot shows a terminal window with the following details:

- Terminal Title:** Terminal
- File Explorer:** Shows a file tree for a project named "simulador\_tienda". The tree includes directories like "carrito", "clientes", "pedidos", "productos", and "test.py", along with their corresponding \_\_pycache\_\_ sub-directories.
- Code Editor:** Displays the content of `carrito.py`. The code defines a class `Carrito` with methods for adding products, removing products, and calculating the total price.
- Terminal Output:** Shows the command `python test.py` being run, followed by the output of the program which lists items in the cart and calculates totals for specific orders.
- Bottom Status:** Shows the terminal path as `~/bootcamp/simulador_tienda`, the current directory as `56:/bin/bash`, and system status like battery level (65%), time (15:33), and signal strength.

Figure 38.1: Tienda Online

Un proyecto interactivo que simula una tienda en línea donde los clientes pueden agregar productos al carrito, realizar pedidos, gestionar inventarios y procesar pagos.

## 38.1 Módulos del Proyecto

### 38.1.1 Módulo de Productos

1. Definir productos con nombre, precio y cantidad en inventario.
2. Actualizar el inventario después de cada compra o cuando se agregan nuevos productos.

### 38.1.2 Módulo de Carrito

1. Permite a los clientes agregar o quitar productos de su carrito.
2. Calcular el costo total de los productos en el carrito.

### **38.1.3 Módulo de Cliente**

1. Gestionar la creación de nuevos clientes.
2. Mantener el historial de compras del cliente.

### **38.1.4 Módulo de Pedido**

1. Procesar un pedido, verificar disponibilidad en inventario, y generar la factura.
2. Actualizar el inventario después de la compra.

# 39 Desarrollo

Creamos la siguiente estructura de carpetas para organizar nuestro proyecto:

```
tienda_online/
    productos/
        __init__.py
        producto.py

    clientes/
        __init__.py
        cliente.py

    carrito/
        __init__.py
        carrito.py

    pedidos/
        __init__.py
        pedido.py
```

Definimos las clases y funciones necesarias para gestionar la tienda en línea.

## 39.1 Productos

En el archivo **producto.py**, definimos la clase **Producto**:

```
# productos/producto.py

class Producto:
    def __init__(self, nombre, precio, inventario):
        self.nombre = nombre
        self.precio = precio
        self.inventario = inventario

    def actualizar_inventario(self, cantidad):
        self.inventario -= cantidad

    def __str__(self):
        return f'{self.nombre} - ${self.precio} (Inventario: {self.inventario})'
```

## 39.2 Carrito

En el archivo **carrito.py**, definimos la clase Carrito:

```
# carrito/carrito.py

class Carrito:
    def __init__(self):
        self.productos = {}

    def agregar_producto(self, producto, cantidad):
        if producto.nombre in self.productos:
            self.productos[producto.nombre] += cantidad
        else:
            self.productos[producto.nombre] = cantidad

    def eliminar_producto(self, producto):
        if producto.nombre in self.productos:
            del self.productos[producto.nombre]

    def total(self):
        return sum(producto.precio * cantidad for producto, cantidad in self.productos.items())

    def __str__(self):
        carrito_str = "Carrito:\n"
        for producto, cantidad in self.productos.items():
            carrito_str += f"{producto}: {cantidad}\n"
        return carrito_str
```

## 39.3 Clientes

En el archivo **cliente.py**, definimos la clase Cliente:

```
# clientes/cliente.py

class Cliente:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email
        self.historial_compras = []

    def agregar_historial(self, pedido):
        self.historial_compras.append(pedido)

    def ver_historial(self):
        if not self.historial_compras:
```

```

        return "No tienes compras aún."
    return "\n".join(str(pedido) for pedido in self.historial_compras)

def __str__(self):
    return f"Cliente: {self.nombre} ({self.email})"

```

## 39.4 Pedidos

En el archivo **pedido.py**, definimos la clase **Pedido**:

```

# pedidos/pedido.py

class Pedido:
    def __init__(self, cliente, carrito):
        self.cliente = cliente
        self.carrito = carrito
        self.total = carrito.total()

    def procesar_pedido(self):
        for producto, cantidad in self.carrito.productos.items():
            producto.actualizar_inventario(cantidad)
        self.cliente.agregar_historial(self)

    def __str__(self):
        return f"Pedido de {self.cliente.nombre} - Total: ${self.total}"

```

## 40 Prueba del Simulador de Tienda Online

En un archivo de prueba test.py, puedes simular una compra en la tienda:

```
# test.py

from productos.producto import Producto
from carrito.carrito import Carrito
from clientes.cliente import Cliente
from pedidos.pedido import Pedido

# Crear productos
producto1 = Producto("Camiseta", 20.0, 50)
producto2 = Producto("Zapatos", 50.0, 20)

# Crear un cliente
cliente = Cliente("Juan Pérez", "juan@example.com")

# Crear un carrito y agregar productos
carrito = Carrito()
carrito.agregar_producto(producto1, 2)
carrito.agregar_producto(producto2, 1)

print(carrito) # Ver contenido del carrito

# Crear y procesar el pedido
pedido = Pedido(cliente, carrito)
pedido.procesar_pedido()

print(pedido) # Ver detalles del pedido
print(cliente.ver_historial()) # Ver historial de compras
```

Al ejecutar el archivo test.py, verás el contenido del carrito, el pedido procesado, y el historial de compras del cliente.

## 41 Extra

- Añadir la funcionalidad de eliminar productos del carrito:

```
def eliminar_producto(self, producto):  
    if producto in self.productos:  
        del self.productos[producto]
```

- Añadir un sistema de descuento:

```
def aplicar_descuento(self, porcentaje):  
    self.total -= self.total * (porcentaje / 100)
```

- Añadir una interfaz gráfica usando Tkinter:

```
import tkinter as tk  
  
root = tk.Tk()  
  
label = tk.Label(root, text="¡Bienvenido a la Tienda Online!")  
label.pack()  
  
root.mainloop()
```

## **42 Conclusión**

Con esta estructura básica de POO, hemos creado un simulador de tienda online donde se gestionan productos, carritos, clientes y pedidos. A medida que avances, puedes agregar más características como métodos de pago, envío, y más opciones de interacción para los clientes.

¡Diviértete desarrollando y mejorando tu tienda online!

## **43 Sistema Universitario**

```
Neo-tree
└─/workspaces/practicas/python/univers
    └─main.py

  main.py      x  65708:/bin/bash x  | 66502:/bin/bash x
12     profesor_pedro = Profesor("Pedro Ramirez", 40, "Masculino", "202020204", "Quimica")
11
10     # Crear los cursos
9      curso_matematicas = Curso("Matematicas", "MAT101", profesor_juan)
8      curso_fisica = Curso("Fisica", "FIS101", profesor_maría)
7      curso_quimica = Curso("Quimica", "QUI101", profesor_pedro)
6
5      # Agregar los cursos a la universidad
4      universidad.agregar_curso(curso_matematicas)
3      universidad.agregar_curso(curso_fisica)
2      universidad.agregar_curso(curso_quimica)
1
78     # Crear el objeto estudiante
1 estudiante_carlos = Estudiante(
2     | "Carlos Perez", 20, "Masculino", "202010101", "Ingenieria en Sistemas Informaticos"
3   )
4

16 Curso: Quimica, Código: QUI101, Profesor: Pedro Ramirez
15
14 Nombre: Carlos Perez, Edad: 20, Sexo: Masculino, Carnet: 202010101, Carrera: Ingenieria en Sistemas Informaticos
13
12 Nombre: Juan Perez, Edad: 30, Sexo: Masculino, Código: 202020202, Especialidad: Matematicas
11
10 Curso: Matematicas, Código: MAT101, Profesor: Juan Perez
9
8 Universidad: Universidad de El Salvador
7 Cursos:
6 Curso: Matematicas, Código: MAT101, Profesor: Juan Perez
5 Curso: Fisica, Código: FIS101, Profesor: Maria Lopez
4 Curso: Quimica, Código: QUI101, Profesor: Pedro Ramirez
3 Curso: Fisica, Código: FIS101, Profesor: Maria Lopez
2
1 ⌂ statick @ fedora in universidad
38
```

Figure 43.1: Universidad

En este laboratorio vamos a aprender a utilizar la POO mediante la creación de un sistema Universitario.

El sistema consiste en definir las clases Persona, Estudiante, Profesor, Curso y Universidad, con los siguientes atributos:

- **Persona:** nombre, edad y sexo.

Tambien se crearan las siguientes clases:

- **Estudiante:** carnet, carrera.
  - **Profesor:** codigo, especialidad.
  - **Curso:** nombre, codigo, profesor.
  - **Universidad:** nombre, cursos.
  - Se crean los objetos universidad, profesores, cursos y estudiante con los datos indicados.
  - Se agregan los cursos a la universidad.
  - Se imprime la información de la universidad, el estudiante, el profesor y el curso de Matemáticas.

## 43.1 Objetivos

- Definir clases en Python.
- Crear objetos de clases.
- Utilizar herencia en clases.
- Mostrar información de objetos. ## Requerimientos
- Conocimientos básicos de programación en Python.
- Conocimientos básicos de programación orientada a objetos.

## 43.2 Instrucciones.

1. **Clase Persona:** Define los atributos comunes nombre, edad y sexo.
2. **Clase Estudiante:** Hereda de Persona y agrega los atributos carnet y carrera.
3. **Clase Profesor:** Hereda de Persona y agrega los atributos codigo y especialidad.
4. **Clase Curso:** Contiene los atributos nombre, codigo y una instancia de Profesor.
5. **Clase Universidad:** Contiene el atributo nombre y una lista de cursos. Incluye un método para agregar cursos.
6. Creación de objetos:
  - Se crea un objeto universidad de la clase Universidad.
  - Se crean los objetos profesor, curso y estudiante con los datos indicados.
  - Se agrega cada curso a la universidad y luego se imprime la universidad con los cursos.
7. Impresión:
  - Se imprime la información de la universidad, el estudiante, el profesor y el curso de Matemáticas, según los requerimientos.

## 43.3 Desarrollo

1. Crear la clase Persona.

```

class Persona:
    def __init__(self, nombre, edad, sexo):
        self.nombre = nombre
        self.edad = edad
        self.sexo = sexo

    def __str__(self):
        return f"Nombre: {self.nombre}, Edad: {self.edad}, Sexo: {self.sexo}"

```

En el código anterior se crea la clase Persona con los atributos nombre, edad y sexo. Además, se crea el método **str** para mostrar la información de la persona.

- Crear la clase Estudiante que hereda de Persona.

```

class Estudiante(Persona):
    def __init__(self, nombre, edad, sexo, carnet, carrera):
        super().__init__(nombre, edad, sexo)
        self.carnet = carnet
        self.carrera = carrera

    def __str__(self):
        return f"{super().__str__()}, Carnet: {self.carnet}, Carrera: {self.carrera}"

```

En el código anterior se crea la clase Estudiante que hereda de Persona. Se añaden los atributos carnet y carrera. Además, se sobreescribe el método **str** para mostrar la información del estudiante.

- Crear la clase Profesor que hereda de Persona.

```

class Profesor(Persona):
    def __init__(self, nombre, edad, sexo, codigo, especialidad):
        super().__init__(nombre, edad, sexo)
        self.codigo = codigo
        self.especialidad = especialidad

    def __str__(self):
        return f"{super().__str__()}, Código: {self.codigo}, Especialidad: {self.especialidad}"

```

En el código anterior se crea la clase Profesor que hereda de Persona. Se añaden los atributos codigo y especialidad. Además, se sobreescribe el método **str** para mostrar la información del profesor.

- Crear la clase Curso.

```

class Curso:
    def __init__(self, nombre, codigo, profesor):
        self.nombre = nombre
        self.codigo = codigo
        self.profesor = profesor

    def __str__(self):
        return f"Curso: {self.nombre}, Código: {self.codigo}, Profesor: {self.profesor}"

```

En el código anterior se crea la clase `Curso` con los atributos `nombre`, `codigo` y `profesor`. Además, se crea el método `str` para mostrar la información del curso.

#### 5. Crear la clase Universidad.

```

class Universidad:
    def __init__(self, nombre):
        self.nombre = nombre
        self.cursos = []

    def agregar_curso(self, curso):
        self.cursos.append(curso)

    def __str__(self):
        cursos_str = "\n".join([str(curso) for curso in self.cursos])
        return f"Universidad: {self.nombre}\nCursos:\n{cursos_str}"

```

En el código anterior se crea la clase `Universidad` con los atributos `nombre` y `cursos`. Se añade el método `agregar_curso` para agregar un curso a la lista de cursos. Además, se sobrescribe el método `str` para mostrar la información de la universidad y los cursos.

#### 6. Crear los objetos

```

# Crear la universidad
universidad = Universidad("Universidad de El Salvador")

# Crear los profesores
profesor_juan = Profesor("Juan Perez", 30, "Masculino", "202020202", "Matematicas")
profesor_maria = Profesor("Maria Lopez", 35, "Femenino", "202020203", "Fisica")
profesor_pedro = Profesor("Pedro Ramirez", 40, "Masculino", "202020204", "Quimica")

# Crear los cursos
curso_matematicas = Curso("Matematicas", "MAT101", profesor_juan)
curso_fisica = Curso("Fisica", "FIS101", profesor_maria)
curso_quimica = Curso("Quimica", "QUI101", profesor_pedro)

# Agregar los cursos a la universidad
universidad.agregar_curso(curso_matematicas)
universidad.agregar_curso(curso_fisica)

```

```
universidad.agregar_curso(curso_quimica)

# Crear el objeto estudiante
estudiante_carlos = Estudiante("Carlos Perez", 20, "Masculino", "202010101", "Ingenieria
```

En el código anterior se crean los objetos de la universidad, profesores, cursos y estudiante.

## 7. Imprimir la información

```
print(universidad)
print()
print(estudiante_carlos)
print()
print(profesor_juan)
print()
print(curso_matematicas)

# Crear un nuevo curso de Fisica y agregarlo a la universidad
curso_nuevo_fisica = Curso("Fisica", "FIS101", profesor_maria)
universidad.agregar_curso(curso_nuevo_fisica)

# Imprimir nuevamente la universidad con el nuevo curso agregado
print()
print(universidad)
```

En el código anterior se imprime la información de la universidad, estudiante, profesor y curso. Luego se crea un nuevo curso de Física y se agrega a la universidad, para finalmente imprimir nuevamente la información de la universidad.

## **44 Conclusión**

En este laboratorio hemos aprendido a utilizar la programación orientada a objetos mediante la creación de un sistema universitario. Hemos definido clases para Persona, Estudiante, Profesor, Curso y Universidad, y hemos creado objetos con los datos indicados. Además, hemos agregado cursos a la universidad y hemos mostrado la información de la universidad, estudiante, profesor y curso.