

# **Bootcamp Desarrollo Web FullStack**

Diego Saavedra

Dec 20, 2024

# Table of contents

<b>1</b>	<b>Bienvenido</b>	<b>18</b>
1.1	¿De qué trata este Bootcamp? . . . . .	18
1.2	¿Para quién es este bootcamp? . . . . .	18
1.3	¿Qué aprenderás? . . . . .	18
1.4	¿Cómo contribuir? . . . . .	18
<b>I</b>	<b>Unidad 0: Introducción a Git y GitHub</b>	<b>20</b>
<b>2</b>	<b>Git y GitHub</b>	<b>21</b>
2.1	¿Qué es Git y GitHub? . . . . .	21
2.2	¿Quiénes utilizan Git? . . . . .	22
2.3	¿Cómo se utiliza Git? . . . . .	22
2.4	¿Para qué sirve Git? . . . . .	23
2.5	¿Por qué utilizar Git? . . . . .	24
2.6	¿Dónde puedo utilizar Git? . . . . .	25
2.7	Pasos Básicos . . . . .	25
2.8	Instalación de Visual Studio Code . . . . .	26
2.8.1	Descarga e Instalación de Git . . . . .	27
2.8.2	Configuración . . . . .	28
2.8.3	Creación de un Repositorio “helloWorld” en Python . . . . .	28
2.8.4	Comandos Básicos de Git . . . . .	29
2.8.5	Estados en Git . . . . .	29
<b>3</b>	<b>Tutorial: Moviendo Cambios entre Estados en Git</b>	<b>30</b>
3.1	Introducción . . . . .	30
3.2	Sección 1: Modificar Archivos en el Repositorio . . . . .	30
3.3	Mover Cambios de Local a Staging: . . . . .	30
3.4	Agregar Cambios de Local a Staging: . . . . .	31
3.5	Sección 2: Confirmar Cambios en un Commit . . . . .	31
3.6	Mover Cambios de Staging a Commit: . . . . .	31
3.7	Sección 3: Creación y Fusión de Ramas . . . . .	31
3.8	Crear una Nueva Rama: . . . . .	31
3.9	Implementar Funcionalidades en la Rama: . . . . .	31
3.10	Fusionar Ramas con la Rama Principal: . . . . .	32
3.11	Sección 4: Revertir Cambios en un Archivo . . . . .	32
3.12	Revertir Cambios en un Archivo: . . . . .	32
3.13	Conclusión . . . . .	32
<b>4</b>	<b>Asignación</b>	<b>33</b>

<b>5</b>	<b>GitHub Classroom</b>	<b>34</b>
5.1	¿Qué es GitHub Classroom?	34
5.1.1	Funcionalidades Principales	34
5.2	Ejemplo Práctico	35
5.2.1	Creación de una Asignación en GitHub Classroom	35
5.3	Trabajo de los Estudiantes	37
<b>II</b>	<b>Unidad 1: Introducción e Instalaciones Necesarias</b>	<b>43</b>
<b>6</b>	<b>Introducción e Instalaciones Necesarias.</b>	<b>44</b>
6.1	Introducción General a la Programación	45
6.2	Instalación de Python	47
6.3	Uso de REPL, PEP 8 y Zen de Python	50
6.3.1	REPL	50
<b>7</b>	<b>Pep 8</b>	<b>51</b>
<b>8</b>	<b>Zen de python.</b>	<b>52</b>
8.1	Entornos de Desarrollo	53
8.2	5 Consejos para mejorar la lógica de programación.	55
8.3	Conclusiones	56
<b>9</b>	<b>Introducción a la Programación con Python</b>	<b>57</b>
9.1	¿Qué es la programación?	57
9.2	¿Qué es Python?	57
9.3	¿Por qué aprender Python?	57
9.4	¿Qué aprenderemos en este bootcamp?	58
9.5	Identación en Python	58
9.6	Comentarios en python	58
9.7	Variables y Variables Múltiples	58
9.8	Concatenación de Cadenas	59
<b>10</b>	<b>Actividad</b>	<b>60</b>
10.1	instrucciones	60
<b>11</b>	<b>Conclusión</b>	<b>61</b>
<b>12</b>	<b>Tipos de Datos</b>	<b>62</b>
12.1	String y Números.	62
12.1.1	String	62
12.1.2	Números	63
12.2	Listas y Tuplas.	63
12.2.1	Listas	63
12.2.2	Tuplas	63
12.3	Diccionarios y Booleanos.	64
12.3.1	Diccionarios	64
12.3.2	Booleanos	64
12.4	Range	64

<b>13 Actividad</b>	<b>65</b>
13.1 Instrucciones . . . . .	65
<b>14 Conclusiones</b>	<b>66</b>
<b>15 Control de Flujo</b>	<b>67</b>
15.1 If y Condicionales . . . . .	67
15.2 If, elif y else . . . . .	68
15.3 And y Or . . . . .	69
15.4 While loop . . . . .	69
15.5 While, break y continue . . . . .	70
15.6 For loop . . . . .	70
<b>16 Actividad</b>	<b>72</b>
16.1 instrucciones . . . . .	72
<b>17 Conclusiones</b>	<b>73</b>
<b>18 Funciones y recursividad.</b>	<b>74</b>
18.1 Introducción a Funciones . . . . .	74
18.2 Parámetros y Argumentos . . . . .	75
18.3 Retorno de valores . . . . .	75
18.4 Recursividad . . . . .	76
<b>19 Actividad</b>	<b>77</b>
19.1 Instrucciones . . . . .	77
<b>20 Conclusiones</b>	<b>78</b>
<b>III Unidad 2: Programación Orientada a Objetos</b>	<b>79</b>
<b>21 Programacion Orientada a Objetos.</b>	<b>80</b>
21.1 Objetos y Clases . . . . .	81
21.2 Atributos . . . . .	81
21.3 ¿Qué es self? . . . . .	81
21.4 Métodos . . . . .	82
21.5 Self, Eliminar Propiedades y Objetos . . . . .	82
21.6 Eliminar Propiedades y Objetos . . . . .	83
21.7 Herencia, Polimorfismo y Encapsulación . . . . .	83
21.7.1 Herencia . . . . .	83
21.7.2 Polimorfismo . . . . .	84
21.7.3 Encapsulación . . . . .	85
21.8 Actividad . . . . .	86
<b>22 Conclusiones</b>	<b>90</b>

<b>IV Unidad 3: Módulos y Paquetes</b>	<b>91</b>
<b>23 Módulos</b>	<b>92</b>
23.1 Introducción a Módulos . . . . .	92
23.2 Creando Módulos Personalizados . . . . .	93
23.3 Usando Módulos en un Archivo Principal . . . . .	93
23.4 Importando y Renombrando Módulos . . . . .	93
23.5 Importando Funciones Específicas de un Módulo . . . . .	94
23.6 Usando Módulos Externos con pip . . . . .	94
23.7 Instalando un módulo con pip . . . . .	94
23.8 Usando el módulo instalado . . . . .	94
23.9 Instalando otro módulo . . . . .	94
23.10 Usando el módulo emojis . . . . .	95
<b>24 Actividad Práctica</b>	<b>96</b>
<b>25 Conclusión</b>	<b>98</b>
<b>V Unidad 4: Docker</b>	<b>99</b>
<b>26 Docker</b>	<b>100</b>
26.1 Ejemplos: . . . . .	101
26.2 Comandos básicos de Docker: . . . . .	102
26.3 Atajos y Comandos Adicionales: . . . . .	103
26.4 Práctica: . . . . .	104
<b>27 Conclusiones</b>	<b>105</b>
<b>28 Dockerfile y Docker Compose</b>	<b>106</b>
28.1 Introducción . . . . .	106
28.1.1 Dockerfile . . . . .	106
28.1.2 Docker Compose . . . . .	106
28.2 Ejemplos: . . . . .	106
28.2.1 server.js . . . . .	107
28.2.2 Dockerfile . . . . .	107
28.2.3 docker-compose.yml . . . . .	108
28.3 Práctica: . . . . .	109
<b>29 Conclusión</b>	<b>110</b>
<b>30 DevContainers</b>	<b>111</b>
30.1 ¿Qué son los DevContainers? . . . . .	111
30.2 Instalación y Uso . . . . .	111
30.3 Ejemplos: . . . . .	112
30.4 Práctica . . . . .	115
30.5 Conclusiones . . . . .	116

<b>VI Unidad 5: Python Avanzado</b>	<b>117</b>
<b>31 Conceptos Avanzados en Python</b>	<b>118</b>
<b>32 Excepciones y Manejo de Errores</b>	<b>119</b>
32.0.1 Conceptos clave . . . . .	119
32.0.2 Ejemplo . . . . .	119
32.0.3 Excepciones personalizadas . . . . .	120
32.0.4 Ejemplo Práctico . . . . .	120
<b>33 Lectura y Escritura de Archivos</b>	<b>122</b>
33.0.1 Conceptos clave . . . . .	122
33.0.2 Archivos binarios . . . . .	123
33.0.3 Ejemplo Práctico . . . . .	123
<b>34 Programación Funcional</b>	<b>124</b>
34.0.1 Conceptos clave . . . . .	124
34.0.2 Comprensión de listas y generadores. . . . .	124
34.0.3 Ejemplo Práctico . . . . .	125
<b>35 Comprensiones y Generadores</b>	<b>127</b>
35.0.1 Conceptos clave . . . . .	127
35.0.2 Ejemplo Práctico . . . . .	128
<b>36 Módulos y Paquetes Avanzados</b>	<b>129</b>
36.0.1 Conceptos clave . . . . .	129
36.0.2 Ejemplo Práctico . . . . .	130
<b>37 Decoradores y Context Managers</b>	<b>131</b>
37.0.1 Conceptos clave . . . . .	131
37.0.2 Ejemplo Práctico . . . . .	132
<b>38 Colecciones de Datos y Estructuras Especializadas</b>	<b>134</b>
38.0.1 Conceptos clave . . . . .	134
38.0.2 Ejemplo Práctico . . . . .	135
<b>39 Manipulación de Fechas y Tiempos</b>	<b>137</b>
39.0.1 Conceptos clave . . . . .	137
39.0.2 Ejemplo Práctico . . . . .	138
<b>40 Concurrencia y Paralelismo</b>	<b>139</b>
40.0.1 Conceptos clave . . . . .	139
40.0.2 Ejemplo Práctico . . . . .	139
<b>41 Pruebas y Debugging</b>	<b>142</b>
41.0.1 Conceptos clave . . . . .	142
41.0.2 Ejemplo Práctico . . . . .	143

<b>VII Unidad 6: Bases de Datos</b>	<b>145</b>
<b>42 Introducción a Bases de Datos</b>	<b>146</b>
42.1 1. Fundamentos de Bases de Datos . . . . .	146
42.1.1 Conceptos Clave . . . . .	146
42.2 Ejemplo Práctico . . . . .	147
42.2.1 Instrucciones: . . . . .	148
<b>43 Conclusiones</b>	<b>149</b>
<b>44 Bases de Datos con SQLite3</b>	<b>150</b>
44.1 Conceptos Clave . . . . .	150
44.2 Ejemplos . . . . .	150
44.3 Ejemplo Práctico . . . . .	153
44.3.1 Instrucciones: . . . . .	154
<b>45 Conclusiones</b>	<b>156</b>
<b>46 Bases de Datos en MySQL</b>	<b>157</b>
46.1 Conceptos Clave . . . . .	157
46.2 Configuración de MySQL con Docker . . . . .	157
46.2.1 Instrucciones . . . . .	157
46.2.2 Parámetros: . . . . .	158
46.3 Ejemplos . . . . .	158
46.4 Ejemplo Práctico . . . . .	162
46.5 Instrucciones: . . . . .	162
<b>47 Conclusiones</b>	<b>164</b>
<b>48 Bases de Datos en PostgreSQL</b>	<b>165</b>
48.1 Conceptos Clave . . . . .	165
48.2 Configuración de PostgreSQL con Docker . . . . .	165
48.2.1 Instrucciones . . . . .	165
48.2.2 Parámetros: . . . . .	166
48.2.3 Acceder al contenedor (opcional): . . . . .	166
48.3 Ejemplos . . . . .	166
48.4 Ejemplo Práctico . . . . .	169
48.5 Instrucciones: . . . . .	170
<b>49 Conclusiones</b>	<b>172</b>
<b>50 Bases de Datos MongoDB</b>	<b>173</b>
50.1 Conceptos Clave . . . . .	173
50.2 Configuración de MongoDB con Docker . . . . .	173
50.2.1 Instrucciones . . . . .	173
50.2.2 Parámetros: . . . . .	174
50.3 Ejemplos . . . . .	174
50.4 Ejemplo Práctico . . . . .	177
50.5 Instrucciones: . . . . .	178

<b>51 Conclusiones</b>	<b>180</b>
<b>VIII Unidad 7: Frameworks en Python</b>	<b>181</b>
<b>52 Introducción a los Frameworks en Python</b>	<b>182</b>
52.1 Creación de Entornos Virtuales . . . . .	183
52.2 Flask . . . . .	184
52.2.1 Ejemplo . . . . .	184
52.3 Django . . . . .	186
52.3.1 Ejemplo . . . . .	187
52.4 FastAPI . . . . .	189
52.4.1 Ejemplo . . . . .	189
52.5 Conclusiones . . . . .	191
<b>53 Introducción a Django</b>	<b>192</b>
<b>54 Historia de Django</b>	<b>193</b>
<b>55 Características de Django</b>	<b>194</b>
55.1 Ejemplo . . . . .	194
<b>56 Buenas Prácticas</b>	<b>199</b>
<b>57 Actividad</b>	<b>200</b>
<b>58 Recursos Adicionales</b>	<b>201</b>
<b>59 Conclusiones</b>	<b>202</b>
<b>60 Estructura de un proyecto en Django</b>	<b>203</b>
<b>61 Estructura de una aplicación en Django</b>	<b>205</b>
61.1 Activiades: . . . . .	206
<b>62 Conclusiones</b>	<b>208</b>
<b>63 Modelos en Django</b>	<b>209</b>
63.1 ¿Qué es ORM? . . . . .	209
63.2 Ventajas de ORM . . . . .	209
63.3 Crear un modelo . . . . .	210
63.4 Migraciones . . . . .	211
63.5 Interactuar con la base de datos . . . . .	212
<b>64 Actividades</b>	<b>214</b>
<b>65 Conclusión</b>	<b>217</b>
<b>66 Forms en Django</b>	<b>218</b>
66.1 Creación de un formulario . . . . .	218
66.2 Procesamiento de formularios en las vistas . . . . .	219

<b>67 Actividades:</b>	<b>220</b>
<b>68 Conclusiones</b>	<b>221</b>
<b>69 Vistas, Templates y Rutas en Django</b>	<b>222</b>
69.1 Modelos y Formularios . . . . .	222
<b>70 Paso 1: Mostrar la Lista de Medicamentos</b>	<b>224</b>
70.1 Vista . . . . .	224
70.2 Template . . . . .	224
70.3 Ruta . . . . .	225
<b>71 Paso 2: Mostrar el Detalle de un Medicamento</b>	<b>226</b>
71.1 Vista . . . . .	226
71.2 Template . . . . .	226
71.3 Ruta . . . . .	227
<b>72 Paso 3: Crear un Medicamento</b>	<b>228</b>
72.1 Vista . . . . .	228
72.2 Template . . . . .	228
72.3 Ruta . . . . .	229
<b>73 Paso 4: Editar un Medicamento</b>	<b>230</b>
73.1 Vista . . . . .	230
73.2 Template . . . . .	230
73.3 Ruta . . . . .	230
<b>74 Paso 5: Eliminar un Medicamento</b>	<b>231</b>
74.1 Vista . . . . .	231
74.2 Ruta . . . . .	231
<b>75 Ejecutar el Proyecto</b>	<b>232</b>
<b>76 Sistema de Herencia de PlantiLLas, Tailwindcss y Archivos Estáticos.</b>	<b>233</b>
<b>77 1. Herencia de Plantillas</b>	<b>234</b>
77.1 Otras Plantillas . . . . .	234
77.1.1 1. crear_medicamento.html: . . . . .	234
77.2 2. detalle_medicamento.html: . . . . .	235
77.3 3. lista_medicamentos.html: . . . . .	236
77.4 4. editar_medicamento.html: . . . . .	237
77.5 4. eliminar_medicamento.html: . . . . .	237
<b>78 2. Integración de Tailwind CSS</b>	<b>239</b>
78.1 2.1 Instalación de Tailwind CSS . . . . .	239
78.2 2.2 Configuración de Tailwind en Django . . . . .	239
78.3 2.3 Configuración del archivo tailwind.config.js . . . . .	240
<b>79 3. Generación del archivo static</b>	<b>241</b>
79.1 3.1 Configuración en settings.py . . . . .	241
79.2 3.2 Generación de output.css . . . . .	242

<b>80</b>	<b>4. Estructura de Archivos Final</b>	<b>243</b>
<b>81</b>	<b>Conclusión</b>	<b>244</b>
<b>82</b>	<b>Introducción a Django REST Framework</b>	<b>245</b>
82.1	Serialización de modelos . . . . .	245
82.1.1	2. Configuración Inicial . . . . .	245
82.1.2	3. Crear la API . . . . .	246
82.1.3	4. Documentación con Swagger . . . . .	248
82.1.4	5. Pruebas con Thunder Client . . . . .	250
82.1.5	6. Pruebas Automatizadas de las APIs . . . . .	251
82.1.6	7. Conclusión . . . . .	253
<b>IX</b>	<b>Proyectos</b>	<b>254</b>
<b>83</b>	<b>Laboratorio: Construcción de un Juego de Ahorcado en Python</b>	<b>255</b>
83.1	Objetivos del Laboratorio . . . . .	255
83.2	Prerrequisitos . . . . .	255
83.3	Paso 1: Crear la Estructura Inicial del Proyecto . . . . .	256
83.3.1	Crear un archivo de Python: . . . . .	256
83.4	Paso 2: Definir las Etapas del Ahorcado en ASCII . . . . .	256
83.4.1	Crear la lista AHORCADO_DIBUJO: . . . . .	256
83.4.2	Prueba del dibujo: . . . . .	257
83.5	Paso 3: Crear la Función para Mostrar el Dibujo del Ahorcado . . . . .	257
83.5.1	Definir la función mostrar_ahorcado: . . . . .	257
83.5.2	Prueba de la función: . . . . .	258
83.6	Paso 4: Crear Funciones para el Flujo del Juego . . . . .	258
83.6.1	Función para Seleccionar Palabra Aleatoria: . . . . .	258
83.6.2	Función para Mostrar el Estado Actual: . . . . .	258
83.6.3	Función para Manejar el Intento del Jugador: . . . . .	259
83.7	Paso 5: Crear la Función Principal del Juego . . . . .	259
83.7.1	Configurar el Juego: . . . . .	259
83.7.2	Ciclo del Juego: . . . . .	259
83.8	Paso 6: Crear Función de Resultado Final con Emojis . . . . .	260
83.8.1	Definir mostrar_resultado: . . . . .	260
83.9	Paso 7: Ejecutar el Juego . . . . .	260
83.9.1	Ejecutar el Juego: . . . . .	260
83.9.2	Prueba Final: . . . . .	261
83.10	Paso 8: Mejoras Opcionales . . . . .	261
83.10.1	Añadir Validación de Entradas: Controla que el jugador solo introduzca letras válidas. . . . .	261
<b>84</b>	<b>Conclusión</b>	<b>262</b>
<b>85</b>	<b>Que aprendimos</b>	<b>266</b>
<b>86</b>	<b>Gestor de Tareas con Prioridades</b>	<b>267</b>
86.1	Módulos del Proyecto . . . . .	267
86.1.1	Módulo de tareas . . . . .	267

86.2	Funciones Clave . . . . .	267
86.2.1	Desarrollo . . . . .	268
<b>87</b>	<b>Extra</b>	<b>270</b>
<b>88</b>	<b>Conclusión</b>	<b>271</b>
<b>89</b>	<b>Reto</b>	<b>272</b>
<b>90</b>	<b>Simulador de Tienda Online</b>	<b>273</b>
90.1	Módulos del Proyecto . . . . .	273
90.1.1	Módulo de Productos . . . . .	273
90.1.2	Módulo de Carrito . . . . .	273
90.1.3	Módulo de Cliente . . . . .	274
90.1.4	Módulo de Pedido . . . . .	274
<b>91</b>	<b>Desarrollo</b>	<b>275</b>
91.1	Productos . . . . .	275
91.2	Carrito . . . . .	276
91.3	Clientes . . . . .	276
91.4	Pedidos . . . . .	277
<b>92</b>	<b>Prueba del Simulador de Tienda Online</b>	<b>278</b>
<b>93</b>	<b>Extra</b>	<b>279</b>
<b>94</b>	<b>Conclusión</b>	<b>280</b>
<b>95</b>	<b>Sistema Universitario</b>	<b>281</b>
95.1	Objetivos . . . . .	282
95.2	Instrucciones. . . . .	282
95.3	Desarrollo . . . . .	282
<b>96</b>	<b>Conclusión</b>	<b>286</b>
<b>97</b>	<b>Laboratorio: DevContainer con NGINX</b>	<b>287</b>
<b>98</b>	<b>Objetivo:</b>	<b>288</b>
98.1	1. Estructura del Proyecto . . . . .	288
98.2	3. Instrucciones de Creación y Ejecución . . . . .	289
98.3	2. Archivos de Configuración . . . . .	290
98.4	Problemas Comunes . . . . .	290
<b>99</b>	<b>Laboratorio: Calculadora en Python</b>	<b>292</b>
99.1	Paso 1: Configuración inicial del proyecto . . . . .	292
99.1.1	Crear el directorio del proyecto . . . . .	292
99.2	Paso 2: Agregar las operaciones básicas (suma, resta, multiplicación, división)	293
99.2.1	Código inicial . . . . .	293
99.2.2	Crear un commit . . . . .	294
99.3	Paso 3: Agregar funcionalidad de radicación y potenciación . . . . .	294
99.3.1	Actualizar main.py . . . . .	294

99.3.2	Crear un commit . . . . .	295
99.4	Paso 4: Refactorización del código en múltiples archivos . . . . .	295
99.4.1	Crear estructura modular . . . . .	295
99.4.2	Actualizar main.py . . . . .	296
99.4.3	Crear un commit . . . . .	296
99.5	Paso 5: Manejo de errores más detallado . . . . .	296
99.5.1	Mejorar el manejo de errores en division . . . . .	296
99.5.2	Crear un commit . . . . .	297
99.6	Paso 6: Testeo automatizado . . . . .	297
99.6.1	Crear pruebas unitarias . . . . .	297
99.6.2	Ejecutar las pruebas . . . . .	298
99.7	Siguientes pasos . . . . .	298
<b>100</b>	<b>Sistema de Gestión de Cursos con Django y DRF</b>	<b>299</b>
100.11.	Configuración del Proyecto . . . . .	299
100.1.1	Paso 1: Crear el proyecto y entorno virtual . . . . .	299
100.1.2	Paso 2: Crear el proyecto y la aplicación . . . . .	299
100.1.3	Paso 3: Configurar INSTALLED_APPS . . . . .	300
100.1.4	Paso 4: Migrar la base de datos . . . . .	300
<b>1012.</b>	<b>Modelos</b>	<b>301</b>
101.0.1	Paso 1: Definir los modelos . . . . .	301
101.0.2	Paso 2: Migrar los modelos . . . . .	301
<b>1023.</b>	<b>Serializers</b>	<b>302</b>
102.0.1	Paso 1: Crear el archivo serializers.py . . . . .	302
<b>1034.</b>	<b>Vistas</b>	<b>303</b>
103.0.1	Paso 1: Crear las vistas . . . . .	303
<b>1045.</b>	<b>Rutas</b>	<b>304</b>
104.0.1	Paso 1: Configurar las rutas de la aplicación . . . . .	304
104.0.2	Paso 2: Incluir las rutas en el proyecto . . . . .	304
<b>1056.</b>	<b>Pruebas de la API</b>	<b>305</b>
105.0.1	Paso 1: Levantar el servidor . . . . .	305
105.0.2	Paso 2: Endpoints disponibles . . . . .	305
<b>106</b>	<b>Tienda Virtual.</b>	<b>306</b>
106.1	Descripción del Proyecto . . . . .	306
106.2	Objetivo General . . . . .	306
106.3	Funcionalidades Principales . . . . .	306
<b>107</b>	<b>Fases del Proyecto</b>	<b>308</b>
107.1	Fase 1: Configuración Inicial . . . . .	308
107.2	Fase 2: Desarrollo del Backend . . . . .	308
107.3	Fase 3: Desarrollo del Frontend . . . . .	309
107.4	Fase 4: Integración de Pagos . . . . .	309
107.5	Fase 5: Pruebas . . . . .	309
107.6	Fase 6: Despliegue . . . . .	309

<b>108</b>	<b>Resultado Esperado</b>	<b>310</b>
<b>109</b>	<b>Fase 1: Recolección de Requisitos</b>	<b>311</b>
109.1	Requisitos Funcionales . . . . .	311
109.1.1	Usuarios . . . . .	311
109.1.2	Productos . . . . .	311
109.1.3	Carrito y Pedidos . . . . .	312
109.1.4	Pagos . . . . .	312
109.1.5	Accesibilidad y SEO . . . . .	312
109.2	Requisitos Técnicos . . . . .	312
109.2.1	Backend . . . . .	312
109.2.2	Frontend . . . . .	313
109.2.3	Infraestructura . . . . .	313
109.2.4	Despliegue . . . . .	313
<b>110</b>	<b>Fase 2: Diseño de la Arquitectura</b>	<b>314</b>
110.11.	Arquitectura General del Sistema . . . . .	314
110.2	Detalles importantes: . . . . .	314
110.32.	Backend (Django con Django REST Framework) . . . . .	314
110.3.1 A.	Estructura del Proyecto: . . . . .	314
110.3.2 B.	Tecnologías: . . . . .	315
110.43.	Frontend (React con Vite) . . . . .	315
110.4.1 A.	Estructura del Proyecto: . . . . .	315
110.4.2 B.	Tecnologías: . . . . .	316
110.54.	Integración del Backend y Frontend . . . . .	316
110.5.1	Comunicación: . . . . .	316
110.5.2	Autenticación: . . . . .	316
110.5.3	Manejo de Errores: . . . . .	317
110.65.	Base de Datos (PostgreSQL) . . . . .	317
110.6.1	Relacionando los Modelos: . . . . .	317
110.76.	Infraestructura (Docker y Contenedores) . . . . .	317
110.87.	Despliegue (Railway para Backend, Vercel para Frontend) . . . . .	318
110.8.1	Escalabilidad: . . . . .	318
110.98.	Seguridad y Buenas Prácticas . . . . .	318
<b>111</b>	<b>Fase 3. Desarrollo de la Aplicación</b>	<b>319</b>
111.11.	Desarrollo del Backend (Django con Django REST Framework) . . . . .	319
111.1.1 A.	Configuración Inicial del Proyecto . . . . .	319
111.1.2 B.	Implementación de Modelos . . . . .	320
111.1.3 C.	Serializadores . . . . .	321
111.1.4 D.	Vistas y Rutas . . . . .	321
111.1.5 E.	Autenticación (JWT) . . . . .	322
111.22.	Desarrollo del Frontend (React con Vite) . . . . .	322
111.2.1 A.	Configuración Inicial del Proyecto . . . . .	322
111.2.2 B.	Estructura de Archivos . . . . .	323
111.2.3 C.	Lógica de Peticiones con Axios . . . . .	323
111.33.	Despliegue y Configuración de Docker . . . . .	324
111.3.1 A.	Configuración de Docker . . . . .	324

<b>112</b>	<b>Fase 4: Integración de Funcionalidades Avanzadas</b>	<b>326</b>
112.11.	Implementación de la Autenticación de Usuarios con JWT . . . . .	326
112.1.1 A.	Configuración de la Autenticación JWT en el Backend . . . . .	326
112.1.2 B.	Configuración del Frontend para JWT . . . . .	327
112.22.	Integración con PayPal . . . . .	328
112.2.1 A.	Configuración del SDK de PayPal . . . . .	328
112.33.	Optimización de Flujo de Trabajo . . . . .	330
112.3.1 A.	Validaciones en Formularios . . . . .	330
112.3.2 B.	Paginación en Listas de Productos . . . . .	330
112.3.3 C.	Optimización con Caching . . . . .	330
112.3.4 D.	Dockerización Completa . . . . .	331
112.44.	Despliegue en Producción . . . . .	331
112.4.1 A.	Configuración de Servidor . . . . .	331
112.4.2 B.	SSL con Let's Encrypt . . . . .	331
112.4.3 C.	Configuración de Base de Datos . . . . .	331
<b>113</b>	<b>Fase 5: Configuración para Despliegue y Pruebas E2E</b>	<b>332</b>
113.11.	Configuración del Entorno de Producción . . . . .	332
113.1.1 A.	Configuración del Servidor con NGINX y Gunicorn . . . . .	332
113.1.2 B.	Configurar Certificados SSL con Let's Encrypt . . . . .	333
113.22.	Pruebas E2E con Cypress . . . . .	334
113.2.1 A.	Instalación de Cypress . . . . .	334
113.2.2 B.	Crear Pruebas E2E . . . . .	334
113.2.3 C.	Pruebas para el Flujo Completo . . . . .	334
113.33.	Dockerización Completa . . . . .	335
113.3.1 A.	Actualización del <code>docker-compose.yml</code> . . . . .	335
<b>X</b>	<b>Ejercicios</b>	<b>337</b>
<b>114</b>	<b>Ejercicios Python - Nivel Intermedio (POO) - Parte 1</b>	<b>338</b>
<b>115</b>	<b>Ejemplo de uso</b>	<b>339</b>
<b>116</b>	<b>Ejemplo de uso</b>	<b>340</b>
<b>117</b>	<b>Ejemplo de uso</b>	<b>341</b>
<b>118</b>	<b>Ejemplo de uso</b>	<b>342</b>
<b>119</b>	<b>Ejemplo de uso</b>	<b>343</b>
<b>120</b>	<b>Ejemplo de uso</b>	<b>344</b>
<b>121</b>	<b>Ejemplo de uso</b>	<b>345</b>
<b>122</b>	<b>Ejemplo de uso</b>	<b>346</b>
<b>123</b>	<b>Ejemplo de uso</b>	<b>347</b>
<b>124</b>	<b>Ejemplo de uso</b>	<b>348</b>

<b>125Ejercicios Python - Nivel Intermedio (POO) - Parte 2</b>	<b>349</b>
<b>126Ejercicios Python - Nivel Intermedio (POO) - Parte 3</b>	<b>354</b>
<b>XI Extras</b>	<b>358</b>
<b>127Laboratorio: Desarrollo de un Sistema de Chat Local Cliente-Servidor</b>	<b>359</b>
127.1Introducción . . . . .	359
127.2Lo que Vamos a Desarrollar . . . . .	360
127.2.1 Análisis de Requisitos: . . . . .	360
127.2.2Historias de Usuario: . . . . .	360
127.2.3Diseño y Arquitectura: . . . . .	360
127.2.4 Codificación: . . . . .	360
127.2.5 Integración Continua: . . . . .	360
127.2.6 Extensión con GUI: . . . . .	360
127.3Objetivos Específicos . . . . .	360
127.4Materiales y Herramientas Necesarias . . . . .	361
127.4.1 Software: . . . . .	361
127.4.2 Conocimientos Previos Requeridos: . . . . .	361
127.4.3 Estructura del Laboratorio . . . . .	361
127.5Resultados Esperados . . . . .	362
<b>128Fase 1: Análisis de Requisitos, Historias de Usuario y Preparación del Proyecto</b>	<b>363</b>
128.1Objetivo . . . . .	363
128.2Conceptos Clave . . . . .	363
128.3Historias de Usuario . . . . .	363
128.3.1 Instrucciones: Fase 1 . . . . .	363
128.3.2 Inicializar un repositorio Git . . . . .	364
128.4Planificar la arquitectura inicial . . . . .	364
128.5Codificar clases base En server.py: . . . . .	364
128.6En client.py: . . . . .	365
128.7Agregar y confirmar cambios en Git . . . . .	365
128.8Pruebas . . . . .	366
<b>129Conclusiones</b>	<b>367</b>
<b>130Fase 2: Implementación del Servidor CLI</b>	<b>368</b>
130.1Objetivo . . . . .	368
130.2Conceptos Clave . . . . .	368
130.3Instrucciones . . . . .	368
130.3.1 1. Actualizar el código del servidor . . . . .	368
130.42. Probar el servidor . . . . .	370
130.4.1 Ejecuta el servidor: . . . . .	370
130.53. Agregar pruebas unitarias básicas . . . . .	370
130.64. Versionar los cambios . . . . .	371
130.7Pruebas . . . . .	371
130.8Conclusiones . . . . .	371

<b>131</b>	<b>Fase 3: Implementación del Cliente CLI</b>	<b>372</b>
131.1	Objetivo . . . . .	372
131.2	Conceptos Clave . . . . .	372
131.3	Instrucciones . . . . .	372
131.3.1.1	Crear la estructura inicial del cliente . . . . .	372
131.3.2.2	Probar el cliente . . . . .	374
131.3.3.3	Agregar pruebas unitarias para el cliente . . . . .	374
131.3.4.4	Versionar los cambios . . . . .	375
131.4	Pruebas . . . . .	375
<b>132</b>	<b>Conclusiones</b>	<b>376</b>
<b>133</b>	<b>Fase 4: Extender la Aplicación con una Interfaz Gráfica para el Cliente usando Tkinter</b>	<b>377</b>
133.1	Objetivo . . . . .	377
133.2	Conceptos Clave . . . . .	377
133.3	Instrucciones . . . . .	377
133.3.1.1	Crear la estructura inicial para la GUI . . . . .	377
133.3.2.2	Probar la interfaz gráfica . . . . .	380
133.4.3	Versionar los cambios . . . . .	381
133.5	Pruebas . . . . .	381
133.6	Conclusiones . . . . .	381
<b>134</b>	<b>Fase 5: Implementación de Pruebas e Integración Continua</b>	<b>382</b>
134.1	Objetivo . . . . .	382
134.2	Conceptos Clave . . . . .	382
134.3	Instrucciones . . . . .	382
134.3.1.1	Configurar un entorno de pruebas . . . . .	382
134.3.2.2	Escribir pruebas unitarias para el servidor . . . . .	383
134.3.3.3	Escribir pruebas para el cliente . . . . .	383
134.3.4.4	Ejecutar las pruebas . . . . .	384
134.3.5.5	Configurar integración continua con GitHub Actions . . . . .	384
134.4	Pruebas y Validación . . . . .	386
134.5	Conclusiones . . . . .	386
<b>135</b>	<b>Fase 6: Documentación del Laboratorio y Reflexión Final</b>	<b>387</b>
135.1	Objetivo . . . . .	387
<b>136</b>	<b>Parte 2: Codificación por Fases</b>	<b>389</b>
136.1	Implementa las clases base: . . . . .	389
136.2	Pruebas: . . . . .	389
136.2.1	Ejecuta las pruebas: . . . . .	389
136.3	Integración continua: . . . . .	389
136.4	Interfaz gráfica (opcional): . . . . .	389
136.5	Diseño y Arquitectura: . . . . .	389
136.6	Pruebas: . . . . .	390
136.7	Integración Continua: . . . . .	390
136.8	Desafíos Técnicos: . . . . .	390
136.9	Resultados Obtenidos: . . . . .	390

136.10	Lecciones Clave:	390
136.11	Próximos Pasos:	390
136.12	Entrega Final	391
136.12	Repositorio GitHub:	391
136.13	Evidencias:	391
<b>XII Pruebas Técnicas</b>		<b>392</b>
<b>137</b>	<b>Prueba Técnica para Programador Junior - Python</b>	<b>393</b>
137.1	Parte 1: Programación Orientada a Objetos (15 puntos)	393
137.1.1	Clase Producto (15 puntos)	393
137.1.2	Ejemplo de uso:	394
137.2	Parte 2: Lógica de Programación y Estructuras de Datos (15 puntos)	394
137.2.1	Fibonacci Recursivo (10 puntos)	394
137.2.2	Ejemplo de uso:	394
137.2.3	Análisis de texto (5 puntos)	394
137.2.4	Ejemplo de uso:	394
<b>138</b>	<b>Criterios de Evaluación</b>	<b>395</b>
138.1	Parte 1: Programación Orientada a Objetos (15 puntos)	395
138.2	Parte 2: Lógica de Programación y Estructuras de Datos (15 puntos)	395
138.3	Notas de calificación:	396
<b>139</b>	<b>Resolución de la Prueba Técnica para Programador Junior - Python</b>	<b>397</b>
139.1	Parte 1: Programación Orientada a Objetos	397
139.1.1	Archivo: parte1/producto.py	397
139.1.2	Implementación de la clase Producto:	397
139.2	Parte 2: Lógica de Programación y Estructuras de Datos	398
139.2.1	Función fibonacci (Parte 2):	398
139.2.2	Implementación de la función fibonacci:	398
139.2.3	Función analizar_texto (Parte 2):	398
139.2.4	Implementación de la función analizar_texto:	399
139.2.5	Archivo runner.py	399
<b>140</b>	<b>Conclusión:</b>	<b>400</b>
140.1	Ahora puedes ejecutar las pruebas de la siguiente manera:	400

# 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 línea, 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?

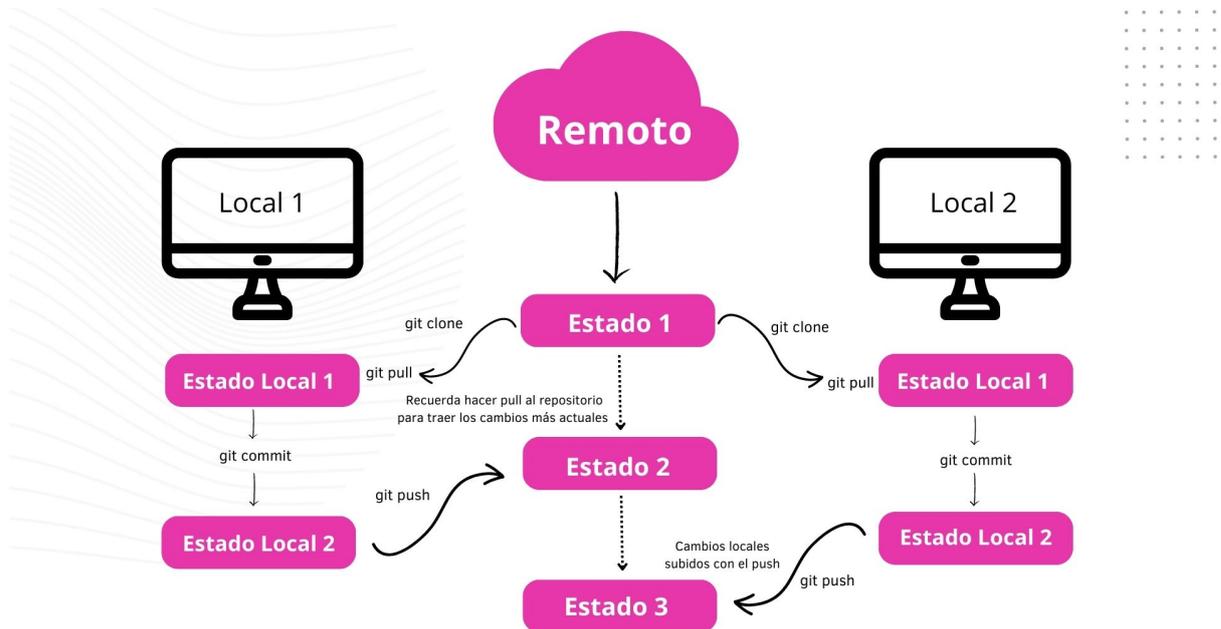


Figure 2.5: Ventajas de Git

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?

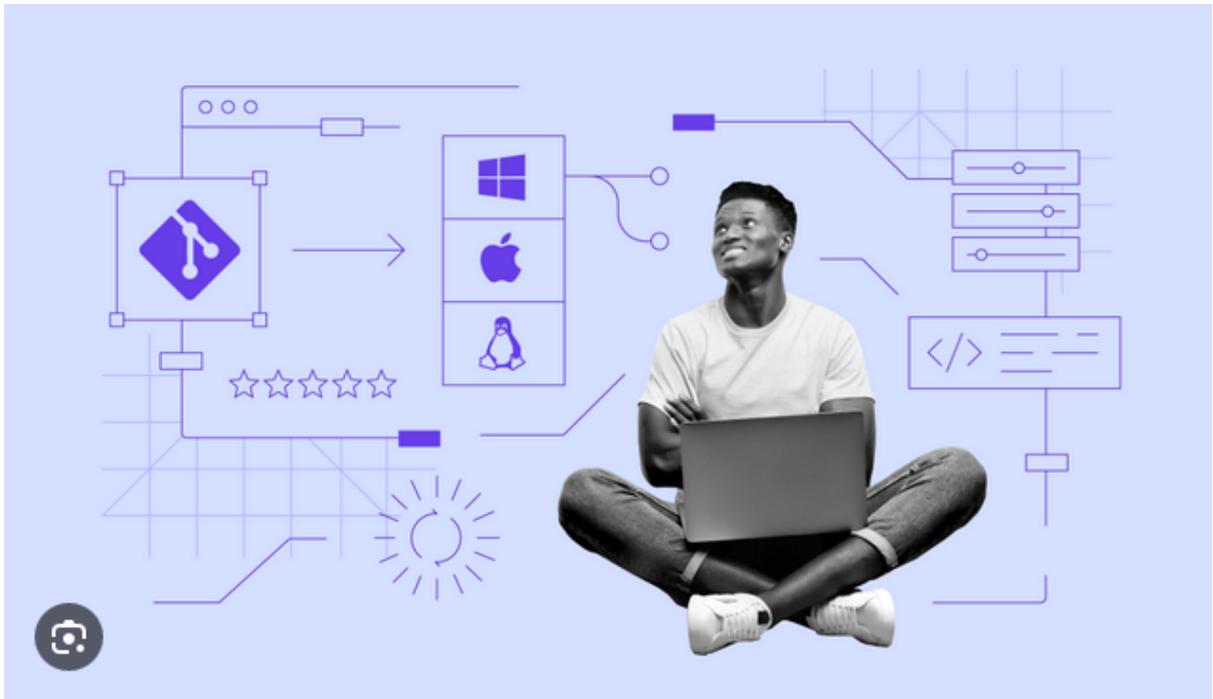


Figure 2.6: Git en Diferentes Sistemas Operativos

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

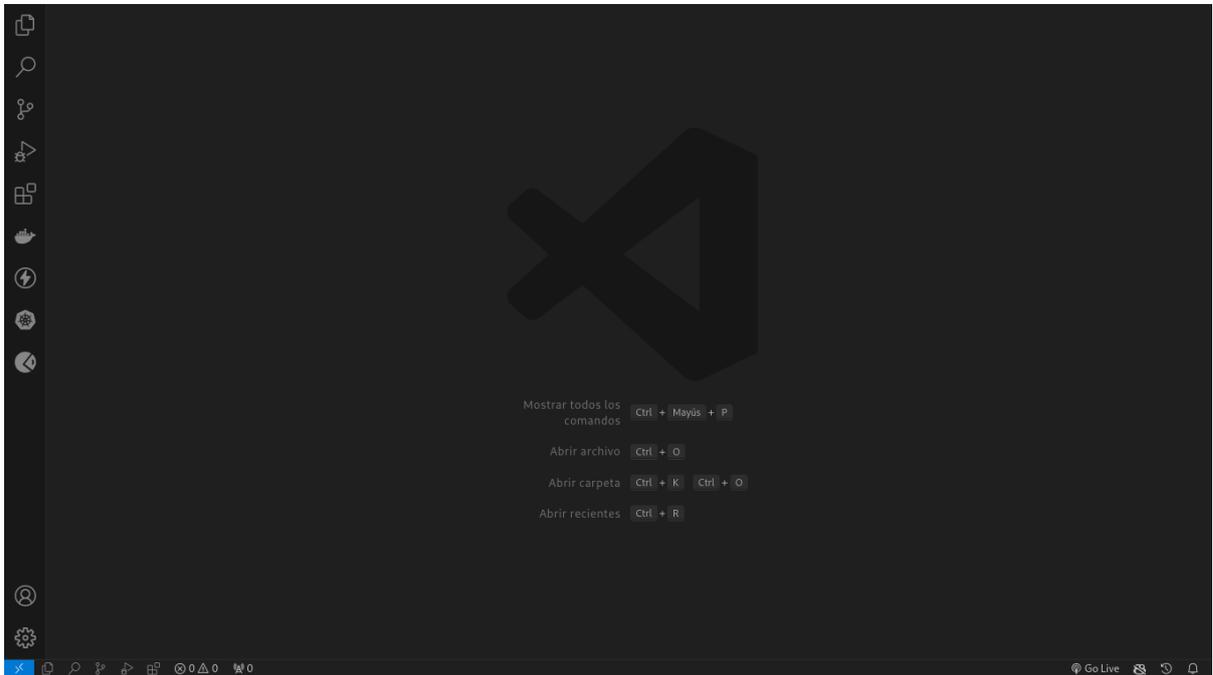


Figure 2.7: Visual Studio Code

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

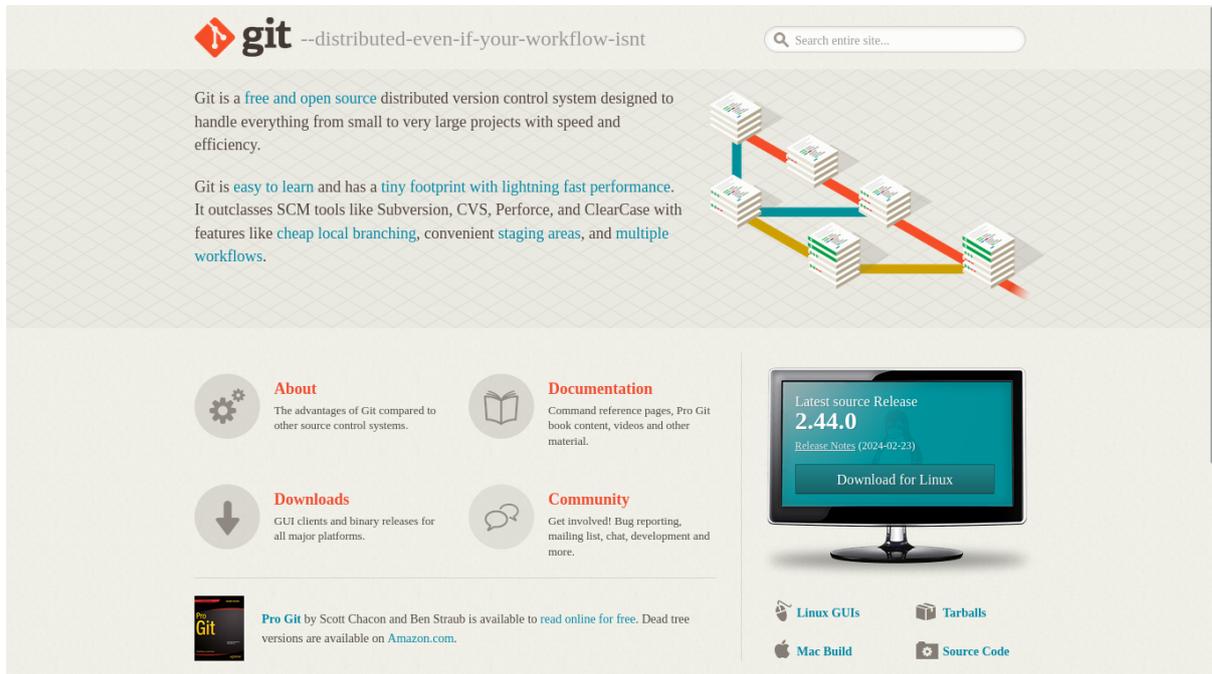


Figure 2.8: Git

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 ábrela en Visual Studio Code.
- Crea un archivo Python llamado **hello\_world.py** y escribe el siguiente código:

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

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

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

```
git init
```

- Añade el archivo al área de preparación con:

```
git add hello_world.py
```

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

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

#### 2.8.4 Comandos Básicos de Git

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

#### 2.8.5 Estados en Git

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

# 3 Tutorial: Moviendo Cambios entre Estados en Git

## 3.1 Introducción

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

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

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

## 3.2 Sección 1: Modificar Archivos en el Repositorio

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

## 3.3 Mover Cambios de Local a Staging:

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

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

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

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

### 3.4 Agregar Cambios de Local a Staging:

```
git add hello_world.py
```

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

### 3.5 Sección 2: Confirmar Cambios en un Commit

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

### 3.6 Mover Cambios de Staging a Commit:

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

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

### 3.7 Sección 3: Creación y Fusión de Ramas

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

### 3.8 Crear una Nueva Rama:

```
git branch feature
```

Hemos creado una nueva rama llamada “feature” para desarrollar una nueva funcionalidad.

### 3.9 Implementar Funcionalidades en la Rama:

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

### 3.10 Fusionar Ramas con la Rama Principal:

```
git merge feature
```

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

### 3.11 Sección 4: Revertir Cambios en un Archivo

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

### 3.12 Revertir Cambios en un Archivo:

```
git reset hello_world.py
```

Hemos revertido los cambios en el archivo **hello\_world.py** y deshecho las modificaciones realizadas.

### 3.13 Conclusión

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

## 4 Asignación

[Hello World!](#)

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

### La Asignación

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

El comando de ejecución del test es:

```
pytest test_hello.py
```

¡Mucha suerte!

## 5 GitHub Classroom



Figure 5.1: Github Classroom

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

### 5.1 ¿Qué es GitHub Classroom?



Figure 5.2: Github Classroom Windows

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

#### 5.1.1 Funcionalidades Principales

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

criterios de evaluación.

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

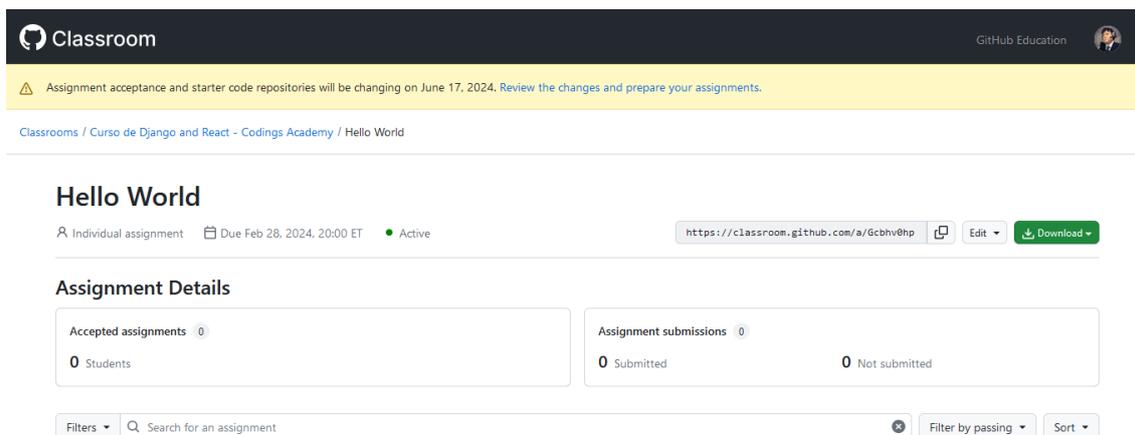
**Seguimiento de Progreso:** Los profesores pueden realizar un seguimiento del progreso de los estudiantes y revisar sus contribuciones a través de solicitudes de extracción (pull requests) y comentarios en el código.

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

## 5.2 Ejemplo Práctico

### 5.2.1 Creación de una Asignación en GitHub Classroom

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



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

- Assignment basics
- Starter code and environment
- Grading and feedback

### Assignment basics

Assignment title \*  
Hello World

Student assignment repositories will have the prefix: `hello-world`

Deadline  
02 / 28 / 2024, 08:00 PM  
(Optional) If left blank, there will be no deadline. Date format: YYYY-MM-DD HH:MM a

This is a cutoff date  
If selected, the student will lose write access to their repository after the date is reached.

Assignment status: Active | Individual or group assignment: Individual assignment  
Assignment type cannot be changed after assignment creation.

Repository visibility

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

**Add a template repository to give students starter code**  
Your assignment will be created with empty student repositories if you don't add starter code. Changes to starter code after students have accepted the assignment will not retroactively change existing student repositories.  
Note: All starter code must use a [template repository](#). Your starter code repository must be either in the same organization as this classroom or a public repository if elsewhere. [Learn about transferring your repositories](#).

Find a GitHub repository  
  
education/autograding-example-python  
GitHub Classroom autograding example repo with Python and Pytest

**GitHub Codespaces**  
Your organization is eligible for GitHub Codespaces. Enable Codespaces in students' repositories to give them a one-click experience for getting started coding, running, and collaborating on their code. [Enable it in Classroom settings](#).

**Supported editor**  
Changing the online IDE after an assignment has been created is not possible.  
 Don't use an online IDE

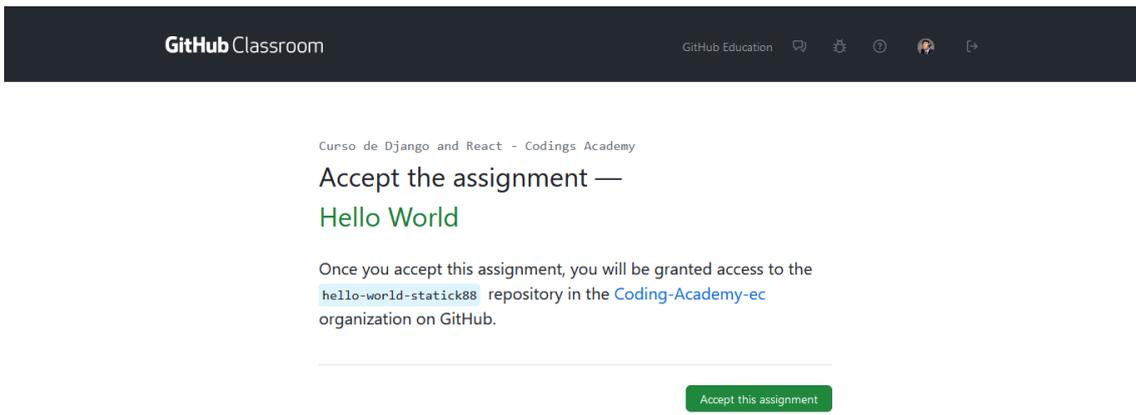
**Grading and feedback**

**Add autograding tests**  
Autograding tests help provide feedback for students immediately upon submission using [GitHub Actions](#). Add a test to enable autograding.

Hello world test

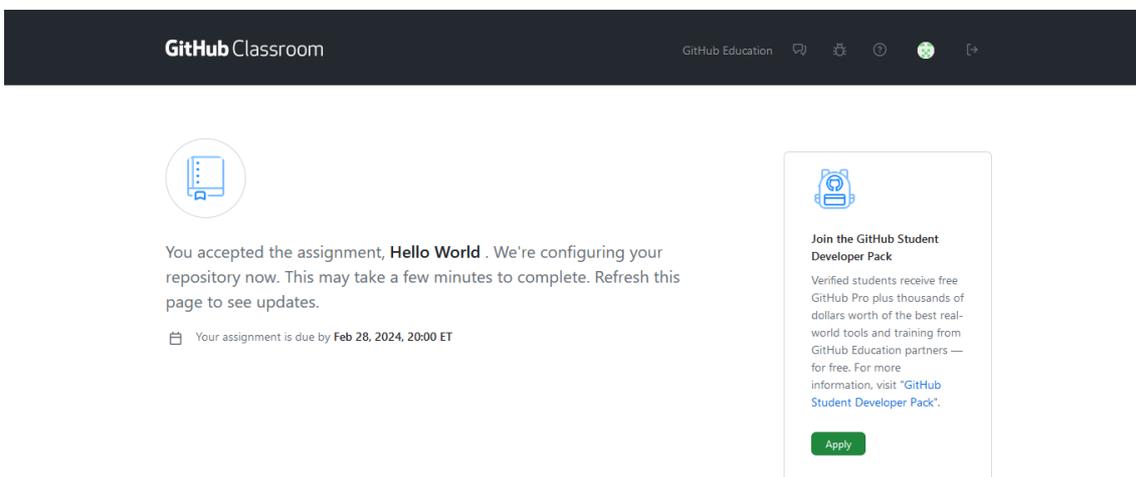
+ Add test

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

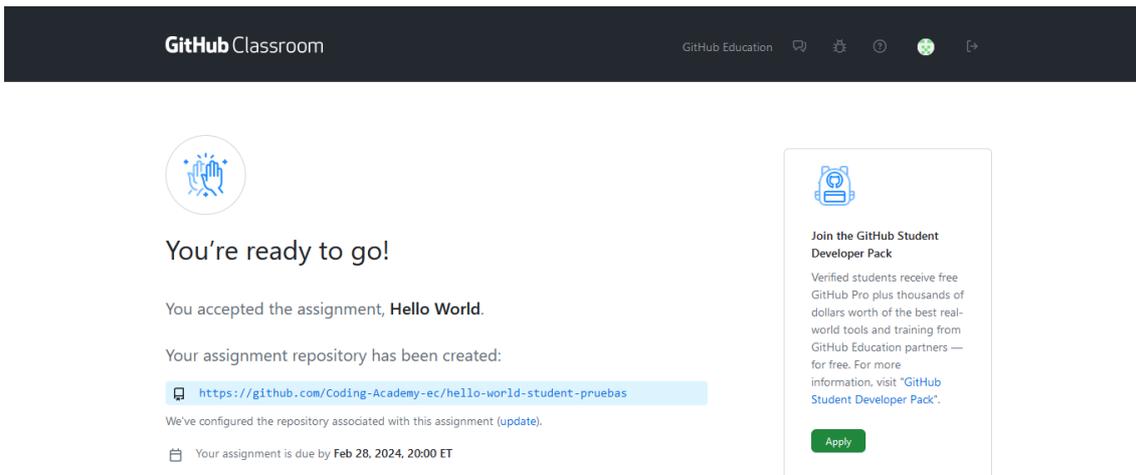


### 5.3 Trabajo de los Estudiantes

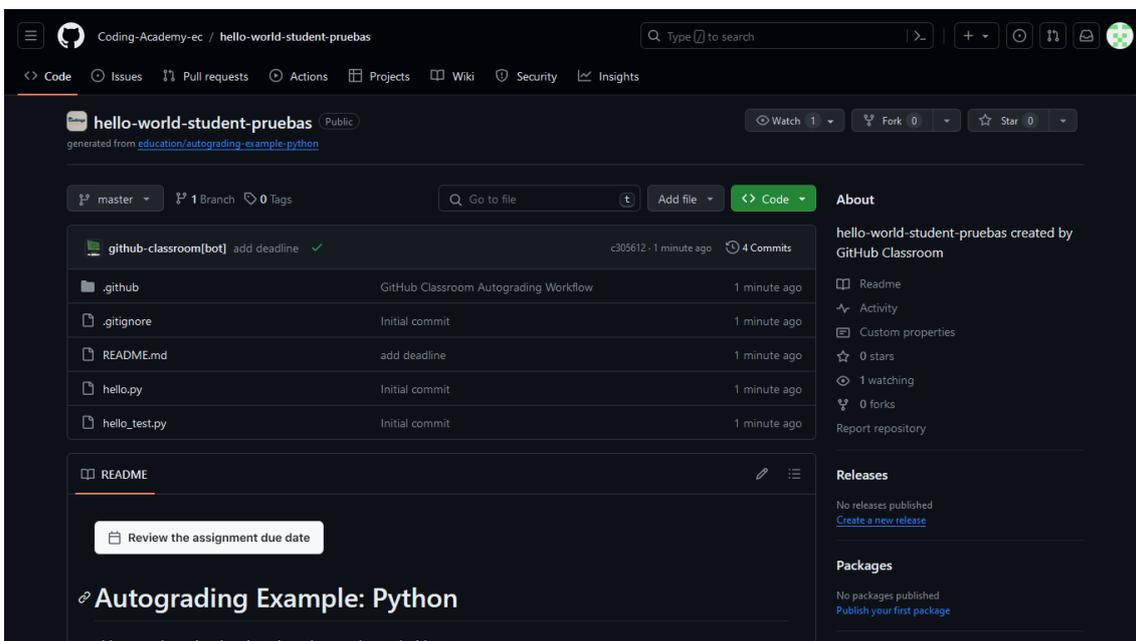
**Aceptar la Asignación:** Los estudiantes reciben el enlace de la asignación y aceptan la tarea, lo que les permite crear un repositorio privado basado en la plantilla proporcionada.



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



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



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

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

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

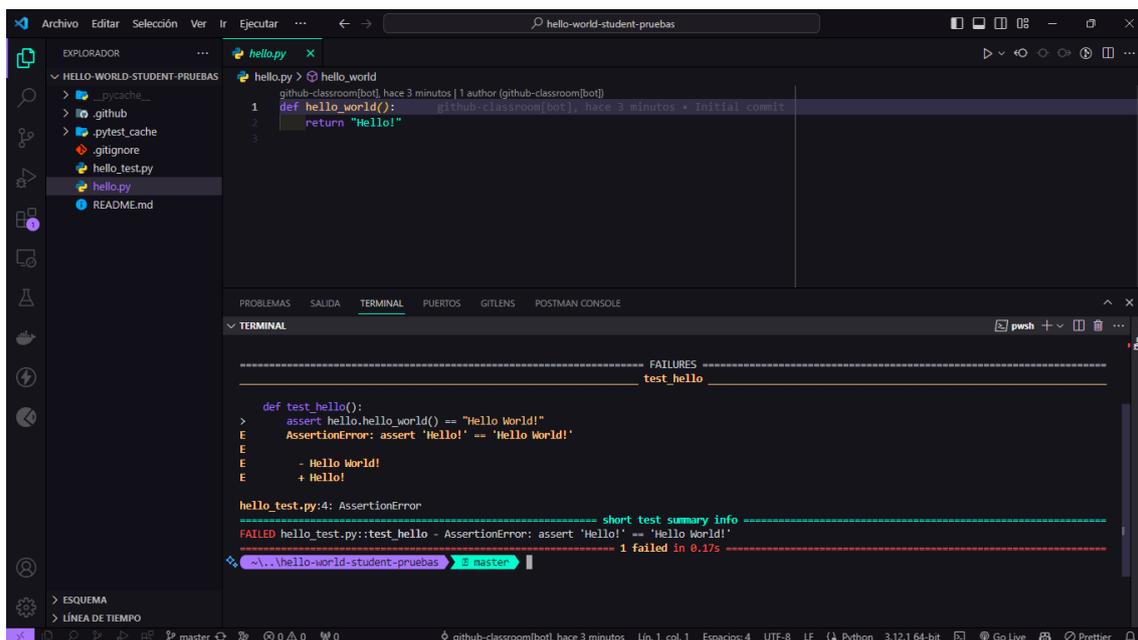
**Desarrollar la Tarea:** Los estudiantes trabajan en la tarea, realizando los cambios necesarios y realizando commits de manera regular para mantener un historial de su trabajo.

### 💡 Tip

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

## pytest

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



```
hello.py
1 def hello_world():
2     return "Hello!"
3

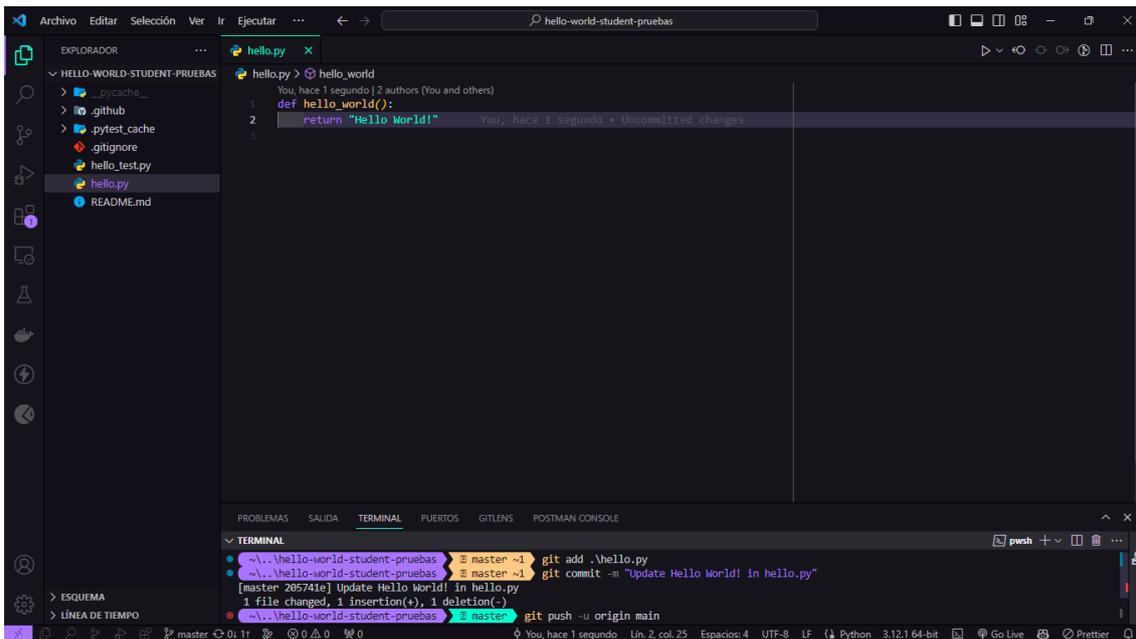
hello_test.py
1 def test_hello():
2     assert hello.hello_world() == "Hello World!"
3

Terminal
----- FAILURES -----
test_hello

def test_hello():
> assert hello.hello_world() == "Hello World!"
E       AssertionError: assert 'Hello!' == 'Hello World!'
E
E       - Hello World!
E       + Hello!

hello_test.py:4: AssertionError
----- short test summary info -----
FAILED hello_test.py::test_hello - AssertionError: assert 'Hello!' == 'Hello World!'
1 failed in 0.17s
```

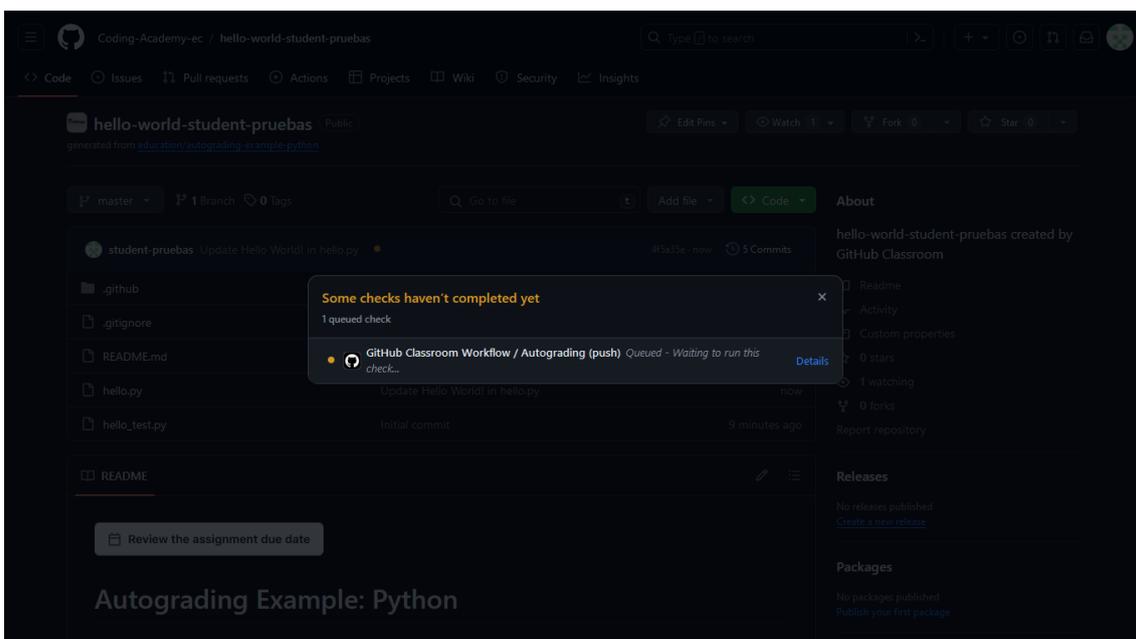
**Enviar la Solicitud de Extracción:** Una vez completada la tarea, los estudiantes envían una solicitud de extracción desde su rama hacia la rama principal del repositorio, solicitando la revisión del profesor.



Una vez realizado el push se envía al 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



Este Action lo que hace es evaluar los cambios realizados

The screenshot shows the Autograding interface. On the left, there's a sidebar with 'Summary', 'Jobs', 'Autograding', 'Run details', 'Usage', and 'Workflow file'. The main area displays the output of a test run for 'Run education/autograding@v1'. The output shows the installation of dependencies (tomli, iniconfig, exceptiongroup, pluggy) and the execution of a test session. The test 'Hello world test' passed successfully, resulting in a score of 100/100. The interface also includes a 'Beta' badge, 'Give Feedback' link, and a search bar for logs.

**Tip**

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.

The screenshot shows a GitHub Classroom assignment page for 'Hello World'. The page includes the assignment title, a search bar, and a 'Download' button. Below the title, there are three summary cards: 'Accepted assignments 1' (1 Student), 'Assignment submissions 1' (1 Submitted, 0 Not submitted), and 'Passed students 1' (1/1 Passed). A search bar and filters are present. The 'Total students' section shows a list of students, with 'student-pruebas' listed as 'Submitted' with a score of 100/100. The footer contains the GitHub logo, copyright information, and various links like Terms, Privacy, Security, Status, Docs, Contact GitHub, Pricing, API, Training, Blog, and About.

 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.

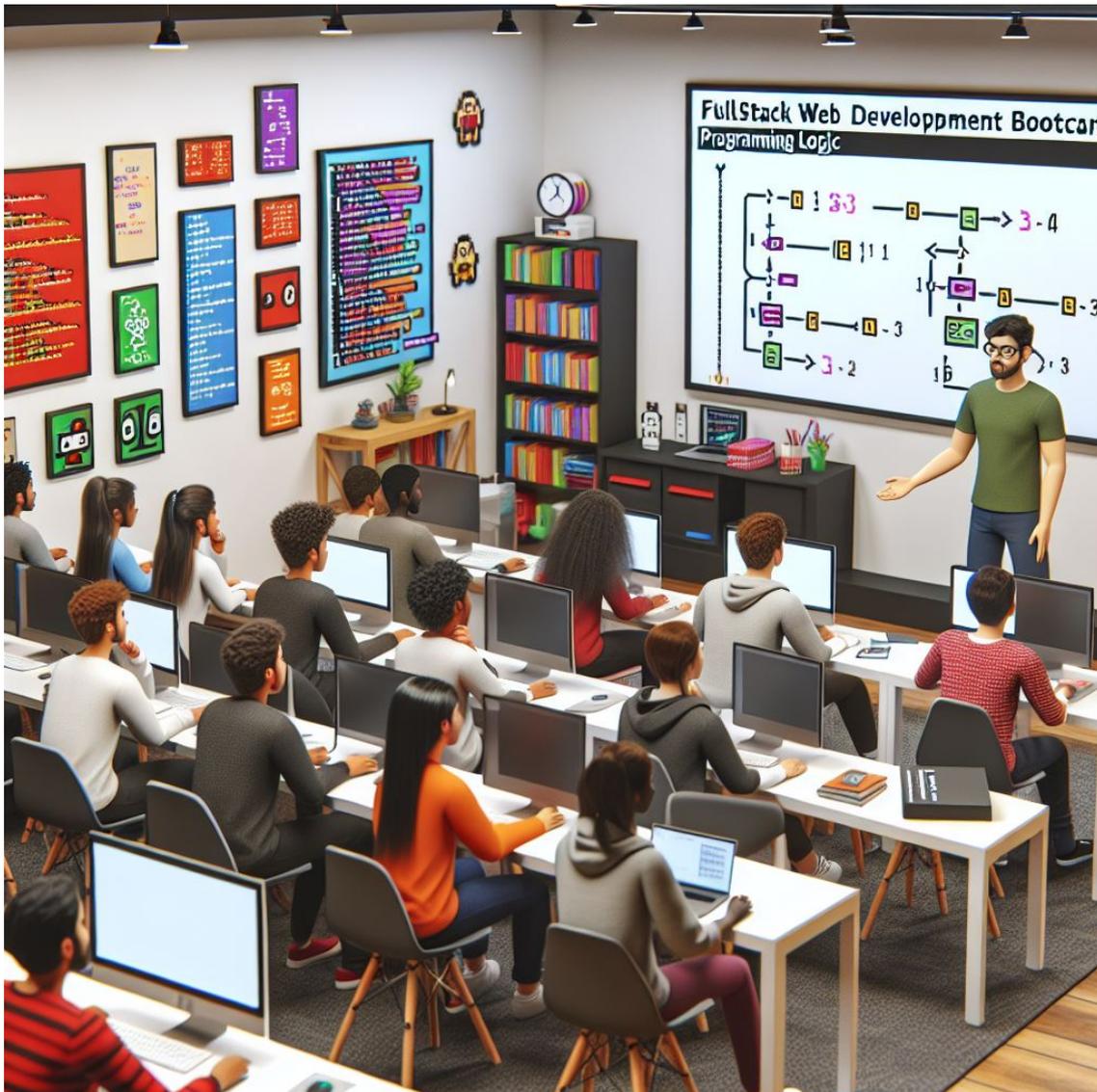


Figure 6.1: Lógica de la Programación

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 también 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 quisiéramos 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](#)
- [Retod 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/>.

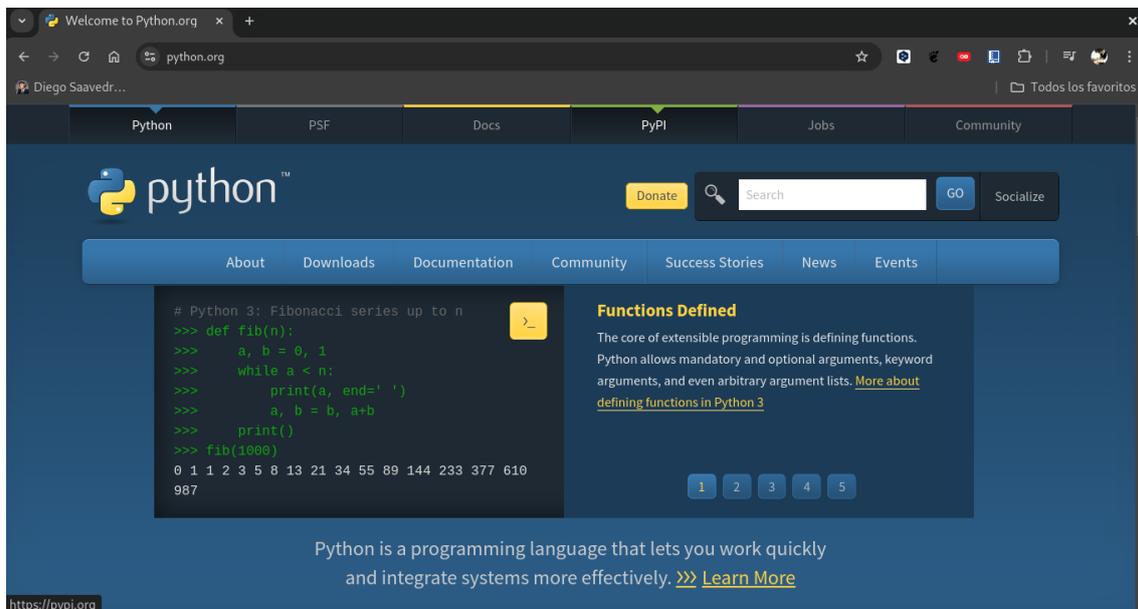


Figure 6.3: Python

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

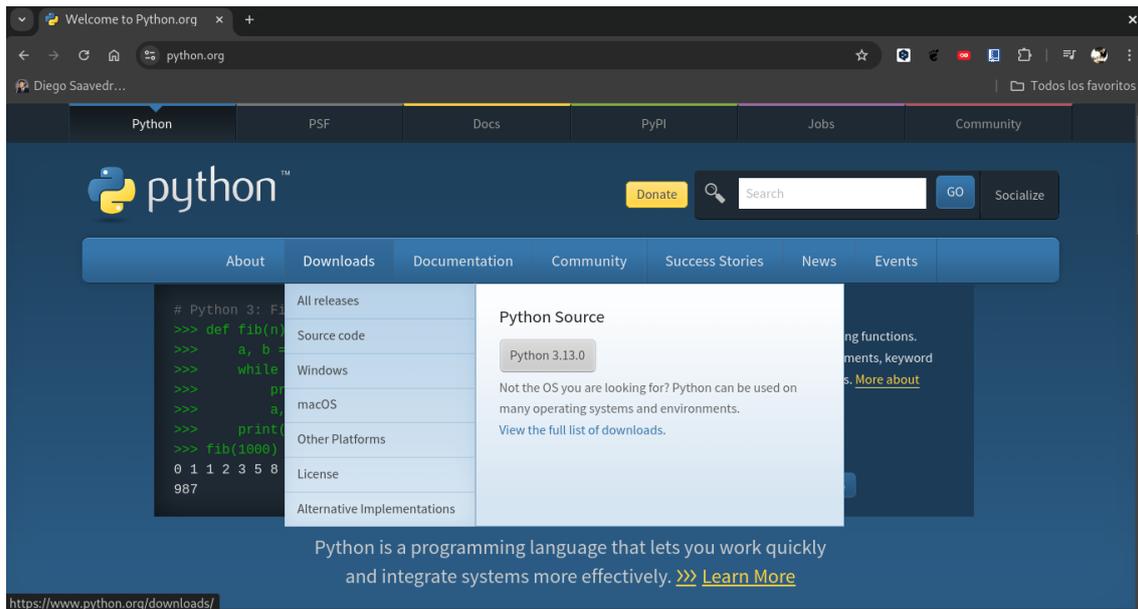


Figure 6.4: Python

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).

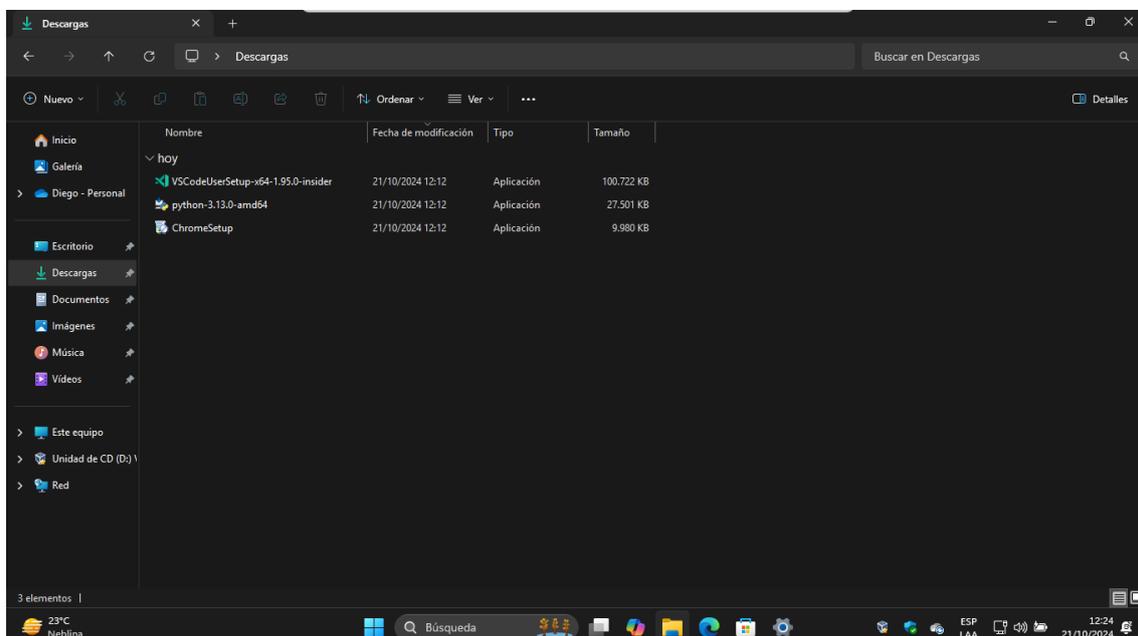


Figure 6.5: Python

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

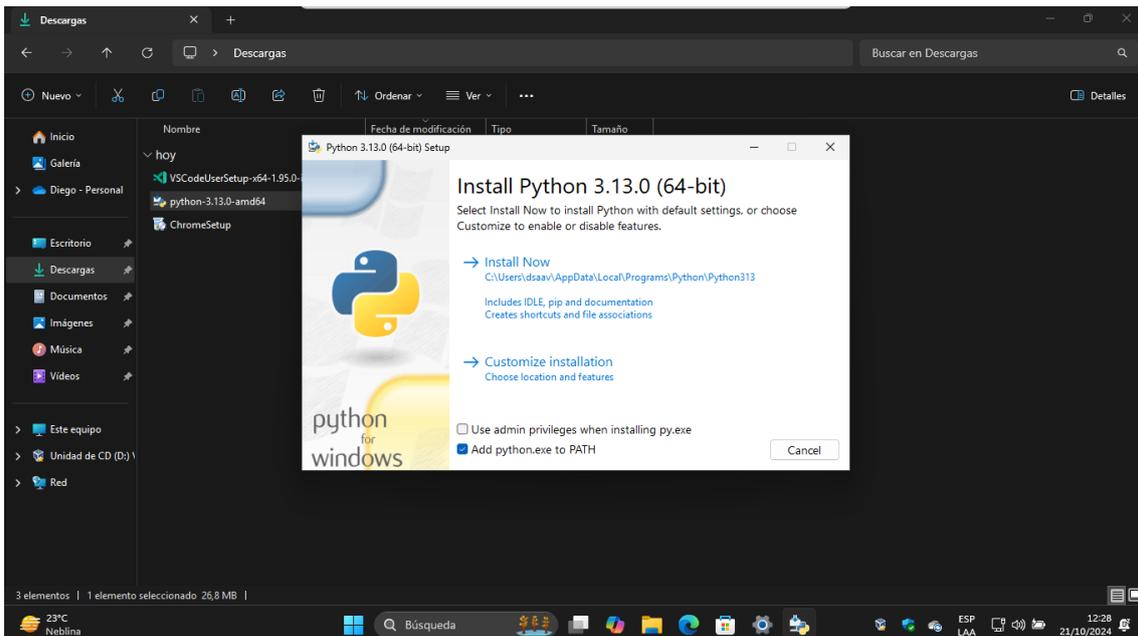


Figure 6.6: Python

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:

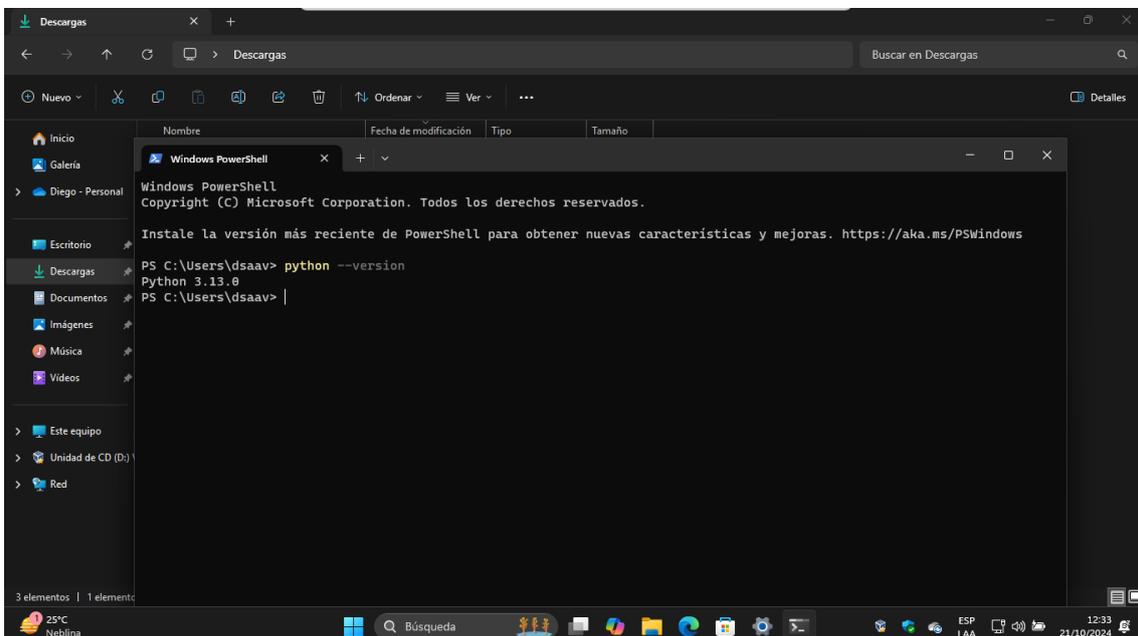


Figure 6.7: Python

```
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.

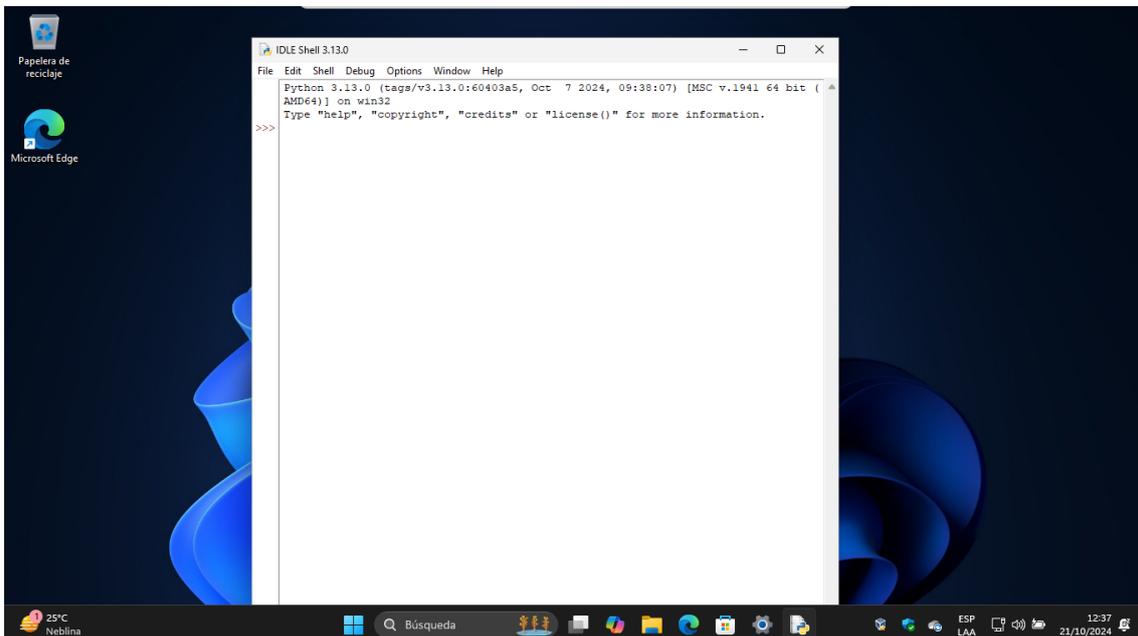


Figure 8.1: IDLE

- **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.

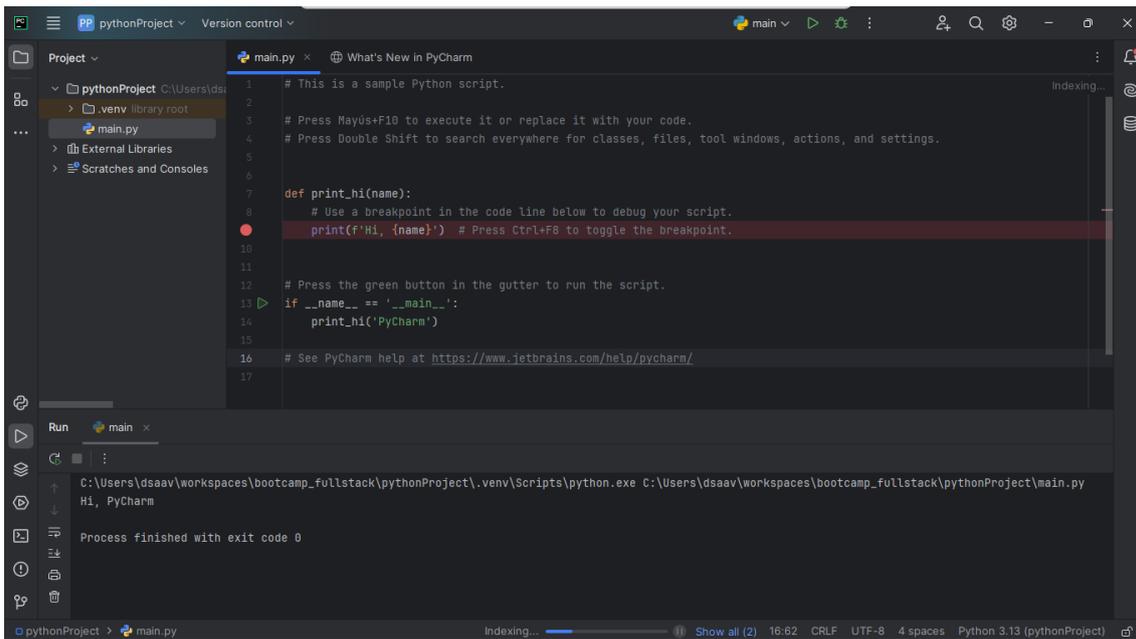


Figure 8.2: PyCharm

- **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.

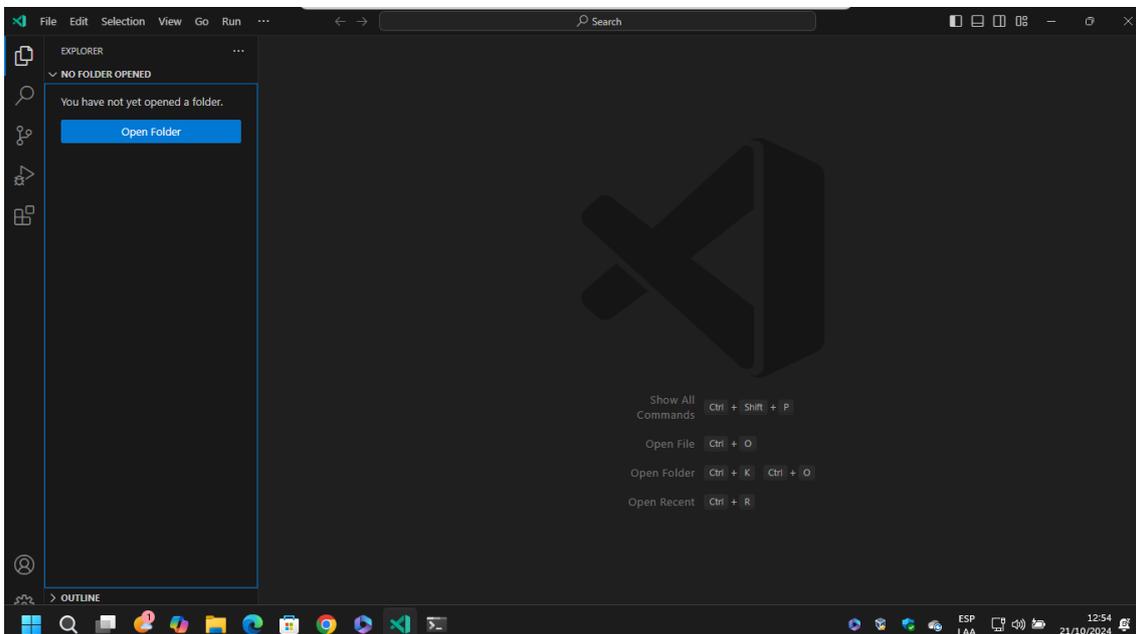


Figure 8.3: Visual Studio Code

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

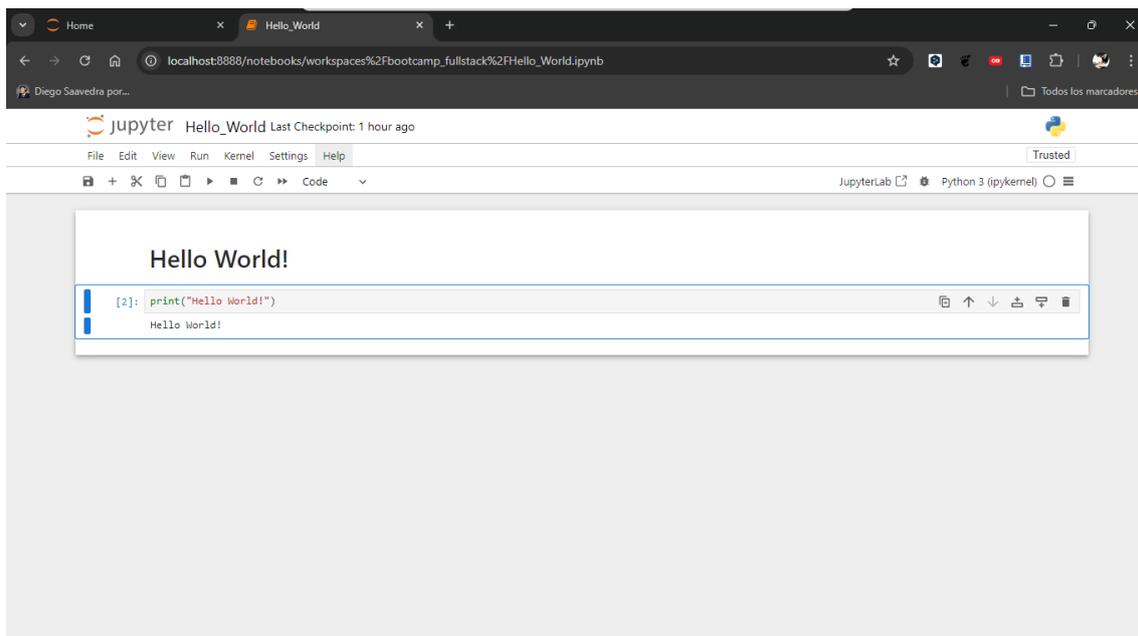


Figure 8.4: Jupyter Notebook

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 indentación para definir bloques de código. La indentación 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 indentada 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 indentada, 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 comentarios
print("Hola, mundo!") # Este es otro comentarios
```

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, indentación, 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.

También 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**, Codigo: **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 quieres 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

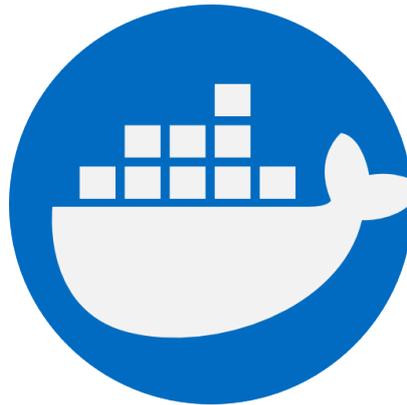


Figure 26.1: Docker

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.



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 aísla 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:

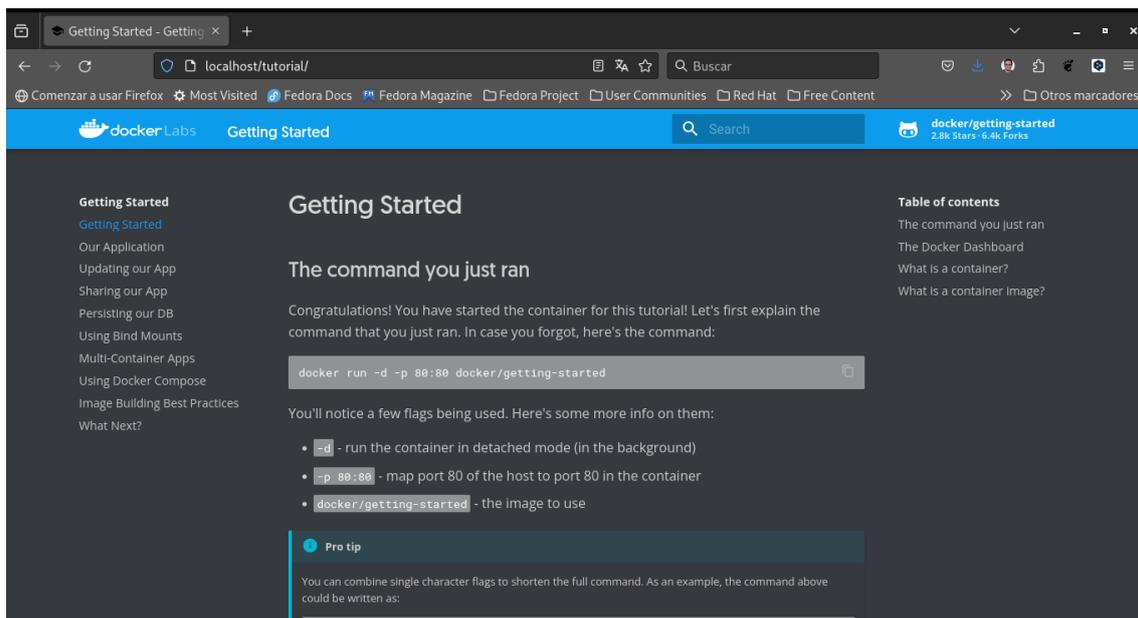
```
docker pull docker/getting-started
```

```
✓ TERMINAL
• statick main U:11 ? :7 ~/workspaces/curso_docker/book docker pull docker/getting-started
Using default tag: latest
latest: Pulling from docker/getting-started
Digest: sha256:d79336f4812b6547a53e735480dde67f8f7071b414fbd9297609ffb989abc1
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 con el ID real del contenedor que obtuviste en el paso anterior.

6. Elimina el contenedor detenido:

```
docker rm <CONTAINER_ID>
```

Reemplaza con el ID real del contenedor.

### Tip

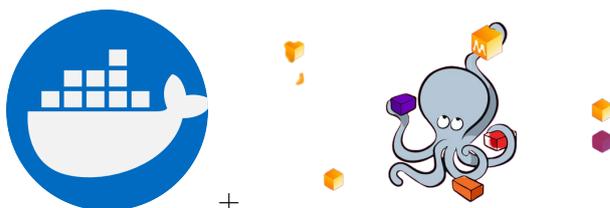
Combina los comandos `docker ps`, `docker stop`, y `docker rm` para gestionar contenedores eficientemente.

¡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

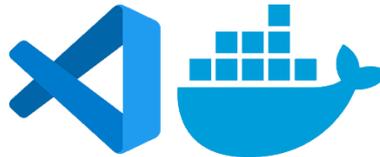


Figure 30.1: DevContainers

### 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.

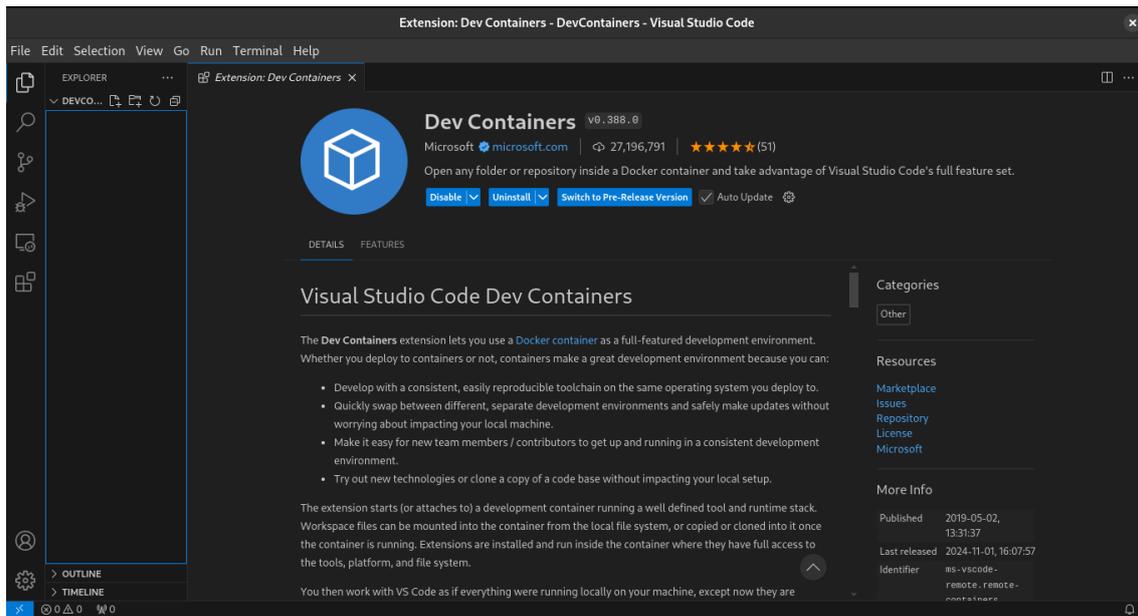


Figure 30.2: DevContainer en Visual Studio Code

### 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.

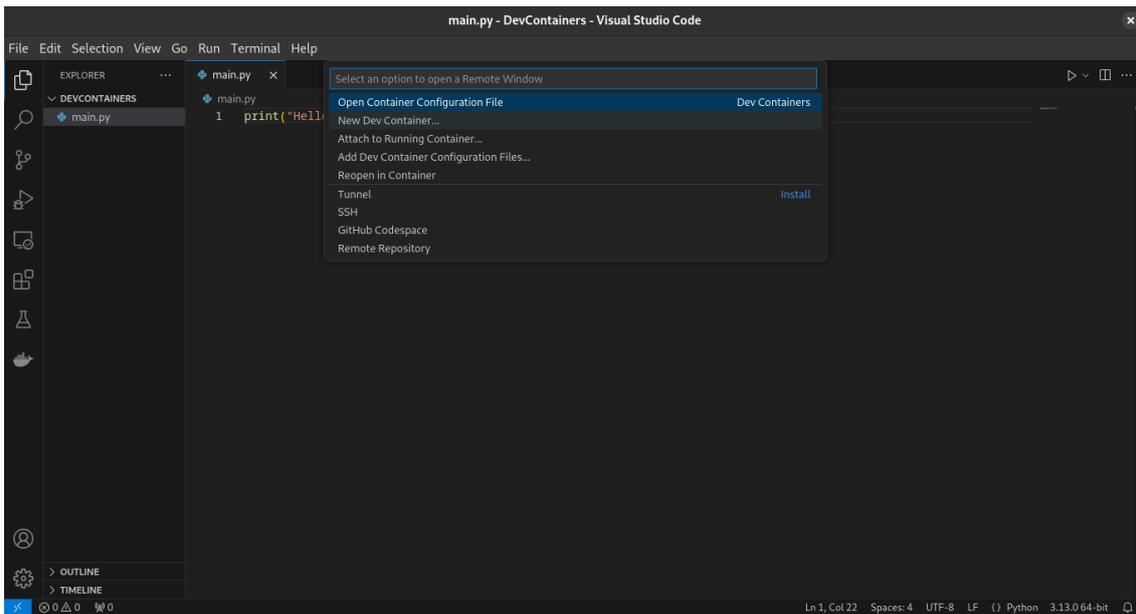


Figure 30.3: New DevContainer

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.

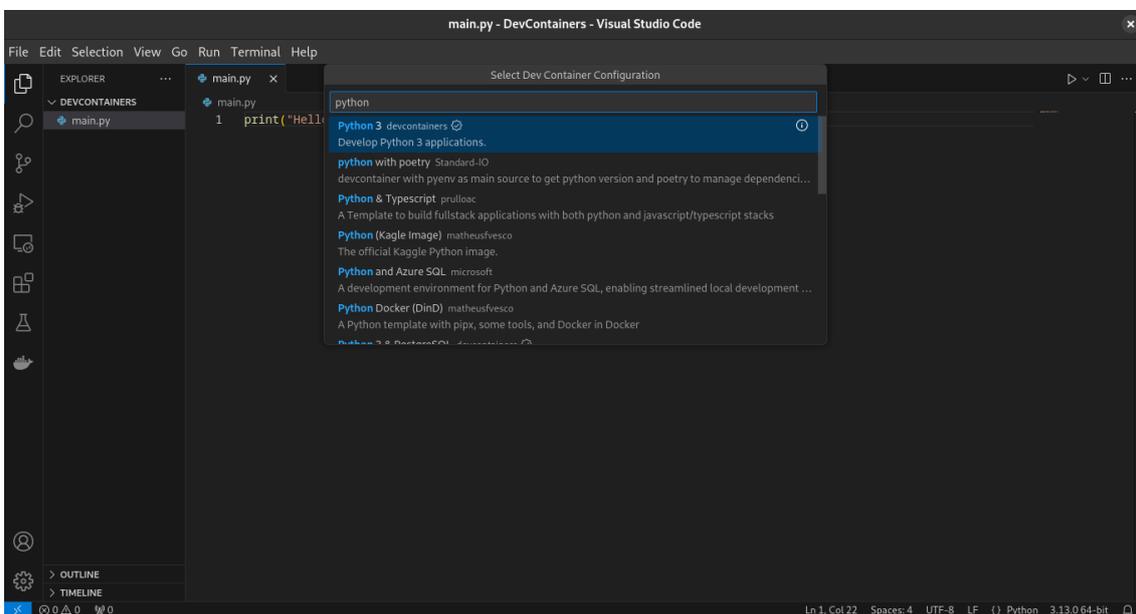


Figure 30.4: Python 3 DevContainer

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.

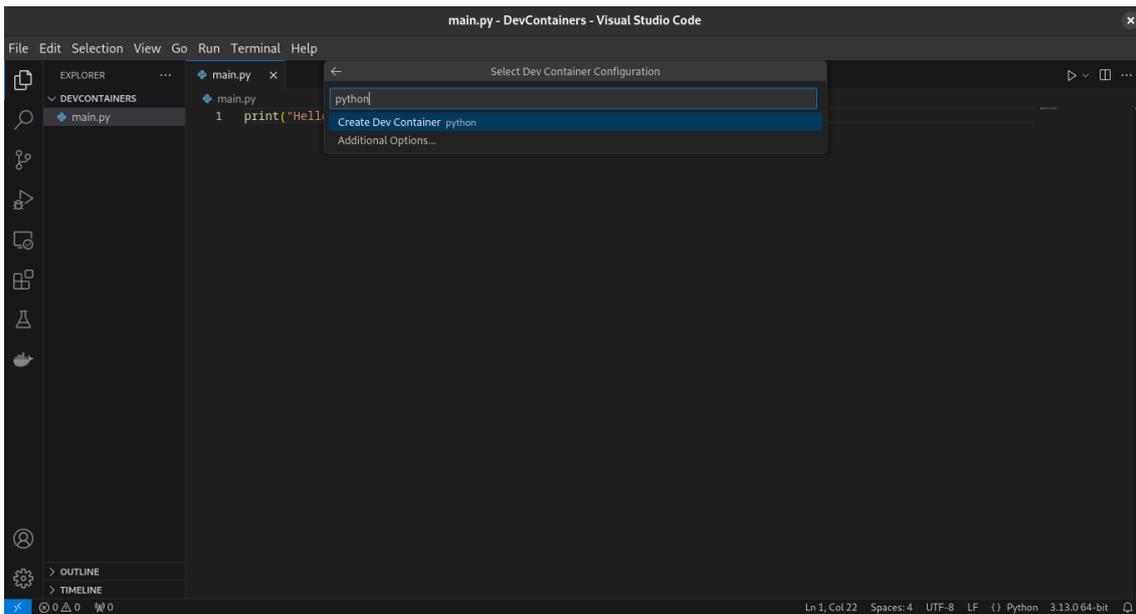


Figure 30.5: Create DevContainer

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

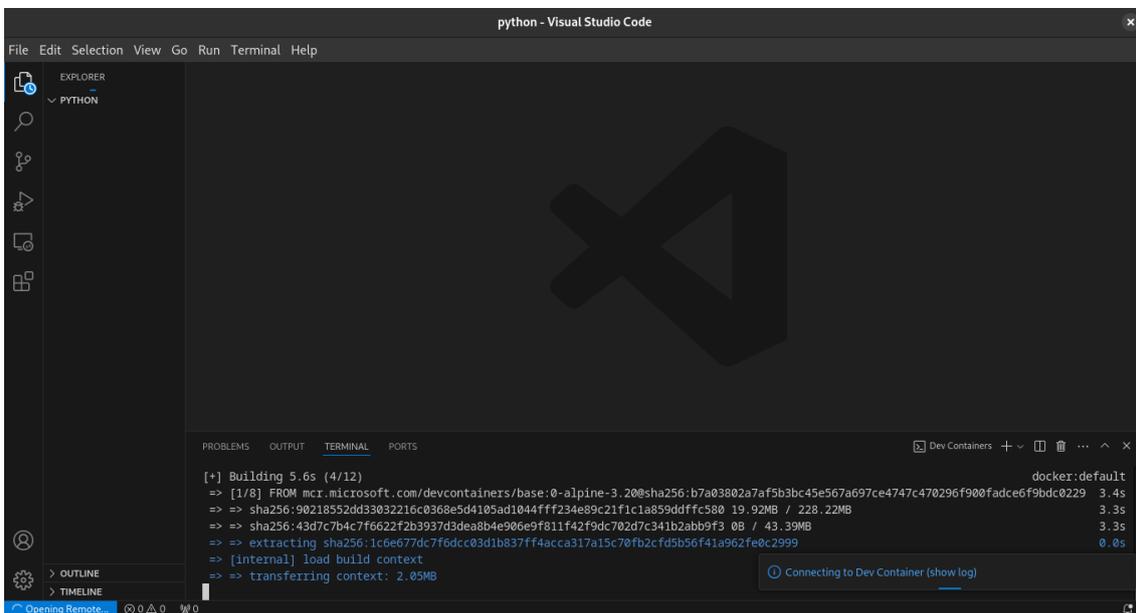


Figure 30.6: Create DevContainer

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.

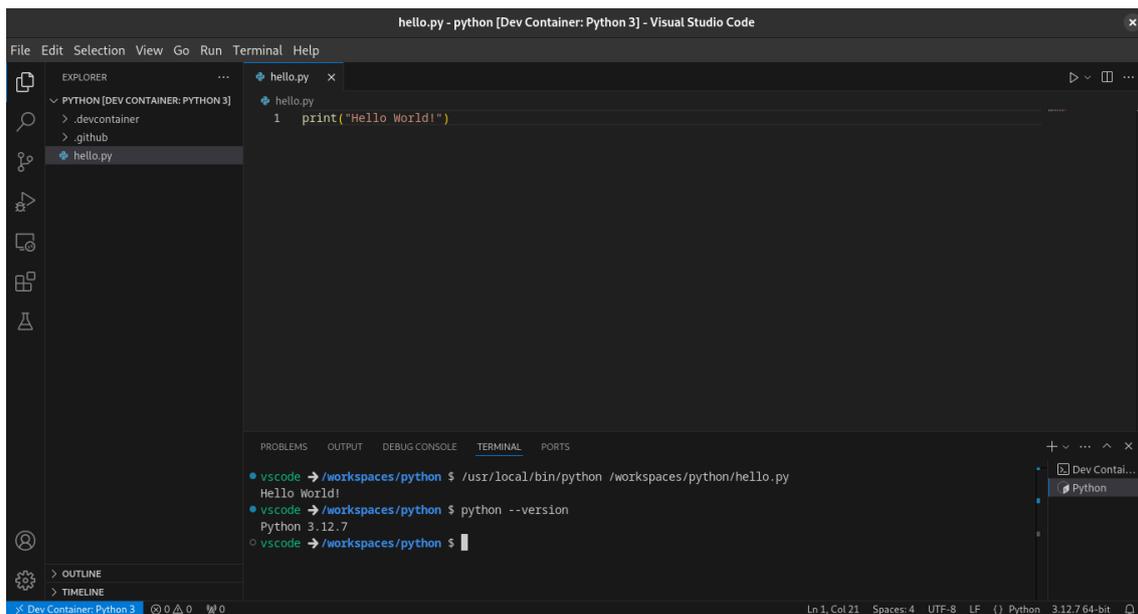


Figure 30.7: Python in DevContainer

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**

# **Unidad 5: Python Avanzado**

# 31 Conceptos Avanzados en Python

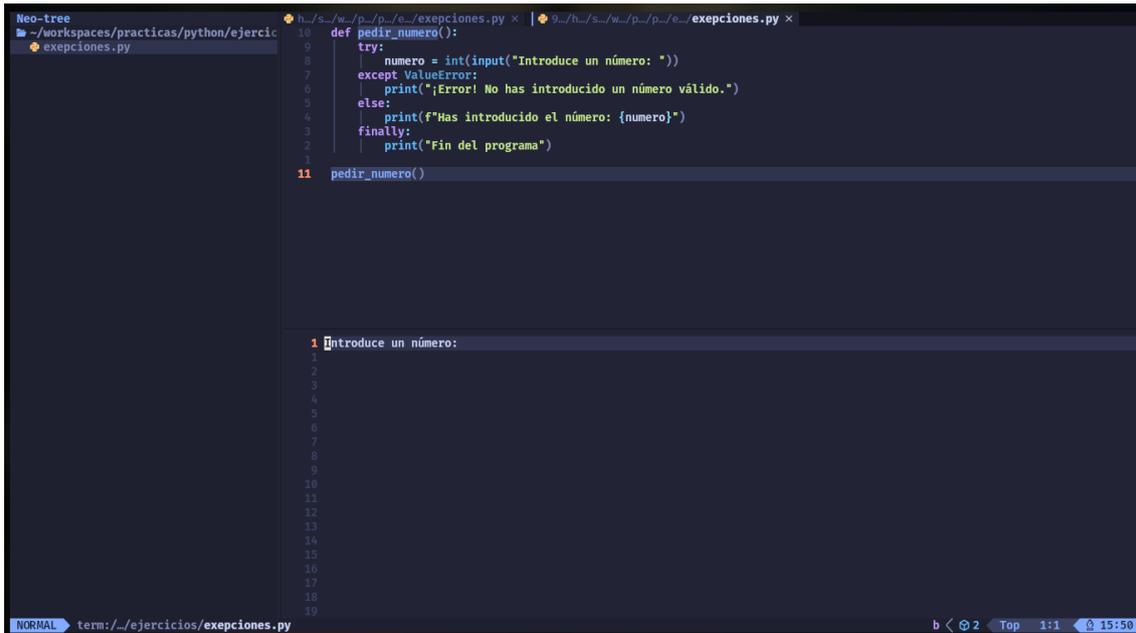
En este capítulo en particular aprenderemos los conceptos avanzados que necesitamos conocer de Python para poder avanzar con temas relacionados al desarrollo de software.

En el mismo aprenderemos:

1. [Excepciones y Manejo de Errores](#)
2. [Lectura y Escritura de Archivos](#)
3. [Programación Funcional](#)
4. [Comprensión y Generadores](#)
5. [Módulos y Paquetes](#)
6. [Decoradores y Context Managers](#)
7. [Colecciones de Datos y Estructuras Especializadas](#)
8. [Manipulación de Fechas y Tiempos](#)
9. [Concurrencia y Paralelismo](#)
10. [Pruebas y Debugging](#)

Este capítulo cubre varios de los aspectos más avanzados de Python, y proporciona una base sólida para desarrollar aplicaciones web fullstack más complejas. Los ejemplos prácticos te ayudarán a entender cómo aplicar estos conceptos en situaciones reales.

## 32 Excepciones y Manejo de Errores



```
def pedir_numero():
    try:
        numero = int(input("Introduce un número: "))
    except ValueError:
        print("Error! No has introducido un número válido.")
    else:
        print(f"Has introducido el número: {numero}")
    finally:
        print("Fin del programa")

pedir_numero()
```

```
1 Introduce un número:
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Figure 32.1: Excepciones y Manejo de Errores

El manejo adecuado de errores es esencial para escribir código robusto. Las excepciones permiten manejar situaciones inesperadas durante la ejecución de un programa sin que este termine abruptamente.

### 32.0.1 Conceptos clave

- **try:** Bloque donde intentamos ejecutar código que puede generar una excepción.
- **except:** Bloque donde capturamos y gestionamos una excepción.
- **else:** Bloque que se ejecuta si no hay excepciones.
- **finally:** Bloque que se ejecuta independientemente de si hubo una excepción o no.

### 32.0.2 Ejemplo

```
try:
    numero = int(input("Introduce un número: "))
except ValueError as e:
    print(f"Error: {e}. Introduce un número válido.")
```

```

else:
    print(f"El número es {numero}.")
finally:
    print("Operación terminada.")

```

### 32.0.3 Excepciones personalizadas

Podemos crear nuestras propias excepciones para situaciones específicas.

```

class MiError(Exception):
    def __init__(self, mensaje):
        self.mensaje = mensaje
        super().__init__(self.mensaje)

try:
    raise MiError("Algo salió mal")
except MiError as e:
    print(f"Capturado: {e}")

```

### 32.0.4 Ejemplo Práctico

#### Objetivo:

Aprender a manejar excepciones en Python para crear un programa robusto que gestione entradas de usuario incorrectas.

**Descripción:** Crear un programa que pida al usuario un número, y en caso de que se ingrese algo que no sea un número, maneje el error de manera adecuada, mostrando un mensaje informativo.

#### Instrucciones:

- Utiliza un bloque try-except para manejar excepciones de tipo ValueError.
- Agrega un bloque else para confirmar la entrada del usuario si es válida.
- Incluye un bloque finally que imprima un mensaje de despedida.

Posibles soluciones

#### Código:

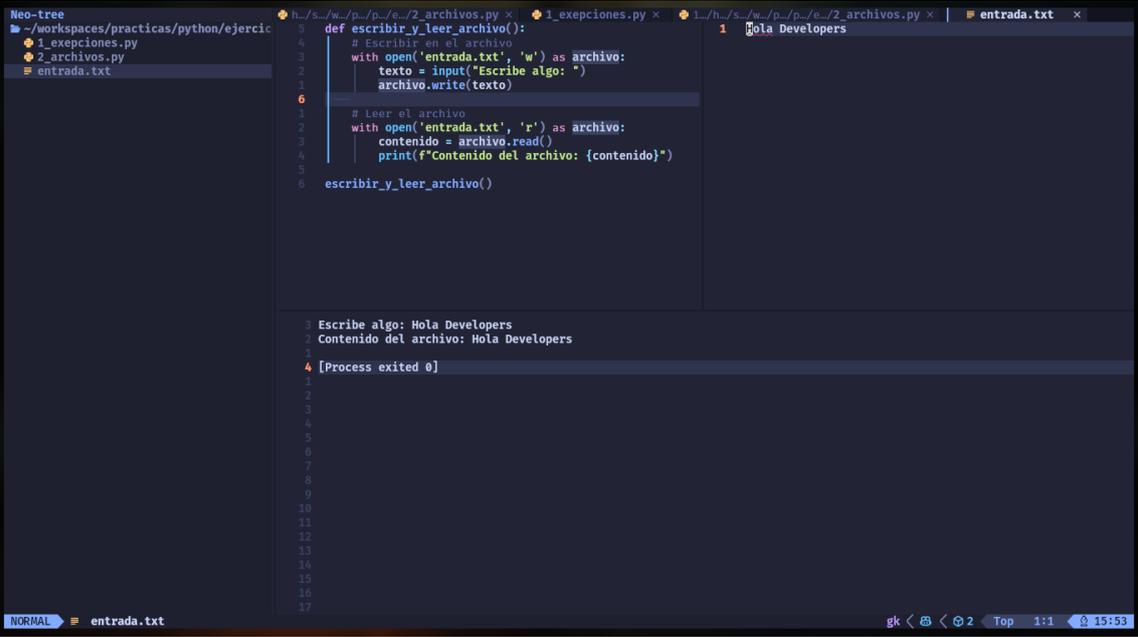
```

def pedir_numero():
    try:
        numero = int(input("Introduce un número: "))
    except ValueError:
        print(";Error! No has introducido un número válido.")
    else:
        print(f"Has introducido el número: {numero}")

```

```
finally:  
    print("Fin del programa")  
  
pedir_numero()
```

## 33 Lectura y Escritura de Archivos



```
def escribir_y_leer_archivo():
    # Escribir en el archivo
    with open('entrada.txt', 'w') as archivo:
        texto = input("Escribe algo: ")
        archivo.write(texto)

    # Leer el archivo
    with open('entrada.txt', 'r') as archivo:
        contenido = archivo.read()
        print(f"Contenido del archivo: {contenido}")

escribir_y_leer_archivo()
```

```
3 Escribe algo: Hola Developers
2 Contenido del archivo: Hola Developers
1
4 [Process exited 0]
```

Figure 33.1: Lectura y Escritura de Archivos

Leer y escribir archivos es una habilidad básica en el desarrollo de aplicaciones, como para guardar configuraciones o almacenar datos de usuarios.

### 33.0.1 Conceptos clave

- **open**: Función para abrir archivos.
- **Modos de apertura**:
  - ‘r’: Lectura.
  - ‘w’: Escritura.
  - ‘a’: Añadir datos.

**El contexto with**: Manejo automático de recursos.

Ejemplo

```
# Escritura en archivo
with open('archivo.txt', 'w') as f:
    f.write("Hola, Mundo!\n")

# Lectura de archivo
with open('archivo.txt', 'r') as f:
    contenido = f.read()
    print(contenido)
```

### 33.0.2 Archivos binarios

Podemos manejar archivos binarios usando el modo `'rb'` o `'wb'`:

```
# Lectura binaria
with open('imagen.jpg', 'rb') as f:
    datos = f.read()
```

### 33.0.3 Ejemplo Práctico

#### Objetivo:

Aprender a leer y escribir archivos de texto en Python.

**Descripción:** Crear un programa que pida al usuario un texto y lo escriba en un archivo de texto. Luego, el programa debe leer el archivo y mostrar su contenido.

#### Instrucciones:

- Pide un texto al usuario.
- Escribe ese texto en un archivo llamado `entrada.txt`.
- Luego, lee el archivo y muestra su contenido en la consola.

Posibles soluciones

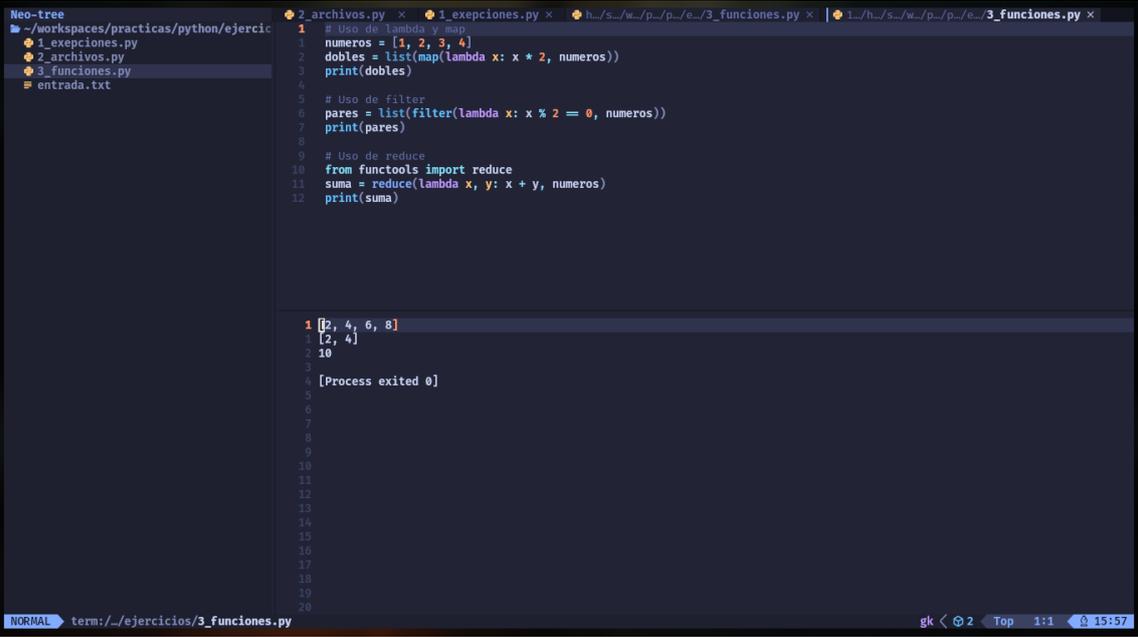
#### Código:

```
def escribir_y_leer_archivo():
    # Escribir en el archivo
    with open('entrada.txt', 'w') as archivo:
        texto = input("Escribe algo: ")
        archivo.write(texto)

    # Leer el archivo
    with open('entrada.txt', 'r') as archivo:
        contenido = archivo.read()
        print(f"Contenido del archivo: {contenido}")

escribir_y_leer_archivo()
```

## 34 Programación Funcional



```
Neo-tree
~/workspaces/practicas/python/ejercic
  1_exepciones.py
  2_archivos.py
  3_funciones.py
  entrada.txt
2_archivos.py x 1_exepciones.py x h../s../w../p../e../3_funciones.py x 1../h../s../w../p../e../3_funciones.py x
1 # Uso de lambda y map
2 numeros = [1, 2, 3, 4]
3 dobles = list(map(lambda x: x * 2, numeros))
4 print(dobles)
5
6 # Uso de filter
7 pares = list(filter(lambda x: x % 2 == 0, numeros))
8 print(pares)
9
10 # Uso de reduce
11 from functools import reduce
12 suma = reduce(lambda x, y: x + y, numeros)
13 print(suma)
14
15
16
17
18
19
20
1 [2, 4, 6, 8]
2 [2, 4]
3 10
4 [Process exited 0]
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
NORMAL term:./ejercicios/3_funciones.py gk < 2 Top 1:1 15:57
```

Figure 34.1: Programación Funcional

Python soporta parcialmente la programación funcional, lo que permite escribir código más limpio y conciso.

### 34.0.1 Conceptos clave

- **lambda**: Funciones anónimas.
- **map**: Aplica una función a cada ítem de un iterable.
- **filter**: Filtra elementos de un iterable según una condición.
- **reduce**: Reducción de un iterable a un único valor.

### 34.0.2 Comprensión de listas y generadores.

Ejemplo

```

# Uso de lambda y map
numeros = [1, 2, 3, 4]
dobles = list(map(lambda x: x * 2, numeros))
print(dobles)

# Uso de filter
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)

# Uso de reduce
from functools import reduce
suma = reduce(lambda x, y: x + y, numeros)
print(suma)

```

### 34.0.3 Ejemplo Práctico

#### Objetivo:

Aprender a utilizar funciones lambda y operaciones como map, filter y reduce para trabajar con colecciones de datos.

#### Descripción:

Crear un programa que utilice una lista de números para aplicar operaciones funcionales usando lambda, map, filter y reduce.

#### Instrucciones:

- Crea una lista de números del 1 al 10.
- Usa map con una función lambda para obtener el doble de cada número.
- Usa filter para filtrar solo los números pares.
- Usa reduce para obtener la suma de todos los números en la lista.

Posibles soluciones

#### Código:

```

from functools import reduce

# Lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Usando map con lambda
dobles = list(map(lambda x: x * 2, numeros))
print(f"Lista de dobles: {dobles}")

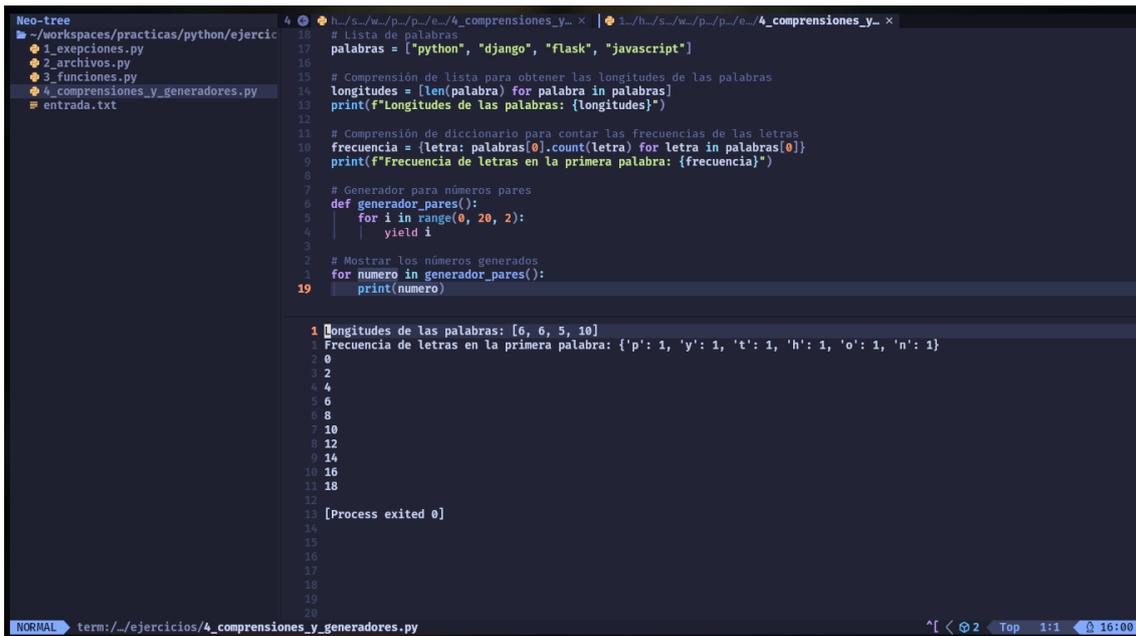
# Usando filter con lambda
pares = list(filter(lambda x: x % 2 == 0, numeros))

```

```
print(f"Números pares: {pares}")

# Usando reduce con lambda
suma = reduce(lambda x, y: x + y, numeros)
print(f"Suma de números: {suma}")
```

## 35 Comprensiones y Generadores



```
Neo-tree
~/workspaces/practicas/python/ejercic
  1_excepciones.py
  2_archivos.py
  3_funciones.py
  4_comprensiones_y_generadores.py
  entrada.txt

4  # Lista de palabras
16  palabras = ["python", "django", "flask", "javascript"]
17
18  # Comprensión de lista para obtener las longitudes de las palabras
19  longitudes = [len(palabra) for palabra in palabras]
20  print(f"Longitudes de las palabras: {longitudes}")
21
22  # Comprensión de diccionario para contar las frecuencias de las letras
23  frecuencia = {letra: palabras[0].count(letra) for letra in palabras[0]}
24  print(f"Frecuencia de letras en la primera palabra: {frecuencia}")
25
26  # Generador para números pares
27  def generador_pares():
28      for i in range(0, 20, 2):
29          yield i
30
31  # Mostrar los números generados
32  for numero in generador_pares():
33      print(numero)
34
35  1 Longitudes de las palabras: [6, 6, 5, 10]
36  2 Frecuencia de letras en la primera palabra: {'p': 1, 'y': 1, 't': 1, 'h': 1, 'o': 1, 'n': 1}
37  3 0
38  4 2
39  5 4
40  6 6
41  7 8
42  8 10
43  9 12
44  10 14
45  11 16
46  12 18
47  13 [Process exited 0]
48  14
49  15
50  16
51  17
52  18
53  19
54  20
```

Figure 35.1: Comprensiones y Generadores

Las comprensiones proporcionan una manera más compacta de crear colecciones. Los generadores permiten trabajar con grandes volúmenes de datos de manera eficiente.

### 35.0.1 Conceptos clave

- Comprensión de listas, diccionarios y conjuntos.
- Generadores y yield.

Ejemplo

```
# Comprensión de lista
cuadrados = [x**2 for x in range(5)]
print(cuadrados)

# Generador
def contador():
    for i in range(5):
        yield i
```

```
gen = contador()
for valor in gen:
    print(valor)
```

### 35.0.2 Ejemplo Práctico

#### Objetivo:

Aprender a crear listas, diccionarios y generadores utilizando comprensiones y el comando yield.

#### Descripción:

Crea un programa que use comprensiones de listas y diccionarios para realizar operaciones sobre una lista de palabras, y usa un generador para crear una secuencia de números.

#### Instrucciones:

- Usa una comprensión de lista para crear una lista de las longitudes de las palabras.
- Usa una comprensión de diccionario para contar la frecuencia de cada letra en un conjunto de palabras.
- Usa un generador para producir los primeros 10 números pares.

Posibles soluciones

#### Código:

```
# Lista de palabras
palabras = ["python", "django", "flask", "javascript"]

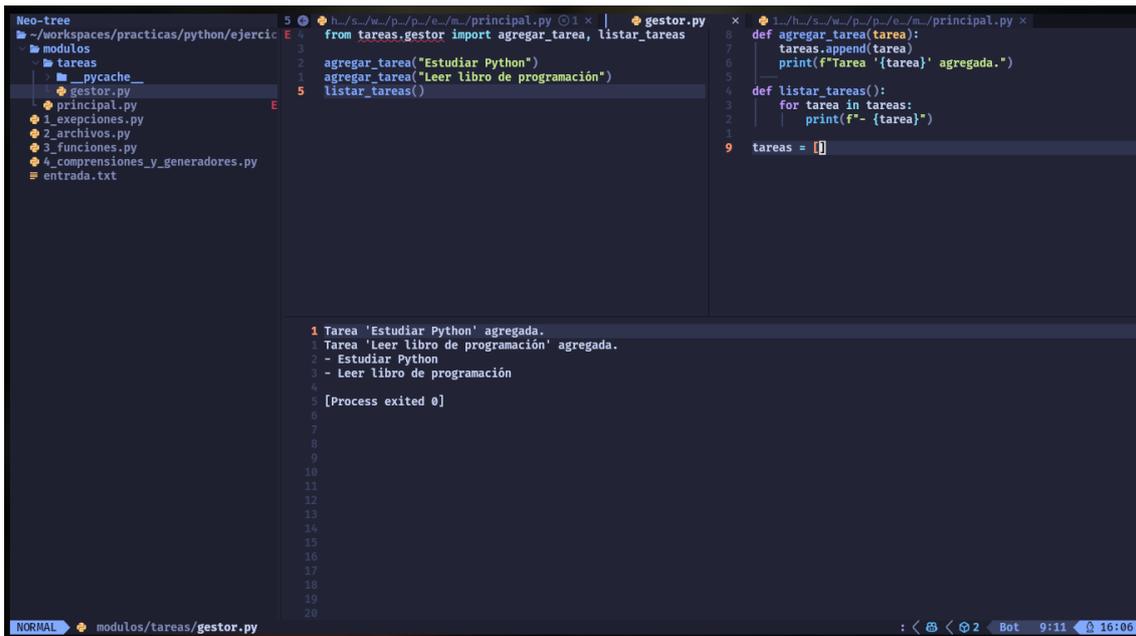
# Comprensión de lista para obtener las longitudes de las palabras
longitudes = [len(palabra) for palabra in palabras]
print(f"Longitudes de las palabras: {longitudes}")

# Comprensión de diccionario para contar las frecuencias de las letras
frecuencia = {letra: palabras[0].count(letra) for letra in palabras[0]}
print(f"Frecuencia de letras en la primera palabra: {frecuencia}")

# Generador para números pares
def generador_pares():
    for i in range(0, 20, 2):
        yield i

# Mostrar los números generados
for numero in generador_pares():
    print(numero)
```

## 36 Módulos y Paquetes Avanzados



The screenshot shows a code editor with three panes. The left pane displays a file tree for a project named 'ejercicio' located at '~/.workspaces/practicas/python/ejercicio'. The tree includes a 'modulos' directory containing 'tareas', which in turn contains 'gestor.py'. Other files in the project include 'principal.py', '1\_excepciones.py', '2\_archivos.py', '3\_funciones.py', '4\_comprensiones\_y\_generadores.py', and 'entrada.txt'. The middle pane shows the code for 'gestor.py':

```
1 from tareas.gestor import agregar_tarea, listar_tareas
2
3 agregar_tarea("Estudiar Python")
4 agregar_tarea("Leer libro de programación")
5 listar_tareas()
```

The right pane shows the code for 'principal.py':

```
1
2
3 def agregar_tarea(tarea):
4     tareas.append(tarea)
5     print(f"Tarea '{tarea}' agregada.")
6
7 def listar_tareas():
8     for tarea in tareas:
9         print(f"- {tarea}")
10
11 tareas = []
```

The bottom pane shows the output of the program:

```
1 Tarea 'Estudiar Python' agregada.
2 Tarea 'Leer libro de programación' agregada.
3 - Estudiar Python
4 - Leer libro de programación
5 [Process exited 0]
```

Figure 36.1: Módulos y Paquetes

Organizar el código en módulos y paquetes es fundamental para proyectos grandes.

### 36.0.1 Conceptos clave

- Importación relativa y absoluta.
- **init.py**: Archivo necesario para que un directorio sea reconocido como un paquete.
- Gestión de dependencias.

Ejemplo

```
# Importación absoluta
import mi_modulo

# Importación relativa
from . import mi_modulo
```

## 36.0.2 Ejemplo Práctico

### Objetivo:

Aprender a organizar el código en módulos y paquetes para proyectos más grandes.

### Descripción:

Creas un proyecto con múltiples archivos Python y organiza el código en módulos. Simula un programa de gestión de tareas.

### Instrucciones:

- Crea una carpeta llamada tareas.
- Dentro de esa carpeta, crea tres archivos:
- `init.py`: Para inicializar el paquete.
- `gestor.py`: Para gestionar tareas.
- `principal.py`: Para ejecutar el programa.

Posibles soluciones

### Código:

#### `gestor.py`:

```
def agregar_tarea(tarea):
    tareas.append(tarea)
    print(f"Tarea '{tarea}' agregada.")

def listar_tareas():
    for tarea in tareas:
        print(f"- {tarea}")

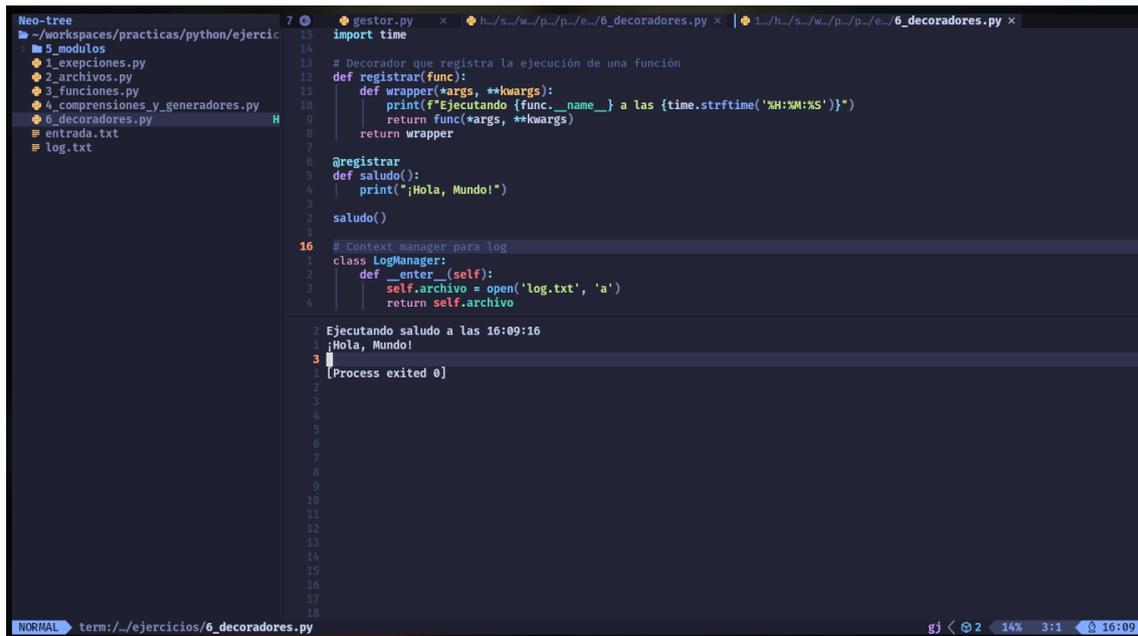
tareas = []
```

#### `principal.py`:

```
from tareas.gestor import agregar_tarea, listar_tareas

agregar_tarea("Estudiar Python")
agregar_tarea("Leer libro de programación")
listar_tareas()
```

## 37 Decoradores y Context Managers



```
Neo-tree
~/workspaces/practicas/python/ejercic
  5_modulos
  1_excepciones.py
  2_archivos.py
  3_funciones.py
  4_comprensiones_y_generadores.py
  6_decoradores.py
  entrada.txt
  log.txt

gestor.py
import time
# Decorador que registra la ejecución de una función
def registrar(func):
    def wrapper(*args, **kwargs):
        print(f"Ejecutando {func.__name__} a las {time.strftime('%H:%M:%S')}")
        return func(*args, **kwargs)
    return wrapper

@registrar
def saludo():
    print("¡Hola, Mundo!")

saludo()

# Context manager para log
class LogManager:
    def __enter__(self):
        self.archivo = open('log.txt', 'a')
        return self.archivo

Ejecutando saludo a las 16:09:16
¡Hola, Mundo!
[Process exited 0]
```

Figure 37.1: Decoradores

Los decoradores permiten modificar el comportamiento de una función, mientras que los context managers gestionan recursos como archivos o conexiones a bases de datos.

### 37.0.1 Conceptos clave

- **@decorator**: Sintaxis para aplicar un decorador.
- **with y enter, exit**: Para crear context managers.

Ejemplo

```
# Decorador
def mi_decorador(func):
    def wrapper():
        print("Antes de la función")
        func()
        print("Después de la función")
    return wrapper
```

```

@mi_decorador
def saludo():
    print("Hola")

saludo()

# Context manager
class MiContexto:
    def __enter__(self):
        print("Entrando al contexto")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Saliendo del contexto")

with MiContexto():
    print("Dentro del contexto")

```

### 37.0.2 Ejemplo Práctico

#### Objetivo:

Aprender a crear decoradores y context managers en Python.

**Descripción:** Crea un decorador que registre la ejecución de una función y un context manager que gestione un archivo de log.

#### Instrucciones:

- Crea un decorador que imprima la fecha y hora de la ejecución de una función.
- Crea un context manager que gestione la apertura y cierre de un archivo de log.

Posibles soluciones

#### Código:

```

import time

# Decorador que registra la ejecución de una función
def registrar(func):
    def wrapper(*args, **kwargs):
        print(f"Ejecutando {func.__name__} a las {time.strftime('%H:%M:%S')}")
        return func(*args, **kwargs)
    return wrapper

@registrar
def saludo():
    print(";Hola, Mundo!")

```

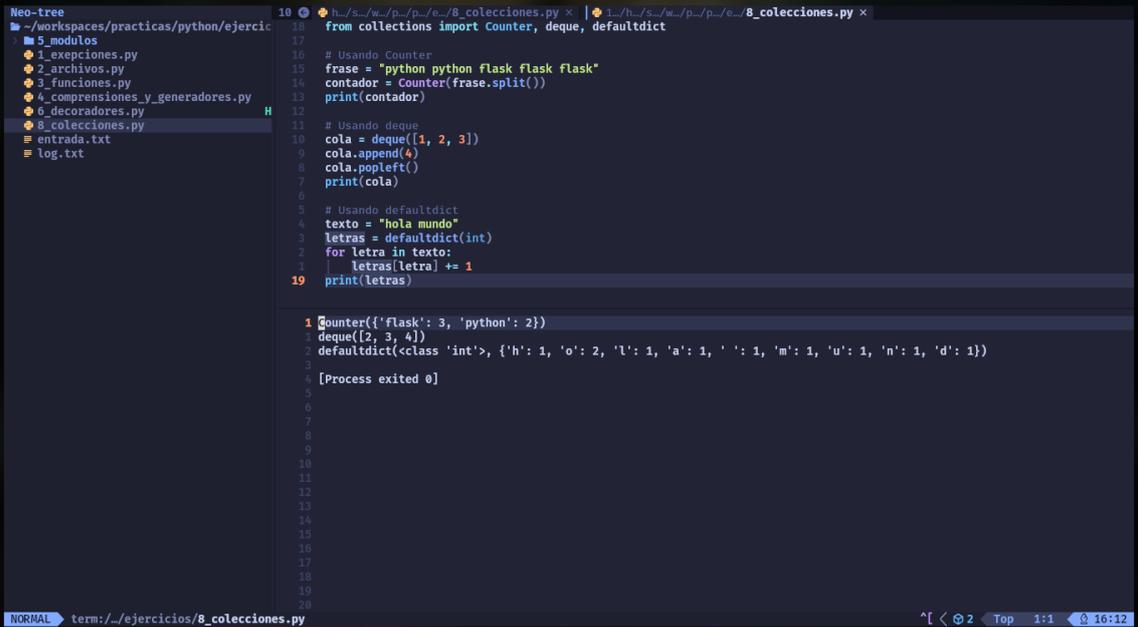
```
saludo()

# Context manager para log
class LogManager:
    def __enter__(self):
        self.archivo = open('log.txt', 'a')
        return self.archivo

    def __exit__(self, exc_type, exc_value, traceback):
        self.archivo.close()

with LogManager() as log:
    log.write(f"Acción registrada a las {time.strftime('%H:%M:%S')}\n")
```

## 38 Colecciones de Datos y Estructuras Especializadas



```
Neo-tree
~/workspaces/practicas/python/ejercic
  5_modulos
  1_excepciones.py
  2_archivos.py
  3_funciones.py
  4_comprensiones_y_generadores.py
  6_decoradores.py
  8_colecciones.py
  entrada.txt
  log.txt
10 from collections import Counter, deque, defaultdict
11
12 # Usando Counter
13 frase = "python python flask flask flask"
14 contador = Counter(frase.split())
15 print(contador)
16
17 # Usando deque
18 cola = deque([1, 2, 3])
19 cola.append(4)
20 cola.popleft()
21 print(cola)
22
23 # Usando defaultdict
24 texto = "hola mundo"
25 letras = defaultdict(int)
26 for letra in texto:
27     letras[letra] += 1
28 print(letras)
29
30 1 Counter({'flask': 3, 'python': 2})
31 2 deque([2, 3, 4])
32 3 defaultdict(<class 'int'>, {'h': 1, 'o': 2, 'l': 1, 'a': 1, ' ': 1, 'm': 1, 'u': 1, 'n': 1, 'd': 1})
33
34 4 [Process exited 0]
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Figure 38.1: Colecciones

La librería collections ofrece estructuras de datos útiles para optimizar el código.

### 38.0.1 Conceptos clave

- **Counter**: Cuenta elementos.
- **deque**: Cola de doble extremo.
- **defaultdict**: Diccionario con valores predeterminados.
- **namedtuple**: Tupla con nombre.

Ejemplo

```
from collections import Counter, deque, defaultdict, namedtuple

# Counter
c = Counter([1, 2, 2, 3])
```

```

print(c)

# deque
d = deque([1, 2, 3])
d.append(4)
print(d)

# defaultdict
dd = defaultdict(int)
dd['a'] += 1
print(dd)

# namedtuple
Persona = namedtuple('Persona', 'nombre edad')
persona = Persona(nombre='Juan', edad=30)
print(persona.nombre)

```

### 38.0.2 Ejemplo Práctico

#### Objetivo:

Aprender a utilizar estructuras de datos avanzadas como Counter, deque y defaultdict.

#### Descripción:

Crea un programa que utilice Counter para contar elementos, deque para manipular una cola y defaultdict para un diccionario con valores predeterminados.

#### Instrucciones:

- Usa Counter para contar las palabras en una frase.
- Usa deque para simular una cola.
- Usa defaultdict para contar ocurrencias de letras en un texto.

Posibles soluciones

#### Código:

```

from collections import Counter, deque, defaultdict

# Usando Counter
frase = "python python flask flask flask"
contador = Counter(frase.split())
print(contador)

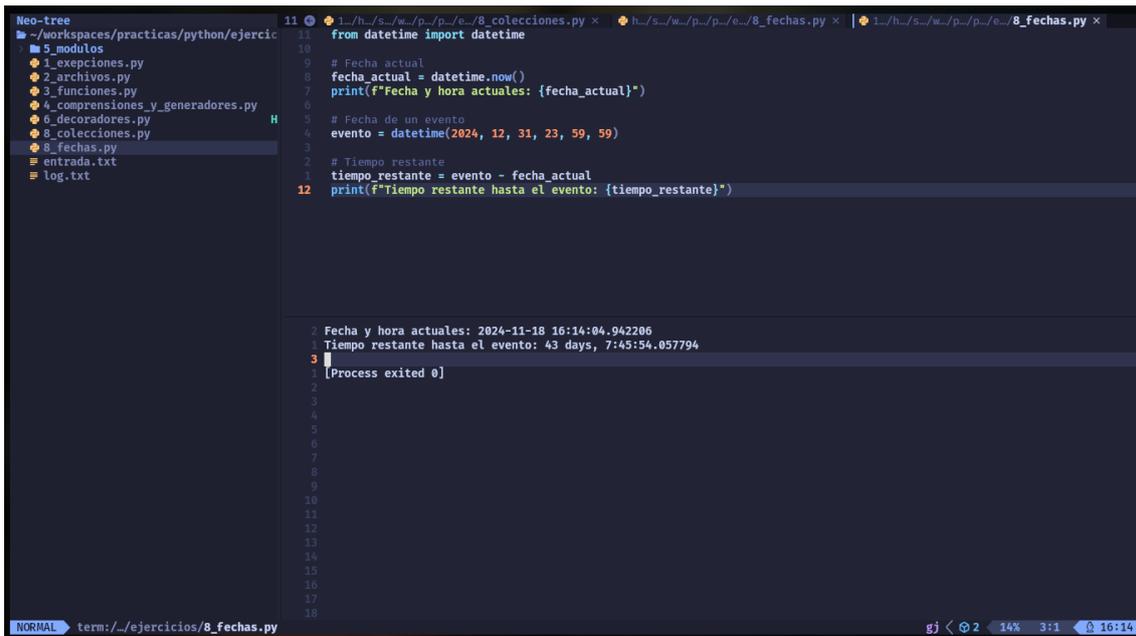
# Usando deque
cola = deque([1, 2, 3])
cola.append(4)
cola.popleft()

```

```
print(cola)

# Usando defaultdict
texto = "hola mundo"
letras = defaultdict(int)
for letra in texto:
    letras[letra] += 1
print(letras)
```

## 39 Manipulación de Fechas y Tiempos



```
Neo-tree
~/workspaces/practicas/python/ejercic
  5_modulos
  1_excepciones.py
  2_archivos.py
  3_funciones.py
  4_comprensiones_y_generadores.py
  6_decoradores.py
  8_colecciones.py
  8_fechas.py
  entrada.txt
  log.txt
11 from datetime import datetime
12
10 # Fecha actual
9 fecha_actual = datetime.now()
8 print(f"Fecha y hora actuales: {fecha_actual}")
7
6 # Fecha de un evento
5 evento = datetime(2024, 12, 31, 23, 59, 59)
4
3 # Tiempo restante
2 tiempo_restante = evento - fecha_actual
1 print(f"Tiempo restante hasta el evento: {tiempo_restante}")
12

2 Fecha y hora actuales: 2024-11-18 16:14:04.942206
1 Tiempo restante hasta el evento: 43 days, 7:45:54.057794
3
1 [Process exited 0]
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
NORMAL term: ~/ejercicios/8_fechas.py
```

Figure 39.1: Fechas

Trabajar con fechas y horas es una parte fundamental en muchas aplicaciones.

### 39.0.1 Conceptos clave

- **datetime**: Para trabajar con fechas y horas.
- **time**: Para trabajar con tiempos.
- **pytz**: Para manejar zonas horarias.

Ejemplo

```
from datetime import datetime

# Fecha y hora actuales
ahora = datetime.now()
print(ahora)

# Formateo de fecha
```

```
fecha_formateada = ahora.strftime("%Y-%m-%d %H:%M:%S")
print(fecha_formateada)
```

### 39.0.2 Ejemplo Práctico

#### Objetivo:

Aprender a trabajar con fechas y horas utilizando el módulo datetime.

**Descripción:** Crea un programa que calcule el tiempo restante hasta un evento futuro.

#### Instrucciones:

- Usa datetime para calcular la fecha y hora actuales.
- Calcula el tiempo restante hasta un evento programado (por ejemplo, fin de año).

Posibles soluciones

#### Código:

```
from datetime import datetime

# Fecha actual
fecha_actual = datetime.now()
print(f"Fecha y hora actuales: {fecha_actual}")

# Fecha de un evento
evento = datetime(2024, 12, 31, 23, 59, 59)

# Tiempo restante
tiempo_restante = evento - fecha_actual
print(f"Tiempo restante hasta el evento: {tiempo_restante}")
```

# 40 Concurrencia y Paralelismo

En aplicaciones que requieren ejecutar múltiples tareas simultáneamente, la concurrencia y el paralelismo permiten mejorar el rendimiento.

## 40.0.1 Conceptos clave

- **threading**: Hilos de ejecución.
- **multiprocessing**: Procesos independientes.
- **asyncio y async/await**: Manejo de tareas asincrónicas.

Ejemplo

```
import threading

def tarea():
    print("Tarea ejecutada por hilo")

# Crear un hilo
hilo = threading.Thread(target=tarea)
hilo.start()
hilo.join()
```

## 40.0.2 Ejemplo Práctico

### Objetivo:

Aprender a utilizar técnicas de concurrencia y paralelismo para ejecutar tareas de manera simultánea y mejorar el rendimiento de las aplicaciones.

### Descripción:

En este ejemplo se utilizan tres enfoques diferentes de concurrencia: threading, multiprocessing y asyncio. Cada uno es útil en diferentes escenarios según la naturaleza de la tarea que se quiere realizar.

### Instrucciones:

- Crea una función simple que imprima un mensaje.
- Implementa la ejecución concurrente de esa función utilizando threading, multiprocessing y asyncio.

### Ejemplos prácticos:

#### 40.0.2.1 1. Uso de threading:

El módulo threading permite ejecutar funciones de forma concurrente en múltiples hilos dentro de un solo proceso.

```
import threading

def tarea():
    print("Tarea ejecutada por hilo")

# Crear un hilo
hilo = threading.Thread(target=tarea)
hilo.start()
hilo.join() # Esperar a que termine la ejecución del hilo
print("Hilo terminado")
```

#### 40.0.2.2 2. Uso de multiprocessing:

El módulo multiprocessing permite ejecutar funciones en múltiples procesos independientes, lo que es útil para tareas que consumen mucho CPU.

```
import multiprocessing

def tarea():
    print("Tarea ejecutada por proceso")

# Crear un proceso
proceso = multiprocessing.Process(target=tarea)
proceso.start()
proceso.join() # Esperar a que termine la ejecución del proceso
print("Proceso terminado")
```

#### 40.0.2.3 3. Uso de asyncio y async/await:

El módulo asyncio permite manejar operaciones de entrada/salida asincrónicas de manera eficiente, sin bloquear el hilo principal.

```
import asyncio

async def tarea():
    print("Tarea asincrónica ejecutada")
    await asyncio.sleep(2) # Simula una tarea asincrónica con espera
    print("Tarea asincrónica terminada")

# Ejecutar tareas asincrónicas
async def main():
```

```
    await asyncio.gather(tarea(), tarea())  
asyncio.run(main()) # Ejecuta el bucle de eventos
```

# 41 Pruebas y Debugging

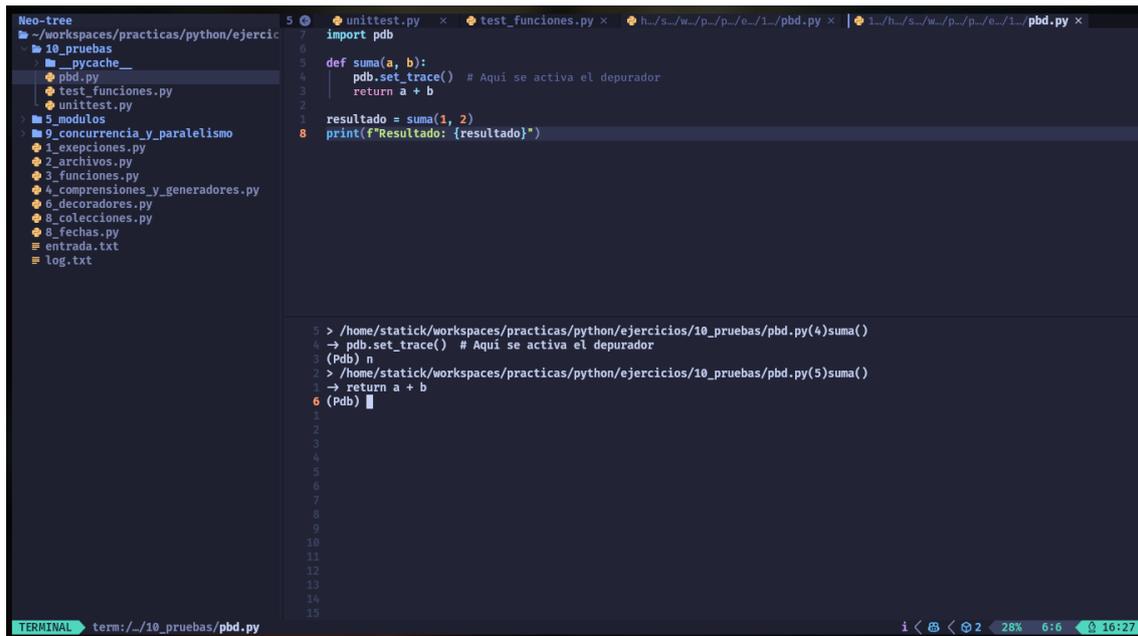


Figure 41.1: Pruebas y Debugging

Escribir pruebas y depurar el código son prácticas esenciales para garantizar la calidad y facilitar el mantenimiento.

## 41.0.1 Conceptos clave

- **unittest** y **pytest**: Frameworks para pruebas.
- **assert**: Para comprobar condiciones.
- **pdb**: Para depuración interactiva.

Ejemplo

```
# Prueba simple con unittest
import unittest

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

```

class TestSuma(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(suma(1, 2), 3)

if __name__ == '__main__':
    unittest.main()

```

## 41.0.2 Ejemplo Práctico

### Objetivo:

Aprender a escribir pruebas unitarias y utilizar herramientas de depuración para asegurar la calidad del código.

### Descripción:

En este tema se cubren pruebas unitarias con unittest, depuración con pdb y el uso de pytest para realizar pruebas automatizadas.

### Instrucciones:

- Escribe pruebas unitarias para una función que realiza una operación matemática (suma).
- Aprende a utilizar el depurador pdb para inspeccionar el flujo de ejecución.

### Ejemplos prácticos:

#### 41.0.2.1 1. Pruebas con unittest:

El módulo unittest permite crear casos de prueba, asegurando que el código funcione correctamente.

Posibles soluciones

```

import unittest

# Función simple que vamos a probar
def suma(a, b):
    return a + b

# Clase de prueba
class TestSuma(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(suma(1, 2), 3) # Verifica que la suma de 1 y 2 sea 3

if __name__ == '__main__':
    unittest.main() # Ejecuta las pruebas

```

### 41.0.2.2 2. Pruebas con pytest:

pytest es una alternativa moderna y más sencilla para realizar pruebas. Aquí utilizamos el mismo ejemplo de la función suma.

Posibles soluciones

```
# Guarda esto en un archivo llamado test_funciones.py

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

def test_suma():
    assert suma(1, 2) == 3 # Verifica que la suma de 1 y 2 sea 3
```

Ejecuta las pruebas con el comando:

```
pytest test_funciones.py
```

### 41.0.2.3 3. Depuración con pdb:

El depurador pdb permite interactuar con el código paso a paso, inspectando variables y el flujo de ejecución.

Posibles soluciones

```
import pdb

def suma(a, b):
    pdb.set_trace() # Aquí se activa el depurador
    return a + b

resultado = suma(1, 2)
print(f"Resultado: {resultado}")
```

Cuando ejecutes el programa, el depurador se activará en `pdb.set_trace()`. Desde ahí, podrás usar comandos como `n` para avanzar a la siguiente línea o `p` para imprimir el valor de una variable.

**Comandos útiles de pdb:**

- **n:** Ejecuta la siguiente línea de código.
- **p variable:** Muestra el valor de una variable.
- **q:** Sale del depurador

## **Part VII**

# **Unidad 6: Bases de Datos**

## 42 Introducción a Bases de Datos



Figure 42.1: Bases de Datos

Las bases de datos son un componente esencial para el desarrollo de software, ya que permiten el almacenamiento, gestión y consulta de información estructurada. En este capítulo, exploraremos los fundamentos y las operaciones básicas de bases de datos en diferentes tecnologías.

### 42.1 1. Fundamentos de Bases de Datos

Las bases de datos son sistemas organizados para almacenar información, permitiendo consultas eficientes, actualizaciones seguras y una administración centralizada de los datos. Son esenciales para casi todas las aplicaciones modernas, desde sistemas empresariales hasta redes sociales.

#### 42.1.1 Conceptos Clave

- **Modelo de datos:** Estructura para definir cómo se almacenarán los datos (relacional, no relacional).
- **Consultas:** Lenguaje para interactuar con los datos (SQL para bases relacionales, JSON para bases no relacionales).
- **ACID:** Propiedades fundamentales para garantizar consistencia en transacciones.
- **Normalización:** Proceso de organización para reducir redundancia y mejorar integridad.

### 42.1.1.1 Ejemplos

**Ejemplo 1:** Estructura básica de una base de datos relacional

```
CREATE TABLE usuarios (  
  id INT PRIMARY KEY,  
  nombre VARCHAR(50),  
  email VARCHAR(100)  
);
```

En el ejemplo anterior, se crea una tabla llamada usuarios con tres columnas: id, nombre y email.

**Ejemplo 2:** Consulta básica

```
SELECT * FROM usuarios WHERE email LIKE '%\@gmail.com';
```

En el ejemplo anterior, se seleccionan todos los usuarios cuyo email contiene "@gmail.com".

**Ejemplo 3:** Transacción básica

```
BEGIN TRANSACTION;  
INSERT INTO usuarios (id, nombre, email) VALUES (1, 'Ana', 'ana\@gmail.com');  
DELETE FROM usuarios WHERE id = 1;  
ROLLBACK;
```

En el ejemplo anterior, se inicia una transacción, se inserta un usuario, se elimina y se revierte la operación.

**Ejemplo 4:** Uso de índices para mejorar consultas

```
CREATE INDEX idx_email ON usuarios(email);  
SELECT * FROM usuarios WHERE email = 'ana@gmail.com';
```

En el ejemplo anterior, se crea un índice en la columna email para acelerar la búsqueda de usuarios por email.

## 42.2 Ejemplo Práctico

**Objetivo:** Crear una base de datos relacional básica y ejecutar una consulta de ejemplo.

**Descripción:** En este ejercicio, se creará una base de datos SQLite para almacenar información de usuarios y se ejecutarán operaciones básicas como inserciones y consultas.

### 42.2.1 Instrucciones:

1. Crea un archivo Python que conecte a una base de datos SQLite.
2. Crea una tabla llamada usuarios.
3. Inserta tres registros.
4. Recupera y muestra los datos.

Posibles soluciones

#### Código:

```
import sqlite3

# Conexión a la base de datos SQLite
conexion = sqlite3.connect("mi_base_datos.db")
cursor = conexion.cursor()

# Crear tabla
cursor.execute("""
CREATE TABLE IF NOT EXISTS usuarios (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    email TEXT NOT NULL
)
""")

# Insertar registros
usuarios = [
    (1, 'Ana', 'ana@gmail.com'),
    (2, 'Carlos', 'carlos@gmail.com'),
    (3, 'Luis', 'luis@gmail.com')
]
cursor.executemany("INSERT INTO usuarios VALUES (?, ?, ?)", usuarios)
conexion.commit()

# Recuperar datos
cursor.execute("SELECT * FROM usuarios")
for fila in cursor.fetchall():
    print(fila)

# Cerrar conexión
conexion.close()
```

## 43 Conclusiones

Las bases de datos son una parte fundamental de la infraestructura tecnológica actual, permitiendo el almacenamiento y gestión eficiente de grandes volúmenes de información. Conocer los conceptos básicos y las operaciones comunes en bases de datos es esencial para cualquier desarrollador de software.

## 44 Bases de Datos con SQLite3



Figure 44.1: SQLite3

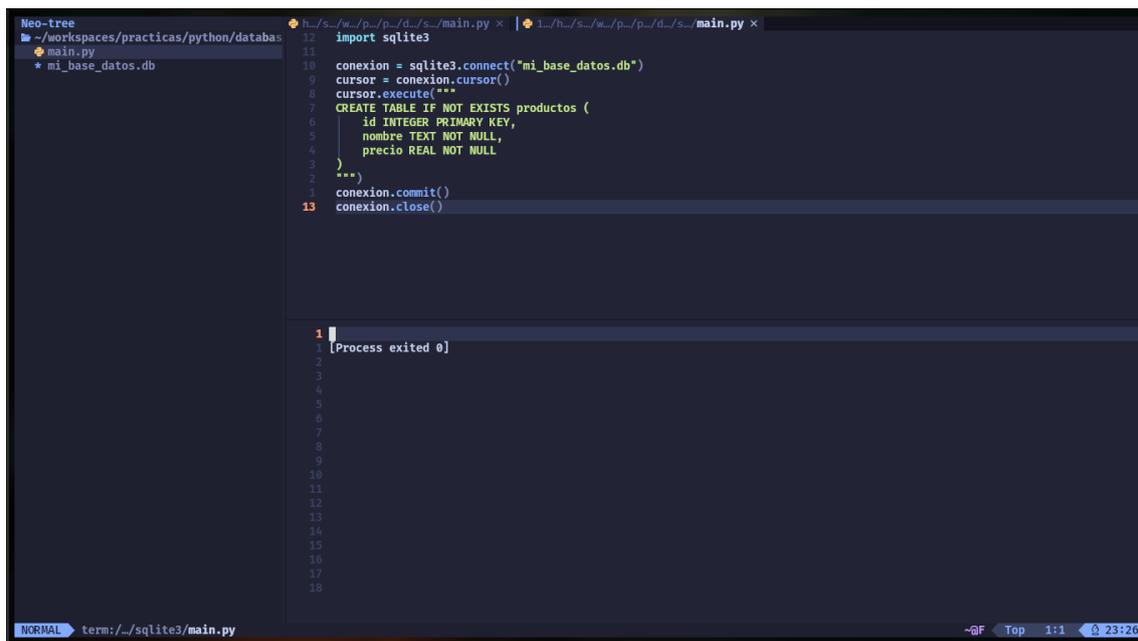
SQLite3 es una base de datos ligera, fácil de configurar y embebida en aplicaciones. Es ideal para prototipos, aplicaciones pequeñas y herramientas locales que no requieren un servidor independiente.

### 44.1 Conceptos Clave

- **Ligereza:** No requiere configuración de servidor, los datos se almacenan en un archivo local.
- **SQL estándar:** Utiliza el lenguaje SQL para consultas y administración.
- **Portabilidad:** Las bases de datos se almacenan en archivos que pueden moverse fácilmente entre sistemas.
- **Uso común:** Perfecta para entornos de prueba o proyectos con requisitos mínimos.

### 44.2 Ejemplos

**Ejemplo 1:** Crear una base de datos y una tabla



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with 'main.py' and 'mi\_base\_datos.db'. The code editor shows the following Python code:

```
12 import sqlite3
11
10 conexion = sqlite3.connect("mi_base_datos.db")
9 cursor = conexion.cursor()
8 cursor.execute("""
7 CREATE TABLE IF NOT EXISTS productos (
6     id INTEGER PRIMARY KEY,
5     nombre TEXT NOT NULL,
4     precio REAL NOT NULL
3 )
2 """)
1 conexion.commit()
13 conexion.close()
```

Below the code editor, there is a terminal window showing the output: "[Process exited 0]".

Figure 44.2: SQLite3 Crear Base de Datos

```
import sqlite3

conexion = sqlite3.connect("mi_base_datos.db")
cursor = conexion.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    precio REAL NOT NULL
)
""")
conexion.commit()
conexion.close()
```

En el ejemplo anterior, se crea una base de datos llamada `mi_base_datos.db` con una tabla `productos` que contiene columnas para `id`, `nombre` y `precio`.

**Ejemplo 2:** Insertar datos

```

10 import sqlite3
11
12 conexion = sqlite3.connect("mi_base_datos.db")
13 cursor = conexion.cursor()
14 cursor.execute("""
15 CREATE TABLE IF NOT EXISTS productos (
16     id INTEGER PRIMARY KEY,
17     nombre TEXT NOT NULL,
18     precio REAL NOT NULL
19 )
20 """)
21 conexion.commit()
22 conexion.close()
23
24 conexion = sqlite3.connect("mi_base_datos.db")
25 cursor = conexion.cursor()
26 cursor.execute("INSERT INTO productos (nombre, precio) VALUES ('Laptop', 1200.50)")
27 conexion.commit()
28 conexion.close()

```

[Process exited 0]

Figure 44.3: SQLite3 Insertar Datos

```

conexion = sqlite3.connect("mi_base_datos.db")
cursor = conexion.cursor()
cursor.execute("INSERT INTO productos (nombre, precio) VALUES ('Laptop', 1200.50)")
conexion.commit()
conexion.close()

```

En este caso, se inserta un nuevo producto en la tabla productos con nombre “Laptop” y precio 1200.50.

### Ejemplo 3: Leer datos

```

10 import sqlite3
11
12 conexion = sqlite3.connect("mi_base_datos.db")
13 cursor = conexion.cursor()
14 cursor.execute("""
15 CREATE TABLE IF NOT EXISTS productos (
16     id INTEGER PRIMARY KEY,
17     nombre TEXT NOT NULL,
18     precio REAL NOT NULL
19 )
20 """)
21 conexion.commit()
22 conexion.close()
23
24 conexion = sqlite3.connect("mi_base_datos.db")
25 cursor = conexion.cursor()
26 cursor.execute("INSERT INTO productos (nombre, precio) VALUES ('Laptop', 1200.50)")
27 conexion.commit()
28 conexion.close()
29
30 conexion = sqlite3.connect("mi_base_datos.db")
31 cursor = conexion.cursor()
32 cursor.execute("SELECT * FROM productos")
33 datos = cursor.fetchall()
34 print(datos)
35
36 conexion.close()

```

[[('Laptop', 1200.5), ('Laptop', 1200.5)]]

[Process exited 0]

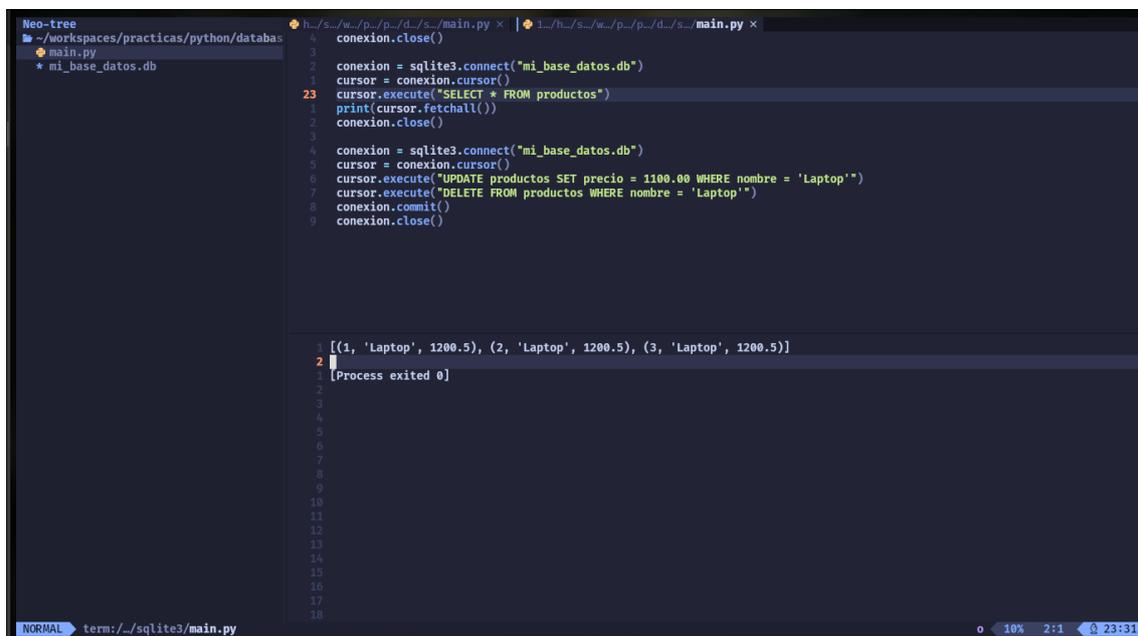
{fig-

```
align="center" width="400}
```

```
conexion = sqlite3.connect("mi_base_datos.db")
cursor = conexion.cursor()
cursor.execute("SELECT * FROM productos")
print(cursor.fetchall())
conexion.close()
```

La consulta `SELECT * FROM productos` recupera todos los registros de la tabla `productos` y los imprime en pantalla.

#### Ejemplo 4: Actualizar y eliminar datos



```
Neo-tree
~/workspaces/practicas/python/databases
├── main.py
└── mi_base_datos.db

h:/s../w../p../d../s../main.py x | h:/s../w../p../d../s../main.py x
4  conexion.close()
3
2  conexion = sqlite3.connect("mi_base_datos.db")
1  cursor = conexion.cursor()
23 cursor.execute("SELECT * FROM productos")
1  print(cursor.fetchall())
2  conexion.close()
3
4  conexion = sqlite3.connect("mi_base_datos.db")
5  cursor = conexion.cursor()
6  cursor.execute("UPDATE productos SET precio = 1100.00 WHERE nombre = 'Laptop'")
7  cursor.execute("DELETE FROM productos WHERE nombre = 'Laptop'")
8  conexion.commit()
9  conexion.close()

1  [(1, 'Laptop', 1200.5), (2, 'Laptop', 1200.5), (3, 'Laptop', 1200.5)]
2
1  [Process exited 0]
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

NORMAL term:~/sqlite3/main.py o 10% 2:1 23:31
```

Figure 44.4: SQLite3 Actualizar y Eliminar Datos

```
conexion = sqlite3.connect("mi_base_datos.db")
cursor = conexion.cursor()
cursor.execute("UPDATE productos SET precio = 1100.00 WHERE nombre = 'Laptop'")
cursor.execute("DELETE FROM productos WHERE nombre = 'Laptop'")
conexion.commit()
conexion.close()
```

### 44.3 Ejemplo Práctico

**Objetivo:** Crear, administrar y consultar una base de datos de “productos”.

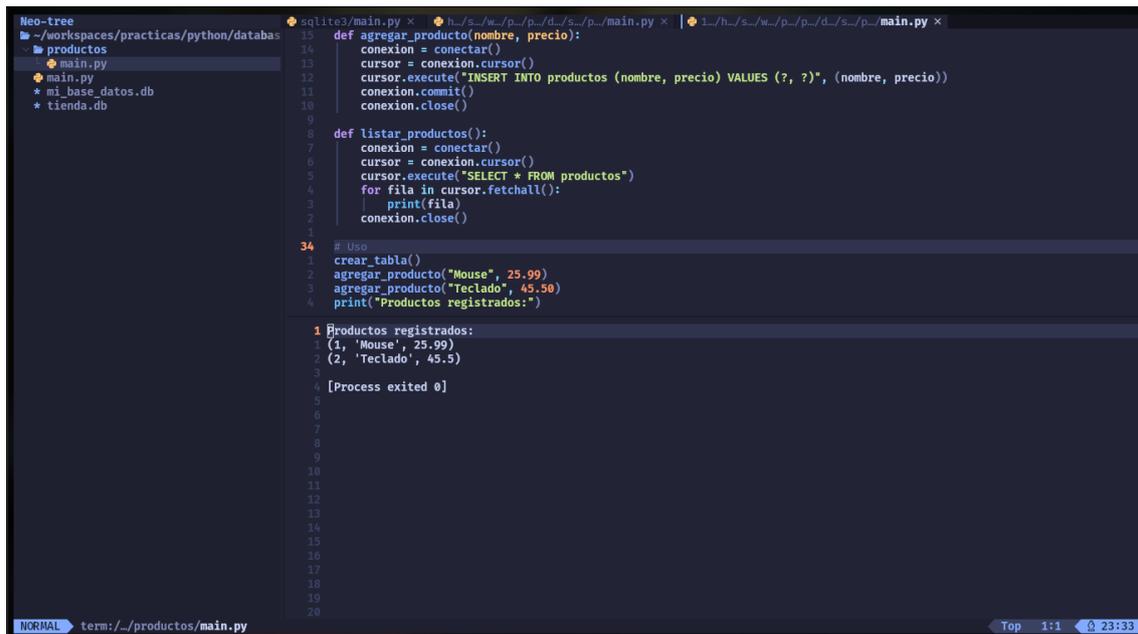
**Descripción:** Se implementará un pequeño sistema que permite agregar, listar y buscar productos utilizando SQLite3.

### 44.3.1 Instrucciones:

1. Crea una base de datos llamada tienda.db.
2. Define una tabla productos con columnas id, nombre y precio.
3. Permite agregar y listar productos desde un script Python.

Posible solución

Código:



```
Neo-tree
~/workspaces/practicas/python/databases
├── productos
│   ├── main.py
│   ├── mi_base_datos.db
│   └── tienda.db
└── main.py

sqlite3/main.py x | h.../s.../w.../p.../d.../s.../p.../main.py x | h.../s.../w.../p.../d.../s.../p.../main.py x
15 def agregar_producto(nombre, precio):
16     conexion = conectar()
17     cursor = conexion.cursor()
18     cursor.execute("INSERT INTO productos (nombre, precio) VALUES (?, ?)", (nombre, precio))
19     conexion.commit()
20     conexion.close()
9
8 def listar_productos():
7     conexion = conectar()
6     cursor = conexion.cursor()
5     cursor.execute("SELECT * FROM productos")
4     for fila in cursor.fetchall():
3         print(fila)
2     conexion.close()
1
34 # Uso
1 crear_tabla()
2 agregar_producto("Mouse", 25.99)
3 agregar_producto("Teclado", 45.50)
4 print("Productos registrados:")
1 Productos registrados:
2 (1, 'Mouse', 25.99)
3 (2, 'Teclado', 45.5)
4 [Process exited 0]
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
NORMAL term:~/productos/main.py Top 1:1 23:33
```

Figure 44.5: SQLite3 Ejemplo Práctico

```
import sqlite3

def conectar():
    return sqlite3.connect("tienda.db")

def crear_tabla():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    precio REAL NOT NULL
)
""")
    conexion.commit()
    conexion.close()
```

```

def agregar_producto(nombre, precio):
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO productos (nombre, precio) VALUES (?, ?)", (nombre, precio))
    conexion.commit()
    conexion.close()

def listar_productos():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM productos")
    for fila in cursor.fetchall():
        print(fila)
    conexion.close()

# Uso
crear_tabla()
agregar_producto("Mouse", 25.99)
agregar_producto("Teclado", 45.50)
print("Productos registrados:")
listar_productos()

```

## 45 Conclusiones

SQLite3 es una excelente opción para proyectos pequeños y prototipos que requieren una base de datos local. Su facilidad de uso y portabilidad lo convierten en una herramienta versátil para el desarrollo de aplicaciones.

## 46 Bases de Datos en MySQL



Figure 46.1: MySQL

MySQL es uno de los sistemas de gestión de bases de datos relacionales más populares. Es ampliamente utilizado en aplicaciones web y empresariales debido a su estabilidad, rendimiento y soporte para múltiples usuarios y transacciones complejas.

### 46.1 Conceptos Clave

**Relacional:** MySQL organiza los datos en tablas que se relacionan entre sí.

**Escalabilidad:** Adecuado para aplicaciones pequeñas y grandes.

**Transacciones:** Admite transacciones para garantizar la integridad de los datos.

**SQL estándar:** Usa SQL para definir, consultar y manipular datos.

**Comunidad activa:** Gran cantidad de documentación y soporte.

### 46.2 Configuración de MySQL con Docker

#### 46.2.1 Instrucciones

**Crear un contenedor de MySQL con Docker:**

Ejecuta el siguiente comando para iniciar un servidor MySQL en Docker.

```
docker run --name mysql-database -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=tienda -p
```

### 46.2.2 Parámetros:

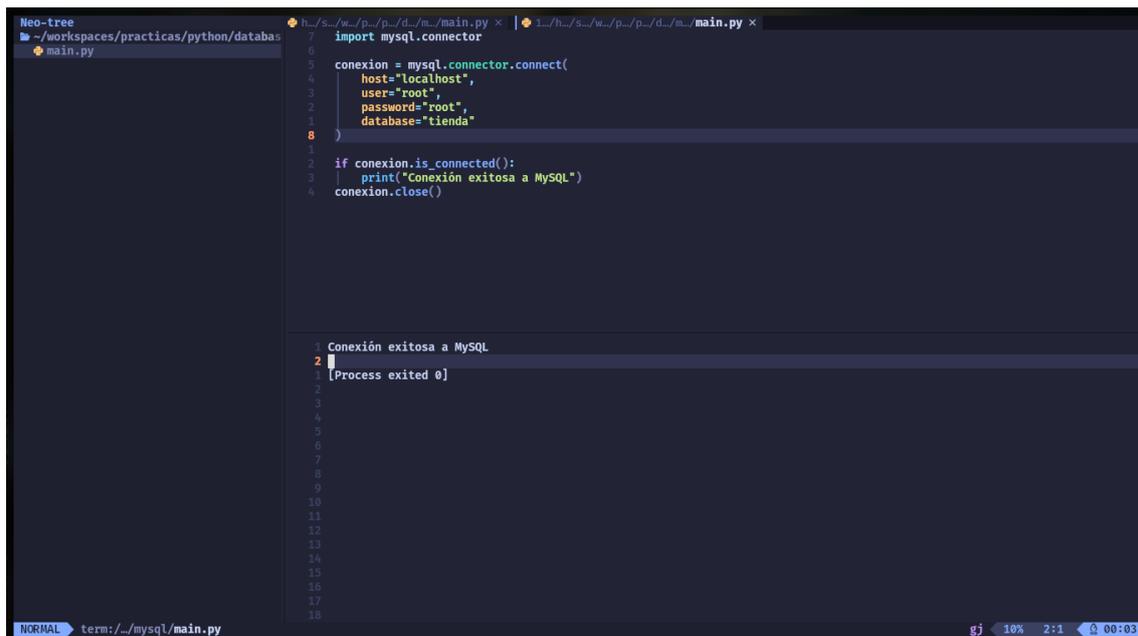
**-name:** Nombre del contenedor. **MYSQL\_ROOT\_PASSWORD:** Contraseña para el usuario root. **MYSQL\_DATABASE:** Nombre de la base de datos que se creará al iniciar. **-p 3306:3306:** Mapea el puerto del contenedor al puerto local. **mysql:8.0:** Imagen oficial de MySQL.

**Acceder al contenedor (opcional):**

```
docker exec -it mysql-database mysql -uroot -proot
```

## 46.3 Ejemplos

**Ejemplo 1:** Conexión a la base de datos desde Python



The screenshot shows a code editor with a Python script for connecting to a MySQL database. The script is as follows:

```
import mysql.connector

conexion = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="tienda"
)

if conexion.is_connected():
    print("Conexión exitosa a MySQL")
conexion.close()
```

The output of the script execution is shown in the terminal window below the code editor:

```
1 Conexión exitosa a MySQL
2
1 [Process exited 0]
```

Figure 46.2: MySQL Conexión

```
import mysql.connector

conexion = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="tienda"
)
```

```

if conexion.is_connected():
    print("Conexión exitosa a MySQL")
conexion.close()

```

En el ejemplo anterior, se establece una conexión a la base de datos MySQL llamada tienda con el usuario root y la contraseña root.

### Ejemplo 2: Crear una tabla

The screenshot shows a code editor with a file explorer on the left and a terminal at the bottom. The code in the editor is as follows:

```

4
3  conexion = mysql.connector.connect(
2      host="localhost",
1      user="root",
17     password="root",
1      database="tienda"
2  )
3  cursor = conexion.cursor()
4  cursor.execute("""
5  CREATE TABLE IF NOT EXISTS productos (
6      id INT AUTO_INCREMENT PRIMARY KEY,
7      nombre VARCHAR(255) NOT NULL,
8      precio DECIMAL(10, 2) NOT NULL
9  )
10 """)
11 conexion.commit()
12 conexion.close()

```

The terminal output shows:

```

1
1 [Process exited 0]
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

```

At the bottom left of the terminal, it says "NORMAL term: /.../mysql/main.py".

Figure 46.3: MySQL Crear Tabla

```

conexion = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="tienda"
)
cursor = conexion.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,

```

```

        precio DECIMAL(10, 2) NOT NULL
    )
    """
    conexion.commit()
    conexion.close()

```

En este caso, se crea una tabla productos con columnas para id, nombre y precio.

### Ejemplo 3: Insertar datos

```

Neo-tree
└─ ~/workspaces/practicas/python/databas
   └─ main.py
h.../S.../W.../P.../D.../M.../main.py × | 1.../h.../S.../W.../P.../D.../M.../main.py ×
4
3  conexion = mysql.connector.connect(
2      host="localhost",
1      user="root",
17     password="root",
1      database="tienda"
2  )
3  cursor = conexion.cursor()
4  cursor.execute("""
5  CREATE TABLE IF NOT EXISTS productos (
6      id INT AUTO_INCREMENT PRIMARY KEY,
7      nombre VARCHAR(255) NOT NULL,
8      precio DECIMAL(10, 2) NOT NULL
9  )
10 """)
11 conexion.commit()
12 conexion.close()

1
1 [Process exited 0]
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
NORMAL term:~/mysql/main.py

```

Figure 46.4: MySQL Insertar Datos

```

conexion = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="tienda"
)
cursor = conexion.cursor()
cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", ("Laptop", 1299.
conexion.commit()
conexion.close()

```

En este caso, se inserta un nuevo producto en la tabla productos con nombre "Laptop" y precio 1299.99.

#### Ejemplo 4: Consultar datos

```
Neo-tree
~/workspaces/practicas/python/databas
main.py

h.../S.../w.../p.../d.../m.../main.py x | 1.../h.../S.../w.../p.../d.../m.../main.py x
4  conexion = mysql.connector.connect(
3     host="localhost",
2     user="root",
1     password="root",
45    database="tienda"
1  )
2  cursor = conexion.cursor()
3  cursor.execute("SELECT * FROM productos")
4  for fila in cursor.fetchall():
5     print(fila)
6  conexion.close()

1  [1, 'Laptop', Decimal('1299.99')]
2  [Process exited 0]
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

NORMAL term:~/mysql/main.py
```

Figure 46.5: MySQL Consultar Datos

```
conexion = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="tienda"
)
cursor = conexion.cursor()
cursor.execute("SELECT * FROM productos")
for fila in cursor.fetchall():
    print(fila)
conexion.close()
```

La consulta `SELECT * FROM productos` recupera todos los registros de la tabla productos y los imprime en pantalla.

## 46.4 Ejemplo Práctico

**Objetivo:** Crear, administrar y consultar productos en una base de datos MySQL usando Python.

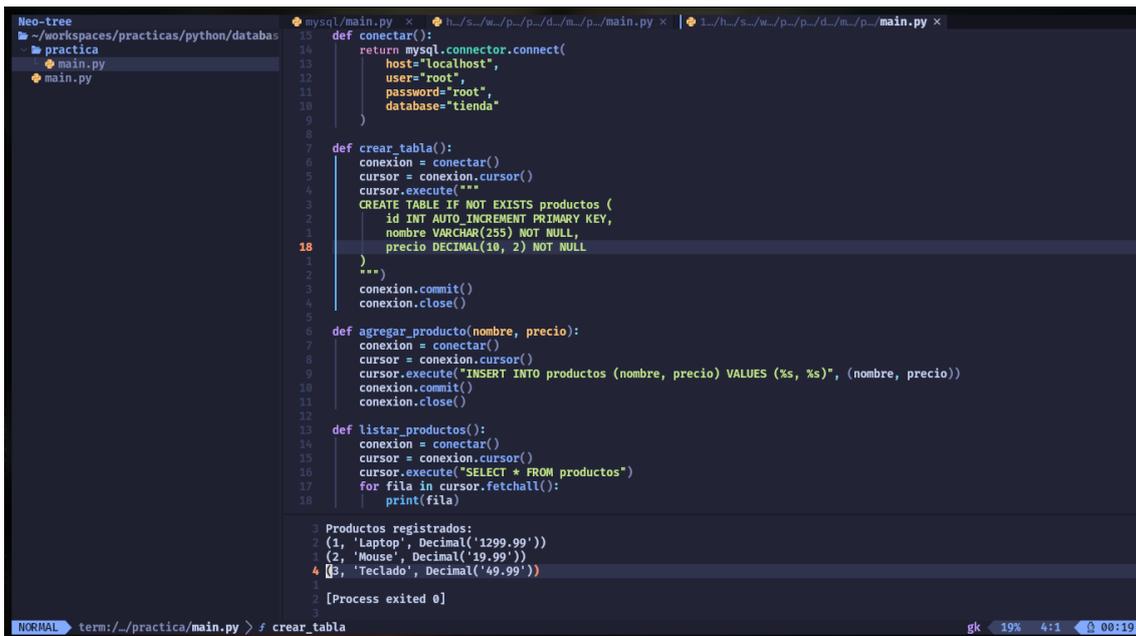
**Descripción:** Se implementará un sistema que permite agregar, listar, y buscar productos. La base de datos será configurada mediante Docker.

## 46.5 Instrucciones:

1. Configura un contenedor de MySQL usando Docker.
2. Crea una tabla productos en la base de datos tienda.
3. Implementa funciones para administrar los productos desde Python.

Posible solución

Código:



```
def conectar():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        database="tienda"
    )

def crear_tabla():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    precio DECIMAL(10, 2) NOT NULL
)
""")
    conexion.commit()
    conexion.close()

def agregar_producto(nombre, precio):
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", (nombre, precio))
    conexion.commit()
    conexion.close()

def listar_productos():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM productos")
    for fila in cursor.fetchall():
        print(fila)

3 Productos registrados:
2 (1, 'Laptop', Decimal('1299.99'))
1 (2, 'Mouse', Decimal('19.99'))
4 [(3, 'Teclado', Decimal('49.99'))]
1
2 [Process exited 0]
3
```

Figure 46.6: MySQL Ejemplo Práctico

```
import mysql.connector

def conectar():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        database="tienda"
    )
```

```

def crear_tabla():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    precio DECIMAL(10, 2) NOT NULL
)
""")
    conexion.commit()
    conexion.close()

def agregar_producto(nombre, precio):
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", (nombre, precio))
    conexion.commit()
    conexion.close()

def listar_productos():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM productos")
    for fila in cursor.fetchall():
        print(fila)
    conexion.close()

# Uso
crear_tabla()
agregar_producto("Mouse", 19.99)
agregar_producto("Teclado", 49.99)
print("Productos registrados:")
listar_productos()

```

## 47 Conclusiones

1. MySQL es una base de datos relacional popular con soporte para múltiples usuarios y transacciones.
2. Docker facilita la configuración de entornos de desarrollo con contenedores aislados.
3. Python se puede utilizar para interactuar con bases de datos MySQL mediante el conector **mysql-connector-python**.

## 48 Bases de Datos en PostgreSQL

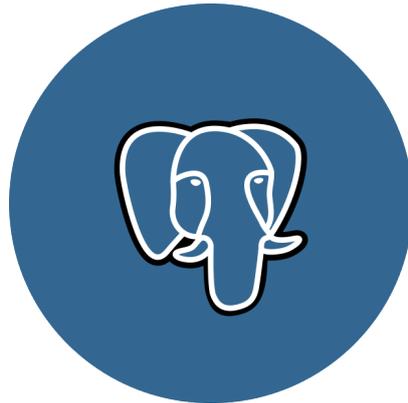


Figure 48.1: PostgreSQL Database

PostgreSQL es una de las bases de datos relacionales más avanzadas y robustas, conocida por su capacidad de manejo de datos complejos y cumplimiento estricto de los estándares SQL. Es ideal para proyectos que requieren transacciones complejas, extensibilidad y consistencia.

### 48.1 Conceptos Clave

**ACID:** Garantiza la confiabilidad de las transacciones.

**Extensibilidad:** Admite tipos de datos personalizados y funciones definidas por el usuario.

**Consultas avanzadas:** Optimiza las consultas complejas.

**Open Source:** Altamente personalizable y gratuito.

**Integridad:** Gestión avanzada de claves foráneas y restricciones.

### 48.2 Configuración de PostgreSQL con Docker

#### 48.2.1 Instrucciones

**Crear un contenedor de PostgreSQL con Docker:**

Ejecuta el siguiente comando para iniciar un servidor PostgreSQL en Docker.

```
docker run --name postgres-database -e POSTGRES_PASSWORD=root -e POSTGRES_DB=tienda -p 54
```

### 48.2.2 Parámetros:

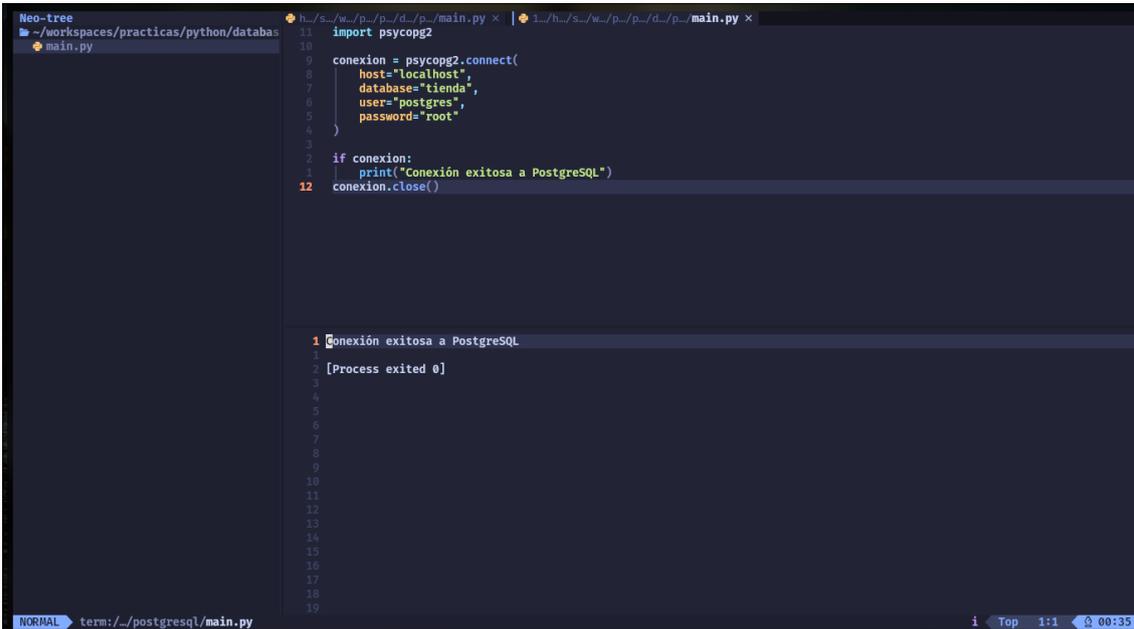
- `--name`: Nombre del contenedor.
- `POSTGRES_PASSWORDx`: Contraseña para el usuario postgres.
- `POSTGRES_DB`: Nombre de la base de datos inicial.
- `-p 5432:5432`: Mapea el puerto del contenedor al puerto local.
- `postgres:15`: Imagen oficial de PostgreSQL.

### 48.2.3 Acceder al contenedor (opcional):

```
docker exec -it postgres-database psql -U postgres
```

## 48.3 Ejemplos

### Ejemplo 1: Conexión a PostgreSQL desde Python



```
Neo-tree
~/workspaces/practicasy/python/databases
├── main.py
└── main.py

11 import psycopg2
12
13 conexion = psycopg2.connect(
14     host="localhost",
15     database="tienda",
16     user="postgres",
17     password="root"
18 )
19
20 if conexion:
21     print("Conexión exitosa a PostgreSQL")
22
23 conexion.close()

1 Conexión exitosa a PostgreSQL
2 [Process exited 0]
```

Figure 48.2: PostgreSQL Conexión

```
import psycopg2

conexion = psycopg2.connect(
    host="localhost",
    database="tienda",
```

```

        user="postgres",
        password="root"
    )

    if conexion:
        print("Conexión exitosa a PostgreSQL")
    conexion.close()

```

En el ejemplo anterior, se establece una conexión a la base de datos PostgreSQL llamada tienda con el usuario postgres y la contraseña root.

### Ejemplo 2: Crear una tabla

The screenshot shows a Neo-tree IDE window with a Python script named 'main.py'. The script connects to a PostgreSQL database on localhost using the 'postgres' user and 'root' password. It then creates a table named 'productos' with the following schema: 'id SERIAL PRIMARY KEY', 'nombre VARCHAR(255) NOT NULL', and 'precio NUMERIC(10, 2) NOT NULL'. The script commits the transaction and closes the connection. The terminal output shows '[Process exited 0]'.

```

conexion = psycopg2.connect(
    host="localhost",
    database="tienda",
    user="postgres",
    password="root"
)

cursor = conexion.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    precio NUMERIC(10, 2) NOT NULL
)
""")
conexion.commit()
conexion.close()

```

{fig-

align="center" width="800}

```

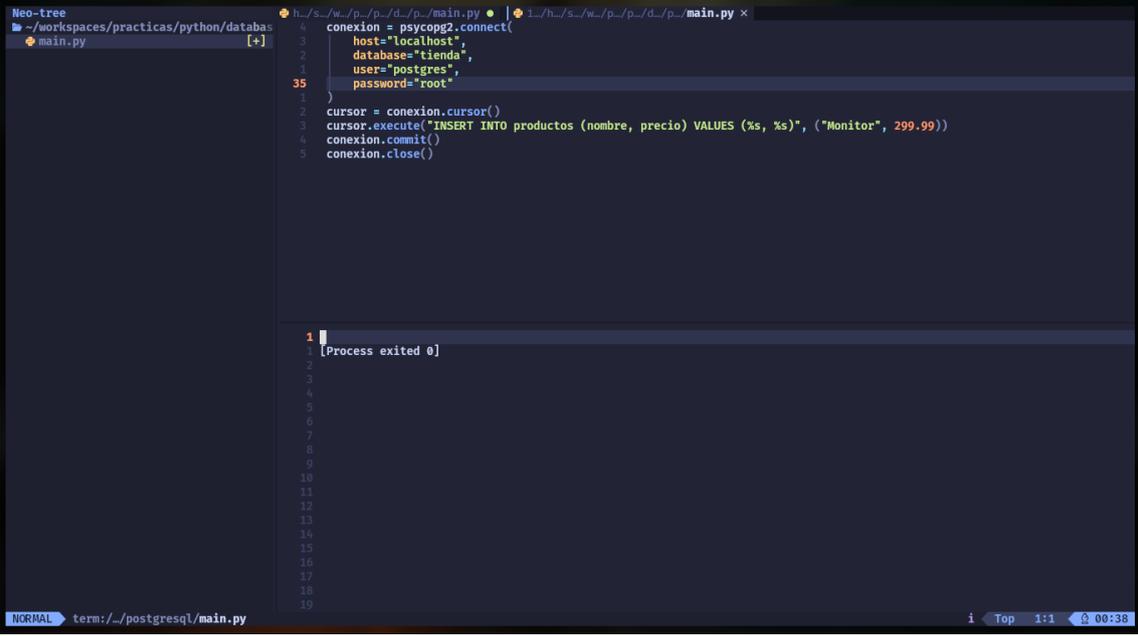
conexion = psycopg2.connect(
    host="localhost",
    database="tienda",
    user="postgres",
    password="root"
)

cursor = conexion.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    precio NUMERIC(10, 2) NOT NULL
)
""")
conexion.commit()
conexion.close()

```

En este caso, se crea una tabla productos con columnas para id, nombre y precio.

### Ejemplo 3: Insertar datos



```
1 conexion = psycopg2.connect(  
2     host="localhost",  
3     database="tienda",  
4     user="postgres",  
5     password="root"  
6 )  
7 cursor = conexion.cursor()  
8 cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", ("Monitor", 299.99))  
9 conexion.commit()  
10 conexion.close()
```

1  
2 [Process exited 0]  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19

term: ~/postgres/main.py i Top 1:1 00:38

Figure 48.3: PostgreSQL Insertar Datos

```
conexion = psycopg2.connect(  
    host="localhost",  
    database="tienda",  
    user="postgres",  
    password="root"  
)  
cursor = conexion.cursor()  
cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", ("Monitor", 299.99))  
conexion.commit()  
conexion.close()
```

En este caso, se inserta un nuevo producto en la tabla productos con nombre “Monitor” y precio 299.99.

### Ejemplo 4: Consultar datos

```
Neo-tree
~/workspaces/practicas/python/databases
  main.py

h.../s.../w.../p.../d.../p.../main.py x | 1.../h.../s.../w.../p.../d.../p.../main.py x
11 # conexion.close()
10
9   conexion = psycopg2.connect(
8       host="localhost",
7       database="tienda",
6       user="postgres",
5       password="root"
4   )
3   cursor = conexion.cursor()
2   cursor.execute("SELECT * FROM productos")
1   for fila in cursor.fetchall():
50     print(fila)
1   conexion.close()
```

```
4
3 [Process exited 0]
2
1
5 |
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

NORMAL term:./.../postgresql/main.py

Figure 48.4: PostgreSQL Consultar Datos

```
conexion = psycopg2.connect(
    host="localhost",
    database="tienda",
    user="postgres",
    password="root"
)
cursor = conexion.cursor()
cursor.execute("SELECT * FROM productos")
for fila in cursor.fetchall():
    print(fila)
conexion.close()
```

La consulta `SELECT * FROM productos` recupera todos los registros de la tabla `productos` y los imprime en pantalla.

## 48.4 Ejemplo Práctico

**Objetivo:** Administrar una base de datos PostgreSQL usando Python para crear, listar y eliminar productos.

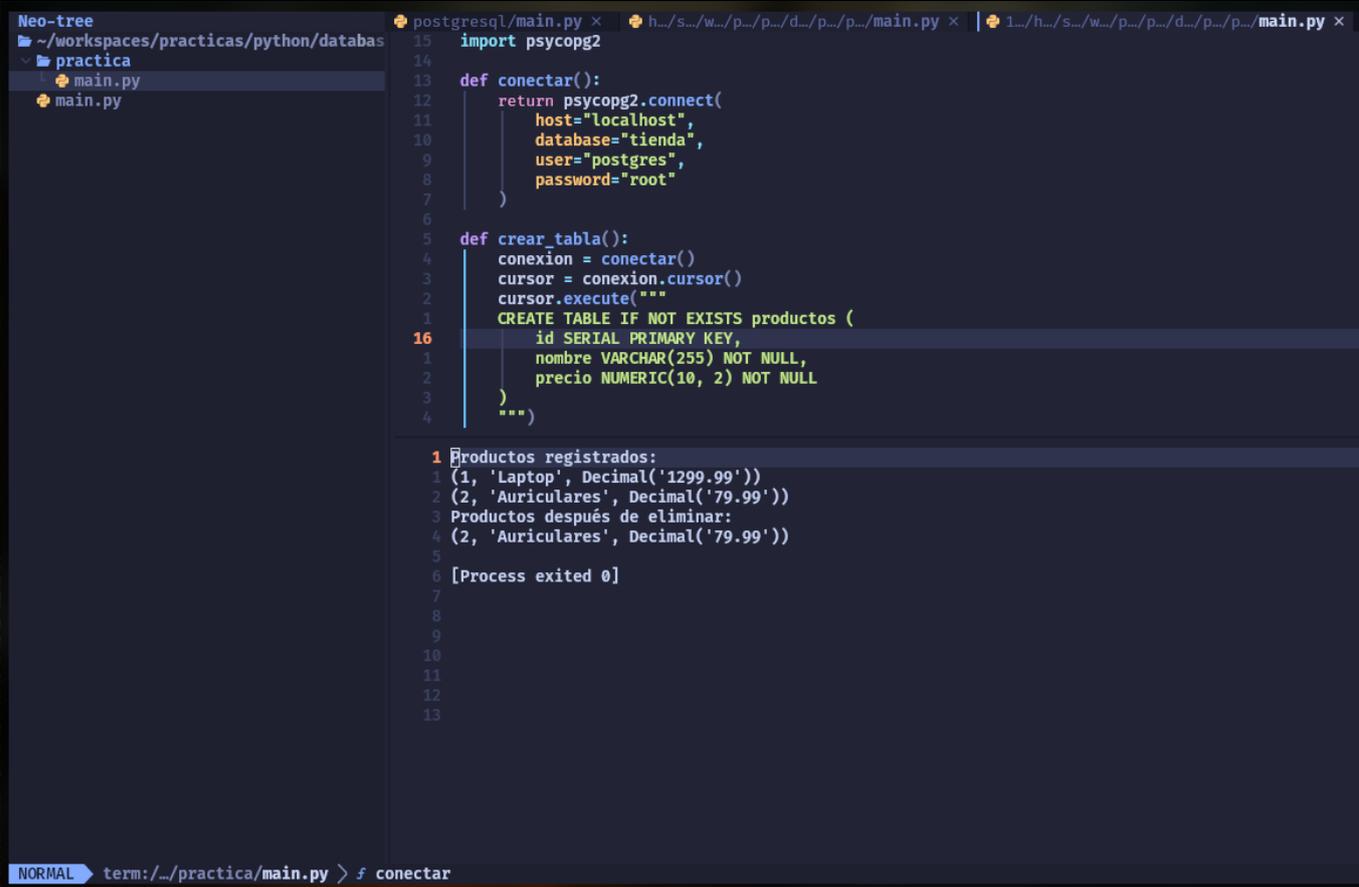
**Descripción:** El ejemplo incluye la configuración del contenedor Docker y el código Python para interactuar con PostgreSQL.

## 48.5 Instrucciones:

1. Configura un contenedor PostgreSQL usando Docker.
2. Crea una tabla productos en la base de datos tienda.
3. Implementa funciones para agregar, listar y eliminar productos desde Python.

Posible solución

Código:



```
Neo-tree
~/workspaces/practicas/python/databases
├── practica
│   └── main.py
└── main.py

postgresql/main.py x  h.../s.../w.../p.../d.../p.../p.../main.py x  | 1.../h.../s.../w.../p.../d.../p.../p.../main.py x
15 import psycopg2
14
13 def conectar():
12     return psycopg2.connect(
11         host="localhost",
10         database="tienda",
9         user="postgres",
8         password="root"
7     )
6
5 def crear_tabla():
4     conexion = conectar()
3     cursor = conexion.cursor()
2     cursor.execute("""
1     CREATE TABLE IF NOT EXISTS productos (
16         id SERIAL PRIMARY KEY,
1         nombre VARCHAR(255) NOT NULL,
2         precio NUMERIC(10, 2) NOT NULL
3     )
4     """)
1 Productos registrados:
1 (1, 'Laptop', Decimal('1299.99'))
2 (2, 'Auriculares', Decimal('79.99'))
3 Productos después de eliminar:
4 (2, 'Auriculares', Decimal('79.99'))
5
6 [Process exited 0]
7
8
9
10
11
12
13

NORMAL term:~/practica/main.py > f conectar
```

Figure 48.5: PostgreSQL Ejemplo Práctico

```
import psycopg2

def conectar():
    return psycopg2.connect(
        host="localhost",
        database="tienda",
```

```

        user="postgres",
        password="root"
    )

def crear_tabla():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("""
CREATE TABLE IF NOT EXISTS productos (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    precio NUMERIC(10, 2) NOT NULL
)
""")
    conexion.commit()
    conexion.close()

def agregar_producto(nombre, precio):
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO productos (nombre, precio) VALUES (%s, %s)", (nombre, pre
    conexion.commit()
    conexion.close()

def listar_productos():
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM productos")
    for fila in cursor.fetchall():
        print(fila)
    conexion.close()

def eliminar_producto(id_producto):
    conexion = conectar()
    cursor = conexion.cursor()
    cursor.execute("DELETE FROM productos WHERE id = %s", (id_producto,))
    conexion.commit()
    conexion.close()

# Uso
crear_tabla()
agregar_producto("Laptop", 1299.99)
agregar_producto("Auriculares", 79.99)
print("Productos registrados:")
listar_productos()
eliminar_producto(1)
print("Productos después de eliminar:")
listar_productos()

```

## 49 Conclusiones

PostgreSQL es una base de datos potente y versátil que ofrece una amplia gama de funcionalidades para el manejo de datos. Su capacidad de extensibilidad y cumplimiento de los estándares SQL la convierten en una excelente opción para proyectos de cualquier tamaño y complejidad.

## 50 Bases de Datos MongoDB

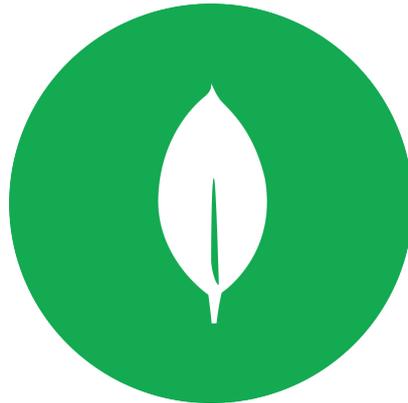


Figure 50.1: MongoDB Database

MongoDB es una base de datos NoSQL diseñada para manejar datos no estructurados y semi-estructurados de manera eficiente. Utiliza un modelo basado en documentos, lo que la hace ideal para aplicaciones que requieren alta flexibilidad y escalabilidad.

### 50.1 Conceptos Clave

- **Documentos:** La unidad básica de datos, almacenados en formato BSON (similar a JSON).
- **Colecciones:** Agrupaciones de documentos similares.
- **NoSQL:** No utiliza tablas o esquemas predefinidos, ofreciendo flexibilidad en los datos.
- **Consultas:** Potentes y basadas en JSON.
- **Escalabilidad:** Compatible con particionamiento horizontal y réplicas.

### 50.2 Configuración de MongoDB con Docker

#### 50.2.1 Instrucciones

**Crear un contenedor de MongoDB con Docker:**

Ejecuta el siguiente comando para iniciar un servidor MongoDB.

```
docker run --name mongodb-container -d -p 27017:27017 mongo:6.0
```

### 50.2.2 Parámetros:

- **-name:** Nombre del contenedor.
- **-d:** Ejecuta el contenedor en segundo plano.
- **-p 27017:27017:** Mapea el puerto del contenedor al puerto local.
- **mongo:6.0:** Imagen oficial de MongoDB.

Conectar a MongoDB desde un cliente (opcional):

Puedes usar herramientas como MongoDB Compass o Visual Studio Code con extensiones para MongoDB.

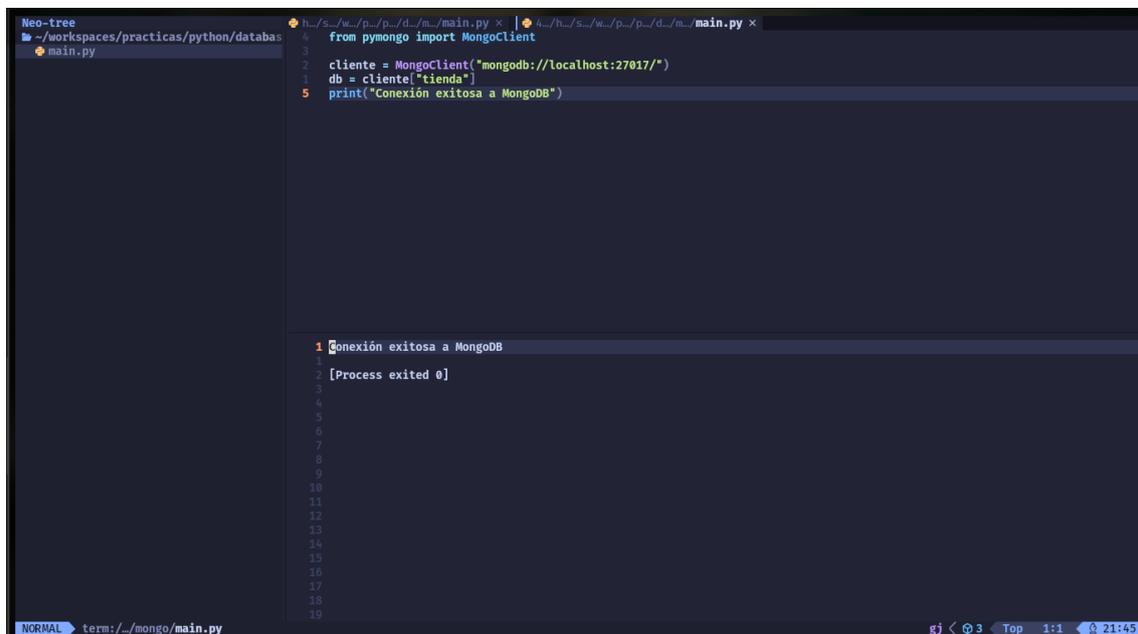
## 50.3 Ejemplos

### Ejemplo 1: Conexión a MongoDB desde Python

Instala la librería pymongo:

```
pip install pymongo
```

Conecta a la base de datos:



```
Neo-tree
├─ /workspaces/practicas/python/databases
│  └─ main.py
└─ ...

h.../s.../w.../p.../d.../m.../main.py x | h.../s.../w.../p.../d.../m.../main.py x
4 from pymongo import MongoClient
5
6 cliente = MongoClient("mongodb://localhost:27017/")
7 db = cliente["tienda"]
8
9 print("Conexión exitosa a MongoDB")

1 Conexión exitosa a MongoDB
2 [Process exited 0]
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

NORMAL term: /.../mongo/main.py gJ < 3 Top 1:1 21:45
```

Figure 50.2: MongoDB Conexión



```
Neo-tree
~/workspaces/prac
main.py

12 from pymongo import MongoClient
11
10 cliente = MongoClient("mongodb://localhost:27017/")
9 db = cliente["tienda"]
8 print("Conexión exitosa a MongoDB")
7
6 coleccion = db["productos"]
5 producto = {"nombre": "Teclado", "precio": 49.99}
4 coleccion.insert_one(producto)
3 print("Documento insertado:", producto)
2
1 for producto in coleccion.find():
13     print(producto)

4 Conexión exitosa a MongoDB
3 Documento insertado: {'nombre': 'Teclado', 'precio': 49.99, '_id': ObjectId('673ff15450542ebea071b190')}
2 {'_id': ObjectId('673ff0e2d80ebea9c8a5d1df'), 'nombre': 'Teclado', 'precio': 49.99}
1 {'_id': ObjectId('673ff15450542ebea071b190'), 'nombre': 'Teclado', 'precio': 49.99}
5
1 [Process exited 0]
```

Figure 50.4: MongoDB Consultar Documentos

```
for producto in coleccion.find():
    print(producto)
```

En este ejemplo, se recuperan todos los documentos de la colección productos y se imprimen en pantalla

**Ejemplo 4:** Actualizar documentos

```
Neo-tree
~/workspaces/prac
main.py
18 from pymongo import MongoClient
17
16 cliente = MongoClient("mongodb://localhost:27017/")
15 db = cliente["tienda"]
14 print("Conexión exitosa a MongoDB")
13
12 coleccion = db["productos"]
11 producto = {"nombre": "Teclado", "precio": 49.99}
10 coleccion.insert_one(producto)
9 print("Documento insertado:", producto)
8
7 for producto in coleccion.find():
6 |     print(producto)
5
4 coleccion.update_one(
3 |     {"nombre": "Teclado"},
2 |     {"$set": {"precio": 39.99}}
1 | )
19 print("Precio actualizado")

1 Conexión exitosa a MongoDB
2 Documento insertado: {'nombre': 'Teclado', 'precio': 49.99, '_id': ObjectId('673ff57403f09d0d04bf43d3')}
3 {'_id': ObjectId('673ff0e2d80ebea9c8a5d1df'), 'nombre': 'Teclado', 'precio': 49.99}
4 {'_id': ObjectId('673ff15450542e071b190'), 'nombre': 'Teclado', 'precio': 49.99}
5 {'_id': ObjectId('673ff57403f09d0d04bf43d3'), 'nombre': 'Teclado', 'precio': 49.99}
6 Precio actualizado
7 [Process exited 0]
8
9
10
11
12
13
14
15
16
17
18
19
20
NORMAL term:~/mongo/main.py
```

Figure 50.5: MongoDB Actualizar Documento

```
coleccion.update_one(
    {"nombre": "Teclado"},
    {"$set": {"precio": 39.99}}
)
print("Precio actualizado")
```

En este caso, se actualiza el precio del producto “Teclado” a 39.99.

## 50.4 Ejemplo Práctico

**Objetivo:** Crear una base de datos MongoDB para gestionar productos, implementando operaciones de inserción, consulta, actualización y eliminación.

**Descripción:** Usaremos Docker para iniciar MongoDB y Python para manipular los datos almacenados en documentos dentro de una colección.

## 50.5 Instrucciones:

1. Configura un contenedor MongoDB utilizando Docker.
2. Conéctate a la base de datos desde Python.
3. Realiza operaciones CRUD en una colección llamada productos.

Posible solución

**Código:**

```
from pymongo import MongoClient

# Conectar a MongoDB
def conectar():
    cliente = MongoClient("mongodb://localhost:27017/")
    return cliente["tienda"]

# Crear colección e insertar documento
def agregar_producto(nombre, precio):
    db = conectar()
    coleccion = db["productos"]
    producto = {"nombre": nombre, "precio": precio}
    coleccion.insert_one(producto)
    print(f"Producto agregado: {producto}")

# Consultar todos los documentos
def listar_productos():
    db = conectar()
    coleccion = db["productos"]
    print("Lista de productos:")
    for producto in coleccion.find():
        print(producto)

# Actualizar documento
def actualizar_producto(nombre, nuevo_precio):
    db = conectar()
    coleccion = db["productos"]
    coleccion.update_one(
        {"nombre": nombre},
        {"$set": {"precio": nuevo_precio}}
    )
    print(f"Producto '{nombre}' actualizado con precio {nuevo_precio}")

# Eliminar documento
def eliminar_producto(nombre):
    db = conectar()
    coleccion = db["productos"]
    coleccion.delete_one({"nombre": nombre})
    print(f"Producto '{nombre}' eliminado")
```

```
# Uso
agregar_producto("Monitor", 199.99)
agregar_producto("Mouse", 29.99)
listar_productos()
actualizar_producto("Monitor", 149.99)
listar_productos()
eliminar_producto("Mouse")
listar_productos()
```

## 51 Conclusiones

En este tutorial, aprendimos a trabajar con bases de datos relacionales y NoSQL utilizando Python. Aprendimos a conectarnos a bases de datos PostgreSQL y MongoDB, y a realizar operaciones CRUD como inserción, consulta, actualización y eliminación de datos.

## **Part VIII**

# **Unidad 7: Frameworks en Python**

## 52 Introducción a los Frameworks en Python



Figure 52.1: Frameworks en Python

Los frameworks son herramientas que facilitan el desarrollo de aplicaciones web al proporcionar una estructura y funcionalidades predefinidas. En Python, existen varios frameworks populares que permiten crear aplicaciones web de forma rápida y eficiente. Algunos de los frameworks más utilizados en Python son Flask, Django y FastAPI. En este tutorial, se presentarán estos frameworks y se mostrarán ejemplos de cómo crear aplicaciones web simples con cada uno de ellos.

## 52.1 Creación de Entornos Virtuales

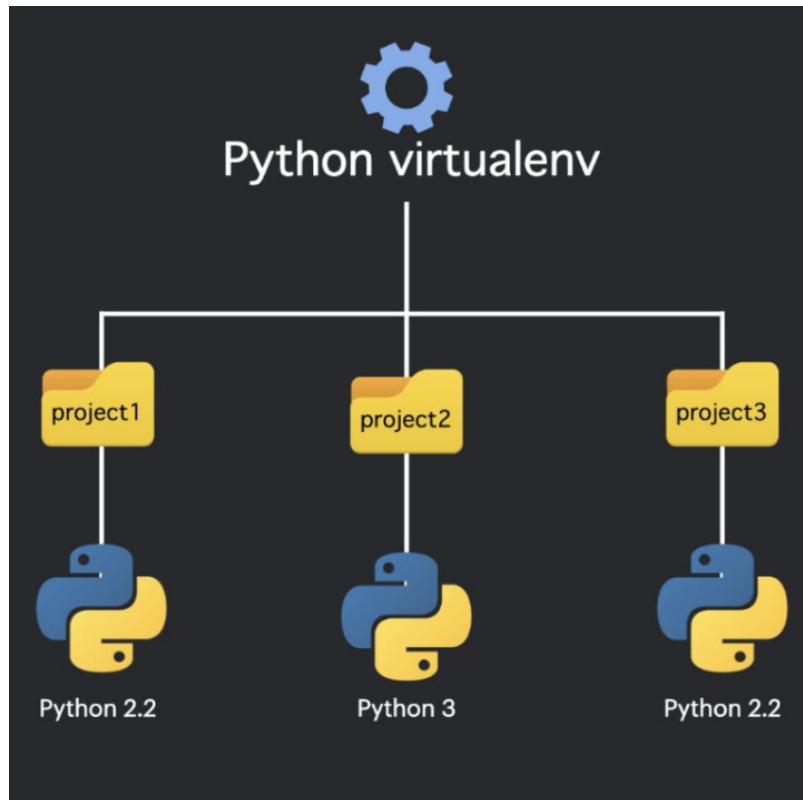


Figure 52.2: Entornos Virtuales

Antes de comenzar a trabajar con los frameworks, es recomendable crear un entorno virtual para cada proyecto. Los entornos virtuales permiten aislar las dependencias de cada proyecto y evitar conflictos entre versiones de paquetes. A continuación, se muestran los comandos para crear un entorno virtual con **venv**:

1. Crea un nuevo directorio para el proyecto:

```
mkdir mi_proyecto
cd mi_proyecto
```

2. Crea un entorno virtual:

```
python -m venv env
```

3. Activa el entorno virtual:

- En Windows:

```
env\Scripts\activate
```

- En macOS y Linux:

```
source env/bin/activate
```

Con el entorno virtual activado, puedes instalar las dependencias específicas de tu proyecto sin afectar al sistema global de Python.

## 52.2 Flask



Figure 52.3: Flask

Flask es un microframework para Python que permite crear aplicaciones web de forma rápida y sencilla. A continuación, se muestra un ejemplo de una aplicación web simple que muestra un mensaje de bienvenida.

### 52.2.1 Ejemplo

1. Instala Flask:

```
pip install Flask
```

2. Crea un archivo **app.py** con el siguiente contenido:

```
from flask import Flask

app = Flask(__name__)

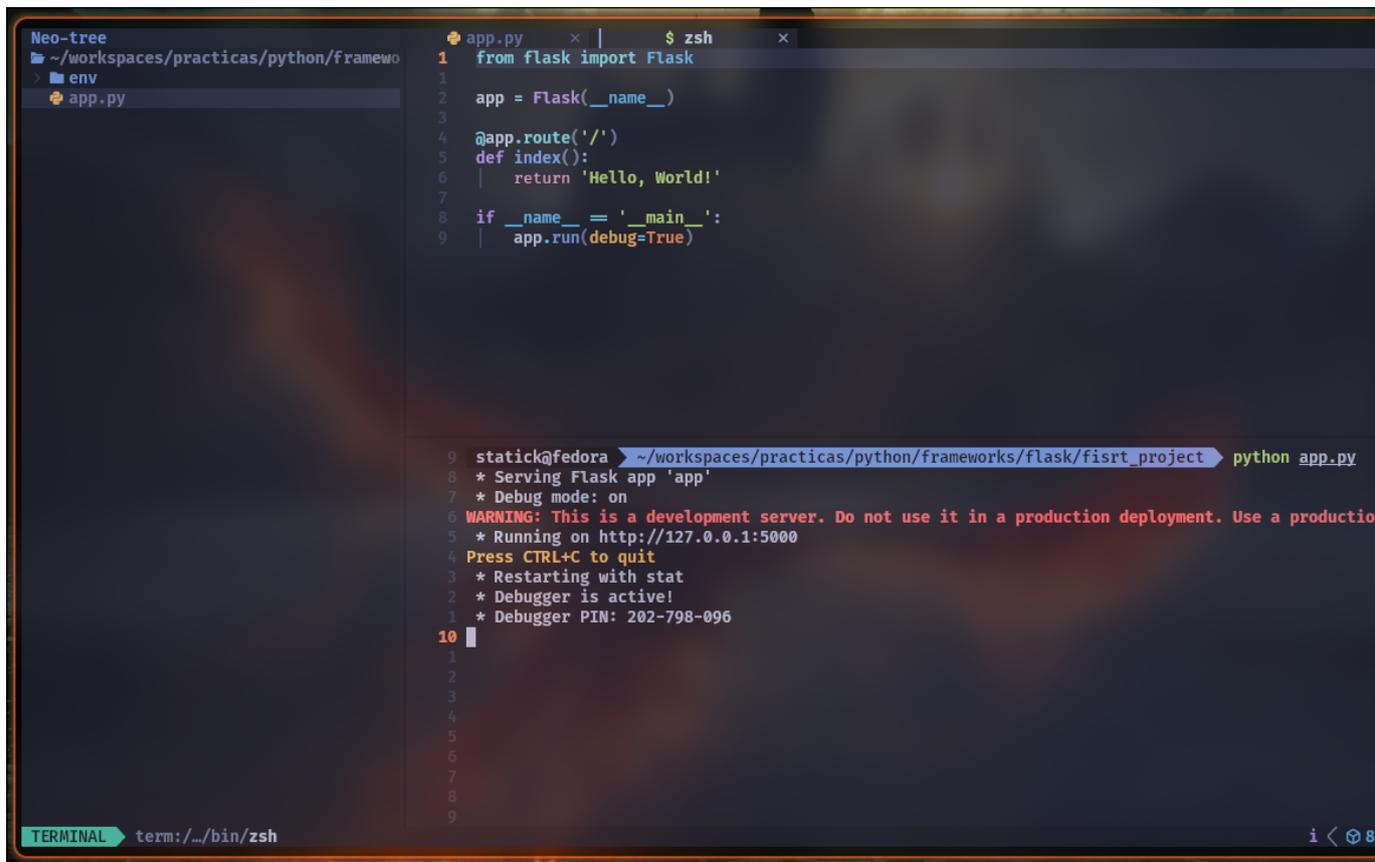
@app.route('/')
def index():
    return 'Hello, World!'
```

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

En el ejemplo anterior, se crea una aplicación web con Flask que muestra el mensaje “Hello, World!” en la ruta raíz (/).

3. Ejecuta la aplicación:

```
python app.py
```



```
Neo-tree  
~/workspaces/practicas/python/frameworko  
├── env  
└── app.py  
app.py x | $ zsh x  
1 from flask import Flask  
2 app = Flask(__name__)  
3  
4 @app.route('/')  
5 def index():  
6     return 'Hello, World!'  
7  
8 if __name__ == '__main__':  
9     app.run(debug=True)  
  
9 statick@fedora ~/workspaces/practicas/python/frameworks/flask/fisrt_project python app.py  
8 * Serving Flask app 'app'  
7 * Debug mode: on  
6 WARNING: This is a development server. Do not use it in a production deployment. Use a production  
5 * Running on http://127.0.0.1:5000  
4 Press CTRL+C to quit  
3 * Restarting with stat  
2 * Debugger is active!  
1 * Debugger PIN: 202-798-096  
10 |  
1  
2  
3  
4  
5  
6  
7  
8  
9  
TERMINAL term:~/bin/zsh
```

Figure 52.4: Flask

En la captura de pantalla anterior, se muestra el código de la aplicación Flask que se ejecuta en el servidor de desarrollo.

4. Abre un navegador web y accede a la dirección <http://localhost:5000> para ver el mensaje de bienvenida.

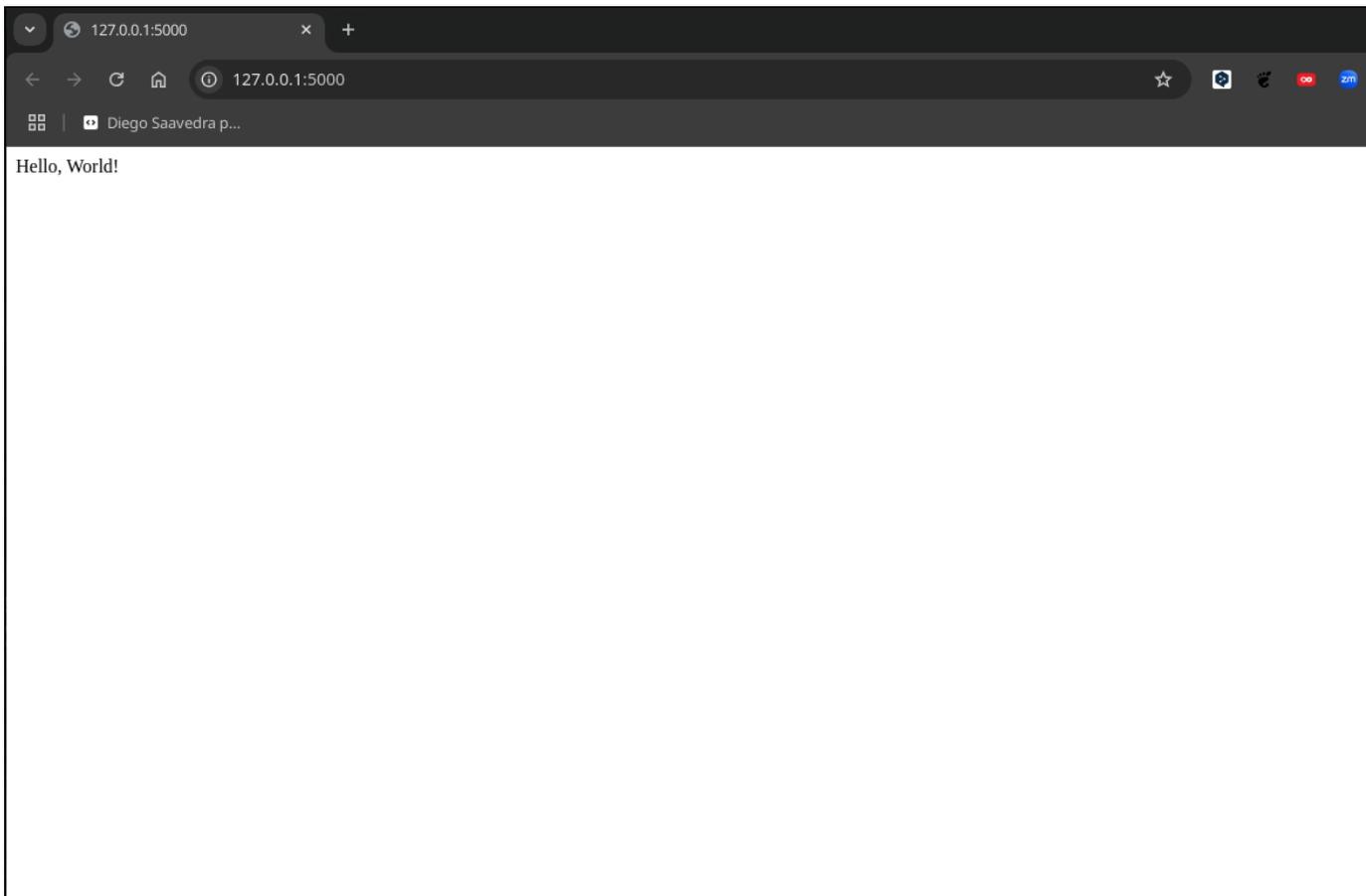


Figure 52.5: Flask

## 52.3 Django



Figure 52.6: Django

Django es un framework web de alto nivel para Python que facilita el desarrollo de aplicaciones web complejas. A continuación, se muestra un ejemplo de una aplicación web simple que muestra un mensaje de bienvenida.

### 52.3.1 Ejemplo

1. Instala Django:

```
pip install Django
```

2. Crea un proyecto Django:

```
django-admin startproject myproject
```

3. Crea una aplicación dentro del proyecto:

```
cd myproject  
python manage.py startapp myapp
```

4. Define una vista en el archivo **views.py** de la aplicación:

```
from django.http import HttpResponse  
  
def index(request):  
    return HttpResponse('¡Hola, Mundo!')
```

5. Configura la URL en el archivo **urls.py** del proyecto:

```
from django.urls import path  
  
from myapp import views  
  
urlpatterns = [  
    path('', views.index),  
]
```

6. Ejecuta el servidor de desarrollo:

```
python manage.py runserver
```

7. Abre un navegador web y accede a la dirección <http://localhost:8000> para ver el mensaje de bienvenida.

```
Neo-tree
~/workspaces/practicas/python/frameworks
├── myapp
│   ├── __pycache__
│   ├── migrations
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
└── myproject
    ├── __pycache__
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    ├── wsgi.py
    ├── db.sqlite3
    └── manage.py

views.py
1 from django.http import HttpResponseRedirect
2
3
4 def index(request):
5     """request is not accessed"""
6     return HttpResponseRedirect("¡Hola, Mundo!")

Terminal
~/workspaces/practicas/python/frameworks/django/first_app/myproject $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
November 22, 2024 - 22:36:33
Django version 5.1.3, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[22/Nov/2024 22:36:39] "GET / HTTP/1.1" 200 14
Not Found: /favicon.ico
[22/Nov/2024 22:36:39] "GET /favicon.ico HTTP/1.1" 404 2205

TERMINAL  term://bin/zsh > f index
```

Figure 52.7: Django

En la captura de pantalla anterior, se muestra el código de la aplicación Django que se ejecuta en el servidor de desarrollo.

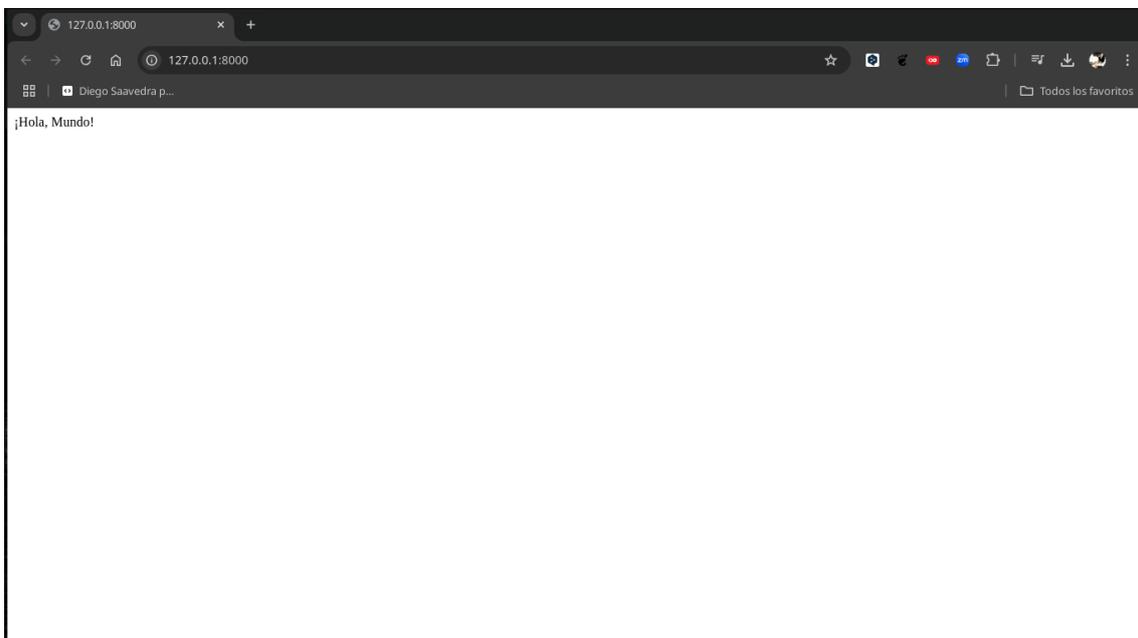


Figure 52.8: Django

En la captura de pantalla anterior, se muestra el mensaje de bienvenida en el navegador web al acceder a la dirección <http://localhost:8000>.

## 52.4 FastAPI



Figure 52.9: FastAPI

FastAPI es un framework web moderno y rápido para Python que permite crear APIs de forma sencilla y eficiente. A continuación, se muestra un ejemplo de una API simple que devuelve un mensaje de bienvenida.

### 52.4.1 Ejemplo

1. Instala FastAPI:

```
pip install fastapi
```

2. Crea un archivo **main.py** con el siguiente contenido:

```
from fastapi import FastAPI

app = FastAPI()

@app.get('/')
def read_root():
    return {'message': '¡Hola, Mundo!'}
```

En el ejemplo anterior, se crea una API con FastAPI que devuelve un diccionario con el mensaje “¡Hola, Mundo!” en la ruta raíz (/).

3. Ejecuta la aplicación:

```
uvicorn main:app --reload
```

4. Abre un navegador web y accede a la dirección <http://localhost:8000> para ver el mensaje de bienvenida.

```
Neo-tree
~/workspaces/practicasy/python/frameworko
├── __pycache__
├── env
└── main.py

main.py
6 from fastapi import FastAPI
5
4 app = FastAPI()
3
2 @app.get('/')
1 def read_root():
7 |     return {'message': '¡Hola, Mundo!'}

4 (env) statick@fedora ~/workspaces/practicasy/python/frameworks/first_app python -m uvicorn m
3
2 INFO: Will watch for changes in these directories: ['/home/statick/workspaces/practicasy/pyth
1 ]
94 INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
1 INFO: Started reloader process [19863] using StatReload
2 INFO: Started server process [19865]
3 INFO: Waiting for application startup.
4 INFO: Application startup complete.
5 INFO: 127.0.0.1:34348 - "GET / HTTP/1.1" 200 OK
6 INFO: 127.0.0.1:34348 - "GET /favicon.ico HTTP/1.1" 404 Not Found
7
8
9
10
11
12
13
14

NORMAL term:~/bin/zsh > f read_root
```

Figure 52.10: FastAPI

En la captura de pantalla anterior, se muestra el código de la API FastAPI que se ejecuta en el servidor de desarrollo.

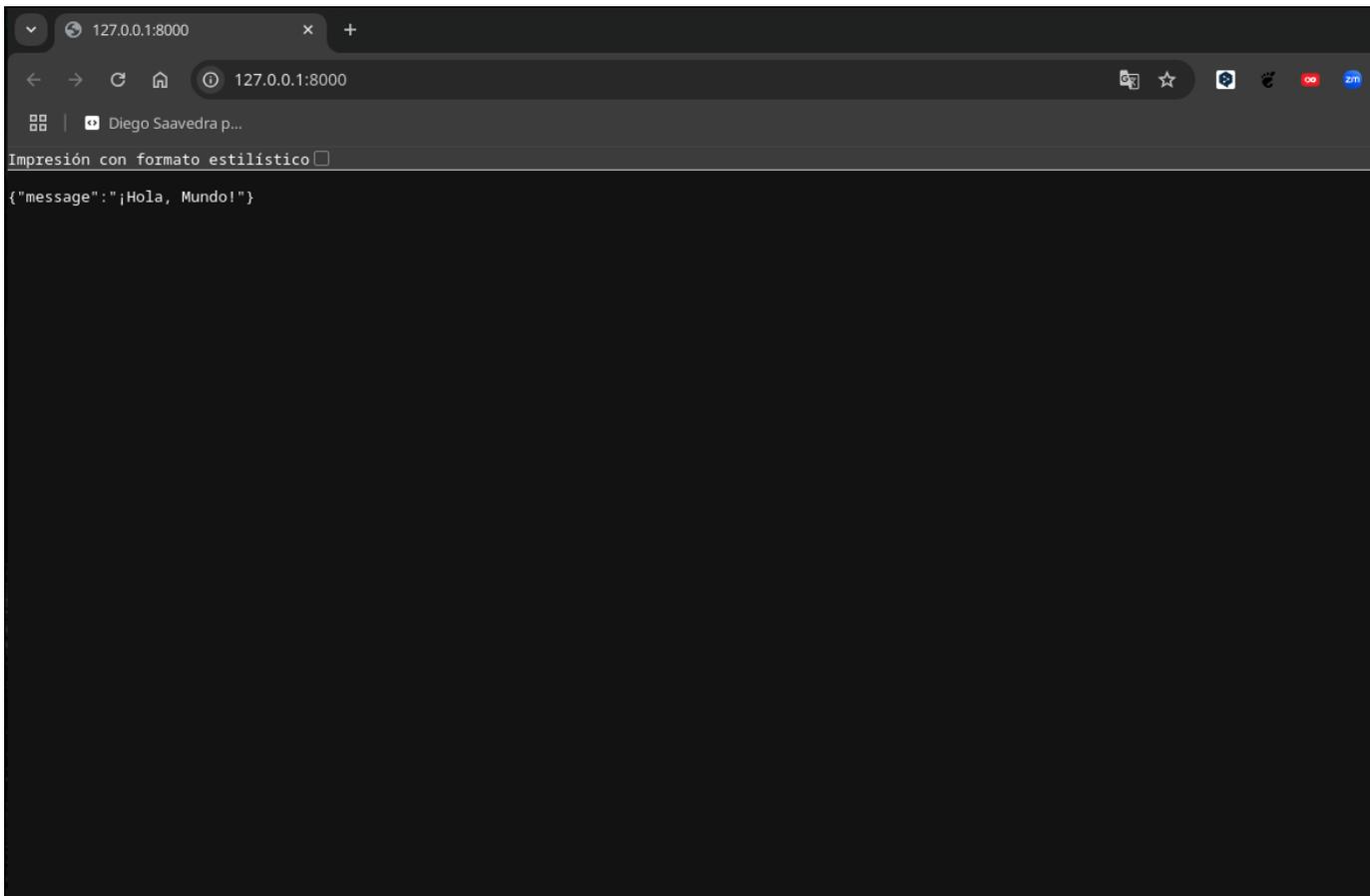


Figure 52.11: FastAPI

## 52.5 Conclusiones

En este tutorial, se presentaron algunos de los frameworks más populares para el desarrollo web en Python, incluyendo Flask, Django y FastAPI. Cada uno de estos frameworks tiene sus propias características y ventajas, por lo que es importante elegir el que mejor se adapte a tus necesidades y preferencias. Con estos frameworks, puedes crear aplicaciones web y APIs de forma rápida y sencilla, lo que te permitirá desarrollar proyectos web de manera eficiente y productiva. ¡Esperamos que este tutorial te haya sido útil y te inspire a explorar más sobre el desarrollo web en Python!

## 53 Introducción a Django



Figure 53.1: Django Framework

Django es un framework web de alto nivel para Python que facilita el desarrollo de aplicaciones web complejas. A continuación, se muestra un ejemplo de una aplicación web simple que muestra un mensaje de bienvenida.

## 54 Historia de Django

Django fue creado por Adrian Holovaty y Simon Willison mientras trabajaban en Lawrence Journal-World, un periódico en Lawrence, Kansas. Django fue lanzado como software de código abierto en julio de 2005 y ha sido mantenido por la Django Software Foundation desde entonces.

## 55 Características de Django

Django tiene las siguientes características:

- **Diseñado para la perfección:** Django sigue el principio de “baterías incluidas”, lo que significa que proporciona una amplia gama de funcionalidades listas para usar.
- **Escalabilidad:** Django es altamente escalable y se puede utilizar para desarrollar aplicaciones web de cualquier tamaño.
- **Seguridad:** Django proporciona protección contra vulnerabilidades comunes, como la inyección de SQL, la falsificación de solicitudes entre sitios (CSRF) y la inyección de scripts entre sitios (XSS).
- **Documentación detallada:** Django tiene una documentación detallada y una comunidad activa que proporciona soporte y recursos adicionales.
- **Versatilidad:** Django se puede utilizar para desarrollar una amplia variedad de aplicaciones web, desde sitios web simples hasta aplicaciones empresariales complejas.

### 55.1 Ejemplo

1. Crear un entorno virtual e instalar Django:

```
python -m venv env
source env/bin/activate # Windows: env\Scripts\activate
pip install Django
```

- 💡 Para instalar una versión específica de Django, puedes ejecutar el siguiente comando:

```
pip install Django==5.0
```

💡 Toma en cuenta el calendario de lanzamientos de Django para elegir la versión adecuada para tu proyecto. Puedes consultar la [documentación oficial de Django](#) para obtener más información.

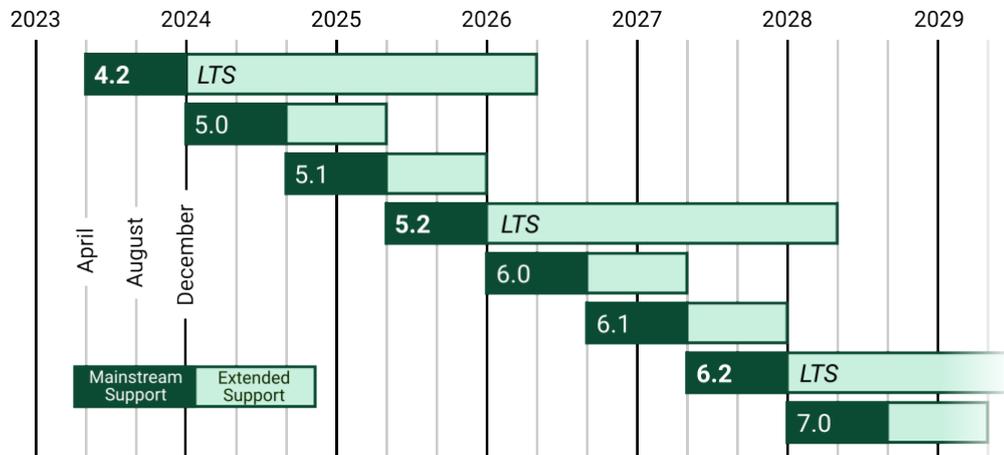


Figure 55.1: Calendario de Lanzamientos de Django

2. Crear un archivo requirements.txt con las dependencias del proyecto:

```
Django==5.1.3
```

3. Crear un archivo .gitignore para ignorar los archivos y directorios innecesarios:

```
env/  
__pycache__/  
*.pyc  
db.sqlite3
```

4. Crear un proyecto Django:

```
django-admin startproject myproject
```

💡 Puedes agregar el `.` al final del comando para crear el proyecto en el directorio actual:

```
django-admin startproject myproject .
```

5. Crear una aplicación dentro del proyecto:

```
python manage.py startapp myapp
```

! Puedes crear **n** aplicaciones dentro del proyecto, donde **n** es el número de aplicaciones que deseas crear.

💡 Agregar el nombre de la aplicación al archivo **INSTALLED\_APPS** en el archivo **settings.py** del proyecto:

```
INSTALLED_APPS = [  
    ...  
    'myapp',  
]
```

6. Definir una vista en el archivo **views.py** de la aplicación:

```
from django.http import HttpResponse  
  
def index(request):  
    return HttpResponse('¡Hola, Mundo!')
```

7. Configurar la URL en el archivo **urls.py** del proyecto:

```
from django.urls import path  
  
from myapp import views  
  
urlpatterns = [  
    path('', views.index),  
]
```

8. Ejecutar el servidor de desarrollo:

```
python manage.py runserver
```

9. Abrir un navegador web y acceder a la dirección <http://localhost:8000> para ver el mensaje de bienvenida.

```
Neo-tree
~/workspaces/practicasython/frameworks/django/first_app/myproject
├── __pycache__
├── migrations
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── myproject
│   ├── __pycache__
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   ├── db.sqlite3
│   └── manage.py
└── views.py
    3 from django.http import HttpResponse
    4 def index(request):
    5     return HttpResponse('Hola, Mundo!')

17 statick@fedora ~/workspaces/practicasython/frameworks/django/first_app/myproject python m
16 Watching for file changes with StatReloader
15 Performing system checks...
14
13 System check identified no issues (0 silenced).
12
11 You have 18 unapplied migration(s). Your project may not work properly until you apply the migrat
10 auth, contenttypes, sessions.
9 Run 'python manage.py migrate' to apply them.
8 November 22, 2024 - 22:36:33
7 Django version 5.1.3, using settings 'myproject.settings'
6 Starting development server at http://127.0.0.1:8000/
5 Quit the server with CONTROL-C.
4
3 [22/Nov/2024 22:36:39] "GET / HTTP/1.1" 200 14
2 Not Found: /favicon.ico
1 [22/Nov/2024 22:36:39] "GET /favicon.ico HTTP/1.1" 404 2205
18
1
```

Figure 55.2: Django

En la captura de pantalla anterior, se muestra el código de la aplicación Django que se ejecuta en el servidor de desarrollo.

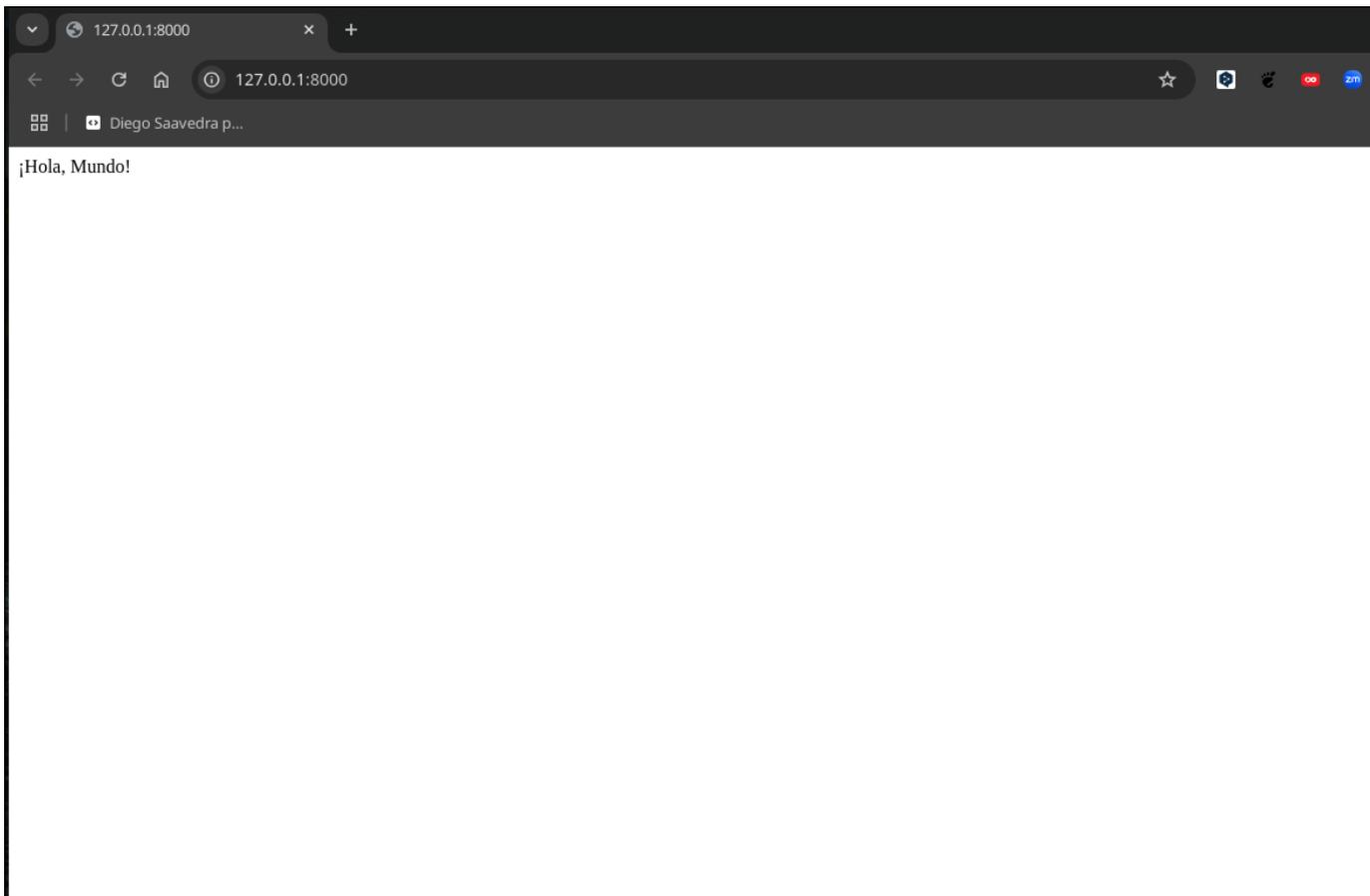


Figure 55.3: Django

En la captura de pantalla anterior, se muestra el mensaje de bienvenida en el navegador web al acceder a la dirección <http://localhost:8000>.

## 56 Buenas Prácticas

- Utiliza un entorno virtual para instalar las dependencias del proyecto.
- Crea un archivo **requirements.txt** con las dependencias del proyecto.
- Crea un archivo **.gitignore** para ignorar los archivos y directorios innecesarios.

## 57 Actividad

1. Crea un proyecto Django llamado **myproject**.
2. Crea una aplicación llamada **myapp** dentro del proyecto.
3. Define una vista en el archivo **views.py** de la aplicación.
4. Configura la URL en el archivo **urls.py** del proyecto.
5. Ejecuta el servidor de desarrollo y accede a la dirección <http://localhost:8000> para ver el mensaje de bienvenida.

## 58 Recursos Adicionales

- [Documentación oficial de Django](#)
- [Calendario de lanzamientos de Django](#)

## 59 Conclusiones

En este tutorial, aprendiste a crear un proyecto Django y una aplicación web simple que muestra un mensaje de bienvenida. También aprendiste a definir una vista y configurar la URL en el proyecto Django. Por último, aprendiste a ejecutar el servidor de desarrollo y acceder a la dirección <http://localhost:8000> para ver el mensaje de bienvenida en un navegador web.

## 60 Estructura de un proyecto en Django



Figure 60.1: Django Project Structure

La estructura de un proyecto en Django es muy sencilla y se basa en un conjunto de directorios y archivos que se crean automáticamente al crear un nuevo proyecto. A continuación se muestra la estructura de un proyecto en Django:

Es necesario tomar en cuenta que para cada proyecto se creará un directorio raíz con el nombre del proyecto, en este caso se llamará **gestion de farmacias**. Sin embargo antes de crear cualquier proyecto se deberá crear:

1. Entorno Virtual
2. Instalar Django
3. Crear el archivo requirements.txt
4. Crear el archivo .gitignore
5. Crear el archivo README.md
6. Crear el archivo LICENSE
7. Crear el archivo .env

Para crear un proyecto en Django se debe ejecutar el siguiente comando:

```
django-admin startproject gestion_de_farmacias .
```

Este comando creará un nuevo proyecto llamado **gestion\_de\_farmacias** en el directorio actual. A continuación se muestra la estructura de un proyecto en Django:

```
gestion_de_farmacias/  
  gestion_de_farmacias/  
    __init__.py  
    asgi.py  
    settings.py  
    urls.py  
    wsgi.py  
  manage.py  
  requirements.txt
```

A continuación se describen los directorios y archivos que se crean al crear un nuevo proyecto en Django:

- **gestion\_de\_farmacias/**: Directorio raíz del proyecto.
  - **gestion\_de\_farmacias/**: Directorio que contiene el código fuente del proyecto.
    - \* **\_\_init\_\_.py**: Archivo que indica que el directorio es un paquete de Python.
    - \* **asgi.py**: Archivo que contiene la configuración de ASGI (Asynchronous Server Gateway Interface).
    - \* **settings.py**: Archivo que contiene la configuración del proyecto.
    - \* **urls.py**: Archivo que contiene las rutas del proyecto.
    - \* **wsgi.py**: Archivo que contiene la configuración de WSGI (Web Server Gateway Interface).
  - **manage.py**: Archivo que se utiliza para administrar el proyecto.
  - **requirements.txt**: Archivo que contiene las dependencias del proyecto.

El archivo **manage.py** es un script que se utiliza para administrar el proyecto. Se puede utilizar para realizar tareas como crear aplicaciones, ejecutar el servidor de desarrollo, ejecutar pruebas, etc.

El archivo **requirements.txt** es un archivo de texto que contiene las dependencias del proyecto. Se puede utilizar para instalar todas las dependencias del proyecto con el comando **pip install -r requirements.txt**.

En resumen, la estructura de un proyecto en Django es muy sencilla y se basa en un conjunto de directorios y archivos que se crean automáticamente al crear un nuevo proyecto.

## 61 Estructura de una aplicación en Django

La estructura de una aplicación en Django es muy sencilla y se basa en un conjunto de directorios y archivos que se crean automáticamente al crear una nueva aplicación. A continuación se muestra la estructura de una aplicación en Django:

Para crear una aplicación en Django se debe ejecutar el siguiente comando:

```
python manage.py startapp farmacia
```

Este comando creará una nueva aplicación llamada **farmacia** en el directorio actual. A continuación se muestra la estructura de una aplicación

```
farmacia/  
  migrations/  
    __init__.py  
  __init__.py  
  admin.py  
  apps.py  
  models.py  
  tests.py  
  views.py
```

A continuación se describen los directorios y archivos que se crean al crear una nueva aplicación en Django:

- **farmacia/**: Directorio raíz de la aplicación.
  - **migrations/**: Directorio que contiene las migraciones de la aplicación.
    - \* **\_\_init\_\_.py**: Archivo que indica que el directorio es un paquete de Python.
  - **\_\_init\_\_.py**: Archivo que indica que el directorio es un paquete de Python.
  - **admin.py**: Archivo que contiene la configuración del panel de administración.
  - **apps.py**: Archivo que contiene la configuración de la aplicación.
  - **models.py**: Archivo que contiene los modelos de la aplicación.
  - **tests.py**: Archivo que contiene las pruebas de la aplicación.
  - **views.py**: Archivo que contiene las vistas de la aplicación.

En resumen, la estructura de una aplicación en Django es muy sencilla y se basa en un conjunto de directorios y archivos que se crean automáticamente al crear una nueva aplicación.

## 61.1 Activiades:

1. Crear un proyecto en Django llamado **gestion\_de\_farmacias**.
2. Crear una aplicación en Django llamada **farmacia**.
3. Crear un archivo **requirements.txt** con las dependencias del proyecto.
4. Crear un archivo **.gitignore** para ignorar los archivos y directorios innecesarios.
5. Crear un archivo **README.md** con la descripción del proyecto.
6. Crear un archivo **LICENSE** con la licencia del proyecto.
7. Crear un archivo **.env** con las variables de entorno del proyecto.

Posible solución

1. Crear un proyecto en Django llamado **gestion\_de\_farmacias**.

```
django-admin startproject gestion_de_farmacias .
```

2. Crear una aplicación en Django llamada **farmacia**.

```
python manage.py startapp farmacia
```

3. Crear un archivo **requirements.txt** con las dependencias del proyecto.

```
Django==4.2.1
```

4. Crear un archivo **.gitignore** para ignorar los archivos y directorios innecesarios.

```
__pycache__/  
*.pyc  
*.sqlite3  
*.env
```

5. Crear un archivo **README.md** con la descripción del proyecto.

```
# Gestion de Farmacias
```

```
Proyecto de gestión de farmacias desarrollado con Django.
```

```
## Instalación
```

1. Clonar el repositorio.
2. Crear un entorno virtual.
3. Instalar las dependencias.
4. Configurar las variables de entorno.

```
## Uso
```

```
Para ejecutar el proyecto, utilizar el comando **python manage.py runserver**.
```

```
## Licencia
```

```
Este proyecto está bajo la licencia MIT.
```

6. Crear un archivo **LICENSE** con la licencia del proyecto.

```
MIT License
```

```
Derechos de autor (c) 2024 Autor
```

```
Se concede permiso, de forma gratuita, a cualquier persona que obtenga una copia de este software y de los archivos de documentación asociados (el "Software"), para tratar el Software sin restricciones, incluidos, entre otros, los derechos para usar, copiar, modificar, fusionar, publicar, distribuir, sublicenciar y / o vender copias del Software, y para permitir a las personas a quienes pertenece el Software amueblado para hacerlo, sujeto a las siguientes condiciones:
```

```
El aviso de copyright anterior y este aviso de permiso se incluirán en todas las copias o partes sustanciales del Software.
```

```
EL SOFTWARE SE PROPORCIONA "TAL CUAL", SIN GARANTÍA DE NINGÚN TIPO, EXPRESA O IMPLÍCITA, INCLUYENDO PERO NO LIMITADO A LAS GARANTÍAS DE COMERCIALIZACIÓN,
```

```
APTITUD PARA UN PROPÓSITO PARTICULAR Y NO INFRACCIÓN. EN NINGÚN CASO EL AUTOR O
```

```
LOS TITULARES DE DERECHOS DE AUTOR SERÁN RESPONSABLES DE CUALQUIER RECLAMO, DAÑO U OTRO RESPONSABILIDAD, YA SEA EN UNA ACCIÓN DE CONTRATO, AGRAVIO O DE OTRO MODO, DERIVADO DE, FUERA DE O EN RELACIÓN CON EL SOFTWARE O EL USO U OTROS TRATOS EN EL SOFTWARE.
```

7. Crear un archivo **.env** con las variables de entorno del proyecto.

```
DEBUG=True
```

```
SECRET_KEY
```

## 62 Conclusiones

En este tutorial se ha mostrado la estructura de un proyecto en Django y la estructura de una aplicación en Django. Se ha creado un proyecto llamado **gestion\_de\_farmacias** y una aplicación llamada **farmacia**. Además, se han creado los archivos **requirements.txt**, **.gitignore**, **README.md**, **LICENSE** y **.env**.

## 63 Modelos en Django



Figure 63.1: Django Models

Los modelos en Django son clases de Python que representan tablas en la base de datos. Cada atributo de la clase representa una columna en la tabla. Django proporciona un ORM (Object-Relational Mapping) que permite interactuar con la base de datos sin escribir SQL.

### 63.1 ¿Qué es ORM?

ORM (Object-Relational Mapping) es una técnica de programación que permite mapear objetos de una aplicación a tablas de una base de datos relacional. En lugar de escribir consultas SQL directamente, se utilizan clases de Python para interactuar con la base de datos.

Por ejemplo, en lugar de escribir una consulta SQL para insertar un registro en una tabla, se crea un objeto de una clase de Python y se guarda en la base de datos. El ORM se encarga de traducir las operaciones de la aplicación a consultas SQL.

### 63.2 Ventajas de ORM

Algunas de las ventajas de utilizar un ORM en Django son las siguientes:

- **Abstracción de la base de datos:** Permite interactuar con la base de datos a través de clases de Python en lugar de escribir consultas SQL.

- Portabilidad: Permite cambiar de un motor de base de datos a otro sin modificar el código de la aplicación.
- Seguridad: Evita la inyección de SQL al utilizar consultas parametrizadas.
- Productividad: Simplifica el desarrollo de aplicaciones al proporcionar una interfaz de programación de alto nivel.
- Mantenimiento: Facilita la actualización y mantenimiento de la base de datos al utilizar modelos en lugar de consultas SQL.

### 63.3 Crear un modelo

En este ejemplo crearemos modelos que nos permitan gestionar una farmacia. Para ello, crearemos un modelo para representar los medicamentos y otro modelo para representar las ventas.

Para crear un modelo en Django se debe definir una clase que herede de **models.Model** y que contenga los atributos que representan las columnas de la tabla. A continuación se muestra un ejemplo de un modelo para representar un medicamento:

```

from django.db import models ①

class Medicamento(models.Model): ②
    nombre = models.CharField(max_length=100) ③
    precio = models.DecimalField(max_digits=10, decimal_places=2) ④
    existencias = models.IntegerField() ⑤

    def __str__(self): ⑥
        return self.nombre

class Venta(models.Model): ⑦
    medicamento = models.ForeignKey(Medicamento, on_delete=models.CASCADE) ⑧
    cantidad = models.IntegerField() ⑨
    fecha = models.DateTimeField(auto_now_add=True) ⑩

    def __str__(self): ⑪
        return f'{self.medicamento} - {self.cantidad}'

```

- ① Se importa el módulo **models** de Django.
- ② Se define una clase **Medicamento** que hereda de **models.Model**.
- ③ Se define un atributo **nombre** de tipo **CharField** que representa el nombre del medicamento.
- ④ Se define un atributo **precio** de tipo **DecimalField** que representa el precio del medicamento.
- ⑤ Se define un atributo **existencias** de tipo **IntegerField** que representa las existencias del medicamento.
- ⑥ Se define un método **\_\_str\_\_** que devuelve el nombre del medicamento.
- ⑦ Se define una clase **Venta** que hereda de **models.Model**.

- ⑧ Se define un atributo **medicamento** de tipo **ForeignKey** que establece una relación con el modelo **Medicamento**.
- ⑨ Se define un atributo **cantidad** de tipo **IntegerField** que representa la cantidad vendida.
- ⑩ Se define un atributo **fecha** de tipo **DateTimeField** que representa la fecha de la venta.
- ⑪ Se define un método `__str__` que devuelve una representación legible de la venta.

## 63.4 Migraciones

Una vez que se han definido los modelos, es necesario crear las tablas en la base de datos. Para ello, se utilizan las migraciones de Django. Las migraciones son archivos de Python que contienen las instrucciones necesarias para crear, modificar o eliminar tablas en la base de datos.

Para crear las migraciones se debe ejecutar el siguiente comando:

```
python manage.py makemigrations
```

Este comando generará un archivo de migración en el directorio **migrations** de cada aplicación. A continuación se muestra un ejemplo de un archivo de migración:

```
from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Medicamento',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False,
                    ('nombre', models.CharField(max_length=100)),
                    ('precio', models.DecimalField(decimal_places=2, max_digits=10)),
                    ('existencias', models.IntegerField()),
            ],
        ),
        migrations.CreateModel(
            name='Venta',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False,
                    ('cantidad', models.IntegerField()),
                    ('fecha', models.DateTimeField(auto_now_add=True)),
```

```
        ('medicamento', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
                                           to='medicamento')),
    ],
),
]
```

Este archivo contiene las instrucciones necesarias para crear las tablas **Medicamento** y **Venta** en la base de datos. Para aplicar las migraciones se debe ejecutar el siguiente comando:

```
python manage.py migrate
```

Este comando creará las tablas en la base de datos y aplicará las migraciones pendientes.

## 63.5 Interactuar con la base de datos

Una de las formas de interactuar con la base de datos es a través del administrador de Django. Para acceder al administrador se debe crear un superusuario con el siguiente comando:

```
python manage.py createsuperuser
```

Este comando solicitará un nombre de usuario, una dirección de correo electrónico y una contraseña para el superusuario. Una vez que se ha creado el superusuario, se puede acceder al administrador de Django en la siguiente URL: **http://127.0.0.1:8000/admin/**.

En el administrador se pueden gestionar los medicamentos y las ventas de la farmacia. Se pueden crear, leer, actualizar y eliminar registros en la base de datos utilizando una interfaz gráfica.

Sin embargo para que los modelos sean accesibles desde el administrador se debe registrar los modelos en el archivo **admin.py** de la aplicación. A continuación se muestra un ejemplo de cómo registrar los modelos **Medicamento** y **Venta** en el archivo **admin.py**:

```
from django.contrib import admin
from .models import Medicamento, Venta

admin.site.register(Medicamento)
admin.site.register(Venta)
```

Una vez que se han registrado los modelos en el administrador, se pueden gestionar los medicamentos y las ventas desde la interfaz gráfica.

Una vez que se han creado las tablas en la base de datos, se pueden realizar operaciones de lectura, escritura, actualización y eliminación utilizando los modelos de Django. A continuación se muestran algunos ejemplos de cómo interactuar con la base de datos:

Para interactuar con la base de datos se debe utilizar la consola interactiva de Django. Para acceder a la consola se debe ejecutar el siguiente comando:

```
python manage.py shell
```

Este comando abrirá una consola interactiva de Python con acceso a los modelos de Django.

Es necesario hacer algunas importaciones para poder interactuar con los modelos:

```
from farmacia.models import Medicamento, Venta
```

En este ejemplo se importan los modelos **Medicamento** y **Venta** de la aplicación **farmacia**.

A continuación se muestran algunos ejemplos de cómo interactuar con la base de datos utilizando los modelos **Medicamento** y **Venta**:

```
# Crear un medicamento
medicamento = Medicamento(nombre='Paracetamol', precio=10.50, existencias=100)
medicamento.save()

# Crear una venta
venta = Venta(medicamento=medicamento, cantidad=10)
venta.save()

# Consultar los medicamentos
medicamentos = Medicamento.objects.all()

# Consultar las ventas
ventas = Venta.objects.all()

# Actualizar un medicamento
medicamento.existencias = 90
medicamento.save()

# Eliminar un medicamento
medicamento.delete()
```

Estos son algunos ejemplos de cómo interactuar con la base de datos utilizando los modelos de Django. Los modelos proporcionan una interfaz de alto nivel para realizar operaciones CRUD (Create, Read, Update, Delete) en la base de datos sin necesidad de escribir consultas SQL.

## 64 Actividades

1. Define un modelo en Django para representar una categoría de productos. La categoría debe tener un nombre y una descripción.
2. Define un modelo en Django para representar un producto. El producto debe tener un nombre, una descripción, un precio y una categoría a la que pertenece.
3. Crea las migraciones y aplica las migraciones a la base de datos.
4. Crea algunos productos y categorías en la base de datos utilizando la consola interactiva de Django.
5. Consulta los productos y categorías en la base de datos utilizando los modelos de Django.
6. Actualiza y elimina algunos productos y categorías en la base de datos utilizando los modelos de Django.

Ver Solución

```
python manage.py startapp farmacia
```

Ahora que hemos creado la aplicación **farmacia**, podemos definir los modelos que representarán los medicamentos y las ventas en nuestra farmacia. Los modelos en Django son clases de Python que representan tablas en la base de datos. Cada atributo de la clase representa una columna en la tabla.

En el archivo **models.py** de la aplicación **farmacia**, definiremos los modelos **Medicamento** y **Venta** que representarán los medicamentos y las ventas en nuestra farmacia, respectivamente.

```
from django.db import models

class Medicamento(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    existencias = models.IntegerField()

    def __str__(self):
        return self.nombre

class Venta(models.Model):
    medicamento = models.ForeignKey(Medicamento, on_delete=models.CASCADE)
    cantidad = models.IntegerField()
```

```

fecha = models.DateTimeField(auto_now_add=True)

def __str__(self):
    return f'{self.medicamento} - {self.cantidad}'

```

En este ejemplo, hemos definido dos modelos: **Medicamento** y **Venta**. El modelo **Medicamento** tiene tres atributos: **nombre**, **precio** y **existencias**. El modelo **Venta** tiene tres atributos: **medicamento**, **cantidad** y **fecha**.

El atributo **medicamento** del modelo **Venta** es una clave foránea que establece una relación con el modelo **Medicamento**. La opción **on\_delete=models.CASCADE** indica que si se elimina un medicamento, también se eliminarán todas las ventas asociadas a ese medicamento.

Una vez que hemos definido los modelos, es necesario crear las migraciones y aplicarlas a la base de datos. Para crear las migraciones, ejecutamos el siguiente comando:

```
python manage.py makemigrations
```

Este comando generará un archivo de migración en el directorio **migrations** de la aplicación **farmacia**. A continuación, aplicamos las migraciones a la base de datos con el siguiente comando:

```
python manage.py migrate
```

Este comando creará las tablas **Medicamento** y **Venta** en la base de datos y aplicará las migraciones pendientes.

Una vez que hemos creado las tablas en la base de datos, podemos interactuar con ellas utilizando los modelos de Django. Podemos crear, leer, actualizar y eliminar registros en la base de datos utilizando los modelos **Medicamento** y **Venta**.

Para interactuar con la base de datos, podemos utilizar la consola interactiva de Django. Para acceder a la consola interactiva, ejecutamos el siguiente comando:

```
python manage.py shell
```

En la consola interactiva, podemos crear medicamentos, ventas y consultar los registros en la base de datos utilizando los modelos **Medicamento** y **Venta**.

```

# Crear un medicamento

medicamento = Medicamento(nombre='Paracetamol', precio=10.50, existencias=100)
medicamento.save()

# Crear una venta

venta = Venta(medicamento=medicamento, cantidad=10)

```

```
venta.save()

# Consultar los medicamentos

medicamentos = Medicamento.objects.all()

# Consultar las ventas

ventas = Venta.objects.all()
```

Estos son algunos ejemplos de cómo interactuar con la base de datos utilizando los modelos de Django. Los modelos proporcionan una interfaz de alto nivel para realizar operaciones CRUD (Create, Read, Update, Delete) en la base de datos sin necesidad de escribir consultas SQL.

## 65 Conclusión

En este tutorial hemos aprendido cómo definir modelos en Django para representar tablas en la base de datos. Los modelos en Django son clases de Python que representan tablas en la base de datos y proporcionan una interfaz de alto nivel para interactuar con la base de datos sin necesidad de escribir consultas SQL.

## 66 Forms en Django

Los formularios en Django son una parte fundamental de cualquier aplicación web. Permiten a los usuarios enviar datos al servidor y procesarlos de manera eficiente. En este tutorial, aprenderemos cómo crear formularios en Django y cómo procesarlos en las vistas.

### 66.1 Creación de un formulario

Para crear un formulario en Django, primero debemos definir una clase que herede de **forms.Form** o **forms.ModelForm**. A continuación, definimos los campos del formulario como atributos de la clase. Por ejemplo, el siguiente formulario permite a los usuarios enviar un mensaje de contacto:

#### Tip

Recordemos los modelos creados en la sección anterior:

```
from django.db import models

class Medicamento(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    existencias = models.IntegerField()

    def __str__(self):
        return self.nombre

class Venta(models.Model):
    medicamento = models.ForeignKey(Medicamento, on_delete=models.CASCADE)
    cantidad = models.IntegerField()
    fecha = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'{self.medicamento} - {self.cantidad}'
```

Para crear un formulario de contacto, podemos hacer lo siguiente:

```
from django import forms ①
from .models import Medicamento, Venta ②
```

```

class ContactForm(forms.Form):
    nombre = forms.CharField(label='Nombre', max_length=100)
    email = forms.EmailField(label='Email', max_length=100)
    mensaje = forms.CharField(label='Mensaje', widget=forms.Textarea)

class MedicamentoForm(forms.ModelForm):
    class Meta:
        model = Medicamento
        fields = ['nombre', 'precio', 'existencias']

class VentaForm(forms.ModelForm):
    class Meta:
        model = Venta
        fields = ['medicamento', 'cantidad']

```

- ① Importamos el módulo **forms** de Django.
- ② Importamos los modelos **Medicamento** y **Venta**.
- ③ Definimos una clase **ContactForm** que hereda de **forms.Form**.
- ④ Definimos un campo de texto para el nombre del usuario.
- ⑤ Definimos un campo de correo electrónico para el email del usuario.
- ⑥ Definimos un campo de texto multilínea para el mensaje del usuario.
- ⑦ Definimos una clase **MedicamentoForm** que hereda de **forms.ModelForm**.
- ⑧ Definimos la clase **Meta** para especificar el modelo y los campos del formulario.
- ⑨ Especificamos el modelo **Medicamento** como base del formulario.
- ⑩ Especificamos los campos del formulario: **nombre**, **precio** y **existencias**.
- ⑪ Definimos una clase **VentaForm** que hereda de **forms.ModelForm**.
- ⑫ Definimos la clase **Meta** para especificar el modelo y los campos del formulario.
- ⑬ Especificamos el modelo **Venta** como base del formulario.
- ⑭ Especificamos los campos del formulario: **medicamento** y **cantidad**.

## 66.2 Procesamiento de formularios en las vistas

## 67 Actividades:

1. Crear un formulario para la creación de un nuevo medicamento.
2. Crear un formulario para la venta de un medicamento.

Posible solución

1. Crear un formulario para la creación de un nuevo medicamento.

```
from django import forms
from .models import Medicamento

class MedicamentoForm(forms.ModelForm):
    class Meta:
        model = Medicamento
        fields = ['nombre', 'precio', 'existencias']
```

2. Crear un formulario para la venta de un medicamento.

```
from django import forms
from .models import Venta

class VentaForm(forms.ModelForm):
    class Meta:
        model = Venta
        fields = ['medicamento', 'cantidad']
```

## 68 Conclusiones

En este tutorial, hemos aprendido cómo crear formularios en Django y cómo procesarlos en las vistas. Los formularios son una parte esencial de cualquier aplicación web y nos permiten interactuar con los usuarios de manera efectiva. En el próximo tutorial, veremos cómo validar los datos de un formulario y mostrar mensajes de error en caso de que haya algún problema.

## 69 Vistas, Templates y Rutas en Django

En este capítulo aprenderás a implementar las **vistas**, **templates** y **rutas** principales de una aplicación de farmacia en Django. Estos pasos son esenciales para que la aplicación sea funcional y permita operaciones **CRUD** básicas sobre los medicamentos.

### 69.1 Modelos y Formularios

Para este tutorial, se reutilizan los siguientes modelos y formularios:

#### Tip

- Modelos (**models.py**)

```
from django.db import models

class Medicamento(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    existencias = models.IntegerField()
    lugar = models.CharField(max_length=100)
    fecha = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.nombre

class Venta(models.Model):
    medicamento = models.ForeignKey(Medicamento, on_delete=models.CASCADE)
    cantidad = models.IntegerField()
    fecha = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.medicamento.nombre} - {self.cantidad} unidades"
```

#### Tip

- Formularios (**forms.py**)

```
from django import forms
from .models import Medicamento, Venta

class ContactForm(forms.Form):
    nombre = forms.CharField(label='Nombre', max_length=100)
    email = forms.EmailField(label='Email', max_length=100)
    mensaje = forms.EmailField(label='Mensaje', widget=forms.Textarea)

class MedicamentoForm(forms.ModelForm):
    class Meta:
        model = Medicamento
        fields = ['nombre', 'precio', 'existencias', 'lugar']

class VentaForm(forms.ModelForm):
    class Meta:
        model = Venta
        fields = ['medicamento', 'cantidad']
from django import forms
from .models import Medicamento
```

# 70 Paso 1: Mostrar la Lista de Medicamentos

## 70.1 Vista

- Agrega la siguiente función en `views.py`:

```
from django.shortcuts import render ①
from .models import Medicamento ②

def lista_medicamentos(request): ③
    medicamentos = Medicamento.objects.all() ④
    return render(request, 'farmacia/lista_medicamentos.html', {'medicamentos': medicamentos})
```

- ① Importa la función `render`.
- ② Importa el modelo `Medicamento`.
- ③ Define la función `lista_medicamentos`.
- ④ Obtiene todos los medicamentos.
- ⑤ Renderiza el template `lista_medicamentos.html` con la lista de medicamentos.

En esta función, se obtienen todos los medicamentos de la base de datos y se pasan al template para mostrarlos en una lista.

## 70.2 Template

- Crea el archivo `templates/farmacia/lista_medicamentos.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Medicamentos</title>
</head>
<body>
    <h1>Lista de Medicamentos</h1>
    <a href="{% url 'crear_medicamento' %}">Añadir Medicamento</a> ①
    <ul>
        {% for medicamento in medicamentos %} ②
        <li>
            {{ medicamento.nombre }} - ${{ medicamento.precio }} ③
        </li>
    </ul>
</body>
</html>
```

```

        ({{ medicamento.existencias }} disponibles) ④
        <a href="{% url 'detalle_medicamento' medicamento.id %}">Detalles</a> ⑤
        <a href="{% url 'editar_medicamento' medicamento.id %}">Editar</a> ⑥
        <a href="{% url 'eliminar_medicamento' medicamento.id %}">Eliminar</a> ⑦
    </li>
    {% endfor %} ⑧
</ul>
</body>
</html>

```

- ① Enlace para crear un nuevo medicamento.
- ② Itera sobre la lista de medicamentos.
- ③ Muestra el nombre y precio del medicamento.
- ④ Muestra las existencias disponibles.
- ⑤ Enlace para ver los detalles del medicamento.
- ⑥ Enlace para editar el medicamento.
- ⑦ Enlace para eliminar el medicamento.

En este template, se muestra una lista de medicamentos con enlaces para ver los detalles, editar y eliminar

## 70.3 Ruta

- En el archivo `urls.py` de la aplicación:

```

from django.urls import path
from . import views ①

urlpatterns = [
    path('medicamentos/', views.lista_medicamentos, name='lista_medicamentos'), ②
]

```

- ① Importa las vistas.
- ② Define la ruta para mostrar la lista de medicamentos.

En esta ruta, se llama a la función `lista_medicamentos` para mostrar la lista de medicamentos en la URL `/medicamentos/`.

# 71 Paso 2: Mostrar el Detalle de un Medicamento

## 71.1 Vista

- Agrega esta función a `views.py`:

```
from django.shortcuts import get_object_or_404 ①

def detalle_medicamento(request, id): ②
    medicamento = get_object_or_404(Medicamento, id=id) ③
    return render(request, 'farmacia/detalle_medicamento.html', {'medicamento': medicamento})
```

- ① Importa la función `get_object_or_404`.
- ② Define la función `detalle_medicamento`.
- ③ Obtiene el medicamento con el ID especificado.

En esta función, se obtiene un medicamento específico por su ID y se pasa al template para mostrar sus detalles.

## 71.2 Template

- Crea el archivo `templates/farmacia/detalle_medicamento.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Detalle del Medicamento</title>
</head>
<body>
    <h1>{{ medicamento.nombre }}</h1> ①
    <p><strong>Precio:</strong> ${{ medicamento.precio }}</p> ②
    <p><strong>Existencias:</strong> {{ medicamento.existencias }}</p> ③
    <p><strong>Lugar:</strong> {{ medicamento.lugar }}</p> ④
    <a href="{% url 'lista_medicamentos' %}">Volver a la lista</a> ⑤
</body>
</html>
```

- ① Muestra el nombre del medicamento.
- ② Muestra el precio del medicamento.

- ③ Muestra las existencias disponibles.
- ④ Muestra el lugar del medicamento.
- ⑤ Enlace para volver a la lista de medicamentos.

En este template, se muestran los detalles de un medicamento específico.

## 71.3 Ruta

- En `urls.py`:

```
path('medicamentos/<int:id>/', views.detalle_medicamento, name='detalle_medicamento'), ①
```

- ① Define la ruta para mostrar el detalle de un medicamento específico.

En esta ruta, se llama a la función `detalle_medicamento` para mostrar los detalles de un medicamento en la URL `/medicamentos/id/`.

## 72 Paso 3: Crear un Medicamento

### 72.1 Vista

```
from django.shortcuts import redirect ①
from .forms import MedicamentoForm ②

def crear_medicamento(request): ③
    if request.method == 'POST': ④
        form = MedicamentoForm(request.POST) ⑤
        if form.is_valid(): ⑥
            form.save() ⑦
            return redirect('lista_medicamentos') ⑧
    else:
        form = MedicamentoForm() ⑨
    return render(request, 'farmacia/crear_medicamento.html', {'form': form}) ⑩
```

- ① Importa la función **redirect**.
- ② Importa el formulario **MedicamentoForm**.
- ③ Define la función **crear\_medicamento**.
- ④ Verifica si la petición es de tipo POST.
- ⑤ Crea un formulario con los datos de la petición.
- ⑥ Verifica si el formulario es válido.
- ⑦ Guarda el medicamento en la base de datos.
- ⑧ Redirige a la lista de medicamentos.
- ⑨ Crea un formulario vacío.
- ⑩ Renderiza el template **crear\_medicamento.html** con el formulario.

En esta función, se crea un nuevo medicamento a partir de los datos enviados por el usuario y se guarda en la base de datos.

### 72.2 Template

- Crea **templates/farmacia/crear\_medicamento.html**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Crear Medicamento</title>
```

```

</head>
<body>
  <h1>Crear Medicamento</h1>
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Guardar</button>
  </form>
  <a href="{% url 'lista_medicamentos' %}">Volver a la lista</a>
</body>
</html>

```

- ① Formulario para crear un nuevo medicamento.
- ② Token de seguridad.
- ③ Campos del formulario.
- ④ Botón para guardar el medicamento.
- ⑤ Enlace para volver a la lista de medicamentos.

En este template, se muestra un formulario para crear un nuevo medicamento.

## 72.3 Ruta

```

path('medicamentos/nuevo/', views.crear_medicamento, name='crear_medicamento'), ①

```

- ① Define la ruta para crear un nuevo medicamento.

En esta ruta, se llama a la función **crear\_medicamento** para mostrar el formulario de creación de medicamentos en la URL `/medicamentos/nuevo/`.

## 73 Paso 4: Editar un Medicamento

### 73.1 Vista

```
def editar_medicamento(request, id): ①
    medicamento = get_object_or_404(Medicamento, id=id) ②
    if request.method == 'POST': ③
        form = MedicamentoForm(request.POST, instance=medicamento) ④
        if form.is_valid(): ⑤
            form.save() ⑥
            return redirect('lista_medicamentos') ⑦
    else:
        form = MedicamentoForm(instance=medicamento) ⑧
    return render(request, 'farmacia/editar_medicamento.html', {'form': form}) ⑨
```

- ① Define la función **editar\_medicamento**.
- ② Obtiene el medicamento con el ID especificado.
- ③ Verifica si la petición es de tipo POST.
- ④ Crea un formulario con los datos del medicamento.
- ⑤ Verifica si el formulario es válido.
- ⑥ Guarda los cambios en el medicamento.
- ⑦ Redirige a la lista de medicamentos.
- ⑧ Crea un formulario con los datos del medicamento.
- ⑨ Renderiza el template **editar\_medicamento.html** con el formulario.

En esta función, se edita un medicamento existente a partir de los datos enviados por el usuario y se guardan los cambios en la base de datos.

### 73.2 Template

- Crea **templates/farmacia/editar\_medicamento.html** con el mismo diseño que el formulario de creación.

### 73.3 Ruta

```
path('medicamentos/editar/<int:id>/', views.editar_medicamento, name='editar_medicamento')
```

- ① Define la ruta para editar un medicamento existente.

## 74 Paso 5: Eliminar un Medicamento

### 74.1 Vista

```
def eliminar_medimento(request, id):  
    medicamento = get_object_or_404(Medicamento, id=id)  
    medicamento.delete()  
    return redirect('lista_medicamentos')
```

- ① Define la función **eliminar\_medimento**.
- ② Obtiene el medicamento con el ID especificado.
- ③ Elimina el medicamento de la base de datos.
- ④ Redirige a la lista de medicamentos.

En esta función, se elimina un medicamento existente de la base de datos.

### 74.2 Ruta

```
path('medicamentos/eliminar/<int:id>/', views.eliminar_medimento, name='eliminar_medica
```

- ① Define la ruta para eliminar un medicamento existente.

En esta ruta, se llama a la función **eliminar\_medimento** para eliminar un medicamento en la URL `/medicamentos/eliminar/id/`.

## 75 Ejecutar el Proyecto

- Inicia el servidor:

```
python manage.py runserver
```

- Visita las rutas mencionadas para probar cada funcionalidad. Con estos pasos, tendrás un CRUD básico funcionando en tu aplicación Django.

Podemos agregar vistas, templates y rutas adicionales para implementar más funcionalidades en la aplicación de farmacia, como la creación de ventas, la generación de reportes, etc.

## 76 Sistema de Herencia de PlantiLLas, Tailwindcss y Archivos Estáticos.

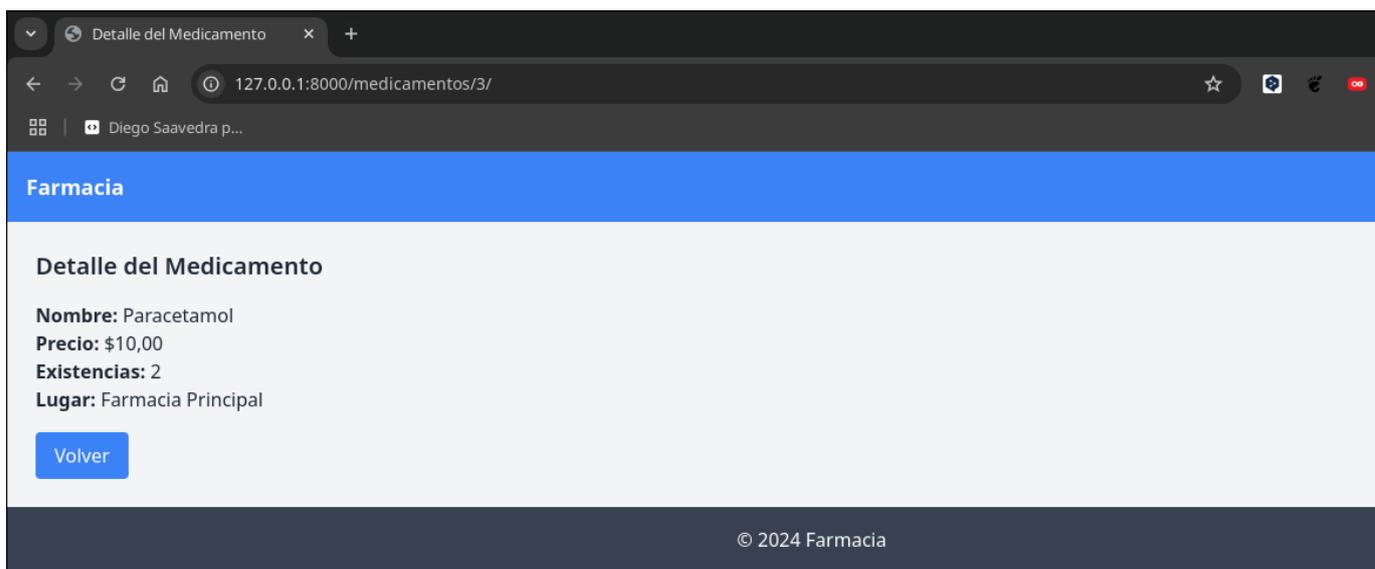


Figure 76.1: Django

En este capítulo, vamos a mejorar la estructura de plantillas de nuestra aplicación de Farmacia, aprovechando el sistema de herencia de plantillas de Django, la integración de Tailwind CSS, la generación del directorio static/ y su configuración en settings.py.

El objetivo es mejorar el diseño de nuestras plantillas HTML mediante el uso de Tailwind CSS y organizar mejor el código, aprovechando la herencia de plantillas. También aseguraremos que todos los archivos estáticos se generen correctamente.

## 77 1. Herencia de Plantillas

En Django, podemos usar un sistema de herencia de plantillas para evitar repetir el mismo código en varias vistas. En este caso, crearemos un archivo base, `base.html`, que servirá como plantilla común para todas nuestras páginas. Las otras plantillas heredarán de este archivo.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>{% block title %}Farmacia{% endblock %}</title>
    {% load static %}
    <link href="{% static 'css/output.css' %}" rel="stylesheet" />
  </head>
  <body class="bg-gray-100 text-gray-800 min-h-screen flex flex-col">
    <header class="bg-blue-500 text-white p-4">
      <h1 class="text-lg font-bold">Farmacia</h1>
    </header>
    <main class="flex-grow p-6">{% block content %}{% endblock %}</main>
    <footer class="bg-gray-700 text-white text-center p-4 mt-auto">
      <p>&copy; 2024 Farmacia</p>
    </footer>
  </body>
</html>
```

Este archivo contiene las partes comunes de la página, como el encabezado, el pie de página y el bloque de contenido principal, que puede ser reemplazado por cada plantilla específica.

### 77.1 Otras Plantillas

Ahora vamos a modificar cada una de las plantillas para que hereden de `base.html` y reemplacen el bloque de contenido principal con su propio contenido.

#### 77.1.1 1. crear\_medimento.html:

```

{% extends 'base.html' %} {% block title %}Nuevo Medicamento{% endblock %}
{% block content %}
<h2 class="text-xl font-semibold mb-4">Nuevo Medicamento</h2>

{% if messages %}
<div class="mb-4">
  {% for message in messages %} {% if message.tags == 'error' %}
  <div class="bg-red-500 text-white p-3 rounded mb-2">{{ message }}</div>
  {% else %}
  <div class="bg-green-500 text-white p-3 rounded mb-2">{{ message }}</div>
  {% endif %} {% endfor %}
</div>
{% endif %}

<form method="POST">
  {% csrf_token %} {{ form.as_p }}

  <button
    type="submit"
    class="bg-green-500 text-white py-2 px-4 rounded hover:bg-green-600"
  >
    Guardar
  </button>
</form>

<a
  href="{% url 'lista_medicamentos' %}"
  class="mt-4 inline-block bg-blue-500 text-white py-2 px-4 rounded hover:bg-blue-600"
>
  Volver
</a>
{% endblock %}

```

En este archivo, hemos reemplazado el bloque de contenido principal con el formulario para crear un nuevo medicamento. También hemos agregado un bloque para mostrar mensajes de error o éxito.

## 77.2 2. detalle\_medicamento.html:

```

{% extends 'base.html' %} {% block title %}Detalle del Medicamento{% endblock %}
{% block content %}
<h2 class="text-xl font-semibold mb-4">Detalle del Medicamento</h2>
<p><strong>Nombre:</strong> {{ medicamento.nombre }}</p>
<p><strong>Precio:</strong> ${{ medicamento.precio }}</p>
<p><strong>Existencias:</strong> {{ medicamento.existencias }}</p>

```

```

<p><strong>Lugar:</strong> {{ medicamento.lugar }}</p>
<a
  href="{% url 'lista_medicamentos' %}"
  class="mt-4 inline-block bg-blue-500 text-white py-2 px-4 rounded hover:bg-blue-600"
  >Volver</a
>
{% endblock %}

```

En este archivo, hemos reemplazado el bloque de contenido principal con los detalles del medicamento y un enlace para volver a la lista de medicamentos.

### 77.3 3. lista\_medicamentos.html:

```

{% extends 'base.html' %} {% block title %}Lista de Medicamentos{% endblock %}
{% block content %}
<h2 class="text-xl font-semibold mb-4">Lista de Medicamentos</h2>
<table class="table-auto w-full border-collapse border border-gray-300">
  <thead>
    <tr class="bg-gray-200">
      <th class="border border-gray-300 px-4 py-2">Nombre</th>
      <th class="border border-gray-300 px-4 py-2">Precio</th>
      <th class="border border-gray-300 px-4 py-2">Acciones</th>
    </tr>
  </thead>
  <tbody>
    {% for medicamento in medicamentos %}
    <tr>
      <td class="border border-gray-300 px-4 py-2">{{ medicamento.nombre }}</td>
      <td class="border border-gray-300 px-4 py-2">
        ${{ medicamento.precio }}
      </td>
      <td class="border border-gray-300 px-4 py-2">
        <a
          href="{% url 'detalle_medicamento' medicamento.id %}"
          class="text-blue-500 hover:underline"
          >Ver</a>
        >
        <a
          href="{% url 'editar_medicamento' medicamento.id %}"
          class="text-yellow-500 hover:underline"
          >Editar</a>
        >
        <a
          href="{% url 'eliminar_medicamento' medicamento.id %}"
          class="text-red-500 hover:underline"

```

```

        >Eliminar</a>
    >
    </td>
</tr>
    {% endfor %}
</tbody>
</table>
<a
  href="{% url 'crear_medicamento' %}"
  class="mt-4 inline-block bg-green-500 text-white py-2 px-4 rounded hover:bg-green-600"
  >Crear Medicamento</a>
>
{% endblock %}

```

En este archivo, hemos reemplazado el bloque de contenido principal con una tabla que muestra la lista de medicamentos y enlaces para ver, editar o eliminar cada medicamento.

#### 77.4 4. editar\_medicamento.html:

```

{% extends 'base.html' %} {% block title %}Editar Medicamento{% endblock %}
{% block content %}
<h2 class="text-2xl font-semibold mb-4">Editar Medicamento</h2>
<form method="post" class="bg-white p-6 rounded shadow-md">
  {% csrf_token %} {{ form.as_p }}
  <button
    type="submit"
    class="bg-blue-500 text-white py-2 px-4 rounded hover:bg-blue-600"
  >
    Guardar
  </button>
</form>
<a
  href="{% url 'lista_medicamentos' %}"
  class="text-blue-500 hover:underline mt-4 inline-block"
  >Volver a la lista</a>
>
{% endblock %}

```

#### 77.5 4. eliminar\_medicamento.html:

```

{% extends 'base.html' %} {% block title %}Confirmar Eliminación{% endblock %}
{% block content %}
<h2 class="text-xl font-semibold text-red-600 mb-4">

```

```

    ¿Estás seguro de que deseas eliminar este medicamento?
</h2>
<p class="mb-4"><strong>{{ medicamento.nombre }}</strong></p>
<form method="post">
    {% csrf_token %}
    <button
        type="submit"
        class="bg-red-500 text-white py-2 px-4 rounded hover:bg-red-600"
    >
        Eliminar
    </button>
    <a
        href="{% url 'lista_medicamentos' %}"
        class="ml-4 bg-gray-500 text-white py-2 px-4 rounded hover:bg-gray-600"
    >Cancelar</a>
    >
</form>
{% endblock %}

```

En este archivo, hemos reemplazado el bloque de contenido principal con un mensaje de confirmación para eliminar un medicamento y botones para confirmar o cancelar la acción.

## 78 2. Integración de Tailwind CSS

Tailwind CSS es un framework de diseño de bajo nivel que facilita la creación de interfaces elegantes sin necesidad de escribir CSS desde cero. Vamos a configurarlo y utilizarlo en nuestras plantillas.

### Tip

Es necesario tener nodejs instalado en tu sistema para poder instalar Tailwind CSS. Para instalar nodejs, puedes seguir las instrucciones en la [página oficial de nodejs](#).

### 78.1 2.1 Instalación de Tailwind CSS

Lo primero que necesitamos es instalar Tailwind CSS globalmente en nuestro sistema. Para ello, utilizaremos npm, el gestor de paquetes de Node.js.

- Instala Tailwind CSS globalmente utilizando npm:

```
npm install -g tailwindcss
```

Ahora es necesario generar un archivo de configuración de Tailwind CSS. Para ello, ejecutamos el siguiente comando en la raíz de nuestro proyecto:

```
npx tailwindcss init
```

Con el comando anterior se generará un archivo **tailwind.config.js** en la raíz de nuestro proyecto. Este archivo es necesario para configurar Tailwind CSS y definir las rutas de los archivos HTML y CSS en los que se utilizarán las clases de Tailwind.

### 78.2 2.2 Configuración de Tailwind en Django

Ahora, configuramos Tailwind CSS en nuestra aplicación Django. Los archivos generados estarán dentro de la carpeta static, por lo que debemos asegurarnos de que Django sirva estos archivos correctamente.

**static/css/style.css**

- Aquí configuramos Tailwind para que lo utilice en nuestras plantillas:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

## 78.3 2.3 Configuración del archivo tailwind.config.js

Este archivo es necesario para configurar las rutas de las plantillas HTML y los archivos CSS en los que Tailwind debe buscar clases CSS dinámicas.

```
module.exports = {
  content: [
    "./farmacia/templates/**/*.html", // Ruta a tus plantillas HTML
    "./static/js/**/*.js", // Si usas JS que contiene clases dinámicas
    "./static/css/**/*.css", // Si tienes más archivos CSS
  ],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

## 79 3. Generación del archivo static

Debemos asegurarnos de que el archivo `output.css` se genere correctamente. Para ello, primero necesitamos configurar y ejecutar Tailwind para generar el archivo CSS final.

### 79.1 3.1 Configuración en `settings.py`

En `settings.py`, aseguramos que Django pueda servir archivos estáticos correctamente. Aquí está una parte de la configuración relevante:

```
# settings.py
import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent
'''
Configuración de templates
'''
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'farmacia/templates/farmacia')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

'''
Configuración de archivos estáticos
'''

STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / 'static']
```

## 79.2 3.2 Generación de output.css

Ahora, ejecutamos Tailwind para compilar nuestro archivo CSS:

```
tailwindcss -i ./static/css/style.css -o ./static/css/output.css --watch
```

Este comando toma el archivo **style.css**, lo procesa con Tailwind y genera el archivo **output.css** que se incluirá en nuestras plantillas HTML.

## 80 4. Estructura de Archivos Final

Tu estructura de archivos debe verse así:

```
farmacia
  templates
    farmacia
      base.html
      crear_medicamento.html
      detalle_medicamento.html
      editar_medicamento.html
      eliminar_medicamento.html
      lista_medicamentos.html
  static
    css
      output.css
      style.css
  tailwind.config.js
  manage.py
  requirements.txt
  settings.py
```

# 81 Conclusión

Con esta configuración hemos logrado:

- Usar herencia de plantillas para evitar la repetición de código en las páginas de la aplicación.
- Integrar Tailwind CSS para mejorar el diseño de las páginas.
- Configurar archivos estáticos en Django y generar el archivo CSS utilizando Tailwind.

Esta configuración asegura una aplicación con un diseño moderno y bien estructurado, facilitando la reutilización de código y mejorando la experiencia del desarrollo en Django.

## 82 Introducción a Django REST Framework

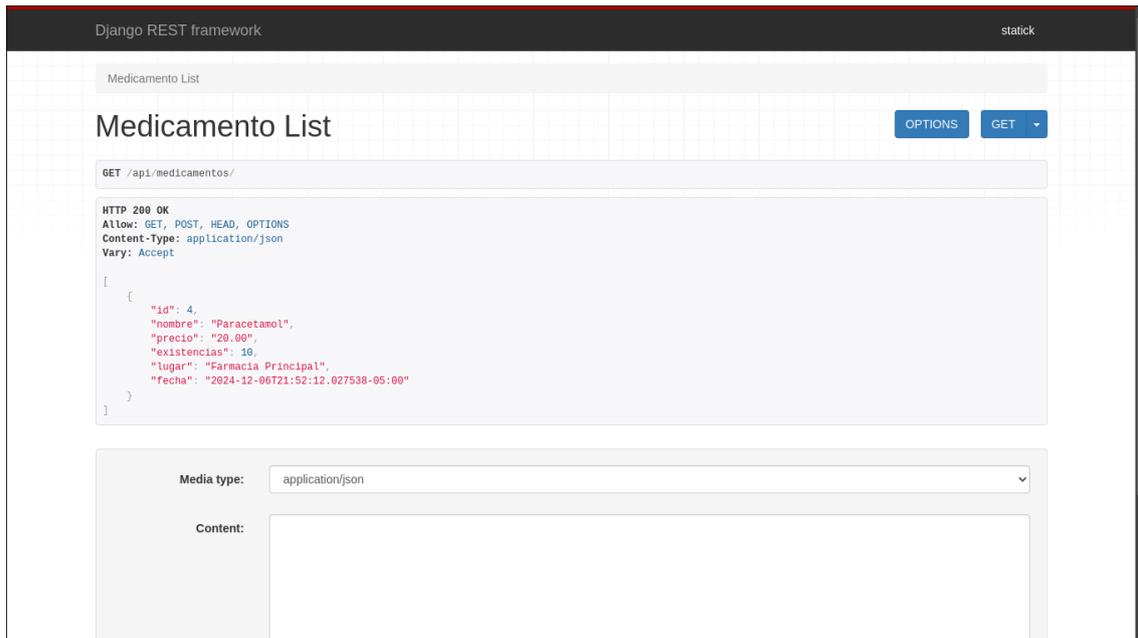


Figure 82.1: Django REST Framework

Django REST Framework es una biblioteca poderosa y flexible para construir APIs web en Django. DRF incluye funcionalidades como:

### 82.1 Serialización de modelos

- Vistas basadas en clases para manejar endpoints
- Autenticación y permisos
- Documentación interactiva

#### 82.1.1 2. Configuración Inicial

- Instalar Django REST Framework:

```
pip install djangorestframework
```

- Agregar DRF a `INSTALLED_APPS` en `settings.py`:

```
INSTALLED_APPS = [
    # Otros apps...
    'rest_framework',
]
```

- Configurar DRF (opcional): En settings.py, puedes definir configuraciones personalizadas:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ],
}
```

- ① **REST\_FRAMEWORK**: Configuración principal de DRF.
- ② **DEFAULT\_AUTHENTICATION\_CLASSES**: Clases de autenticación predeterminadas.
- ③ **SessionAuthentication**: Autenticación basada en sesiones.
- ④ **DEFAULT\_PERMISSION\_CLASSES**: Clases de permisos predeterminadas.
- ⑤ **IsAuthenticatedOrReadOnly**: Permite a los usuarios autenticados realizar operaciones de escritura.

### 82.1.2 3. Crear la API

- Crear un serializador para el modelo Medicamento:

```
from rest_framework import serializers
from .models import Medicamento

class MedicamentoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Medicamento
        fields = '__all__'
```

- ① **serializers**: Módulo de DRF para serializar y deserializar datos.
- ② **Medicamento**: Modelo de Django.
- ③ **MedicamentoSerializer**: Clase para serializar el modelo Medicamento.
- ④ **Meta**: Clase interna para configurar el serializador.
- ⑤ **model**: Modelo que se serializará.
- ⑥ **fields**: Campos del modelo que se incluirán en la serialización.

### 💡 Tip

Al serializar los modelos de Django, puedes especificar los campos que deseas incluir en la respuesta JSON. En este caso, `fields = '__all__'` incluye todos los campos del modelo Medicamento. Este proceso de serizlización se conoce permite convertir los datos de un modelo en un formato que se puede enviar a través de la red, como JSON.

- Crear vistas para las APIs:

```
from .models import Medicamento ①
from .serializers import MedicamentoSerializer ②
from rest_framework import viewsets ③

class MedicamentoView(viewsets.ModelViewSet): ④
    serializer_class = MedicamentoSerializer ⑤
    queryset = Medicamento.objects.all() ⑥
```

- ① **Medicamento**: Modelo de Django.
- ② **MedicamentoSerializer**: Serializador para el modelo Medicamento.
- ③ **viewsets**: Módulo de DRF para vistas basadas en conjuntos.
- ④ **MedicamentoView**: Vista para el modelo Medicamento.
- ⑤ **serializer\_class**: Clase de serializador para la vista.
- ⑥ **queryset**: Conjunto de datos de medicamentos.

Con estas vistas basadas en conjuntos, DRF proporciona automáticamente las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para el modelo Medicamento.

- Configurar las rutas de la API: En `urls.py`:

```
from django.urls import path, include ①
from rest_framework.routers import DefaultRouter ②
from .views import MedicamentoView ③

router = DefaultRouter() ④
router.register('medicamentos', MedicamentoView) ⑤

urlpatterns = [ ⑥
    path('api/', include(router.urls)), ⑦
]
```

- ① **include**: Función para incluir rutas de Django.
- ② **DefaultRouter**: Enrutador de DRF para vistas basadas en conjuntos.
- ③ **MedicamentoView**: Vista para el modelo Medicamento.
- ④ **router**: Enrutador predeterminado de DRF.
- ⑤ **router.register**: Registra la vista MedicamentoView en el enrutador.
- ⑥ **urlpatterns**: Lista de rutas de Django.
- ⑦ **include(router.urls)**: Incluye las rutas del enrutador en la ruta `/api/`.

### 82.1.3 4. Documentación con Swagger

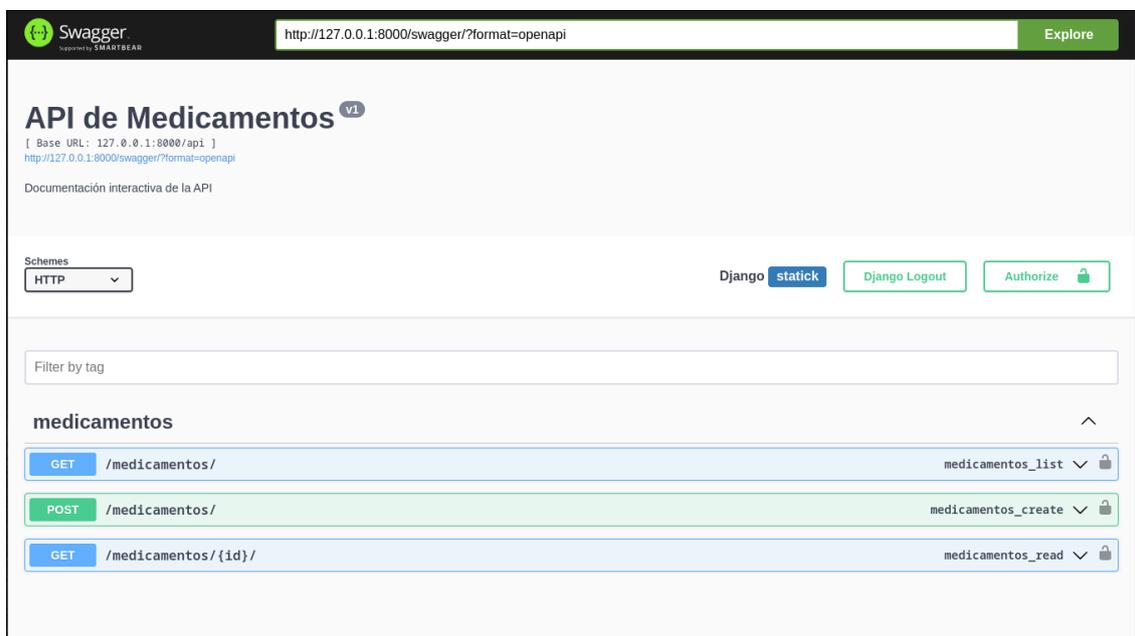


Figure 82.2: Swagger

Swagger es una herramienta para documentar APIs de forma interactiva. Puedes integrar Swagger en tu proyecto de Django REST Framework para proporcionar una documentación clara y detallada de tus APIs.

- Instalar dependencias para Swagger:

```
pip install drf-yasg
```

- Configurar Swagger en `urls.py`:

```
from rest_framework import permissions ①
from drf_yasg.views import get_schema_view ②
from drf_yasg import openapi ③

schema_view = get_schema_view( ④
    openapi.Info( ⑤
        title="API de Medicamentos", ⑥
        default_version='v1', ⑦
        description="Documentación interactiva de la API", ⑧
    ),
    public=True, ⑨
    permission_classes=(permissions.AllowAny,), ⑩
)

urlpatterns += [ ⑪
```

```
path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger')
]
```

- ① **permissions**: Módulo de DRF para permisos.
- ② **get\_schema\_view**: Función para obtener la vista de Swagger.
- ③ **openapi**: Módulo de drf-yasg para definir esquemas OpenAPI.
- ④ **schema\_view**: Vista de Swagger para tu API.
- ⑤ **Info**: Clase para definir información de la API.
- ⑥ **title**: Título de la API.
- ⑦ **default\_version**: Versión predeterminada de la API.
- ⑧ **description**: Descripción de la API.
- ⑨ **public**: Indica si la documentación es pública.
- ⑩ **permission\_classes**: Clases de permisos para acceder a la documentación.
- ⑪ **urlpatterns**: Lista de rutas de Django.
- ⑫ **schema\_view.with\_ui('swagger', cache\_timeout=0)**: Vista de Swagger con interfaz interactiva.

En el archivo `settings.py`, agrega `'drf_yasg'` a `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    # Otros apps...  
    'drf_yasg',  
]
```

- ① **drf\_yasg**: App de Django para integrar Swagger.

Ver la documentación interactiva: Inicia el servidor y ve a <http://localhost:8000/swagger/>. Aquí podrás explorar y probar los endpoints de tu API directamente desde el navegador.

## Tip

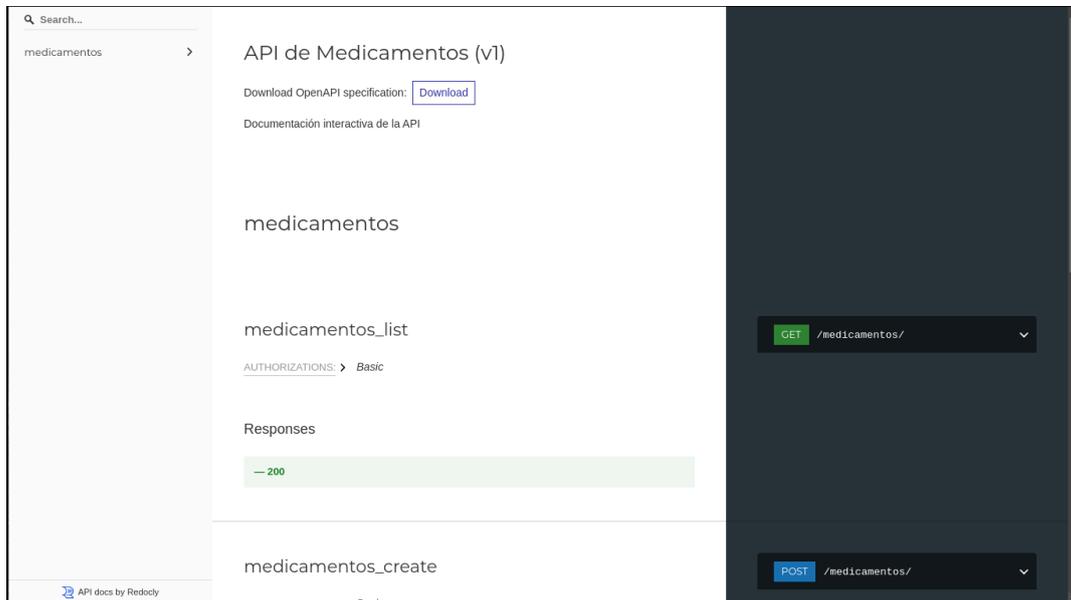


Figure 82.3: Redoc

Otra documentación popular para APIs es **Redoc**, que puedes integrar en tu proyecto de DRF siguiendo los pasos en la documentación oficial de [drf-yasg](#). Para que funcione en nuestro proyecto en el archivo `urls.py` solo hace falta agregar la siguiente línea:

```
path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
```

### 82.1.4 5. Pruebas con Thunder Client

Thunder Client es una extensión de VS Code para probar APIs similar a Postman, pero integrada en el editor.

1. Instalar Thunder Client:
  - Abre VS Code.
  - Ve a la pestaña de extensiones (Ctrl+Shift+X).
  - Busca Thunder Client e instálalo.
2. Crear una colección de pruebas:
  - Abre Thunder Client desde la barra lateral.
  - Haz clic en New Request y selecciona el método (GET, POST, etc.).
  - Ingresa la URL del endpoint, por ejemplo: `http://localhost:8000/api/medicamentos/`.
3. Enviar una solicitud GET:

- Configura la URL y haz clic en Send.
- Verifica la respuesta en el panel de resultados.

#### 4. Prueba de solicitudes POST:

- Cambia el método a POST y selecciona el cuerpo (Body).
- Ingresa un JSON como este:

```
{
  "nombre": "Amoxicilina",
  "precio": 20.5,
  "existencias": 30,
  "lugar": "Almacén C"
}
```

- Haz clic en Send para verificar si el servidor crea correctamente un nuevo medicamento.

### 82.1.5 6. Pruebas Automatizadas de las APIs

- Agrega pruebas unitarias para validar los endpoints.

```
from rest_framework.test import APITestCase ①
from rest_framework import status ②
from .models import Medicamento ③

class MedicamentoAPITests(APITestCase): ④
    def setUp(self): ⑤
        self.medicamento = Medicamento.objects.create( ⑥
            nombre="Paracetamol", ⑦
            precio=10, ⑧
            existencias=50, ⑨
            lugar="Almacén A" ⑩
        )

    def test_lista_medicamentos(self): ⑪
        response = self.client.get('/api/medicamentos/') ⑫
        self.assertEqual(response.status_code, status.HTTP_200_OK) ⑬

    def test_crear_medicamento(self): ⑭
        data = { ⑮
            "nombre": "Ibuprofeno", ⑯
            "precio": 15, ⑰
            "existencias": 30, ⑱
            "lugar": "Almacén B" ⑲
        }
}
```

```

response = self.client.post('/api/medicamentos/', data) ②0
self.assertEqual(response.status_code, status.HTTP_201_CREATED) ②1
self.assertEqual(response.data['nombre'], data['nombre']) ②2
self.assertEqual(response.data['precio'], data['precio']) ②3
self.assertEqual(response.data['existencias'], data['existencias']) ②4
self.assertEqual(response.data['lugar'], data['lugar']) ②5

def test_detalle_medicamento(self): ②6
    response = self.client.get(f'/api/medicamentos/{self.medicamento.id}/') ②7
    self.assertEqual(response.status_code, status.HTTP_200_OK) ②8
    self.assertEqual(response.data['nombre'], "Paracetamol") ②9

```

- ① **APITestCase**: Clase base para pruebas de API.
- ② **status**: Módulo de DRF para códigos de estado HTTP.
- ③ **Medicamento**: Modelo de Django.
- ④ **MedicamentoAPITests**: Clase de pruebas para la API de Medicamentos.
- ⑤ **setUp**: Método para configurar datos de prueba.
- ⑥ **Medicamento.objects.create**: Crea un medicamento en la base de datos.
- ⑦ **test\_lista\_medicamentos**: Prueba para listar medicamentos.
- ⑧ **self.client.get**: Realiza una solicitud GET a la API.
- ⑨ **test\_crear\_medicamento**: Prueba para crear un medicamento.
- ⑩ **data**: Datos del medicamento a crear.
- ⑪ **self.client.post**: Realiza una solicitud POST a la API.
- ⑫ **test\_detalle\_medicamento**: Prueba para obtener detalles de un medicamento.
- ⑬ **self.client.get**: Realiza una solicitud GET a la API.
- ⑭ **response.data**: Datos de la respuesta de la API.
- ⑮ **response.status\_code**: Código de estado de la respuesta.
- ⑯ **response.data['nombre']**: Nombre del medicamento en la respuesta.
- ⑰ **response.data['precio']**: Precio del medicamento en la respuesta.
- ⑱ **response.data['existencias']**: Existencias del medicamento en la respuesta.
- ⑲ **response.data['lugar']**: Lugar del medicamento en la respuesta.
- ⑳ **self.assertEqual**: Compara dos valores en la prueba.
- ㉑ **self.medicamento.id**: ID del medicamento creado en la base de datos.
- ㉒ **response.data['nombre']**: Nombre del medicamento en la respuesta.
- ㉓ **response.data['precio']**: Precio del medicamento en la respuesta.
- ㉔ **response.data['existencias']**: Existencias del medicamento en la respuesta.
- ㉕ **response.data['lugar']**: Lugar del medicamento en la respuesta.
- ㉖ **self.client.get**: Realiza una solicitud GET a la API.
- ㉗ **response.status\_code**: Código de estado de la respuesta.
- ㉘ **response.data['nombre']**: Nombre del medicamento en la respuesta.
- ㉙ **status.HTTP\_200\_OK**: Código de estado para una respuesta exitosa.

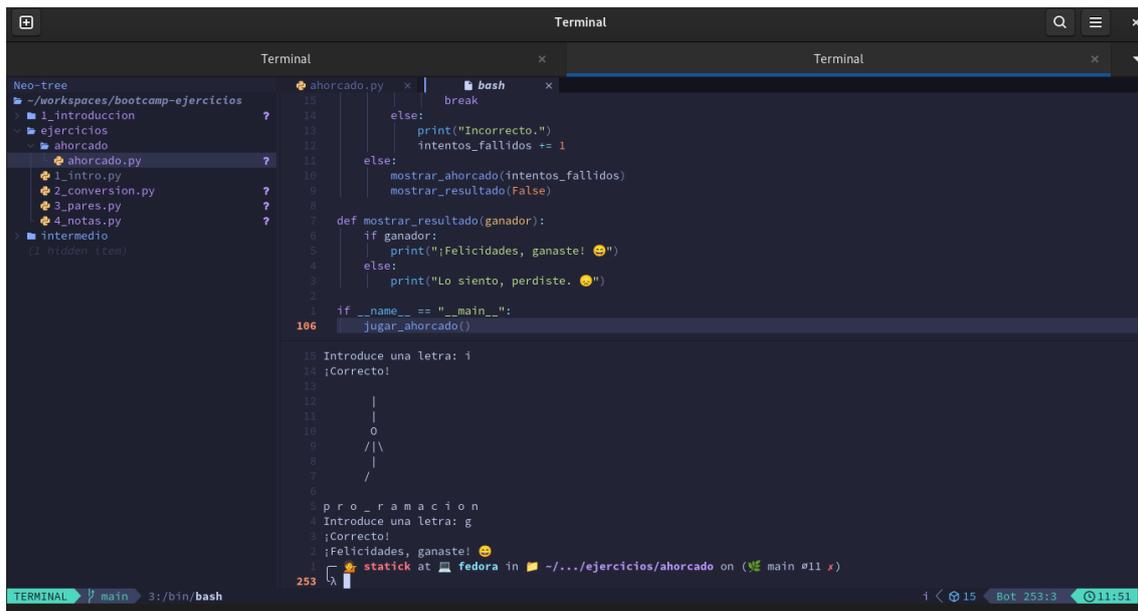
## **82.1.6 7. Conclusión**

Con Django REST Framework, Swagger y Thunder Client, puedes desarrollar, documentar y probar APIs de manera eficiente. Este flujo de trabajo asegura que tus APIs sean robustas, fáciles de usar y bien documentadas para cualquier desarrollador o cliente que las consuma.

**Part IX**

**Proyectos**

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



```
Neo-tree
~/workspaces/bootcamp-ejercicios
├── 1_introduccion
├── ejercicios
│   └── ahorcado
│       ├── ahorcado.py
│       ├── 1_intro.py
│       ├── 2_conversion.py
│       ├── 3_pares.py
│       └── 4_notas.py
└── intermedio
    (1 hidden item)

ahorcado.py
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("¡Felicidades, 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 p r o _ r a m a c i o n
4 Introduce una letra: g
3 ¡Correcto!
2 ¡Felicidades, ganaste! 🎉
1
statick at fedora in ~/.../ejercicios/ahorcado on (main #11 x)
253
```

Figure 83.1: Ahorcado

### 83.1 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.

### 83.2 Prerrequisitos

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

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

### 83.3.1 Crear un archivo de Python:

Abre tu editor de texto o IDE favorito (se recomienda utilizar Vscodex) 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
```

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

### 83.4.1 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
   /|\
    |
   / \
"""]
```

### 83.4.2 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.

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

### 83.5.1 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])
```

### 83.5.2 Prueba de la función:

Llama a `mostrar_ahorcado` varias veces con diferentes valores para verificar que cada etapa se muestra correctamente.

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

### 83.6.1 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.

### 83.6.2 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](#)

### 83.6.3 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_ahdivinadas):
    if letra in palabra:
        letras_ahdivinadas.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.

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

### 83.7.1 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_ahdivinadas = 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.

### 83.7.2 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_ahdivinadas)

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

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

```

    if intentar_letra(palabra, letra, letras_ahorcadas):
        print("¡Correcto!")
        if all(l in letras_ahorcadas 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.

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

### 83.8.1 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("¡Felicidades, 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.

## 83.9 Paso 7: Ejecutar el Juego

### 83.9.1 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.

### 83.9.2 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
```

## 83.10 Paso 8: Mejoras Opcionales

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

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

## 84 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_ahdivinadas):
    progreso = [letra if letra in letras_ahdivinadas else '_' for letra in palabra]
    print(" ".join(progreso))

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

def jugar_ahorcado():
    palabra = seleccionar_palabra()
    letras_ahdivinadas = 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("¡Felicidades, 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.

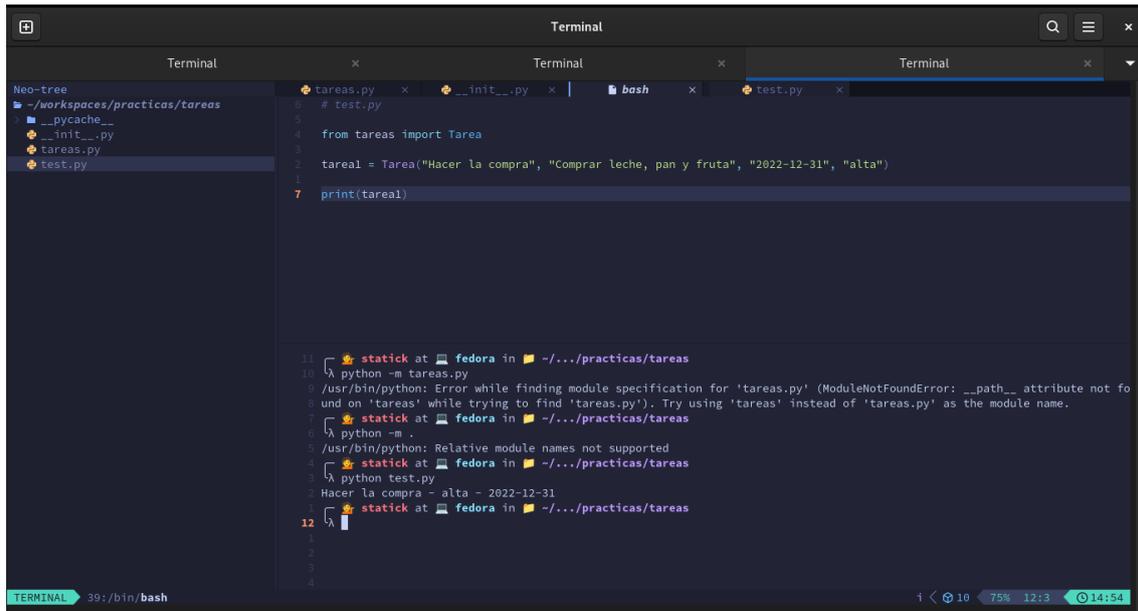
- **Personalizar Mensajes:** Cambia los mensajes de victoria y derrota para hacerlos más divertidos.
- **Agregar Sonidos:** Añade sonidos o efectos de sonido al juego para mejorar la experiencia del jugador.
- **Diseño Gráfico:** Crea un diseño gráfico más elaborado para el ahorcado y las letras adivinadas.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

## 85 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! ¡Felicidades por completar el laboratorio!

## 86 Gestor de Tareas con Prioridades



```
Neo-tree
~/workspaces/practicas/tareas
├── __pycache__
├── __init__.py
├── tareas.py
└── test.py

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

test.py
1
2
3
4
5
6
7
8
9
10
11
12

Terminal
~/workspaces/practicas/tareas
λ python -m tareas.py
/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.
λ python -m .
/usr/bin/python: Relative module names not supported
λ python test.py
Hacer la compra - alta - 2022-12-31
λ
```

Figure 86.1: Gestor de Tareas

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

### 86.1 Módulos del Proyecto

#### 86.1.1 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 .

### 86.2 Funciones Clave

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

## 86.2.1 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
```

## 87 Extra

- Añadir la funcionalidad de editar y eliminar tareas.

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

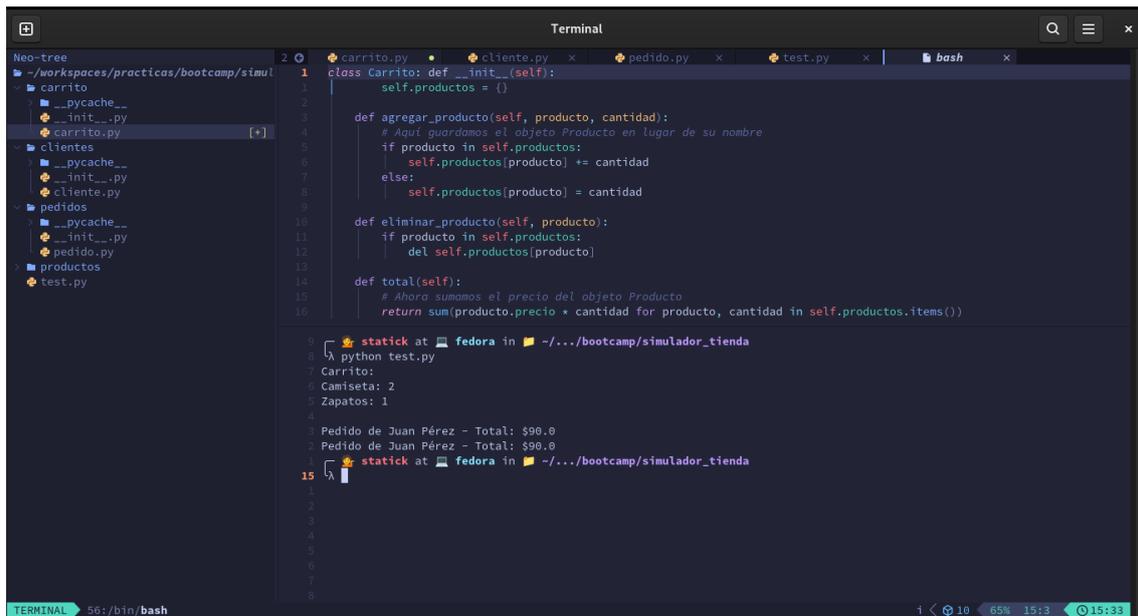
## 88 Conclusión

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.

## 89 Reto

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

## 90 Simulador de Tienda Online



```
Neo-tree
├─ /workspaces/practicas/bootcamp/simu
│   └─ carrito
│       ├── __pycache__
│       ├── __init__.py
│       └─ carrito.py
│   └─ clientes
│       ├── __pycache__
│       ├── __init__.py
│       └─ cliente.py
│   └─ pedidos
│       ├── __pycache__
│       ├── __init__.py
│       └─ pedido.py
│   └─ productos
│       └─ test.py
└─ Terminal
    class Carrito:
    def __init__(self):
        self.productos = {}

    def agregar_producto(self, producto, cantidad):
        # Aquí guardamos el objeto Producto en lugar de su nombre
        if producto in self.productos:
            self.productos[producto] += cantidad
        else:
            self.productos[producto] = cantidad

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

    def total(self):
        # Ahora sumamos el precio del objeto Producto
        return sum(producto.precio * cantidad for producto, cantidad in self.productos.items())

static at fedora in ~/../bootcamp/simulador_tienda
python test.py
Carrito:
Camiseta: 2
Zapatos: 1
Pedido de Juan Pérez - Total: $90.0
Pedido de Juan Pérez - Total: $90.0
static at fedora in ~/../bootcamp/simulador_tienda
```

Figure 90.1: 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.

### 90.1 Módulos del Proyecto

#### 90.1.1 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.

#### 90.1.2 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.

### **90.1.3 Módulo de Cliente**

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

### **90.1.4 Módulo de Pedido**

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

# 91 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.

## 91.1 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})"
```

## 91.2 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
```

## 91.3 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})"

```

## 91.4 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}"

```

## 92 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.

## 93 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:

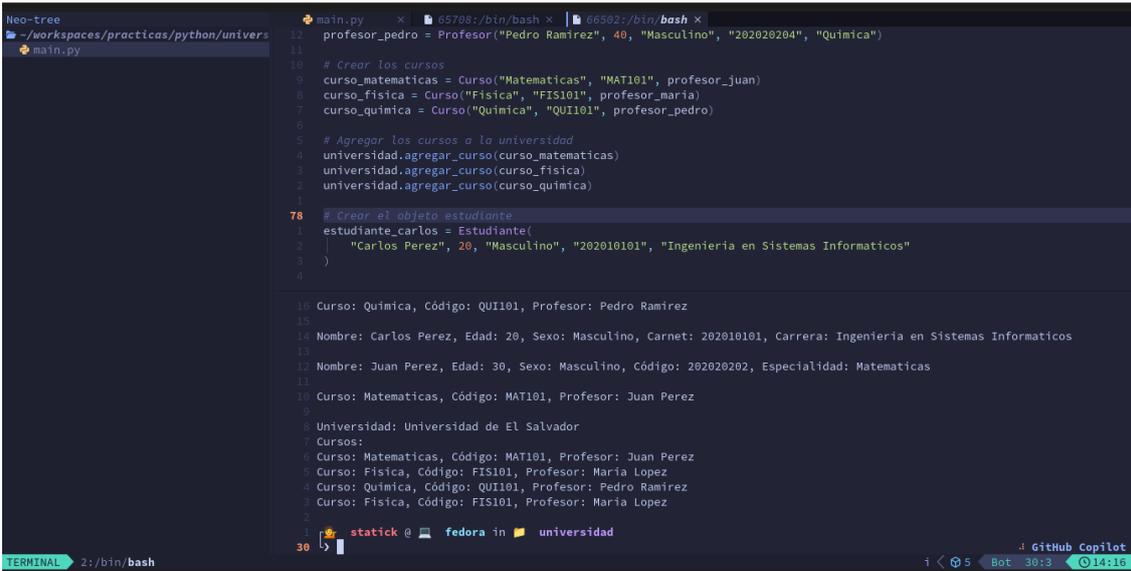
```
import tkinter as tk  
  
root = tk.Tk()  
  
label = tk.Label(root, text="¡Bienvenido a la Tienda Online!")  
label.pack()  
  
root.mainloop()
```

## 94 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!

## 95 Sistema Universitario



```
Neo-tree
- /workspaces/practicas/python/univers
  main.py
main.py
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_maria)
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
30 statick @ fedora in universidad
TERMINAL 2: /bin/bash
```

Figure 95.1: 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.

## 95.1 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.

## 95.2 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.

## 95.3 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.

2. 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 sobrescribe el método **str** para mostrar la información del estudiante.

3. 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 código y especialidad. Además, se sobrescribe el método **str** para mostrar la información del profesor.

4. 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.no

```

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. Car 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.

## 96 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.

## **97 Laboratorio: DevContainer con NGINX**

## 98 Objetivo:

Crear un entorno de desarrollo dentro de un contenedor Docker que ejecute NGINX para servir una página estática simple.

### 98.1 1. Estructura del Proyecto

El proyecto tendrá la siguiente estructura de directorios y archivos:

```
DevContainers/  
  
  .devcontainer/  
    Dockerfile  
    devcontainer.json  
  
  html/  
    index.html
```

Ahora vamos a crear el archivo **index.html/**

**html/index.html**

El archivo HTML que NGINX servirá cuando accedas a <http://localhost:80>.

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <title>Document</title>  
</head>  
<body>  
  <h1>Hello World!</h1>  
</body>  
</html>
```

Este es un archivo HTML simple que contiene un título y un encabezado.

Ahora vamos a crear el archivo **Dockerfile**.

**.devcontainer/Dockerfile**

Este archivo Dockerfile se utiliza para construir la imagen del contenedor que ejecutará NGINX.

```
# Usa una imagen base de Ubuntu
FROM mcr.microsoft.com/devcontainers/base:jammy

# Instala NGINX
RUN apt-get update && apt-get install -y nginx && apt-get clean

# Expone el puerto 80 para NGINX
EXPOSE 80

# Comando para iniciar NGINX en primer plano
CMD ["nginx", "-g", "daemon off;"]
```

El Dockerfile está basado en Ubuntu (jammy) e instala NGINX. Luego, expone el puerto 80 y configura NGINX para que se ejecute en primer plano (esto es necesario para que el contenedor no termine inmediatamente después de iniciarse).

Ahora crearemos el archivo **devcontainer.json**.

#### **.devcontainer/devcontainer.json**

El archivo de configuración del DevContainer, que especifica cómo se debe construir y configurar el contenedor.

```
{
  "name": "DevContainer with NGINX",
  "build": {
    "dockerfile": "Dockerfile"
  },
  "forwardPorts": [80],
  "postCreateCommand": "echo 'DevContainer with NGINX is ready!'",
  "remoteUser": "root",
  "mounts": [
    "source=${localWorkspaceFolder}/html,target=/usr/share/nginx/html,type=bind"
  ]
}
```

- **build:** Este campo indica que se debe usar el Dockerfile para construir la imagen del contenedor.
- **forwardPorts:** Mapea el puerto 80 del contenedor al puerto 80 de la máquina local para que NGINX sea accesible desde el navegador.
- **mounts:** Se vincula el directorio local html al directorio `/usr/share/nginx/html` del contenedor, para que el archivo **index.html** sea servido por NGINX.

## **98.2 3. Instrucciones de Creación y Ejecución**

**Crea el proyecto:** Crea un directorio de trabajo, por ejemplo DevContainers, y dentro de él agrega los archivos de configuración mencionados anteriormente.

**Abre el proyecto en Visual Studio Code:** Abre la carpeta DevContainers en Visual Studio Code.

**Construye el contenedor:** Al abrir el proyecto en VS Code, si tienes configurado DevContainers, automáticamente debería preguntar si deseas abrirlo en un contenedor. Selecciona “Reopen in Container”.

**Verifica la creación del contenedor:** El contenedor se construirá utilizando el Dockerfile y configurará NGINX automáticamente. El puerto 80 será accesible en tu máquina local.

**Accede a tu página web:** Una vez que el contenedor se haya iniciado, abre tu navegador y ve a <http://localhost:80>. Deberías ver la página con el mensaje “**Hello World!**”.

## 98.3 2. Archivos de Configuración

Es posible que tengamos que configurar el archivo `/etc/nginx/sites-available/default`

Este archivo de configuración de NGINX para servir los archivos desde la carpeta `/usr/share/nginx/html`.

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html; # Asegúrate de que esté apuntando a esta carpeta
    index index.html index.htm;

    location / {
        try_files $uri $uri/ =404; # Verifica que esta línea esté configurada para servi
    }
}
```

Este archivo configura el servidor para escuchar en el **puerto 80** y servir archivos desde la carpeta `/usr/share/nginx/html`.

Si el archivo solicitado no se encuentra, se mostrará un error 404.

## 98.4 Problemas Comunes

**NGINX no se inicia:** Si después de construir el contenedor NGINX no está corriendo, puedes iniciarlo manualmente con el siguiente comando:

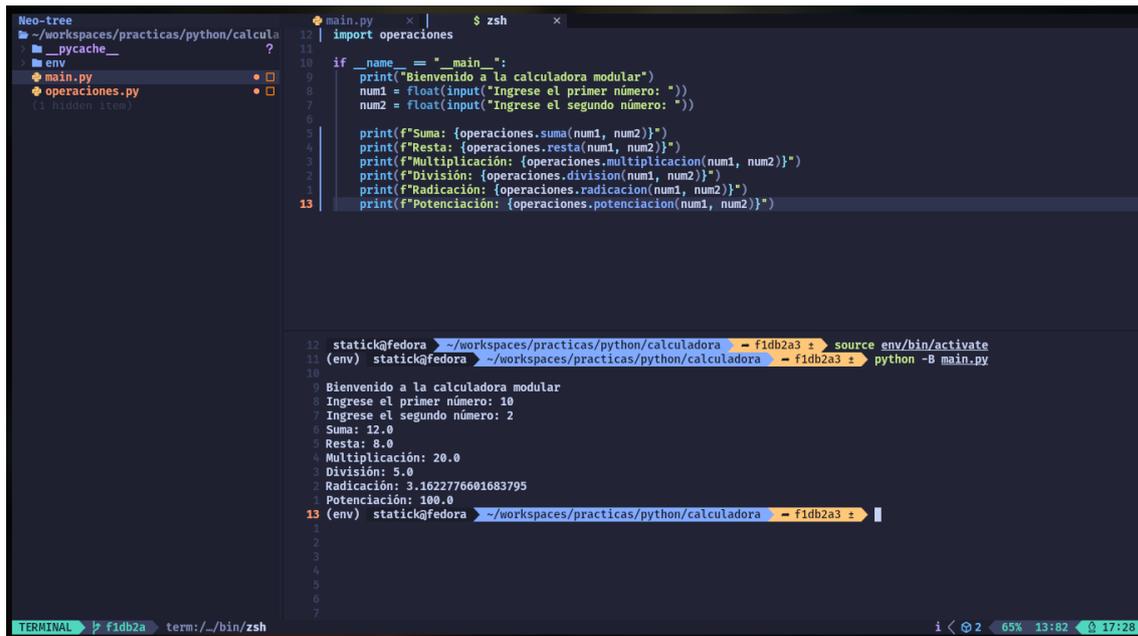
```
service nginx start
```

**Cambios no reflejados:** Si haces cambios en el archivo `index.html`, es posible que necesites **reiniciar NGINX** para que se apliquen:

```
service nginx restart
```

**No se mapea el puerto correctamente:** Si no puedes acceder a la página en el navegador, verifica que el puerto esté correctamente mapeado en la configuración del contenedor. Revisa el archivo `devcontainer.json` y asegúrate de que `forwardPorts` esté configurado correctamente como [80].

# 99 Laboratorio: Calculadora en Python



The screenshot shows a code editor with a file explorer on the left and a terminal at the bottom. The file explorer shows a directory structure with files like `main.py` and `operaciones.py`. The code in `main.py` is as follows:

```
12 import operaciones
11
10 if __name__ == "__main__":
9     print("Bienvenido a la calculadora modular")
8     num1 = float(input("Ingrese el primer número: "))
7     num2 = float(input("Ingrese el segundo número: "))
6
5     print(f"Suma: {operaciones.suma(num1, num2)}")
4     print(f"Resta: {operaciones.resta(num1, num2)}")
3     print(f"Multiplicación: {operaciones.multiplicacion(num1, num2)}")
2     print(f"División: {operaciones.division(num1, num2)}")
1     print(f"Radicación: {operaciones.radicacion(num1, num2)}")
13     print(f"Potenciación: {operaciones.potenciacion(num1, num2)}")
```

The terminal output shows the execution of the program with the following steps:

```
12 statick@fedora > ~/workspaces/practicasython/calculadora - f1db2a3 ± source env/bin/activate
11 (env) statick@fedora > ~/workspaces/practicasython/calculadora - f1db2a3 ± python -B main.py
10
9 Bienvenido a la calculadora modular
8 Ingrese el primer número: 10
7 Ingrese el segundo número: 2
6 Suma: 12.0
5 Resta: 8.0
4 Multiplicación: 20.0
3 División: 5.0
2 Radicación: 3.1622776601683795
1 Potenciación: 100.0
13 (env) statick@fedora > ~/workspaces/practicasython/calculadora - f1db2a3 ±
```

Figure 99.1: Calculadora en Python

## 99.1 Paso 1: Configuración inicial del proyecto

### 99.1.1 Crear el directorio del proyecto

Comienza creando un nuevo directorio para el proyecto:

```
mkdir calculadora
cd calculadora
```

Inicializar un repositorio Git Inicializa un repositorio Git en la carpeta del proyecto:

```
git init
Crear el archivo main.py
```

Crema un archivo **main.py** que será el punto de entrada del programa:

```
touch main.py
Primer commit
```

Añadimos el archivo inicial al control de versiones:

```
git add main.py
git commit -m "Inicio del proyecto: archivo main.py creado"
```

## 99.2 Paso 2: Agregar las operaciones básicas (suma, resta, multiplicación, división)

### 99.2.1 Código inicial

Abre el archivo `main.py` y añade el siguiente código para implementar las operaciones básicas:

```
def suma(a, b):
    """Devuelve la suma de dos números."""
    return a + b

def resta(a, b):
    """Devuelve la resta de dos números."""
    return a - b

def multiplicacion(a, b):
    """Devuelve la multiplicación de dos números."""
    return a * b

def division(a, b):
    """Devuelve la división de dos números. Maneja división entre cero."""
    try:
        return a / b
    except ZeroDivisionError:
        return "Error: No se puede dividir entre cero."

# Punto de entrada
if __name__ == "__main__":
    print("Bienvenido a la calculadora básica")
    num1 = float(input("Ingrese el primer número: "))
    num2 = float(input("Ingrese el segundo número: "))

    print(f"Suma: {suma(num1, num2)}")
```

```
print(f"Resta: {resta(num1, num2)}")
print(f"Multiplicación: {multiplicacion(num1, num2)}")
print(f"División: {division(num1, num2)}")
```

### 99.2.2 Crear un commit

Guarda los cambios y realiza un commit con la descripción del avance:

```
git add main.py
git commit -m "Implementadas las operaciones básicas"
```

## 99.3 Paso 3: Agregar funcionalidad de radicación y potenciación

### 99.3.1 Actualizar main.py

Añadimos funciones para radicación y potenciación, y las integramos al flujo del programa:

```
def radicacion(base, indice):
    """Devuelve la raíz de un número dado el índice."""
    try:
        return base ** (1 / indice)
    except ZeroDivisionError:
        return "Error: El índice de la raíz no puede ser cero."

def potenciacion(base, exponente):
    """Devuelve la potencia de un número dado un exponente."""
    return base ** exponente

# Punto de entrada actualizado
if __name__ == "__main__":
    print("Bienvenido a la calculadora extendida")
    num1 = float(input("Ingrese el primer número: "))
    num2 = float(input("Ingrese el segundo número: "))

    print(f"Suma: {suma(num1, num2)}")
    print(f"Resta: {resta(num1, num2)}")
    print(f"Multiplicación: {multiplicacion(num1, num2)}")
    print(f"División: {division(num1, num2)}")
    print(f"Radicación: {radicacion(num1, num2)}")
    print(f"Potenciación: {potenciacion(num1, num2)}")
```

### 99.3.2 Crear un commit

Guarda los cambios y realiza un commit:

```
git add main.py
git commit -m "Añadidas las operaciones de radicación y potenciación"
```

## 99.4 Paso 4: Refactorización del código en múltiples archivos

### 99.4.1 Crear estructura modular

Organizamos las operaciones en un archivo separado llamado **operaciones.py**.

```
touch operaciones.py
```

En el archivo **operaciones.py**, coloca las funciones:

```
# operaciones.py

def suma(a, b):
    """Devuelve la suma de dos números."""
    return a + b

def resta(a, b):
    """Devuelve la resta de dos números."""
    return a - b

def multiplicacion(a, b):
    """Devuelve la multiplicación de dos números."""
    return a * b

def division(a, b):
    """Devuelve la división de dos números. Maneja división entre cero."""
    try:
        resultado = a / b
    except ZeroDivisionError:
        return "Error: No se puede dividir entre cero."
    else:
        return resultado
    finally:
        pass
```

```

def radicacion(base, indice):
    """Devuelve la raíz de un número dado el índice."""
    try:
        return base ** (1 / indice)
    except ZeroDivisionError:
        return "Error: El índice de la raíz no puede ser cero."

def potenciacion(base, exponente):
    """Devuelve la potencia de un número dado un exponente."""
    return base ** exponente

```

### 99.4.2 Actualizar main.py

Actualiza `main.py` para importar las funciones desde `operaciones.py`:

```

# main.py
from operaciones import suma, resta, multiplicacion, division, radicacion, potenciacion

if __name__ == "__main__":
    print("Bienvenido a la calculadora modular")
    num1 = float(input("Ingrese el primer número: "))
    num2 = float(input("Ingrese el segundo número: "))

    print(f"Suma: {suma(num1, num2)}")
    print(f"Resta: {resta(num1, num2)}")
    print(f"Multiplicación: {multiplicacion(num1, num2)}")
    print(f"División: {division(num1, num2)}")
    print(f"Radicación: {radicacion(num1, num2)}")
    print(f"Potenciación: {potenciacion(num1, num2)}")

```

### 99.4.3 Crear un commit

Guarda los cambios y realiza un commit:

```

git add main.py operaciones.py
git commit -m "Refactorización: código modularizado en main.py y operaciones.py"

```

## 99.5 Paso 5: Manejo de errores más detallado

### 99.5.1 Mejorar el manejo de errores en division

Modifica la función `division` para agregar un bloque `else` para manejar operaciones exitosas y un `finally` para mostrar un mensaje final:

```

def division(a, b):
    """Devuelve la división de dos números."""
    try:
        resultado = a / b
    except ZeroDivisionError:
        print("Operación de división intentada.") # Mostrar mensaje solo si ocurre un error
        return "Error: No se puede dividir entre cero."
    else:
        return resultado
    finally:
        pass # El bloque finally se puede dejar vacío o eliminarlo si no es necesario

```

### 99.5.2 Crear un commit

Realiza un nuevo commit con los cambios:

```

git add operaciones.py
git commit -m "Mejorado el manejo de errores con else y finally en división"

```

## 99.6 Paso 6: Testeo automatizado

### 99.6.1 Crear pruebas unitarias

Ahora implementamos pruebas unitarias para validar cada operación usando el módulo unittest. Crea un archivo `test_calculadora.py` con el siguiente contenido:

```

# test_calculadora.py
import unittest
from operaciones import suma, resta, multiplicacion, division, radicacion, potenciacion

class TestCalculadora(unittest.TestCase):

    def test_suma(self):
        self.assertEqual(suma(2, 3), 5)

    def test_resta(self):
        self.assertEqual(resta(5, 3), 2)

    def test_multiplicacion(self):
        self.assertEqual(multiplicacion(2, 3), 6)

    def test_division(self):
        self.assertEqual(division(6, 3), 2)

    def test_division_por_cero(self):

```

```
        self.assertEqual(division(6, 0), "Error: No se puede dividir entre cero")

    def test_radicaion(self):
        self.assertEqual(radicaion(16, 4), 2)

    def test_potenciacion(self):
        self.assertEqual(potenciacion(2, 3), 8)

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

### 99.6.2 Ejecutar las pruebas

Para ejecutar las pruebas, usa el siguiente comando:

```
python -m unittest test_calculadora.py
```

## 99.7 Sigüientes pasos

- **Interfaz de usuario:** Agregar un menú para que el usuario elija las operaciones. Esto mejorará la interacción con la calculadora.
- **Mejoras adicionales:** Explorar la posibilidad de agregar operaciones avanzadas como trigonometría o logaritmos.

# 100 Sistema de Gestión de Cursos con Django y DRF

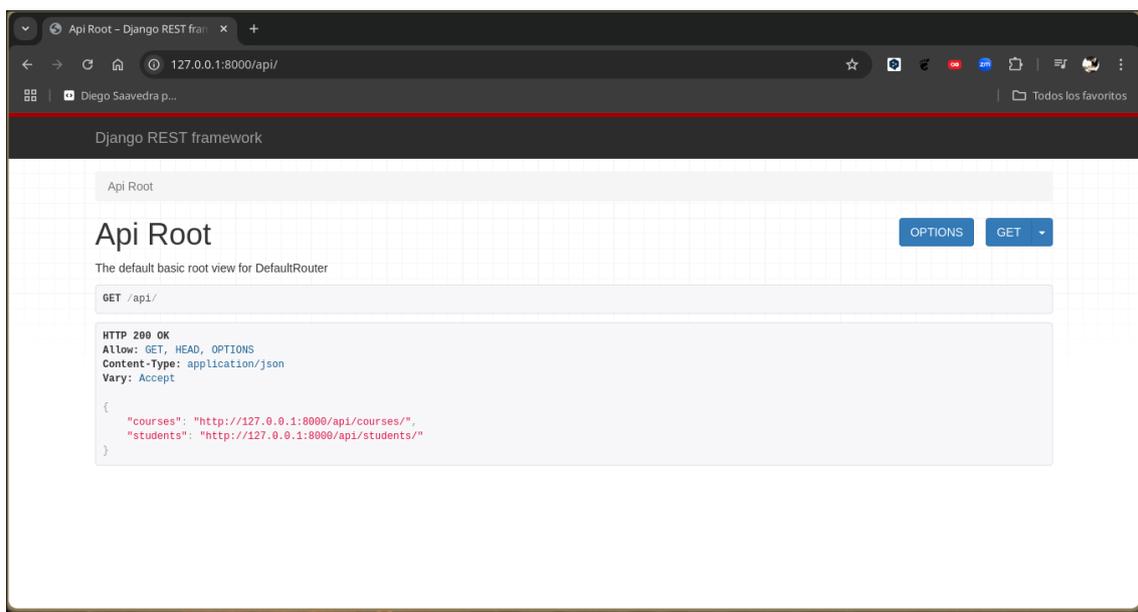


Figure 100.1: Django

## 100.1 1. Configuración del Proyecto

### 100.1.1 Paso 1: Crear el proyecto y entorno virtual

```
mkdir course_management
cd course_management
python -m venv env
source env/bin/activate
pip install django djangorestframework
```

### 100.1.2 Paso 2: Crear el proyecto y la aplicación

```
django-admin startproject config .
python manage.py startapp courses
```

### 100.1.3 Paso 3: Configurar INSTALLED\_APPS

Editar config/settings.py:

```
INSTALLED_APPS = [  
    ...  
    'rest_framework',  
    'courses',  
]
```

### 100.1.4 Paso 4: Migrar la base de datos

```
python manage.py migrate
```

# 101 2. Modelos

## 101.0.1 Paso 1: Definir los modelos

Editar courses/models.py:

```
from django.db import models

class Course(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()

    def __str__(self):
        return self.title

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    courses = models.ManyToManyField(Course, related_name='students')

    def __str__(self):
        return self.name
```

## 101.0.2 Paso 2: Migrar los modelos

```
python manage.py makemigrations
python manage.py migrate
```

## 102 3. Serializers

### 102.0.1 Paso 1: Crear el archivo serializers.py

Crear y editar courses/serializers.py:

```
from rest_framework import serializers
from .models import Course, Student

class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = '__all__'

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = '__all__'
```

## 103 4. Vistas

### 103.0.1 Paso 1: Crear las vistas

Editar `courses/views.py`:

```
from rest_framework.viewsets import ModelViewSet
from .models import Course, Student
from .serializers import CourseSerializer, StudentSerializer

class CourseViewSet(ModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

class StudentViewSet(ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

## 104 5. Rutas

### 104.0.1 Paso 1: Configurar las rutas de la aplicación

Crear y editar `courses/urls.py`:

```
from rest_framework.routers import DefaultRouter
from .views import CourseViewSet, StudentViewSet

router = DefaultRouter()
router.register('courses', CourseViewSet)
router.register('students', StudentViewSet)

urlpatterns = router.urls
```

### 104.0.2 Paso 2: Incluir las rutas en el proyecto

Editar `config/urls.py`:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('courses.urls')),
]
```

# 105 6. Pruebas de la API

## 105.0.1 Paso 1: Levantar el servidor

```
python manage.py runserver
```

## 105.0.2 Paso 2: Endpoints disponibles

- Listar Cursos: GET <http://127.0.0.1:8000/api/courses/>
- Crear Curso: POST <http://127.0.0.1:8000/api/courses/>

```
{  
  "title": "Curso Django",  
  "description": "Aprende Django desde cero."  
}
```

- Listar Estudiantes: GET <http://127.0.0.1:8000/api/students/>
- Crear Estudiante: POST <http://127.0.0.1:8000/api/students/>

## 106 Tienda Virtual.



Figure 106.1: Tienda Virtual

### 106.1 Descripción del Proyecto

El objetivo es construir una tienda virtual básica utilizando Django para el backend, React con Vite y TailwindCSS para el frontend, y PostgreSQL como base de datos. Se empleará Docker para un entorno de desarrollo consistente, con contenedores separados para el backend y base de datos. El sistema incluirá funcionalidades básicas para gestión de usuarios, productos y pedidos.

### 106.2 Objetivo General

Crear una tienda virtual funcional y accesible, enfocada en simplicidad y rendimiento, con un diseño modular y tecnología moderna.

### 106.3 Funcionalidades Principales

1. Usuarios:
  - Registro e inicio de sesión.
  - Gestión de perfiles.

## 2. Productos:

- CRUD de productos con imágenes.

## 3. Carrito y Pedidos:

- Agregar productos al carrito.
- Crear y gestionar pedidos.

## 4. Pagos:

- Integración básica con PayPal Sandbox para validación de pagos.

## 5. Frontend:

- Diseño responsivo y accesible con TailwindCSS.

## 6. Infraestructura:

- Configuración con Docker Compose para desarrollo local.

# 107 Fases del Proyecto

## 107.1 Fase 1: Configuración Inicial

1. Crear estructura del proyecto:
  - Backend: Django con Django REST Framework.
  - Frontend: React con Vite y TailwindCSS.
2. Configurar un archivo docker-compose.yml para el backend y base de datos.
3. Crear archivo .env con variables esenciales:
  - Backend: DATABASE\_URL, SECRET\_KEY.
  - Frontend: URL del backend.

## 107.2 Fase 2: Desarrollo del Backend

1. Modelos:
  - User: Roles (cliente y administrador).
  - Product: Nombre, precio, inventario, imágenes.
  - Order: Asociado a usuarios y productos.
2. Endpoints:
  - Usuarios: Registro, login, perfil.
  - Productos: CRUD.
  - Pedidos: Crear y listar.
3. Autenticación:
  - Implementar autenticación con JWT.

### 107.3 Fase 3: Desarrollo del Frontend

1. Configurar React con Vite y TailwindCSS.
2. Crear componentes básicos:
  - Navbar, ProductList, ProductDetails, Cart, CheckoutForm.
3. Implementar rutas principales con React Router:
  - Home: Lista de productos.
  - Product Details: Detalles de un producto.
  - Cart: Resumen del carrito.
  - Checkout: Confirmación de pedido.
4. Consumir APIs del backend con Axios.

### 107.4 Fase 4: Integración de Pagos

1. Configurar PayPal Sandbox en el backend.
2. Implementar lógica básica en el frontend para manejar pagos.

### 107.5 Fase 5: Pruebas

1. Pruebas unitarias en backend (modelos y vistas).
2. Pruebas de integración básica:
  - Flujo completo (registro → carrito → pedido).

### 107.6 Fase 6: Despliegue

1. Configurar despliegue en:
  - Backend: Railway.
  - Frontend: Vercel.
2. Validar funcionalidad en producción.

## **108 Resultado Esperado**

Una tienda virtual básica funcional, lista para ser extendida en el futuro con más características como métodos de pago adicionales, monitoreo y accesibilidad avanzada.

# 109 Fase 1: Recolección de Requisitos

## 109.1 Requisitos Funcionales

### 109.1.1 Usuarios

1. Registro e inicio de sesión:
  - Los usuarios deben poder registrarse, iniciar sesión y cerrar sesión.
  - Roles: Cliente y Administrador.
2. Gestión de perfiles:
  - Los usuarios pueden editar su información personal (nombre, correo, contraseña).
  - El administrador puede gestionar los usuarios (consultar, actualizar, deshabilitar cuentas).
3. Recuperación de contraseña:
  - Los usuarios deben poder recuperar su contraseña mediante un enlace enviado a su correo.
4. Autenticación de dos factores (2FA):
  - Implementación futura para mayor seguridad. Integración opcional según las necesidades del proyecto.

### 109.1.2 Productos

1. CRUD de productos:
  - El administrador puede crear, leer, actualizar y eliminar productos.
  - Atributos: nombre, descripción, precio, inventario, imágenes.
2. Gestión de imágenes:
  - El sistema almacenará las imágenes en el directorio `/static/images` del backend utilizando el sistema de archivos de Railway.

### 109.1.3 Carrito y Pedidos

1. Gestión del carrito de compras:
  - Los clientes pueden agregar, editar o eliminar productos en su carrito.
2. Gestión de pedidos:
  - Los clientes pueden realizar pedidos y consultar su estado (pendiente, procesado, enviado, cancelado).
  - Seguimiento de pedidos: Los clientes podrán recibir notificaciones o actualizaciones del estado del pedido en tiempo real.

### 109.1.4 Pagos

1. Integración con PayPal Sandbox:
  - Los usuarios podrán realizar pagos utilizando PayPal Sandbox.
  - Los pagos serán validados a través de un servidor externo que confirmará la transacción.
2. Validación y actualización de pagos:
  - Una vez validado el pago, el sistema actualizará automáticamente el estado del pedido según el resultado de la transacción (pendiente, procesado, cancelado).

### 109.1.5 Accesibilidad y SEO

1. Frontend accesible:
  - Cumplir con los estándares de accesibilidad (WCAG 2.1).
  - Aplicar buenas prácticas de SEO, como el uso de etiquetas semánticas, metadatos, y optimización de la velocidad de carga.

## 109.2 Requisitos Técnicos

### 109.2.1 Backend

1. Frameworks y herramientas:
  - Django como framework principal.
  - Django REST Framework para API RESTful.
2. Autenticación:
  - Basada en JWT con roles de usuario.
3. Base de datos:

- PostgreSQL para almacenamiento seguro y escalable.

#### 4. Caché:

- Uso de Redis para mejorar el rendimiento.

### 109.2.2 Frontend

#### 1. Frameworks y tecnologías:

- React con Vite para una configuración rápida.
- TailwindCSS para diseño moderno y responsivo.
- JavaScript para tipado estático y código robusto.

#### 2. Optimización:

- Lazy loading y code splitting para mejorar el rendimiento.
- Axios para el consumo de APIs.

### 109.2.3 Infraestructura

#### 1. Contenedores:

- Docker Compose para orquestación.
- Backend, frontend y base de datos en contenedores separados.

#### 2. Variables de entorno:

- Manejo seguro con archivos .env.

### 109.2.4 Despliegue

#### 1. Hosting:

- Railway para backend.
- Vercel para frontend.

#### 2. CI/CD:

- Flujo automatizado para pruebas y despliegue.

# 110 Fase 2: Diseño de la Arquitectura

## 110.1 1. Arquitectura General del Sistema

La arquitectura del sistema se basará en un enfoque cliente-servidor, donde el cliente (frontend) interactúa con el servidor (backend) a través de una API RESTful. El backend se encargará de gestionar la lógica de negocio, la autenticación, el almacenamiento de datos y la comunicación con servicios externos como PayPal.

## 110.2 Detalles importantes:

- **API RESTful:** El backend proporcionará endpoints RESTful para interactuar con recursos como usuarios, productos, pedidos y pagos. Se detallará la estructura de las rutas y la respuesta esperada de cada uno de los endpoints (por ejemplo, GET /api/products/ para obtener la lista de productos).
- **Servicios Externos (PayPal):** La integración con PayPal será a través de su API RESTful. Las credenciales y detalles del procesamiento de pagos se manejarán con variables de entorno y secretos seguros.

## 110.3 2. Backend (Django con Django REST Framework)

### 110.3.1 A. Estructura del Proyecto:

La estructura del proyecto se organizará de la siguiente manera:

```
backend/  
  auth/  
    migrations/  
    models.py  
    serializers.py  
    views.py  
    tests/  
  products/  
    migrations/  
    models.py  
    serializers.py  
    views.py  
    tests/
```

```
orders/
  migrations/
  models.py
  serializers.py
  views.py
  tests/
settings.py
urls.py
wsgi.py
requirements.txt
.env
```

### 110.3.2 B. Tecnologías:

- **Django:** Framework principal para la gestión del backend.
- **Django REST Framework (DRF):** Para la creación de APIs RESTful que interactúan con el frontend.
- **JWT (JSON Web Tokens):** Para la autenticación de usuarios basada en tokens. Implementaremos una política de Refresh Tokens para permitir la renovación de tokens sin que el usuario tenga que iniciar sesión constantemente.
- **PostgreSQL:** Base de datos relacional para almacenar la información de los usuarios, productos y pedidos.
- **Redis:** Para la gestión de caché, sesiones y optimización de consultas.
- **PayPal API:** Para la integración de pagos. Los detalles de las credenciales de la API se almacenarán en archivos .env utilizando herramientas de seguridad como Vault para la gestión de secretos.
- **Validaciones:** Implementaremos validaciones tanto en el frontend como en el backend para asegurar la integridad de los datos. El backend usará serializers en Django para validar los datos recibidos.
- **Cifrado de Contraseñas:** Usaremos bcrypt para el cifrado de contraseñas y asegurarnos de que la seguridad de los usuarios esté garantizada.

## 110.4 3. Frontend (React con Vite)

### 110.4.1 A. Estructura del Proyecto:

La estructura del frontend se organizará de la siguiente manera:

```
frontend/
  src/
    components/
    pages/
    services/
  App.js
  index.js
```

```
store/  
public/  
tailwind.config.js  
vite.config.js  
.env
```

#### 110.4.2 B. Tecnologías:

- **React:** Librería para la construcción de la interfaz de usuario.
- **Vite:** Herramienta para la configuración rápida y eficiente de aplicaciones React.
- **TailwindCSS:** Framework CSS para un diseño limpio y moderno.
- **Axios:** Para hacer peticiones HTTP a la API del backend. Se incluirán mecanismos para manejar errores globalmente (por ejemplo, mostrando alertas de error si una petición falla).
- **React Router:** Para la gestión de rutas en la aplicación de una sola página.
- **React Context / Redux:** Usaremos React Context o Redux para la gestión de estado a nivel global, especialmente para manejar la autenticación de usuarios, productos y pedidos.
- **Optimización de Carga:** Implementaremos Lazy Loading para componentes pesados y Code Splitting para mejorar el rendimiento y tiempo de carga del frontend.
- **Manejo de Errores:** Se implementará un sistema global de manejo de errores, tanto en las peticiones HTTP (con Axios) como en la interfaz de usuario (con Toast Notifications o alertas).

### 110.5 4. Integración del Backend y Frontend

#### 110.5.1 Comunicación:

- El frontend se comunicará con el backend a través de peticiones HTTP utilizando Axios para obtener datos y realizar acciones (CRUD) sobre los productos, pedidos, etc. Se asegurará de que todas las peticiones que requieren autenticación incluyan el JWT en las cabeceras.

#### 110.5.2 Autenticación:

- La autenticación de los usuarios se manejará mediante JWT. El token de acceso se incluirá en las cabeceras de las peticiones HTTP que requieran autenticación. También se implementará un sistema de Refresh Tokens para renovar el token de acceso sin necesidad de que el usuario vuelva a iniciar sesión.

### 110.5.3 Manejo de Errores:

- Se implementarán manejadores de errores tanto en el backend como en el frontend. En el backend, usaremos excepciones personalizadas para manejar errores en los endpoints de la API, mientras que en el frontend se mostrarán mensajes adecuados de error usando un sistema de notificaciones.

## 110.6 5. Base de Datos (PostgreSQL)

- **Modelo de Datos:** Los modelos de datos se organizarán en tablas de usuarios, productos y pedidos.
  - **Usuarios:** Almacenará información sobre clientes y administradores. Incluye un campo de rol para diferenciarlos.
  - **Productos:** Almacenará la información de los productos (nombre, precio, inventario, imágenes, descripción, etc.).
  - **Pedidos:** Almacenará los detalles de los pedidos realizados por los clientes (productos, cantidades, fecha de compra, total, estado de pago).

### 110.6.1 Relacionando los Modelos:

- Se establecerán relaciones adecuadas entre los modelos, como Foreign Keys entre pedidos y productos, y Many-to-Many entre productos y categorías, si es necesario.

## 110.7 6. Infraestructura (Docker y Contenedores)

- **Docker Compose:** Para orquestar los contenedores del backend, frontend y base de datos.
  - **Contenedor Backend:** Contendrá la aplicación Django, configurada para interactuar con PostgreSQL y Redis.
  - **Contenedor Frontend:** Contendrá la aplicación React, configurada para comunicarse con la API Django.
  - **Contenedor Base de Datos:** Contendrá PostgreSQL.
  - **Contenedor de Cache:** Contendrá Redis.
- **Variables de Entorno:** Los detalles de configuración (como credenciales de base de datos, claves de API) se almacenarán en archivos `.env` para mantener la seguridad y facilitar la configuración.
- Implementaremos un archivo `.env.example` para que los desarrolladores puedan copiarlo y configurar sus entornos locales.

## 110.8 7. Despliegue (Railway para Backend, Vercel para Frontend)

- **Backend en Railway:** El backend se desplegará en Railway, donde se gestionarán las bases de datos y la API.
- **Frontend en Vercel:** El frontend se desplegará en Vercel para un acceso rápido y escalabilidad automática.

### 110.8.1 Escalabilidad:

- A medida que el proyecto crezca, podremos escalar el backend en Railway o cambiar a un servicio como AWS o Heroku si es necesario.
- Vercel ofrece escalabilidad automática, pero también es importante evaluar el uso de un CDN para optimizar la entrega de contenido estático en el frontend.

## 110.9 8. Seguridad y Buenas Prácticas

- **Autenticación de Usuarios:** Implementación de JWT para la autenticación de usuarios, con roles de cliente y administrador. La política de renovación de tokens mediante Refresh Tokens será implementada.
- **Seguridad en los Pagos:** Integración con PayPal para procesar pagos de manera segura. Las credenciales y claves de la API de PayPal se mantendrán seguras mediante variables de entorno.
- **Validación de Datos:** Uso de validaciones tanto en el frontend como en el backend para garantizar la integridad de los datos.
- **Cifrado:** Cifrado de contraseñas en el backend utilizando bcrypt.
- **Manejo de Errores y Logs:** Se configurarán registros (logs) detallados para facilitar la detección de errores y auditoría.

# 111 Fase 3. Desarrollo de la Aplicación

En esta fase, vamos a estructurar cómo se desarrollarán los distintos módulos del sistema, tanto en el backend como en el frontend, para implementar las funcionalidades esenciales del proyecto.

## 111.1 1. Desarrollo del Backend (Django con Django REST Framework)

### 111.1.1 A. Configuración Inicial del Proyecto

1 **Instalación de Dependencias:** Instalaremos las dependencias necesarias en el entorno virtual de Python.

```
pip install django djangorestframework psycopg2-binary django-environ djangorestframework
```

2. Configuración del Proyecto Django:

- Crear el proyecto Django con el comando:

```
django-admin startproject backend .
```

- Crear las aplicaciones para autenticación, productos y pedidos:

```
python manage.py startapp auth
python manage.py startapp products
python manage.py startapp orders
```

- Configuración de la Base de Datos:

3. En el archivo **settings.py**, configura la conexión a PostgreSQL utilizando el paquete **django-environ** para gestionar variables de entorno. Asegúrate de tener un archivo **.env** con las credenciales necesarias:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
        'HOST': os.environ.get('DB_HOST'),
```

```
        'PORT': os.environ.get('DB_PORT', 5432),
    }
}
```

Y en el archivo `.env`, agrega las variables de entorno:

```
DB_NAME=mydatabase
DB_USER=myuser
DB_PASSWORD=mypassword
DB_HOST=localhost
DB_PORT=5432
```

## 111.1.2 B. Implementación de Modelos

### 1. Modelo de Usuario:

- En el archivo `auth/models.py`, crea un modelo de usuario si decides personalizarlo (si no, puedes usar el modelo `User` por defecto de Django).

### 2. Modelo de Producto:

- En `products/models.py`, define el modelo para los productos con campos como **nombre**, **precio**, **inventario**, **imagen**, etc.

```
class Producto(models.Model):
    nombre = models.CharField(max_length=255)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    descripcion = models.TextField()
    inventario = models.IntegerField()
    imagen = models.ImageField(upload_to='productos/')
```

### 3. Modelo de Pedido:

En `orders/models.py`, crea un modelo para los pedidos, con campos como **usuario**, **productos**, **fecha**, **estado**, etc.

```
class Pedido(models.Model):
    usuario = models.ForeignKey(User, on_delete=models.CASCADE)
    productos = models.ManyToManyField(Producto)
    fecha = models.DateTimeField(auto_now_add=True)
    estado = models.CharField(max_length=50)
    total = models.DecimalField(max_digits=10, decimal_places=2)
```

### 111.1.3 C. Serializadores

En el backend, usarás **serializers** para transformar los datos entre los modelos y las respuestas JSON:

1. Serializador de Producto (en **products/serializers.py**):

```
from rest_framework import serializers
from .models import Producto

class ProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Producto
        fields = ['id', 'nombre', 'precio', 'descripcion', 'inventario', 'imagen']
```

2. Serializador de Pedido (en **orders/serializers.py**):

```
from rest_framework import serializers
from .models import Pedido
from products.serializers import ProductoSerializer

class PedidoSerializer(serializers.ModelSerializer):
    productos = ProductoSerializer(many=True)

    class Meta:
        model = Pedido
        fields = ['id', 'usuario', 'productos', 'fecha', 'estado', 'total']
```

### 111.1.4 D. Vistas y Rutas

1. Vistas para el Producto (en **products/views.py**):

```
from rest_framework import viewsets
from .models import Producto
from .serializers import ProductoSerializer

class ProductoViewSet(viewsets.ModelViewSet):
    queryset = Producto.objects.all()
    serializer_class = ProductoSerializer
```

2. Vistas para el Pedido (en **orders/views.py**):

```
from rest_framework import viewsets
from .models import Pedido
from .serializers import PedidoSerializer

class PedidoViewSet(viewsets.ModelViewSet):
    queryset = Pedido.objects.all()
    serializer_class = PedidoSerializer
```

### 3. Rutas (en `urls.py`):

```
from django.urls import include, path
from rest_framework.routers import DefaultRouter
from products.views import ProductoViewSet
from orders.views import PedidoViewSet

router = DefaultRouter()
router.register(r'productos', ProductoViewSet)
router.register(r'pedidos', PedidoViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

## 111.1.5 E. Autenticación (JWT)

### 1. Configuración de JWT:

- Instala el paquete `django-rest-framework-jwt` para la autenticación con JWT.

```
pip install django-rest-framework-jwt
```

- En `settings.py`, agrega la configuración para JWT:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}
```

### 2. Vistas de Login/Logout:

- Crea vistas para manejar el login y logout de los usuarios, utilizando JWT.

## 111.2 2. Desarrollo del Frontend (React con Vite)

### 111.2.1 A. Configuración Inicial del Proyecto

#### 1. Crear el Proyecto React con Vite:

```
npm create vite@latest frontend --template react
cd frontend
npm install
```

#### 2. Instalar Dependencias:

```
npm install axios react-router-dom
```

### 111.2.2 B. Estructura de Archivos

En el frontend, la estructura del proyecto será la siguiente:

```
frontend/  
  src/  
    components/  
      Header.js  
      ProductList.js  
    pages/  
      HomePage.js  
      ProductPage.js  
    services/  
      api.js  
    App.js  
    index.js  
    store/  
  public/  
  tailwind.config.js  
  vite.config.js  
  .env
```

### 111.2.3 C. Lógica de Peticiones con Axios

1. Servicio API (en `services/api.js`):

```
import axios from 'axios';  
  
const API_URL = 'http://localhost:8000/api/';  
  
export const getProductos = async () => {  
  const response = await axios.get(`${API_URL}productos/`);  
  return response.data;  
};  
  
export const getPedido = async (id) => {  
  const response = await axios.get(`${API_URL}pedidos/${id}/`);  
  return response.data;  
};
```

2. Componente de Lista de Productos (en `components/ProductList.js`):

```

import React, { useEffect, useState } from 'react';
import { getProductos } from '../services/api';

const ProductList = () => {
  const [productos, setProductos] = useState([]);

  useEffect(() => {
    const fetchProductos = async () => {
      const data = await getProductos();
      setProductos(data);
    };

    fetchProductos();
  }, []);

  return (
    <div>
      <h2>Productos</h2>
      <ul>
        {productos.map((producto) => (
          <li key={producto.id}>{producto.nombre}</li>
        ))}
      </ul>
    </div>
  );
};

export default ProductList;

```

## 111.3 3. Despliegue y Configuración de Docker

### 111.3.1 A. Configuración de Docker

#### 1. Dockerfile para el Backend:

```

# Dockerfile para Backend
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

```

```
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

## 2. Dockerfile para el Frontend:

```
# Dockerfile para Frontend
FROM node:16

WORKDIR /app

COPY package.json .
COPY package-lock.json .

RUN npm install

COPY . .

CMD ["npm", "run", "dev"]
```

## 3. docker-compose.yml:

```
services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    depends_on:
      - db

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend

  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
```

## 112 Fase 4: Integración de Funcionalidades Avanzadas

En esta fase, vamos a agregar características avanzadas para hacer que el sistema sea más funcional y completo, incluyendo autenticación JWT, integración con pasarelas de pago (como PayPal), y optimización del flujo de trabajo para los usuarios.

### 112.1 1. Implementación de la Autenticación de Usuarios con JWT

#### 112.1.1 A. Configuración de la Autenticación JWT en el Backend

1. **Configuración del paquete SimpleJWT:** Instala la biblioteca necesaria si aún no lo has hecho:

```
pip install djangorestframework-simplejwt
```

Luego, actualiza `settings.py` para incluir la configuración de autenticación JWT:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}
```

2. **Vistas para Autenticación (Login y Refresh Tokens):** En el archivo `auth/views.py`, crea las vistas para manejar tokens de acceso y refresco:

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView
from django.urls import path

urlpatterns = [
    path('login/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

3. **Protección de las Rutas API:** Asegúrate de que las vistas relacionadas con productos y pedidos estén protegidas mediante JWT:

```

from rest_framework.permissions import IsAuthenticated

class ProductoViewSet(viewsets.ModelViewSet):
    queryset = Producto.objects.all()
    serializer_class = ProductoSerializer
    permission_classes = [IsAuthenticated]

```

4. **Pruebas para la Autenticación:** Agrega pruebas en el backend para validar que solo los usuarios autenticados puedan acceder a las rutas protegidas.

### 112.1.2 B. Configuración del Frontend para JWT

1. **Manejo del Login en el Frontend:** Crea un componente **Login.js** para autenticar al usuario:

```

import React, { useState } from 'react';
import axios from 'axios';

const Login = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('http://localhost:8000/api/login/', {
        username: email,
        password: password,
      });
      localStorage.setItem('access', response.data.access);
      localStorage.setItem('refresh', response.data.refresh);
      alert('Inicio de sesión exitoso');
    } catch (error) {
      console.error('Error al iniciar sesión:', error);
      alert('Credenciales incorrectas');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Correo electrónico"
      />
      <input

```

```

        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        placeholder="Contraseña"
      />
      <button type="submit">Iniciar sesión</button>
    </form>
  );
};

export default Login;

```

2. **Autenticación Automática en Axios:** Configura un interceptor en Axios para agregar automáticamente el token JWT a las solicitudes:

```

import axios from 'axios';

const API = axios.create({
  baseURL: 'http://localhost:8000/api/',
});

API.interceptors.request.use((config) => {
  const token = localStorage.getItem('access');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

export default API;

```

## 112.2 2. Integración con PayPal

### 112.2.1 A. Configuración del SDK de PayPal

1. **Instalar el SDK en el Frontend:** Instala el paquete de PayPal:

```
npm install @paypal/react-paypal-js
```

2. **Configurar el Componente de Pago:** Crea un componente PaypalButton.js que permita realizar pagos:

```

import React from 'react';
import { PayPalScriptProvider, PayPalButtons } from '@paypal/react-paypal-js';

const PaypalButton = ({ total }) => {
  const handleApprove = (data, actions) => {
    return actions.order.capture().then((details) => {
      alert(`Pago realizado por ${details.payer.name.given_name}`);
    });
  };

  return (
    <PayPalScriptProvider
      options={{ "client-id": "YOUR_PAYPAL_CLIENT_ID" }}
    >
      <PayPalButtons
        createOrder={({data, actions}) => {
          return actions.order.create({
            purchase_units: [
              {
                amount: {
                  value: total.toString(),
                },
              },
            ],
          });
        }}
        onApprove={handleApprove}
      />
    </PayPalScriptProvider>
  );
};

export default PaypalButton;

```

3. **Actualizar la Página de Pedido:** Integra el botón de PayPal en la página de resumen del pedido (OrderPage.js):

```

import React from 'react';
import PaypalButton from '../components/PaypalButton';

const OrderPage = ({ order }) => {
  return (
    <div>
      <h2>Resumen del Pedido</h2>
      <ul>
        {order.productos.map((producto) => (
          <li key={producto.id}>
            {producto.nombre} - ${producto.precio}
          </li>
        ))}
      </ul>
    </div>
  );
};

```

```

        </li>
    ))}
</ul>
<h3>Total: ${order.total}</h3>
<PaypalButton total={order.total} />
</div>
);
};

export default OrderPage;

```

## 112.3 3. Optimización de Flujo de Trabajo

### 112.3.1 A. Validaciones en Formularios

- Implementa validaciones en el frontend (con herramientas como **Formik** o **React Hook Form**) para asegurar que los formularios contengan datos correctos antes de enviarse.

### 112.3.2 B. Paginación en Listas de Productos

- Implementa la paginación en el backend utilizando **LimitOffsetPagination**:

```

from rest_framework.pagination import LimitOffsetPagination

class ProductoPagination(LimitOffsetPagination):
    default_limit = 10
    max_limit = 100

```

- Configura la paginación en el frontend con estados que gestionen el desplazamiento entre páginas.

### 112.3.3 C. Optimización con Caching

- Configura un sistema de caching en el backend utilizando Redis para mejorar el rendimiento de las solicitudes frecuentes.

```

pip install django-redis

```

En `settings.py`:

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
    }
}
```

### 112.3.4 D. Dockerización Completa

- Asegúrate de que tanto el backend como el frontend estén correctamente empaquetados con sus dependencias para el despliegue. Puedes agregar un contenedor para Redis en el archivo `docker-compose.yml`.

## 112.4 4. Despliegue en Producción

### 112.4.1 A. Configuración de Servidor

- Usa NGINX como servidor proxy inverso para manejar las solicitudes HTTP y redirigirlas al backend y frontend.

### 112.4.2 B. SSL con Let's Encrypt

- Configura certificados SSL para tu dominio utilizando Certbot.

### 112.4.3 C. Configuración de Base de Datos

- Asegúrate de que la base de datos PostgreSQL esté alojada en un entorno seguro y optimizado para producción.

# 113 Fase 5: Configuración para Despliegue y Pruebas E2E

En esta fase, vamos a preparar el proyecto para ser desplegado en un entorno de producción y agregar pruebas E2E (end-to-end) para garantizar que el sistema funcione correctamente desde la perspectiva del usuario final.

## 113.1 1. Configuración del Entorno de Producción

### 113.1.1 A. Configuración del Servidor con NGINX y Gunicorn

1. **Instalar NGINX y Gunicorn:** En el servidor de producción, instala las herramientas necesarias:

```
sudo apt update
sudo apt install nginx python3-pip
pip install gunicorn
```

2. **Configurar Gunicorn:** Crea un archivo `gunicorn.service` para iniciar Gunicorn como un servicio de sistema:

```
sudo nano /etc/systemd/system/gunicorn.service
```

Contenido del archivo:

```
[Unit]
Description=gunicorn daemon
After=network.target

[Service]
User=your_user
Group=www-data
WorkingDirectory=/path/to/your/project
ExecStart=/usr/bin/gunicorn --workers 3 --bind unix:/path/to/your/project.sock your_proje

[Install]
WantedBy=multi-user.target
```

Reinicia y habilita el servicio:

```
sudo systemctl start gunicorn
sudo systemctl enable gunicorn
```

3. **Configurar NGINX:** Crea un archivo de configuración en `/etc/nginx/sites-available/your_project`:

```
server {
    listen 80;
    server_name your_domain.com www.your_domain.com;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /path/to/your/project;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/path/to/your/project.sock;
    }
}
```

Habilita el archivo de configuración:

```
sudo ln -s /etc/nginx/sites-available/your_project /etc/nginx/sites-enabled
sudo nginx -t
sudo systemctl restart nginx
```

### 113.1.2 B. Configurar Certificados SSL con Let's Encrypt

1. **Instalar Certbot:** Instala Certbot y su plugin para NGINX:

```
sudo apt install certbot python3-certbot-nginx
```

2. **Obtener Certificados SSL:** Ejecuta Certbot para tu dominio:

```
sudo certbot --nginx -d your_domain.com -d www.your_domain.com
```

3. **Renovación Automática:** Agrega Certbot al cron para renovar automáticamente los certificados:

```
sudo certbot renew --dry-run
```

## 113.2 2. Pruebas E2E con Cypress

### 113.2.1 A. Instalación de Cypress

1. Instalar Cypress en el Frontend: Desde el directorio del frontend, instala Cypress:

```
npm install cypress --save-dev
```

2. **Inicializar Cypress:** Ejecuta el siguiente comando para crear la configuración básica de Cypress:

```
npx cypress open
```

### 113.2.2 B. Crear Pruebas E2E

1. **Configuración del Archivo de Pruebas:** Crea un archivo de prueba, por ejemplo, `cypress/e2e/login.cy.js`:

```
describe('Pruebas de Autenticación', () => {
  it('El usuario puede iniciar sesión correctamente', () => {
    cy.visit('http://localhost:3000/login');
    cy.get('input[type="email"]').type('usuario@example.com');
    cy.get('input[type="password"]').type('contraseña123');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });

  it('El usuario recibe un error con credenciales incorrectas', () => {
    cy.visit('http://localhost:3000/login');
    cy.get('input[type="email"]').type('usuario@example.com');
    cy.get('input[type="password"]').type('contraseñaIncorrecta');
    cy.get('button[type="submit"]').click();
    cy.contains('Credenciales incorrectas').should('be.visible');
  });
});
```

Ejecutar Pruebas: Lanza las pruebas con:

```
npx cypress run
```

### 113.2.3 C. Pruebas para el Flujo Completo

1. **Simulación de un Pedido:** Crea pruebas para simular el flujo completo, desde agregar productos al carrito hasta realizar el pago:

```

describe('Flujo Completo de Pedido', () => {
  it('El usuario puede realizar un pedido', () => {
    cy.visit('http://localhost:3000/');
    cy.get('.producto').first().click();
    cy.get('button.agregar-al-carrito').click();
    cy.get('button.ver-carrito').click();
    cy.get('button.proceder-al-pago').click();
    cy.get('input[type="email"]').type('usuario@example.com');
    cy.get('input[type="password"]').type('contraseña123');
    cy.get('button[type="submit"]').click();
    cy.contains('Pago realizado con éxito').should('be.visible');
  });
});

```

## 113.3 3. Dockerización Completa

### 113.3.1 A. Actualización del docker-compose.yml

Incluye contenedores para:

- Backend
- Frontend
- Redis
- Base de datos PostgreSQL
- NGINX

Ejemplo de configuración para **docker-compose.yml**:

```

services:
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: your_db
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:6
    ports:
      - "6379:6379"

  backend:
    build:

```

```
    context: ./backend
command: gunicorn your_project.wsgi:application --bind 0.0.0.0:8000
volumes:
  - ./backend:/app
ports:
  - "8000:8000"
depends_on:
  - db
  - redis

frontend:
  build:
    context: ./frontend
  ports:
    - "3000:3000"
  stdin_open: true
  tty: true

nginx:
  image: nginx:latest
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf
  ports:
    - "80:80"
    - "443:443"
  depends_on:
    - backend
    - frontend

volumes:
  db_data:
```

**Part X**

**Ejercicios**

# 114 Ejercicios Python - Nivel Intermedio (POO) - Parte 1

## Ejercicio 1

Crear una clase Persona con atributos nombre y edad. Implementar un método que devuelva si la persona es mayor de edad.

## Solución

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

    def es_mayor_de_edad(self):
        return self.edad >= 18
```

## 115 Ejemplo de uso

```
persona = Persona("Juan", 20)
print(persona.es_mayor_de_edad()) # True
```

### Ejercicio 2

Crear una clase Círculo con un atributo radio. Implementar métodos para calcular el área y el perímetro.

### Solución

```
import math

class Círculo:
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return math.pi * self.radio ** 2

    def perimetro(self):
        return 2 * math.pi * self.radio
```

## 116 Ejemplo de uso

```
circulo = Circulo(5)
print(circulo.area())      # 78.54...
print(circulo.perimetro()) # 31.41...
```

### Ejercicio 3

Crear una clase Rectángulo con atributos base y altura. Implementar un método que determine si es un cuadrado.

### Solución

```
class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def es_cuadrado(self):
        return self.base == self.altura
```

## 117 Ejemplo de uso

```
rectangulo = Rectangulo(5, 5)
print(rectangulo.es_cuadrado()) # True
```

### Ejercicio 4

Crear una clase Vehículo con atributos marca y modelo. Crear una clase hija Coche con un atributo adicional velocidad\_maxima.

### Solución

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

class Coche(Vehiculo):
    def __init__(self, marca, modelo, velocidad_maxima):
        super().__init__(marca, modelo)
        self.velocidad_maxima = velocidad_maxima
```

## 118 Ejemplo de uso

```
coche = Coche("Toyota", "Corolla", 180)
print(coche.marca)           # Toyota
print(coche.velocidad_maxima) # 180
```

Ejercicio 5

Crear una clase CuentaBancaria con métodos para depositar, retirar y consultar el saldo.

Solución

```
class CuentaBancaria:
    def __init__(self, saldo_inicial=0):
        self.saldo = saldo_inicial

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

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

    def consultar_saldo(self):
        return self.saldo
```

## 119 Ejemplo de uso

```
cuenta = CuentaBancaria(100)
cuenta.depositar(50)
cuenta.retirar(30)
print(cuenta.consultar_saldo()) # 120
```

### Ejercicio 6

Crear una clase Empleado con atributos nombre y salario. Implementar un método que calcule el salario anual.

### Solución

```
class Empleado:
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario

    def salario_anual(self):
        return self.salario * 12
```

## 120 Ejemplo de uso

```
empleado = Empleado("Ana", 1500)
print(empleado.salario_anual()) # 18000
```

### Ejercicio 7

Crear una clase Animal con un método hablar. Crear subclases como Perro y Gato que implementen el método hablar.

### Solución

```
class Animal:
    def hablar(self):
        raise NotImplementedError("Este método debe ser implementado por la subclase.")

class Perro(Animal):
    def hablar(self):
        return "Guau"

class Gato(Animal):
    def hablar(self):
        return "Miau"
```

## 121 Ejemplo de uso

```
perro = Perro()
gato = Gato()
print(perro.hablar()) # Guau
print(gato.hablar()) # Miau
```

### Ejercicio 8

Crear una clase Tienda que gestione un inventario. Implementar métodos para agregar, eliminar y mostrar productos.

### Solución

```
class Tienda:
    def __init__(self):
        self.inventario = {}

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

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

    def mostrar_inventario(self):
        return self.inventario
```

## 122 Ejemplo de uso

```
tienda = Tienda()
tienda.agregar_producto("Manzanas", 10)
tienda.agregar_producto("Peras", 5)
tienda.eliminar_producto("Peras")
print(tienda.mostrar_inventario()) # {'Manzanas': 10}
```

### Ejercicio 9

Crear una clase Fracción con atributos numerador y denominador. Implementar un método para sumar dos fracciones.

### Solución

```
class Fraccion:
    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador

    def sumar(self, otra):
        nuevo_num = self.numerador * otra.denominador + otra.numerador * self.denominador
        nuevo_den = self.denominador * otra.denominador
        return Fraccion(nuevo_num, nuevo_den)

    def __str__(self):
        return f"{self.numerador}/{self.denominador}"
```

## 123 Ejemplo de uso

```
f1 = Fraccion(1, 2)
f2 = Fraccion(1, 3)
resultado = f1.sumar(f2)
print(resultado) # 5/6
```

### Ejercicio 10

Crear una clase Libro que contenga un atributo autores como una lista. Implementar un método para agregar y listar los autores.

### Solución

```
class Libro:
    def __init__(self, titulo):
        self.titulo = titulo
        self.autores = []

    def agregar_autor(self, autor):
        self.autores.append(autor)

    def listar_autores(self):
        return self.autores
```

## 124 Ejemplo de uso

```
libro = Libro("Python Intermedio")
libro.agregar_autor("Autor 1")
libro.agregar_autor("Autor 2")
print(libro.listar_autores()) # ['Autor 1', 'Autor 2']
```

## 125 Ejercicios Python - Nivel Intermedio (POO) - Parte 2

### Ejercicio 6

Implementa una clase base `Animal` con un método `hablar`. Crea una subclase `Perro` que sobrescriba este método.

### Solución

```
class Animal:
    def hablar(self):
        raise NotImplementedError("Este método debe ser implementado por la subclase.")

class Perro(Animal):
    def hablar(self):
        return "Guau"
```

### Ejemplo de uso

```
perro = Perro()
print(perro.hablar()) # Guau
```

### Ejercicio 7

Crea una clase `Empleado` y una subclase `Gerente` que utilice `super()` para llamar al constructor de la clase base.

### Solución

```
class Empleado:
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario

class Gerente(Empleado):
    def init(self, nombre, salario, departamento):
        super().init(nombre, salario)
        self.departamento = departamento
```

### Ejemplo de uso

```
gerente = Gerente("Carlos", 5000, "Ventas")
print(gerente.nombre)      # Carlos
print(gerente.departamento) # Ventas
```

#### Ejercicio 8

Crea una clase Matematica con un método estático que calcule el área de un círculo dado el radio.

Solución

```
import math

class Matematica:

    @staticmethod # Método estático, no recibe una instancias como argumento
    def area_circulo(radio):
    return math.pi * radio ** 2
```

Ejemplo de uso

```
print(Matematica.area_circulo(5)) # 78.5398...
```

#### Ejercicio 9

Implementa una clase Vehiculo con un contador de instancias.

Solución

```
class Vehiculo:
    contador = 0

    def __init__(self):
        Vehiculo.contador += 1
```

Ejemplo de uso

```
vehiculo1 = Vehiculo()
vehiculo2 = Vehiculo()
print(Vehiculo.contador) # 2
```

#### Ejercicio 10

Crea una clase Vector que permita sumar dos vectores usando el operador +.

Solución

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro):
        return Vector(self.x + otro.x, self.y + otro.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

```

Ejemplo de uso

```

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3) # (4, 6)

```

Ejercicio 11

Crea una clase Rectangulo que calcule su área como una propiedad.

Solución

```

class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    @property # Propiedad, las propiedades no se llaman como métodos
    def area(self):
        return self.base * self.altura

```

Ejemplo de uso

```

rectangulo = Rectangulo(5, 3)
print(rectangulo.area) # 15

```

Ejercicio 12

Crea una clase abstracta Figura con un método area que deba implementarse en las sub-clases.

Solución

```

from abc import ABC, abstractmethod

class Figura(ABC):

    @abstractmethod # Método abstracto, las subclases deben implementarlo
    def area(self):
        pass

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return 3.1416 * self.radio ** 2

```

Ejemplo de uso

```

circulo = Circulo(4)
print(circulo.area()) # 50.2656...

```

Ejercicio 13

Creando una excepción personalizada llamada SaldoInsuficienteError para una clase de cuenta bancaria.

Solución

```

class SaldoInsuficienteError(Exception):
    pass

class CuentaBancaria:
    def init(self, saldo_inicial=0):
        self.saldo = saldo_inicial

    def retirar(self, cantidad):
        if cantidad > self.saldo:
            raise SaldoInsuficienteError("Fondos insuficientes")
        self.saldo -= cantidad

```

Ejemplo de uso

```

cuenta = CuentaBancaria(100)
try:
    cuenta.retirar(200)
except SaldoInsuficienteError as e:
    print(e) # Fondos insuficientes

```

#### Ejercicio 14

Crea una clase Libro que implemente estos métodos para mostrar información de manera clara.

Solución

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo self.autor = autor

    def __str__(self):
        return f"Libro: {self.titulo} por {self.autor}"

    def __repr__(self):
        return f"Libro({self.titulo!r}, {self.autor!r})"
```

Ejemplo de uso

```
libro = Libro("Python Avanzado", "Juan Pérez")
print(str(libro)) # Libro: Python Avanzado por Juan Pérez
print(repr(libro)) # Libro('Python Avanzado', 'Juan Pérez')
```

#### Ejercicio 15

Implementa una clase Producto usando @dataclass.

Solución

```
from dataclasses import dataclass

@dataclass # Decorador para crear clases de datos
class Producto:
    nombre: str
    precio: float
    cantidad: int
```

Ejemplo de uso

```
producto = Producto("Camiseta", 20.5, 100)
print(producto) # Producto(nombre='Camiseta', precio=20.5, cantidad=100)
```

## 126 Ejercicios Python - Nivel Intermedio (POO) - Parte 3

Ejercicio 16

Crea una clase Contador que actúe como un iterador para contar hasta un número dado.

Solución

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.contador = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.contador < self.limite:
            self.contador += 1
            return self.contador
        else:
            raise StopIteration
```

Ejemplo de uso

```
contador = Contador(3)
for numero in contador:
    print(numero) # 1 2 3
```

Ejercicio 17

Crea una clase LoggerMixin para agregar registro de actividades a otras clases.

Solución

```
class LoggerMixin:
    def log(self, mensaje):
        print(f"Log: {mensaje}")

class Usuario(LoggerMixin):
    def init(self, nombre):
        self.nombre = nombre
```

```
def realizar_actividad(self):
    self.log(f"{self.nombre} realizó una actividad")
```

Ejemplo de uso

```
usuario = Usuario("Carlos")
usuario.realizar_actividad() # Log: Carlos realizó una actividad
```

Ejercicio 18

Crema una clase `Convertor` que permita convertir entre unidades (por ejemplo, kilómetros a millas).

Solución

```
class Convertor:

    @staticmethod
    def kilometros_a_millas(km):
        return km * 0.621371

    @classmethod
    def millas_a_kilometros(cls, millas):
        return millas / 0.621371
```

Ejemplo de uso

```
print(Convertor.kilometros_a_millas(5)) # 3.106855
print(Convertor.millas_a_kilometros(3)) # 4.828032
```

Ejercicio 19

Crema una jerarquía de clases: `Vehiculo` como base, con subclases `Auto` y `Moto`.

Solución

```
class Vehiculo:

    def __init__(self, marca):
        self.marca = marca

class Auto(Vehiculo):
    def __init__(self, marca, puertas):
        super().__init__(marca)
        self.puertas = puertas

class Moto(Vehiculo):
    def __init__(self, marca, cilindrada):
        super().__init__(marca)
        self.cilindrada = cilindrada
```

Ejemplo de uso

```
auto = Auto("Toyota", 4)
moto = Moto("Yamaha", 250)
print(auto.marca) # Toyota
print(moto.cilindrada) # 250
```

Ejercicio 20

Crea una clase Punto cuyos atributos sean de solo lectura.

Solución

```
class Punto:
    def __init__(self, x, y):
        self._x = x self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y
```

Ejemplo de uso

```
punto = Punto(5, 10)
print(punto.x) # 5
print(punto.y) # 10
punto.x = 10 # AttributeError: can't set AttributeError
```

Ejercicio 21

Implementa la clase Fraccion que permita comparar fracciones usando operadores (<, >, ==).

Solución

```
class Fraccion:
    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador

    def __eq__(self, otra):
        return self.numerador * otra.denominador == self.denominador * otra.numerador

    def __lt__(self, otra):
        return self.numerador * otra.denominador < self.denominador * otra.numerador
```

Ejemplo de uso

```
f1 = Fraccion(1, 2)
f2 = Fraccion(1, 3)
print(f1 == f2) # False
print(f1 < f2) # False
```

Ejercicio 22

Crea una clase Calculadora que permita llamar a sus instancias como si fueran funciones.

Solución

```
class Calculadora:
    def __call__(self, a, b):
        return a + b
```

Ejemplo de uso

```
calculadora = Calculadora()
print(calculadora(2, 3)) # 5
```

**Part XI**

**Extras**

# 127 Laboratorio: Desarrollo de un Sistema de Chat Local Cliente-Servidor

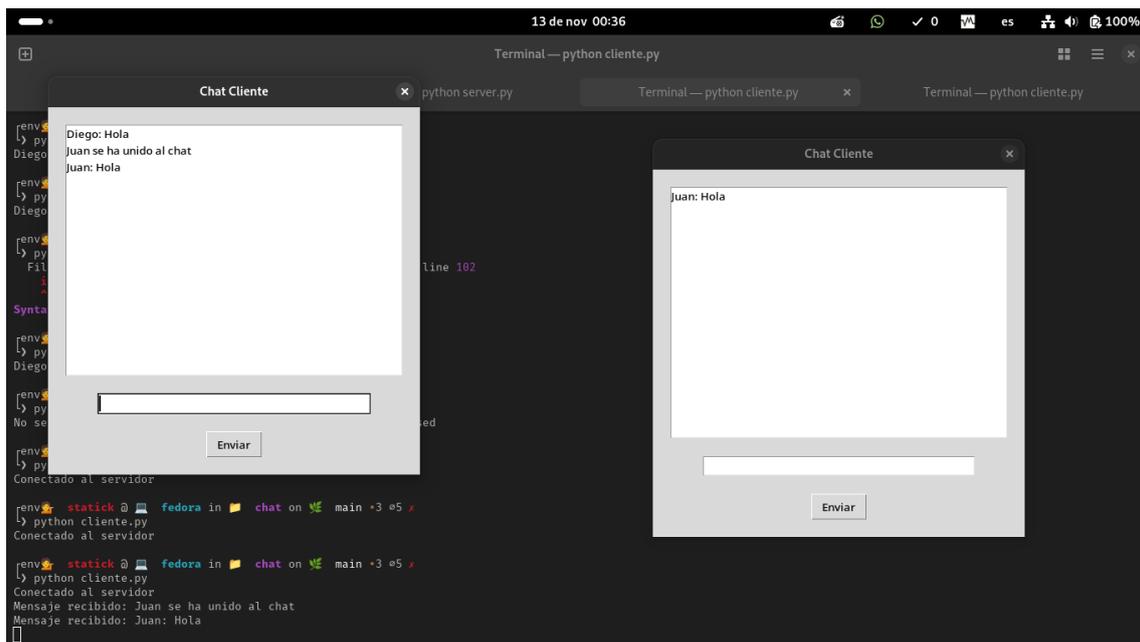


Figure 127.1: Chat local con Sockets y Tkinter

## 127.1 Introducción

En este laboratorio práctico, los estudiantes aprenderán a desarrollar un sistema de chat cliente-servidor utilizando Python. Este laboratorio está diseñado para simular un flujo de trabajo profesional, combinando técnicas modernas de desarrollo como pruebas automatizadas, integración continua y control de versiones con Git y GitHub.

El laboratorio está dividido en varias fases que permiten a los estudiantes comprender y aplicar los fundamentos del desarrollo de aplicaciones de red, utilizando un enfoque incremental para construir funcionalidad de manera ordenada.

## 127.2 Lo que Vamos a Desarrollar

### 127.2.1 Análisis de Requisitos:

- Identificar las funcionalidades clave del sistema, como enviar y recibir mensajes en tiempo real.

### 127.2.2 Historias de Usuario:

- Desarrollar casos prácticos para simular cómo interactuarán los usuarios con el sistema.

### 127.2.3 Diseño y Arquitectura:

- Crear la estructura del proyecto con carpetas organizadas y archivos bien definidos.
- Implementar un diseño modular que facilite la colaboración y el mantenimiento.

### 127.2.4 Codificación:

- Construir el servidor y cliente del sistema en Python utilizando socket.
- Desarrollar pruebas unitarias e integración con pytest.

### 127.2.5 Integración Continua:

- Configurar GitHub Actions para ejecutar pruebas automáticamente con cada cambio en el código.

### 127.2.6 Extensión con GUI:

- Implementar una interfaz gráfica para el cliente utilizando Tkinter.

## 127.3 Objetivos Específicos

- **Entender el Proceso de Desarrollo:** Aplicar principios de diseño modular y buenas prácticas de programación.
- **Desarrollar Habilidades en Python:** Usar bibliotecas estándar como socket y tkinter.
- **Aprender Herramientas Profesionales:** Configurar entornos virtuales, pruebas automatizadas e integración continua.
- **Simular el Trabajo en Equipo:** Utilizar Git y GitHub para la gestión de código y colaboración.

## 127.4 Materiales y Herramientas Necesarias

### 127.4.1 Software:

- Python 3.10 o superior.
- Git y una cuenta de GitHub.
- Editor de código (Visual Studio Code, PyCharm, o similar).
- Librerías y Dependencias:
- pytest para pruebas.
- tkinter para la interfaz gráfica.

### 127.4.2 Conocimientos Previos Requeridos:

- Conceptos básicos de Python.
- Familiaridad con Git y GitHub.
- Conocimientos básicos de programación orientada a objetos (POO).

### 127.4.3 Estructura del Laboratorio

#### 127.4.3.1 Fase 1: Preparación del Entorno y Repositorio.

- Configuración inicial del proyecto con Git y creación del entorno virtual.

#### 127.4.3.2 Fase 2: Creación de Clases Base.

- Implementar las clases del cliente y servidor con métodos vacíos.

#### 127.4.3.3 Fase 3: Comunicación Cliente-Servidor.

- Desarrollar y probar la funcionalidad de envío y recepción de mensajes.

#### 127.4.3.4 Fase 4: Pruebas Automatizadas.

- Crear pruebas unitarias e integración para garantizar la funcionalidad.

#### 127.4.3.5 Fase 5: Configuración de CI/CD.

- Configurar GitHub Actions para pruebas automatizadas.

#### 127.4.3.6 Fase 6: Implementación de Interfaz Gráfica.

- Extender el cliente para incluir una GUI con Tkinter.

#### **127.4.3.7 Fase 7: Documentación y Reflexión.**

- Elaborar conclusiones y responder preguntas de reflexión.

### **127.5 Resultados Esperados**

Al finalizar este laboratorio, los estudiantes tendrán:

- Un sistema de chat cliente-servidor funcional.
- Un repositorio GitHub estructurado con código bien documentado.
- Pruebas automatizadas y un pipeline de CI funcionando correctamente.
- Experiencia en herramientas y técnicas utilizadas en entornos profesionales.

¡Comencemos con el desarrollo!

# 128 Fase 1: Análisis de Requisitos, Historias de Usuario y Preparación del Proyecto

## 128.1 Objetivo

Que los estudiantes comprendan el flujo de desarrollo de una aplicación profesional, desde la planificación hasta la implementación, utilizando herramientas de control de versiones, entornos virtuales y mejores prácticas en Python.

## 128.2 Conceptos Clave

- **Programación en red:** Uso de sockets para la comunicación cliente-servidor.
- **Diseño modular:** Organización del código en clases y métodos.
- **Entornos virtuales:** Aislamiento de dependencias con venv.
- **Control de versiones:** Uso de Git y GitHub.
- **Interfaces gráficas:** Uso de tkinter para desarrollar la GUI del cliente.
- **Historias de usuario:** Identificación de requerimientos clave mediante ejemplos prácticos.

## 128.3 Historias de Usuario

- **HU1:** Como usuario, quiero conectarme al servidor con un apodo único, para identificarme en el chat.
- **HU2:** Como usuario, quiero enviar mensajes al chat y que otros los reciban, para comunicarme con los demás.
- **HU3:** Como usuario, quiero recibir mensajes enviados por otros usuarios en tiempo real, para estar informado de las conversaciones.
- **HU4:** Como administrador, quiero registrar y gestionar conexiones de clientes, para mantener el control del servidor.

### 128.3.1 Instrucciones: Fase 1

- Crear un directorio de proyecto

```
mkdir chat_app
cd chat_app
```

### 128.3.2 Inicializar un repositorio Git

```
git init
git branch -M main
```

- Configurar un entorno virtual

```
python3 -m venv env
source env/bin/activate # En Windows: env\Scripts\activate
```

- Crear el archivo **.gitignore** Contenido sugerido para ignorar archivos innecesarios:

```
env/
__pycache__/
*.pyc
*.pyo
```

Crear un archivo **requirements.txt** Inicialmente vacío. Agregaremos dependencias más adelante.

Estructurar el proyecto

```
mkdir src
touch src/server.py src/client.py
mkdir tests
touch tests/test_server.py tests/test_client.py
```

### 128.4 Planificar la arquitectura inicial

- El servidor manejará conexiones y permitirá la comunicación entre clientes.
- El cliente tendrá una interfaz de línea de comandos (CLI) en esta fase inicial.

### 128.5 Codificar clases base En server.py:

```
"""
Módulo: server.py
Descripción: Define el servidor del sistema de chat local.
"""
import socket
import threading

class ChatServer:
    def __init__(self, host, port):
```

```

    """Inicializa el servidor con la dirección y puerto especificados."""
    pass

def start_server(self):
    """Inicia el servidor y espera conexiones entrantes."""
    pass

def handle_client(self, client_socket, client_address):
    """Maneja la comunicación con un cliente específico."""
    pass

def broadcast(self, message, sender_socket):
    """Envía un mensaje a todos los clientes excepto al remitente."""
    pass

```

## 128.6 En client.py:

```

"""
Módulo: client.py
Descripción: Define el cliente del sistema de chat local.
"""

import socket

class ChatClient:
    def __init__(self, host, port):
        """Inicializa el cliente con la dirección del servidor y el puerto."""
        pass

    def connect_to_server(self):
        """Establece conexión con el servidor."""
        pass

    def send_message(self, message):
        """Envía un mensaje al servidor."""
        pass

    def receive_messages(self):
        """Recibe mensajes del servidor."""
        pass

```

## 128.7 Agregar y confirmar cambios en Git

```
git add .  
git commit -m "Fase 1: Configuración inicial y clases base con métodos vacíos"
```

## 128.8 Pruebas

- En esta fase, no se ejecuta ningún código, pero verifica que los archivos y directorios existen.
- Asegúrate de que el entorno virtual esté activo antes de cualquier ejecución futura.

## 129 Conclusiones

En esta primera fase, los estudiantes aprenden a configurar un proyecto profesional, organizarlo en módulos y versionarlo con Git. Esto sienta las bases para futuras fases en las que se llenarán los métodos y se implementará la lógica del servidor y cliente.

# 130 Fase 2: Implementación del Servidor CLI

## 130.1 Objetivo

Implementar la lógica básica del servidor para gestionar conexiones y comunicaciones entre clientes. Este servidor será accesible a través de la línea de comandos.

## 130.2 Conceptos Clave

- **Sockets TCP:** Configuración de sockets para aceptar conexiones y transmitir datos.
- **Hilos (Threads):** Manejo concurrente de clientes usando la biblioteca threading.
- **Estrategias de Broadcasting:** Comunicación eficiente con múltiples clientes.

## 130.3 Instrucciones

### 130.3.1 1. Actualizar el código del servidor

Abrir `src/server.py` y completar las clases con la lógica básica:

```
"""
Módulo: server.py
Descripción: Define el servidor del sistema de chat local.
"""

import socket
import threading

class ChatServer:
    def __init__(self, host, port):
        """
        Inicializa el servidor con la dirección y puerto especificados.

        :param host: Dirección IP del servidor.
        :param port: Puerto del servidor.
        """
        self.host = host
```

```

self.port = port
self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.clients = {}

def start_server(self):
    """
    Inicia el servidor, permitiendo conexiones entrantes de clientes.
    """
    self.server_socket.bind((self.host, self.port))
    self.server_socket.listen()
    print(f"Servidor escuchando en {self.host}:{self.port}")

    while True:
        client_socket, client_address = self.server_socket.accept()
        print(f"Conexión aceptada de {client_address}")
        threading.Thread(
            target=self.handle_client, args=(client_socket, client_address)
        ).start()

def handle_client(self, client_socket, client_address):
    """
    Maneja la interacción con un cliente específico.

    :param client_socket: Socket del cliente.
    :param client_address: Dirección del cliente.
    """
    nickname = client_socket.recv(1024).decode("utf-8")
    self.clients[client_socket] = nickname
    print(f"{nickname} ({client_address}) se ha conectado")

    self.broadcast(f"{nickname} se ha unido al chat", client_socket)

    while True:
        try:
            message = client_socket.recv(1024).decode("utf-8")
            if not message:
                break
            print(f"Mensaje de {nickname}: {message}")
            self.broadcast(f"{nickname}: {message}", client_socket)
        except Exception as e:
            print(f"Error en cliente {nickname}: {e}")
            break

    print(f"{nickname} ({client_address}) se ha desconectado")
    self.broadcast(f"{nickname} se ha desconectado", client_socket)
    client_socket.close()
    del self.clients[client_socket]

```

```

def broadcast(self, message, sender_socket):
    """
    Envía un mensaje a todos los clientes conectados excepto al remitente.

    :param message: Mensaje a transmitir.
    :param sender_socket: Socket del remitente.
    """
    for client_socket in self.clients.keys():
        if client_socket != sender_socket:
            try:
                client_socket.sendall(message.encode("utf-8"))
            except Exception as e:
                print(f"Error al enviar mensaje: {e}")

if __name__ == "__main__":
    SERVER_HOST = "127.0.0.1"
    SERVER_PORT = 12345
    server = ChatServer(SERVER_HOST, SERVER_PORT)
    server.start_server()

```

## 130.4 2. Probar el servidor

Asegúrate de que el entorno virtual esté activo:

```
source venv/bin/activate # En Windows: venv\Scripts\activate
```

### 130.4.1 Ejecuta el servidor:

```
python src/server.py
```

Verifica que el servidor comienza a escuchar en la dirección configurada.

## 130.5 3. Agregar pruebas unitarias básicas

Edita `tests/test_server.py` para agregar un caso inicial (simulación simple de conexión):

```

import unittest
from src.server import ChatServer

class TestChatServer(unittest.TestCase):

```

```

def setUp(self):
    self.server = ChatServer("127.0.0.1", 12345)

def test_server_initialization(self):
    self.assertEqual(self.server.host, "127.0.0.1")
    self.assertEqual(self.server.port, 12345)
    self.assertIsInstance(self.server.clients, dict)

if __name__ == "__main__":
    unittest.main()

```

Ejecuta las pruebas:

```
python -m unittest discover -s tests
```

## 130.6 4. Versionar los cambios

Agrega y confirma los archivos modificados:

```
git add .
git commit -m "Fase 2: Implementación del servidor básico"
```

(Opcional) Crea un repositorio remoto en GitHub:

```
git remote add origin <URL_DEL_REPOSITORIO>
git push -u origin main
```

## 130.7 Pruebas

- Ejecuta el servidor y asegúrate de que no haya errores.
- Ejecuta las pruebas unitarias.

## 130.8 Conclusiones

En esta fase, se implementó un servidor funcional que puede aceptar múltiples conexiones de clientes y gestionar sus mensajes de forma concurrente. Los estudiantes ahora comprenden la importancia de los sockets y los hilos para crear aplicaciones escalables.

# 131 Fase 3: Implementación del Cliente CLI

## 131.1 Objetivo

Crear la lógica básica del cliente para conectarse al servidor de chat, enviar mensajes y recibir transmisiones, inicialmente usando la línea de comandos (CLI).

## 131.2 Conceptos Clave

**Sockets TCP Cliente:** Uso de sockets para conectar con el servidor. **Hilos en el Cliente:** Permitir el envío y recepción simultánea de mensajes. **Comunicación entre Cliente y Servidor:** Manejo de datos y estructuras de mensajes.

## 131.3 Instrucciones

### 131.3.1 1. Crear la estructura inicial del cliente

Edita `src/client.py` y completa la clase con las funciones esenciales:

```
"""
Módulo: client.py
Descripción: Define el cliente del sistema de chat local.
"""

import socket
import threading

class ChatClient:
    def __init__(self, host, port):
        """
        Inicializa el cliente con la dirección y puerto del servidor.

        :param host: Dirección IP del servidor.
        :param port: Puerto del servidor.
        """
        self.host = host
        self.port = port
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```

self.running = True

def connect_to_server(self):
    """
    Conecta el cliente al servidor especificado.
    """
    try:
        self.client_socket.connect((self.host, self.port))
        print("Conectado al servidor")
        nickname = input("Ingresa tu nickname: ")
        self.client_socket.sendall(nickname.encode("utf-8"))
    except Exception as e:
        print(f"No se pudo conectar al servidor: {e}")
        self.running = False

def send_message(self):
    """
    Envía mensajes al servidor de manera continua.
    """
    while self.running:
        try:
            message = input()
            self.client_socket.sendall(message.encode("utf-8"))
        except Exception as e:
            print(f"Error al enviar mensaje: {e}")
            self.running = False
            break

def receive_messages(self):
    """
    Recibe mensajes del servidor de manera continua.
    """
    while self.running:
        try:
            message = self.client_socket.recv(1024).decode("utf-8")
            if message:
                print(message)
            else:
                break
        except Exception as e:
            print(f"Error al recibir mensaje: {e}")
            self.running = False
            break

def start_client(self):
    """
    Inicia el cliente, manejando la conexión y las operaciones.
    """

```

```

        self.connect_to_server()

        if self.running:
            threading.Thread(target=self.receive_messages, daemon=True).start()
            self.send_message()

if __name__ == "__main__":
    SERVER_HOST = "127.0.0.1"
    SERVER_PORT = 12345
    client = ChatClient(SERVER_HOST, SERVER_PORT)
    client.start_client()

```

### 131.3.2 2. Probar el cliente

Asegúrate de que el servidor está corriendo:

```
python src/server.py
```

En una nueva terminal, ejecuta el cliente:

```
python src/client.py
```

Ingresa un nickname y verifica que el cliente se conecte al servidor.

Abre múltiples terminales y ejecuta el cliente en cada una para probar el sistema.

Envía mensajes entre clientes y confirma que el servidor los retransmite correctamente.

### 131.3.3 3. Agregar pruebas unitarias para el cliente

Edita `tests/test_client.py` para verificar casos básicos:

```

import unittest
from src.client import ChatClient

class TestChatClient(unittest.TestCase):
    def setUp(self):
        self.client = ChatClient("127.0.0.1", 12345)

    def test_client_initialization(self):
        self.assertEqual(self.client.host, "127.0.0.1")
        self.assertEqual(self.client.port, 12345)

    def test_client_socket_creation(self):

```

```
self.assertIsNotNone(self.client.client_socket)

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

Ejecuta las pruebas:

```
python -m unittest discover -s tests
```

#### 131.3.4 4. Versionar los cambios

Agrega y confirma los archivos modificados:

```
git add .
git commit -m "Fase 3: Implementación del cliente CLI"
```

Sube los cambios al repositorio remoto:

```
git push origin main
```

### 131.4 Pruebas

- Conecta múltiples clientes al servidor y verifica que los mensajes se retransmitan correctamente.
- Realiza pruebas unitarias para garantizar la creación adecuada del cliente.

## 132 Conclusiones

En esta fase, se desarrolló un cliente funcional basado en CLI que puede conectarse al servidor y participar en un chat en tiempo real. Los estudiantes pudieron observar cómo manejar la comunicación bidireccional utilizando sockets y threading.

## 133 Fase 4: Extender la Aplicación con una Interfaz Gráfica para el Cliente usando Tkinter

### 133.1 Objetivo

Desarrollar una interfaz gráfica para el cliente del chat usando la biblioteca Tkinter. La interfaz permitirá una experiencia más amigable para enviar y recibir mensajes en tiempo real.

### 133.2 Conceptos Clave

- **Interfaz Gráfica (GUI):** Uso de Tkinter para construir interfaces gráficas básicas en Python.
- **Interactividad\*\*:** Implementar botones, entradas de texto y listas para mostrar mensajes.
- **Hilos en la GUI:** Asegurar que la aplicación gráfica sea responsiva mientras maneja múltiples tareas como enviar y recibir mensajes.

### 133.3 Instrucciones

#### 133.3.1 1. Crear la estructura inicial para la GUI

Edita el archivo `src/client.py` para añadir funcionalidades gráficas. Comienza reemplazando la clase `ChatClient` por una versión ampliada con Tkinter:

```
"""
Módulo: client.py
Descripción: Cliente de chat con interfaz gráfica (Tkinter).
"""

import socket
import tkinter as tk
import threading

class ChatClientGUI:
```

```

def __init__(self, host, port):
    """
    Inicializa el cliente con la dirección y puerto del servidor, y configura la GUI

    :param host: Dirección IP del servidor.
    :param port: Puerto del servidor.
    """
    self.host = host
    self.port = port
    self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.nickname = ""
    self.running = True
    self.chat_window = None
    self.message_box = None
    self.entry_box = None

def connect_to_server(self):
    """
    Conecta el cliente al servidor y envía el nickname.
    """
    try:
        self.client_socket.connect((self.host, self.port))
        print("Conectado al servidor")
    except Exception as e:
        print(f"No se pudo conectar al servidor: {e}")

def send_message(self, message):
    """
    Envía un mensaje al servidor.
    :param message: Mensaje a enviar.
    """
    try:
        self.client_socket.sendall(message.encode("utf-8"))
    except Exception as e:
        print(f"Error al enviar mensaje: {e}")

def receive_messages(self):
    """
    Recibe mensajes del servidor y los muestra en la GUI.
    """
    while self.running:
        try:
            message = self.client_socket.recv(1024).decode("utf-8")
            if message:
                self.chat_window.after(0, self.display_message, message)
            else:
                break
        except Exception as e:

```

```

        print(f"Error al recibir mensaje: {e}")
        break

def display_message(self, message):
    """
    Muestra un mensaje en la ventana de chat.
    :param message: Mensaje recibido.
    """
    self.message_box.insert(tk.END, message)
    self.message_box.yview(tk.END)

def setup_gui(self):
    """
    Configura la ventana principal de la interfaz gráfica.
    """
    self.window = tk.Tk()
    self.window.title("Chat Cliente")

    # Ventana de ingreso de nickname
    self.nickname_window = tk.Frame(self.window)
    self.nickname_window.pack(padx=10, pady=10)

    self.nickname_label = tk.Label(self.nickname_window, text="Nick:")
    self.nickname_label.pack(padx=10, pady=10)

    self.nickname_entry = tk.Entry(self.nickname_window)
    self.nickname_entry.pack(padx=10, pady=10)

    self.nickname_button = tk.Button(
        self.nickname_window, text="OK", command=self.set_nickname
    )
    self.nickname_button.pack(padx=10, pady=10)

    self.window.mainloop()

def set_nickname(self):
    """
    Configura el nickname del usuario y prepara la ventana de chat.
    """
    self.nickname = self.nickname_entry.get()
    if self.nickname:
        self.client_socket.sendall(self.nickname.encode("utf-8"))
        self.nickname_window.pack_forget() # Oculta la ventana de nickname
        self.create_chat_window() # Muestra la ventana de chat

def create_chat_window(self):
    """
    Crea la ventana de chat principal con opciones para enviar y recibir mensajes.

```

```

"""
self.chat_window = tk.Frame(self.window)
self.chat_window.pack(padx=10, pady=10)

self.message_box = tk.Listbox(self.chat_window, height=15, width=50)
self.message_box.pack(padx=10, pady=10)

self.entry_box = tk.Entry(self.chat_window, width=40)
self.entry_box.pack(padx=10, pady=10)

self.send_button = tk.Button(
    self.chat_window, text="Enviar", command=self.on_send_message
)
self.send_button.pack(padx=10, pady=10)

# Hilo para recibir mensajes
threading.Thread(target=self.receive_messages, daemon=True).start()

def on_send_message(self):
    """
    Envía un mensaje al servidor y lo muestra en el cliente.
    """
    message = self.entry_box.get()
    if message:
        self.send_message(message)
        self.display_message(f"{self.nickname}: {message}")
        self.entry_box.delete(0, tk.END)

if __name__ == "__main__":
    SERVER_HOST = "127.0.0.1"
    SERVER_PORT = 12345
    client = ChatClientGUI(SERVER_HOST, SERVER_PORT)
    client.connect_to_server()
    client.setup_gui()

```

### 133.3.2 2. Probar la interfaz gráfica

Asegúrate de que el servidor está ejecutándose:

```
python src/server.py
```

Inicia el cliente con la nueva interfaz gráfica:

```
python src/client.py
```

Conecta múltiples clientes y verifica que puedan enviar y recibir mensajes. Asegúrate de que la interfaz es funcional.

### 133.4 3. Versionar los cambios

Agrega y confirma los archivos modificados:

```
git add .  
git commit -m "Fase 4: Implementación de la interfaz gráfica del cliente con Tkinter"
```

Sube los cambios al repositorio remoto:

```
git push origin main
```

### 133.5 Pruebas

- Verifica que el cliente gráfico se conecta correctamente al servidor.
- Confirma que los mensajes enviados y recibidos se muestran correctamente en la interfaz.
- Realiza pruebas con múltiples clientes para asegurar la funcionalidad del sistema completo.

### 133.6 Conclusiones

En esta fase, se extendió el cliente del chat incorporando una interfaz gráfica interactiva. Esto proporciona una experiencia de usuario más amigable y permite manejar tareas de comunicación en segundo plano sin afectar la responsividad de la interfaz.

# 134 Fase 5: Implementación de Pruebas e Integración Continua

## 134.1 Objetivo

Agregar pruebas unitarias y de integración para asegurar la funcionalidad del sistema. Configurar un flujo de integración continua (CI) usando GitHub Actions para garantizar que los cambios en el código no rompan la aplicación.

## 134.2 Conceptos Clave

- **Pruebas Unitarias:** Validan funcionalidades específicas de componentes individuales del sistema.
- **Pruebas de Integración:** Aseguran que los componentes interactúen correctamente entre sí.
- **Integración Continua (CI):** Automatiza la ejecución de pruebas en cada cambio del código mediante herramientas como GitHub Actions.

## 134.3 Instrucciones

### 134.3.1 1. Configurar un entorno de pruebas

Asegúrate de que tienes las siguientes dependencias instaladas:

```
pip install pytest pytest-mock
```

Crea un archivo **requirements.txt** para incluir las dependencias de desarrollo:

```
pytest  
pytest-mock
```

Agrega este archivo al control de versiones:

```
git add requirements-dev.txt  
git commit -m "Añadido archivo requirements-dev.txt para dependencias de desarrollo"
```

### 134.3.2 2. Escribir pruebas unitarias para el servidor

Crea un directorio `tests/` y un archivo `tests/test_server.py` para las pruebas del servidor:

```
"""
Pruebas unitarias para server.py
"""

import socket
import threading
import pytest
from src.server import ChatServer

@pytest.fixture
def server():
    """
    Configura una instancia del servidor para pruebas.
    """
    chat_server = ChatServer("127.0.0.1", 12345)
    threading.Thread(target=chat_server.start, daemon=True).start()
    return chat_server

def test_server_initialization(server):
    """
    Prueba que el servidor se inicializa correctamente.
    """
    assert server.host == "127.0.0.1"
    assert server.port == 12345
    assert server.clients == []

def test_client_connection(server):
    """
    Prueba que un cliente puede conectarse al servidor.
    """
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server.host, server.port))
    assert client_socket
    client_socket.close()
```

### 134.3.3 3. Escribir pruebas para el cliente

Crea el archivo `tests/test_client.py`:

```

"""
Pruebas unitarias para client.py
"""

import pytest
from src.client import ChatClientGUI

@pytest.fixture
def client():
    """
    Configura un cliente de chat para pruebas.
    """
    return ChatClientGUI("127.0.0.1", 12345)

def test_client_initialization(client):
    """
    Prueba que el cliente se inicializa correctamente.
    """
    assert client.host == "127.0.0.1"
    assert client.port == 12345
    assert client.nickname == ""

```

#### 134.3.4 4. Ejecutar las pruebas

Ejecuta las pruebas localmente para verificar que todo funcione correctamente:

```
pytest tests/
```

Si todo está correcto, verás un resultado como este:

```

===== test session starts =====
collected 3 items

tests/test_client.py .. [ 66%]
tests/test_server.py .. [100%]

===== 3 passed in 0.10s =====

```

#### 134.3.5 5. Configurar integración continua con GitHub Actions

Crea el archivo `.github/workflows/ci.yml` con la siguiente configuración:

```

name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.12"

      - name: Install dependencies
        run: |
          python -m venv venv
          . venv/bin/activate
          pip install -r requirements.txt
          pip install -r requirements-dev.txt

      - name: Run tests
        run: |
          . venv/bin/activate
          pytest tests/

```

Sube los cambios al repositorio:

```

git add .github/workflows/ci.yml
git commit -m "Añadido pipeline de CI con GitHub Actions"
git push origin main

```

Verifica en GitHub que las pruebas se ejecuten automáticamente al hacer un push o pull request.

## 134.4 Pruebas y Validación

- Realiza cambios controlados en el código y verifica que GitHub Actions detecta errores.
- Asegúrate de que las pruebas pasen tanto localmente como en el flujo de CI.

## 134.5 Conclusiones

La incorporación de pruebas unitarias e integración continua asegura la calidad del software y evita errores al integrar cambios. Este flujo refleja prácticas profesionales utilizadas en la industria.

# 135 Fase 6: Documentación del Laboratorio y Reflexión Final

## 135.1 Objetivo

Crear una documentación completa para el laboratorio, detallando cada paso realizado en las fases anteriores, junto con preguntas de reflexión para evaluar el aprendizaje de los estudiantes.

- **Sección 1:** Introducción al Laboratorio
  - Título del Laboratorio
  - Desarrollo de un Chat Local Cliente-Servidor con Pruebas e Integración Continua.
- Descripción
  - En este laboratorio, los estudiantes desarrollarán una aplicación de chat local utilizando Python. El proyecto sigue un flujo profesional, incluyendo análisis de requisitos, diseño, codificación, pruebas, y despliegue de un pipeline de integración continua con GitHub Actions.
- Objetivo General
  - Comprender y aplicar un flujo profesional de desarrollo de software para construir aplicaciones cliente-servidor con Python.
- Sección 2: Instrucciones Prácticas
  - Parte 1: Configuración Inicial
- Crear el repositorio:
  - Usa GitHub para crear un nuevo repositorio.
  - Clona el repositorio en tu máquina local:

```
git clone <URL del repositorio>
```

- Estructura del proyecto:

Crema los siguientes directorios y archivos iniciales:

```
chat-project/  
  .github/  
    workflows/  
      ci.yml  
  src/  
    client.py  
    server.py  
  tests/  
    test_client.py  
    test_server.py  
requirements.txt  
requirements-dev.txt  
.gitignore  
README.md
```

- Entorno virtual:
  - Crea un entorno virtual y activa:

```
python -m venv venv  
source venv/bin/activate # Linux/Mac  
venv\Scripts\activate # Windows
```

- Instala dependencias:
  - Añade las librerías necesarias a requirements.txt y requirements-dev.txt, e instálalas:

```
pip install -r requirements.txt  
pip install -r requirements-dev.txt
```

## 136 Parte 2: Codificación por Fases

### 136.1 Implementa las clases base:

- Define las clases ChatServer y ChatClientGUI con métodos iniciales vacíos (pass).
- Añade funcionalidades gradualmente, como se explicó en fases anteriores.

### 136.2 Pruebas:

- Implementa pruebas unitarias y de integración en la carpeta tests/.

#### 136.2.1 Ejecuta las pruebas:

```
pytest tests/
```

### 136.3 Integración continua:

- Configura el pipeline de CI en GitHub Actions para validar las pruebas automáticamente.

### 136.4 Interfaz gráfica (opcional):

- Implementa la GUI en client.py usando Tkinter.

### Sección 3: Preguntas de Reflexión

### 136.5 Diseño y Arquitectura:

- ¿Qué ventajas tiene dividir el proyecto en módulos (src/ y tests/)?
- ¿Cómo mejora el diseño la inclusión de docstrings en las clases y funciones?

## 136.6 Pruebas:

- ¿Por qué es importante incluir pruebas unitarias e integración en proyectos colaborativos?
- ¿Qué aprendiste al ejecutar y depurar tus pruebas?

## 136.7 Integración Continua:

- ¿Cómo asegura la CI la calidad del código en un proyecto con múltiples desarrolladores?
- ¿Qué beneficios aporta GitHub Actions en comparación con ejecutar pruebas localmente?

## 136.8 Desafíos Técnicos:

- ¿Qué desafíos enfrentaste al implementar la comunicación entre cliente y servidor?
- ¿Cómo los resolviste?

**Sección 4:** Conclusión

## 136.9 Resultados Obtenidos:

- Los estudiantes construyeron un sistema cliente-servidor funcional con comunicación en tiempo real.
- Implementaron pruebas automatizadas y configuraron CI, simulando un flujo profesional de desarrollo.

## 136.10 Lecciones Clave:

- La planificación y estructura del proyecto son esenciales para su éxito.
- La automatización de pruebas y flujos de trabajo asegura la calidad del software.

## 136.11 Próximos Pasos:

- Implementar características avanzadas como manejo de usuarios, historial de chat, o cifrado de mensajes.
- Extender el proyecto para que funcione a través de redes externas.

## **136.12 Entrega Final**

### **136.12.1 Repositorio GitHub:**

- Sube todo el proyecto a un repositorio en GitHub.
- Asegúrate de que el README.md incluya instrucciones claras para ejecutar la aplicación y el flujo de CI.

### **136.13 Evidencias:**

- Capturas de pantalla de las pruebas exitosas.
- URL del repositorio de GitHub.

¡Felicidades por completar el laboratorio!

**Part XII**

**Pruebas Técnicas**

# 137 Prueba Técnica para Programador Junior - Python

**Duración total:** 30 minutos

**Objetivo:** Evaluar conocimientos básicos de programación orientada a objetos, lógica de programación y uso de estructuras de datos en Python.

## Tip

- Por favor leer claramente las instrucciones antes de empezar
- Crea un fork de este repositorio y trabaja en tu propio espacio

[https://github.com/statick88/prueba\\_tecnica\\_junior\\_python](https://github.com/statick88/prueba_tecnica_junior_python)

## 137.1 Parte 1: Programación Orientada a Objetos (15 puntos)

**Duración recomendada:** 15 minutos

### 137.1.1 Clase Producto (15 puntos)

- Crea una clase Producto con los siguientes atributos y métodos:
- **Atributos:**
  - nombre (str)
  - precio (float)
  - cantidad (int)
- **Métodos:**
  - `__init__`: Inicializa los atributos.
  - **calcular\_total**: Devuelve el precio total del producto (precio × cantidad).
  - Implementa un método `__str__` para mostrar la información del producto en el formato:

```
Producto(nombre=Nombre, precio=Precio, cantidad=Cantidad)
```

### 137.1.2 Ejemplo de uso:

```
p1 = Producto("Laptop", 1500.0, 2)
print(p1.calcular_total()) # 3000.0
print(p1) # Producto(nombre=Laptop, precio=1500.0, cantidad=2)
```

## 137.2 Parte 2: Lógica de Programación y Estructuras de Datos (15 puntos)

### 137.2.1 Fibonacci Recursivo (10 puntos)

Escribe una función `fibonacci(n)` que calcule el  $n$ -ésimo número de la secuencia de Fibonacci de forma recursiva.

#### Tip

La secuencia de Fibonacci es:  
0, 1, 1, 2, 3, 5, 8, ....

### 137.2.2 Ejemplo de uso:

```
print(fibonacci(6)) # 8
```

### 137.2.3 Análisis de texto (5 puntos)

Crea una función `analizar_texto(texto)` que reciba un `texto` como `cadena` y devuelva un `diccionario` con:

- Número de palabras.
- Número de letras.

### 137.2.4 Ejemplo de uso:

```
texto = "Hola mundo"
print(analizar_texto(texto))
# {'palabras': 2, 'letras': 9}
```

- **Puntaje total:** 30 puntos
- **Aprobación mínima:** 20 puntos

## 138 Criterios de Evaluación

### 138.1 Parte 1: Programación Orientada a Objetos (15 puntos)

1. Definición de clase y atributos (3 puntos):
  - La clase Producto está correctamente definida con los atributos nombre, precio, y cantidad.
2. Método `__init__` (3 puntos):
  - El constructor inicializa correctamente los atributos.
3. Método `calcular_total` (4 puntos):
  - Calcula correctamente el precio total ( $\text{precio} \times \text{cantidad}$ ).
4. Método `__str__` (5 puntos):
  - Devuelve una representación clara y en el formato especificado:

```
Producto(nombre=Nombre, precio=Precio, cantidad=Cantidad)
```

- No hay errores al mostrar el objeto con `print()`.

### 138.2 Parte 2: Lógica de Programación y Estructuras de Datos (15 puntos)

1. Fibonacci Recursivo (10 puntos):
  - Lógica correcta (6 puntos):
    - La función calcula correctamente el n-ésimo número de Fibonacci.
  - Eficiencia básica (2 puntos):
    - Se aplica recursividad correctamente sin errores de lógica.
  - Resultados esperados (2 puntos):
    - La función produce los resultados esperados para diferentes valores de n.
2. Análisis de Texto (5 puntos):

- Cálculo del número de palabras (2 puntos):
  - Se cuenta correctamente el número de palabras en el texto.
- Cálculo del número de letras (2 puntos):
  - Se cuentan correctamente las letras (excluyendo espacios).
- Formato del diccionario (1 punto):
  - La función devuelve un diccionario con las claves palabras y letras correctamente estructurado.

### 138.3 Notas de calificación:

- **Puntaje total:** 30 puntos.
- **Aprobación mínima:** 20 puntos.
- **Errores menores:** Se puede restar hasta 1 punto por detalles menores (nombres incorrectos de variables o errores de formato).
- **Errores críticos:** Fallos que generan excepciones o resultados incorrectos en los ejemplos restan la totalidad de los puntos asignados al criterio correspondiente.

# 139 Resolución de la Prueba Técnica para Programador Junior - Python

## 139.1 Parte 1: Programación Orientada a Objetos

### 139.1.1 Archivo: parte1/producto.py

Descripción de la clase Producto:

- Atributos:
  - nombre: String
  - precio: Float
  - cantidad: Integer
- Métodos:
  - `__init__(self, nombre, precio, cantidad)`: Constructor que inicializa los atributos.
  - `calcular_total(self)`: Calcula y devuelve el precio total, que es `precio * cantidad`.
  - `__str__(self)`: Devuelve una representación en formato "Producto(nombre=Nombre, precio=Precio, cantidad=Cantidad)".

### 139.1.2 Implementación de la clase Producto:

```
class Producto:
    def __init__(self, nombre, descripcion, precio, cantidad):
        if cantidad < 0:
            raise ValueError("La cantidad no puede ser negativa")
        self.nombre = nombre
        self.descripcion = descripcion
        self.precio = precio
        self.cantidad = cantidad

    def calcular_precio_total(self):
        return self.precio * self.cantidad

    def __str__(self):
        return f"Producto(nombre={self.nombre}, descripcion={self.descripcion}, precio={self.precio}, cantidad={self.cantidad})"
```

- Se ha implementado la clase Producto con sus atributos y métodos.

- El método `__init__` inicializa los atributos de la clase.
- El método `calcular_precio_total` calcula el precio total del producto.
- El método `__str__` devuelve una representación en formato “Producto(nombre=Nombre, precio=Precio, cantidad=Cantidad)”.

## 139.2 Parte 2: Lógica de Programación y Estructuras de Datos

### 139.2.1 Función fibonacci (Parte 2):

La función fibonacci debe ser recursiva. El enésimo número de la secuencia de Fibonacci se calcula como:

- $\text{fibonacci}(0) = 0$
- $\text{fibonacci}(1) = 1$
- $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$  para  $n > 1$

### 139.2.2 Implementación de la función fibonacci:

```
import string

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

def fibonacci_memo(n, memo={0: 0, 1: 1}):
    if n not in memo:
        memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]
```

- Se ha implementado la función fibonacci de forma recursiva.
- Se ha implementado la función `analizar_texto` que cuenta el número de palabras y letras en un texto.

### 139.2.3 Función analizar\_texto (Parte 2):

La función `analizar_texto` recibe un texto y devuelve un diccionario con dos claves:

- ‘palabras’: Número de palabras en el texto.
- ‘letras’: Número de letras en el texto (sin contar espacios ni signos de puntuación).

#### 139.2.4 Implementación de la función analizar\_texto:

```
def analizar_texto(texto):
    palabras = texto.split()

    letras = sum(len(''.join(c for c in palabra if c.isalpha()))
                 for palabra in palabras)

    return {'palabras': len(palabras), 'letras': letras}
```

- Se ha implementado la función analizar\_texto que cuenta el número de palabras y letras en un texto.
- La función split divide el texto en palabras.
- Se utiliza una expresión generadora para contar el número de letras en cada palabra.
- El resultado se devuelve como un diccionario con las claves 'palabras' y 'letras'.

#### 139.2.5 Archivo runner.py

## 140 Conclusión:

- Se ha implementado la clase Producto con sus métodos en la Parte 1.
- Se han implementado las funciones fibonacci y analizar\_texto en la Parte 2.
- El script runner.py permite ejecutar todas las pruebas y calcular la puntuación final.

### 140.1 Ahora puedes ejecutar las pruebas de la siguiente manera:

```
python3 runner.py
```

Esto ejecutará todas las pruebas y generará un resumen con la puntuación final.