Django y React 2024

Diego Saavedra

Sep 13, 2024

Table of contents

1 Bienvenido **23**
1.1 ¿De qué trata este curso? . . . 23
1.2 ¿Para quién es este curso? . . . 23
1.3 ¿Cómo contribuir? . . . 23

I Unidad 1: Introducción a Python **25**

2 Git y GitHub **26**
2.1 ¿Qué es Git y GitHub? . . . 26
2.2 ¿Quiénes utilizan Git? . . . 27
2.3 ¿Cómo se utiliza Git? . . . 27
2.4 ¿Para qué sirve Git? . . . 28
2.5 ¿Por qué utilizar Git? . . . 29
2.6 ¿Dónde puedo utilizar Git? . . . 30
2.7 Pasos Básicos . . . 30
2.8 Instalación de Visual Studio Code . . . 31
2.8.1 Descarga e Instalación de Git . . . 32
2.8.2 Configuración . . . 33
2.8.3 Creación de un Repositorio “helloWorld” en Python . . . 33
2.8.4 Comandos Básicos de Git . . . 34
2.8.5 Estados en Git . . . 34

3 Tutorial: Moviendo Cambios entre Estados en Git **35**
3.1 Introducción . . . 35
3.2 Sección 1: Modificar Archivos en el Repositorio . . . 35
3.3 Mover Cambios de Local a Staging: . . . 35
3.4 Agregar Cambios de Local a Staging: . . . 36
3.5 Sección 2: Confirmar Cambios en un Commit . . . 36
3.6 Mover Cambios de Staging a Commit: . . . 36
3.7 Sección 3: Creación y Fusión de Ramas . . . 36
3.8 Crear una Nueva Rama: . . . 36
3.9 Implementar Funcionalidades en la Rama: . . . 36
3.10 Fusionar Ramas con la Rama Principal: . . . 37
3.11 Sección 4: Revertir Cambios en un Archivo . . . 37
3.12 Revertir Cambios en un Archivo: . . . 37
3.13 Conclusión . . . 37

4 Asignación **38**

5 GitHub Classroom **39**
5.1 ¿Qué es GitHub Classroom? 39
5.1.1 Funcionalidades Principales 39
5.2 Ejemplo Práctico 40
5.2.1 Creación de una Asignación en GitHub Classroom 40
5.3 Trabajo de los Estudiantes 42

6 Docker **48**

7 Conceptos Básicos de Docker **49**
7.1 Imagen 49
7.2 Contenedor 49
7.3 Dockerfile 49
7.4 Docker Compose 50

8 Uso de Docker **51**
8.1 Definir un Dockerfile 51
8.2 Construir la Imagen 51
8.3 Ejecutar un Contenedor 52
8.4 Gestionar Contenedores 52
8.5 Docker Compose 52

II Unidad 2: Python Básico **54**

9 Hola Mundo en Python **55**

10 Introdución **56**
10.0.1 Paso 1: Instalación de Python 56
10.0.2 Paso 2: Instalación de Python en Windows 56
10.0.3 Paso 3: Crear nuestro primer “Hola Mundo” en Python 59

11 Sintaxis Básica **62**

12 Comentarios **63**

13 Variables y Tipos de Datos **64**

14 Tipos de Datos **65**

15 Operadores **67**

16 Estructura de Control **71**

17 Funciones **76**

18 Llamada a Funciones **77**

19 Parámetros y Argumentos **78**

20 Retorno **79**

21 Ejemplo **80**

22 Asignación **82**
22.1 Objetivo 82
22.2 ¿Qué debes hacer? 82
22.3 Pruebas 82
22.4 Ejecución 83

III Unidad 3: Python Intermedio **84**

23 Listas **85**

24 Tuplas **86**

25 Manipulación de Listas y Tuplas **87**

26 Funciones ingegradas para Listas y Tuplas **88**

27 Listas Anidadas **89**

28 Listas y Tuplas como Argumentos de Funciones **90**

29 Listas y Tuplas como Retorno de Funciones **91**

30 Asignación **92**
30.1 Descripción de la Asignación 92
30.2 Tarea Pendiente: 92
30.3 Cómo Ejecutar el Código 92
30.4 Ejemplo de salida: 92

31 Diccionarios **94**

32 Conjuntos **99**

33 Operaciones con Diccionarios y Conjuntos **103**

34 Asignación **105**
34.1 Descripción 105
34.2 Criterios de Evaluación 105

IV Unidad 4: Python Avanzado **107**

35 Programación Orientada a Objetos **109**
35.1 Asignación 115
35.2 Instrucciones 115
35.3 Contenido del Repositorio 115
35.4 Cómo Ejecutar las Pruebas 116

36 Módulos **118**

37 Paquetes **121**

38 Creación y Uso de Módulos **124**
38.1 Creación de Módulos 124
38.2 Uso de Módulos 124
38.3 Importar Funciones Específicas 124
38.4 Importar con Alias 125
38.5 Importar Todas las Funciones 125

39 Creación y Uso de Paquetes **127**
39.1 Creación de Paquetes 127
39.2 Uso de Paquetes 127
39.3 Importar con Alias 127
39.4 Importar Todas las Funciones 127

40 Asignación Calculadora Pythonica **130**
40.1 Instrucciones 130
40.2 Contenido del Repositorio 130
40.3 Ejercicio 130
40.4 Cómo Ejecutar el Programa 130
40.5 Cómo Ejecutar las Pruebas 131

41 Uso de Pypi **133**

42 Utilizar algún paquete de Pypi **135**

43 Conclusión **138**

44 Públicar un paquete en Pypi **140**

45 Creación del archivo README.md **144**

46 Creación del archivo __main__.py **147**

47 Creación del archivo LICENSE **149**

48 Creación del archivo .gitignore **151**

49 Creación de la cuenta en Pypi **153**

50 Publicar el paquete en Pypi **155**

51 Instalar el paquete **158**

52 Uso del paquete **160**

53 Conclusión **162**

54 Desarrollo de un Sistema de Gestión de Inventarios en Python **164**
54.1 Objetivos 164
54.2 Entregables 164

54.3 Instrucciones . . . 164
54.3.1 1. Crear la Estructura de Datos . . . 164
54.3.2 2. Agregar Productos . . . 165
54.3.3 3. Búsqueda y Filtrado . . . 165
54.3.4 4. Actualización de Inventario . . . 165
54.3.5 5. Generación de Informes . . . 166
54.3.6 6. Pruebas Unitarias . . . 166
54.3.7 7. Documentación y GitHub Classroom . . . 166
54.3.8 Evaluación . . . 166
54.4 Asignación . . . 167

V Unidad 5: Django 168

55 Introducción a Django 169
55.1 Conceptos Importantes . . . 169
55.1.1 Entornos Virtuales . . . 170
55.1.2 Modelo Template View (MTV) . . . 171
55.1.3 Formularios . . . 172
55.1.4 Administrador de Django . . . 173
55.1.5 Middleware . . . 173
55.1.6 Autenticación y Autorización . . . 173
55.1.7 Internacionalización . . . 173
55.1.8 Seguridad . . . 174
55.1.9 Testing . . . 174
55.1.10 Despliegue . . . 174

56 Configuración inicial de un proyecto. 175
56.1 1. Crear un entorno virtual . . . 175
56.2 2. Activar el entorno virtual . . . 175
56.3 3. Instalar Django . . . 176
56.4 4. Crear un proyecto de Django . . . 176
56.5 5. Crear una aplicación de Django . . . 177
56.6 6. Crear una vista . . . 177
56.7 7. Configurar las URL . . . 178
56.8 8. Ejecutar el servidor de desarrollo . . . 178
56.9 9. Crear una migración . . . 179
56.10 10. Aplicar una migración . . . 180
56.11 12. Crear un superusuario . . . 180
56.12 13. Acceder al panel de administración . . . 181

57 Ejercicio 183

58 Asignación 185

59 Estructura de archivos y carpetas 186

60 Creación de una aplicación Django 188

61 Configuración de la base de datos 189

62 Crear una vista **190**

63 Crear una plantilla **191**

64 Configurar las rutas **192**

65 Correr el servidor de desarrollo **193**

66 Acceder a la aplicación **194**

67 Acceder a la aplicación **195**

68 Asignación **196**

69 Referencias **198**

70 Modelos **199**

71 Registramos la aplicacion en admin.py **201**

72 Vistas en Django **202**
72.1 Listar productos . . . 202
72.2 Agregar producto . . . 202
72.3 Actualizar producto . . . 203
72.4 Eliminar producto . . . 203
72.5 Buscar producto . . . 204

73 Templates **205**
73.1 Base . . . 205
73.2 Listar . . . 206
73.3 Agregar . . . 206
73.4 Actualizar . . . 207
73.5 Eliminar . . . 208
73.6 Buscar . . . 208

74 URLs **210**
74.1 URLs en la aplicación y el proyecto . . . 210

75 Ejecutar el servidor **212**

76 Conclusiones **213**

77 Django Rest Framework **214**
77.1 ¿Qué es una API REST? . . . 214
77.2 Instalación . . . 214
77.3 Actualizar el archivo requirements.txt . . . 214
77.4 Serializers . . . 215
77.5 Views . . . 215
77.6 URLs de la Aplicación . . . 216
77.7 Configuración URLs del Proyecto . . . 216
77.8 Migraciones . . . 216

77.9 Instalación de setuptools 217
77.10Ejecución 217

78 Documentación de la API con drf-yasg **218**

79 Documentación de la API con CoreAPI **220**
79.1 Primero instalamos CoreAPI: 220

80 Bases de Dato en Django **221**
80.1 Objetivos 221

81 Creación del Entorno Virtual **223**

82 Instalación de Django **224**

83 Creación del Proyecto **225**

84 Configuración de la Base de Datos **226**
84.1 Base de Datos Relacional (PostgreSQL) 226

85 Creación de las variables de entorno **228**
85.1 Configuración de la Base de Datos en Django 229

86 Instalación de los drivers de PostgreSQL y MongoDB **230**

87 Modelos de Base de Datos **231**
87.1 Modelo de Libro 231
87.2 Modelo de Autor 231

88 Migraciones **233**

89 Django REST Framework **234**

90 Serializadores **235**

91 Vistas **236**

92 Rutas **237**

93 Documentación de la API **238**

94 Instalación de setuptools **239**

95 Actualización de requirements.txt **240**

96 Ejecución del Servidor **241**

97 Probar la API con Thunder Client **242**

98 Reto **244**

99 Conclusión **245**

100 Testing **246**

101 Corrección de tests en Django Rest Framework **248**
101.1 Introducción . . . 248
101.2 Pasos . . . 248
101.2.1 1. Crear un entorno virtual . . . 248
101.2.2 2. Activar el entorno virtual . . . 249
101.2.3 3. Instalar las dependencias . . . 249
101.2.4 4. Correr los tests . . . 249
101.2.5 5. Corregir los tests . . . 249
101.2.6 6. Correr los tests nuevamente . . . 249
101.2.7 7. Desactivar el entorno virtual . . . 249
101.3 Conclusión . . . 249
101.4 Referencias . . . 250

VI Unidad 6: Frontend **251**

102 HTML **252**

103 Resumen **253**
103.1 Visión general de HTML . . . 253
103.2 Una breve introducción a los conceptos clave en HTML. . . . 253
103.2.1 Encabezado de un documento HTML . . . 254
103.2.2 Párrafo . . . 254
103.2.3 Tabla . . . 254
103.3 Estructura del documento . . . 254
103.3.1 Aprende cómo estructurar tus documentos HTML con una base sólida. 254
103.4 Metadatos . . . 255
103.4.1 Cómo utilizar etiquetas meta para proporcionar información sobre tus documentos. . . . 255
103.5 HTML semántico . . . 255
103.5.1 Usar los elementos HTML correctos para describir el contenido de tu documento. . . . 255
103.6 Encabezados y secciones . . . 256
103.6.1 Cómo utilizar correctamente los elementos de sección para darle significado a tu contenido. . . . 256
103.7 Atributos . . . 257
103.7.1 Aprende sobre los diferentes atributos globales junto con los atributos específicos de elementos HTML particulares. . . . 257
103.8 HTML particulares. . . . 258
103.8.1 Aprende sobre los diferentes atributos globales junto con los atributos específicos de elementos HTML particulares. . . . 258
103.9 Conceptos básicos de texto . . . 258
103.9.1 Cómo formatear texto utilizando HTML. . . . 258
103.10 Enlaces . . . 259
103.10.1 Todo lo que necesitas saber sobre enlaces. . . . 259
103.11 Listas . . . 259
103.11.1 Cómo crear listas en HTML. . . . 259

103.12 Navegación . . . 260
103.12.1 La navegación es un elemento clave de cualquier sitio o aplicación, y comienza con HTML. . . . 260
103.13 Tablas . . . 260
103.13.1 Comprender cómo utilizar tablas para marcar datos tabulares. . . . 260
103.14 Formularios . . . 261
103.14.1 Una visión general de los formularios en HTML. . . . 261
103.15 Imágenes . . . 262
103.15.1 Una visión general de las imágenes en HTML. . . . 262
103.16 Audio y video . . . 262
103.16.1 Descubre cómo trabajar con medios HTML como audio y video. . . . 262
103.17 Plantilla, ranura y sombra . . . 263
103.17.1 Una explicación de plantilla, ranura y sombra. . . . 263
103.18 APIs de HTML . . . 263
103.18.1 Aprende cómo se puede exponer y manipular información HTML utilizando JavaScript. . . . 263
103.19 Enfoque . . . 264
103.19.1 Cómo gestionar el orden de enfoque en tus documentos HTML. . . . 264
103.20 Otros elementos de texto en línea . . . 264
103.20.1 Una introducción a la variedad de elementos utilizados para marcar texto. . . . 264
103.21 Detalles y resumen . . . 265
103.21.1 Descubre cómo funcionan los elementos de detalles y resumen, muy útiles, y dónde usarlos. . . . 265
103.22 Diálogo . . . 265
103.22.1 El elemento es un elemento útil para representar cualquier tipo de diálogo en HTML, descubre cómo funciona. . . . 265
103.23 Elementos de agrupación . . . 266
103.23.1 Descubre cómo utilizar los elementos de agrupación en HTML. . . . 266
103.24 Elementos de contenido incrustado . . . 267
103.24.1 Descubre cómo incrustar contenido en tu documento HTML. . . . 267
103.25 Elementos de formulario . . . 267
103.25.1 Descubre cómo utilizar los elementos de formulario en HTML. . . . 267
103.26 Elementos de marco . . . 268
103.26.1 Descubre cómo utilizar los elementos de marco en HTML. . . . 268
103.27 Elementos de interacción . . . 268
103.27.1 Descubre cómo utilizar los elementos de interacción en HTML. . . . 268
103.28 Elementos de lista . . . 269
103.28.1 Descubre cómo utilizar los elementos de lista en HTML. . . . 269

104 Reto **270**

105 Conclusiones **272**

106 CSS **273**
106.1 Actividad . . . 273
106.2 Actividad . . . 274
106.3 Actividad . . . 275

107 Selectores **277**
107.1 Actividad 278

108 Comentarios **280**
108.1 Actividad 280

109 Colores, valores y convenciones **281**
109.1 Actividad 281

110 Border **284**
110.1 Actividad 284

111 Border en profundidad **286**
111.1 Actividad 287

112 Unidades de medida **288**
112.1 Actividad 288

113 Background **290**
113.1 Actividad 290

114 Box model, margin, padding, overflow **292**
114.1 Actividad 293

115 Outline, text-align, text-decoration, text-shadow **294**
115.1 Actividad 295

116 Fuentes custom **297**
116.1 Actividad 297

117 Links y sus estados **299**
117.1 Actividad 299

118 Listas **301**
118.1 Actividad 301

119 Tablas **303**
119.1 Actividad 303

120 Display, max-width y position **305**
120.1 Actividad 312

121 Float **318**
121.1 Actividad 319

122 inline-block **324**
122.1 Actividad 325

123 Centrar un elemento **330**
123.1 Actividad 333

124 Proyecto: Danto estilo a un portafolio personal de HTML **339**
124.1 Actividad 342

125 Frameworks de CSS **345**

126 Proyecto: Danto estilo a un portafolio personal de HTML con Bootstrap, **346**
126.1 Actividad 348

127 Proyecto: Danto estilo a un portafolio personal de HTML con Tailwind CSS **350**
127.1 Actividad 352

128 Proyecto: Danto estilo a un portafolio personal de HTML con Bulma. **354**
128.1 Actividad 356

129 Reto **358**

130 Conclusiones **364**

131 Recursos **365**

132 JavaScript **366**
132.1 ¿Qué es JavaScript? 366
132.2 ¿Qué es programar? 367
132.3 ¿Por qué aprender JavaScript? 367
132.4 Quiz 367

133 La consola del navegador **368**
133.1 Cómo abrir la consola 368
133.2 Quiz 369

134 Editores y entornos de desarrollo **370**

135 Tipos de Datos **372**
135.1 Tipos Primitivos. 372
135.2 Números 372
135.3 Operadores aritméticos 373
135.4 ¿Qué significa el // que ves en los ejemplos? 373
135.5 Cadenas de texto 373
135.6 Concatenación 374
135.7 Booleanos 374
135.8 Quiz 374

136 Los operadores de comparación **376**
136.1 Actividad 376
136.2 Comparando cadenas de texto 377
136.3 ¿Y si usamos el operador > con cadenas de texto? 377
136.4 Comparando booleanos 377
136.5 Comparando valores de diferentes tipos 378
136.6 Quiz 378

137 Operadores lógicos en JavaScript **379**
137.1 Operador lógico AND && . . . 379
137.2 Operador lógico OR || . . . 379
137.3 Operador lógico NOT ! . . . 379
137.4 Combinando operadores lógicos, aritméticos y de comparación . . . 380
137.5 Actividad . . . 380

138 Dos o más operandos **382**
138.1 Quiz . . . 382

139 Variables **383**
139.1 Actividad . . . 384
139.2 Constantes const . . . 384
139.3 Actividad . . . 384
139.4 Variables var . . . 385
139.5 El nombre de las variables . . . 385
139.6 Convenciones y nomenclaturas . . . 385
139.7 Quiz . . . 386
139.8 null y undefined . . . 387
139.9 La diferencia entre null y undefined . . . 387
139.10 Actividad . . . 388
139.11 Comparaciones con null y undefined . . . 388
139.12 Quiz . . . 389

140 Operador typeof **390**
140.1 Actividad . . . 390
140.2 Usando con operadores de comparación . . . 391
140.3 Actividad . . . 391
140.4 Quiz . . . 391
140.5 Comentarios . . . 392
140.6 Comentarios de una sola línea // . . . 392
140.7 Comentarios de varias líneas /* */ . . . 392
140.8 Quiz . . . 393
140.9 console.log() . . . 393
140.10 Sintaxis . . . 393
140.11 Más métodos de console . . . 394
140.12 Quiz . . . 395

141 Código Condicional con if **396**
141.1 else . . . 396
141.2 Anidación de condicionales . . . 397
141.3 La importancia de las llaves . . . 398
141.4 Quiz . . . 399
141.5 Actividad . . . 399

142 Bucles con while **400**
142.1 Sintaxis . . . 400
142.2 Ejemplo de uso de while . . . 400
142.3 Cuidado con los bucles infinitos . . . 401

142.4 Saliendo de un bucle con break 401
142.5 Saltando una iteración con continue 402
142.6 Anidación de bucles 403
142.7 Quiz 404

143 Bucles con do while **406**

144 confirm **407**

145 Usando do while **408**
145.1 Quiz 408

146 Expresiones y declaraciones **409**

147 Declaraciones **410**

148 Expresiones **411**

149 ¿Por qué es importante la diferencia? **412**

150 Quiz **413**

151 Bucles con for **414**

152 Ejemplo de uso de for **415**

153 Incrementando números con ++ **416**

154 Iterando al revés **417**

155 Llevando más allá el bucle for **418**

156 continue y break **419**

157 Bucles anidados con for **420**

158 Quiz **421**

159 Switch **422**

160 Sintaxis **423**

161 switch vs if **425**

162 Agrupando cases **426**

163 ¡No olvides el break! **427**

164 El patrón switch(true) **428**

165 Quiz **429**

166 Tu primera función **431**

167 Devolviendo un resultado 432
168 Una función realmente útil 433
169 Quiz 435
170 Parámetros 436
171 Parámetros y argumentos 437
172 El orden de los parámetros importa 438
173 Quiz 439
174 Function Expression 440
175 ¿Qué es una function expression? 441
176 Hoisting 442
177 ¿Y qué pasa con las function expression? 443
178 Quiz 444
179 Funciones flecha 445
180 Ventajas de las funciones flecha 446
181 Return implícito 447
182 Quiz 448
183 Recursividad 449
184 Ejemplo de recursividad 450
185 Usando recursividad y devolviendo un valor 451
186 Quiz 452
187 Arrays: colecciones de elementos 453
188 Declaración y asignación de arrays 454
189 Acceso a los elementos de un array 455
190 Modificar elementos de un array 456
191 Quiz 457
192 Métodos y propiedades de Array 458
193 La longitud de un array 459

194 Métodos de arrays 460

195 .push() 461

196 .pop() 462

197 .shift() 463

198 .unshift() 464

199 Concatenar arrays 465

200 Actividad Práctica 466

201 Iteración de Arrays en JavaScript 467

202 Bucle while 468

203 Bucle for 469

204 Bucle for...of 470

205 El método array.forEach() 471

206 Actividad Práctica 473

207 Búsqueda en Arrays con sus métodos 474

208 indexOf: ¿En qué posición está el elemento? 475

209 includes: ¿El elemento existe en el Array? 476

210 some: ¿Alguno de los elementos cumple con la condición? 477

211 every: ¿Todos los elementos cumplen con la condición? 478

212 find: Devuelve el primer elemento que cumple con la condición 479

213 findIndex: Devuelve el índice del primer elemento que cumple con la condición 480

214 Actividad Práctica 481

215 Ordenamiento de Arrays en JavaScript 482

216 Ordenamiento básico con sort() 483

217 Ordenamiento personalizado con sort() 484

218 sort() y toSorted() 485

219 Actividad Práctica 486

220 Transformación de Arrays en JavaScript 487

221filter: Un nuevo Array con los elementos que cumplan una condición **488**

222map: Un nuevo Array con los elementos transformados **489**

223map + filter: Un nuevo Array con los elementos transformados y filtrados **490**

224reduce: Un único valor a partir de un array **491**

225Actividad Práctica **493**

226Matrices **494**
226.1Creación de Matrices 494

227Acceso a los Elementos de una Matriz **495**

228Iteración sobre Matrices **496**

229Ejemplo práctico de Matrices **497**

230Actividad Práctica **499**

231Algoritmos con Arrays **500**

232Algoritmo para encontrar el número mayor **501**

233Complejidad algorítmica **502**

234Búsqueda binaria **503**

235Implementación de la búsqueda binaria **504**

236Actividad Práctica **506**

237Objetos **507**

238Declaración y asignación de objetos **508**

239Acceder a propiedades y métodos de un objeto **510**

240Añadir y modificar propiedades de un objeto **512**

241Eliminar propiedades de un objeto **513**

242Actividad Práctica **514**

243Atajos para trabajar con objetos **515**
243.1Atajo al crear un objeto 515

244Destructuración: el atajo al recuperar propiedades **516**

245Renombrar variables y valores por defecto **518**

246Objetos anidados y atajo **519**

247 Quiz **520**

248 Iterando Objetos en JavaScript **522**
248.1 Iterando con for...in . . . 522

249 Transformar un objeto en un array **523**
249.1 Iterar con Object.keys() . . . 523
249.2 Iterando con Object.values() . . . 524
249.3 Iterando con Object.entries() . . . 524

250 Actividad Práctica **526**

251 Operador de Encadenamiento Opcional **527**

252 El error fatal de intentar leer una propiedad de undefined **528**

253 Evitar el error con if **529**
253.1 El operador in para comprobar si una propiedad existe . . . 529
253.2 El operador de Encadenamiento Opcional, ?. . . . 530
253.3 Beneficios del Operador de Encadenamiento Opcional . . . 531

254 Actividad Práctica **532**

255 Rest Operator **533**
255.1 Sintaxis del operador de propagación . . . 533
255.2 Usos comunes del operador de propagación . . . 533
255.3 Rest Operator vs Spread Operator . . . 534

256 Actividad Práctica **535**

257 Promesas y async/await **536**
257.1 Crear una promesa . . . 536
257.2 Encadenar promesas . . . 536
257.3 async/await . . . 537
257.4 Beneficios de async/await . . . 538

258 Actividad Práctica **539**

259 Quiz **541**

260 Módulos **542**
260.1 Exportar e importar . . . 542
260.2 Exportar e importar por defecto . . . 542
260.3 Exportar e importar todo . . . 543
260.4 Módulos en el navegador . . . 543
260.5 Módulos en Node.js . . . 544
260.6 Módulos en CommonJS . . . 544

261 Actividad Práctica **546**

262 Quiz **547**

263 Nodejs **548**

264 1. Introducción a Node.js **549**
264.1 ¿Por qué es Node.js? . . . 549
264.2 Historia y Evolución de Node.js . . . 549
264.3 Características de Node.js . . . 549

265 2. Manejadores de Paquetes en Node.js **550**
265.1 npm (Node Package Manager) . . . 550
265.2 yarn . . . 550
265.3 pnpm . . . 550

266 3. Manejadores de Versiones de Node.js **551**
266.1 nvm (Node Version Manager) . . . 551
266.2 fnm (Fast Node Manager) . . . 551

267 4. Entornos de Ejecución de JavaScript **552**

268 Node.js **553**
268.1 Deno . . . 553
268.2 Bun . . . 553

269 5. Fundamentos del Asincronismo en Node.js **554**

270 6. Callbacks **555**
270.1 Ejemplo de Callback: . . . 555

271 7. Promesas **556**
271.1 Ejemplo de Promesa: . . . 556

272 8. Async/Await **557**
272.1 Ejemplo de async/await: . . . 557

273 9. Introducción a Node.js en la práctica **558**
273.1 Crear un Servidor HTTP Asíncrono . . . 558

274 10. Gestión de Paquetes con NPM **559**
274.1 Comandos básicos de NPM: . . . 559
274.2 Instalar create-react-app: . . . 559

275 Tutorial de Fetch **560**
275.1 Requisitos . . . 560
275.2 Instrucciones . . . 560

276 Deploy **564**

277 Reto **565**

278 Fetch con Fake Store API **566**

279 Reto **569**

280 Fetch con Rick and Morty API. 570

281 Reto 573

282 Fetch con otras APIs 574

283 Reto 577
283.1 Resumen . . . 577
283.2 Conclusión . . . 577
283.3 Referencias . . . 578

284 Introducción a React 579

285 1. ¿Qué es React? 580
285.1 Características clave de React: . . . 580

286 2. Evolución de React: De create-react-app a Vite 581
286.1 React con create-react-app (Antes). . . . 581
286.2 Desventajas de create-react-app: . . . 583
286.3 React con Vite (Ahora) . . . 583
286.3.1 Ventajas de Vite: . . . 583
286.4 Crear un proyecto React con Vite: . . . 584

287 3. Conceptos Básicos de React 586
287.1 3.1. Componentes en React . . . 586
287.2 3.2. JSX: JavaScript + HTML . . . 588
287.3 3.3. Props y Estado . . . 590
287.4 3.4. Hooks . . . 592
287.5 Ejemplo con useState y useEffect: . . . 593
287.6 4. Diferencias entre el Uso Antiguo y el Actual de React . . . 596
287.7 Ventajas del enfoque moderno con Hooks: . . . 596

288 Microproyecto con React 597
288.1 1. Crear un Proyecto con Vite . . . 597
288.2 2. Crear Componentes . . . 598
288.3 3. Usar el Componente TodoList . . . 599
288.4 4. Ejecutar el Proyecto . . . 600

289 Reto 601

290 Conclusión: React Antes y Ahora con Vite 614

291 Consumir una API con React con Fetch. 615
291.1 1. Crear un Proyecto React con Vite y Tailwind CSS . . . 615
291.1.1 Pasos: . . . 616
291.2 2. Estructura del Proyecto . . . 617
291.3 3. Creación de los Componentes . . . 617
291.3.1 ProductCard.jsx . . . 617
291.3.2 ProductList.jsx . . . 618
291.4 4. Integración en App.jsx . . . 619

291.55. Estilizado con Tailwind CSS 619
291.66. Ejecución del Proyecto 619

292 Extra **621**
292.1 Agregar el Cambio de Tema de Oscuro a Claro 621
292.1.1 Actualiza App.jsx: 621
292.1.2 Cambios importantes: 622
292.1.3 Asegúrate de Configurar Tailwind para el Modo Oscuro: 622
292.1.4 Actualiza los Estilos en los Componentes: 622
292.1.5 Cambios importantes: 623
292.2 Crear un Build de Producción 623

293 Conclusiones **624**

294 Consumir una API con React con Axios. **625**
294.11. Crear un Proyecto React con Vite y Tailwind CSS 625
294.2 Pasos: 626
294.32. Estructura del Proyecto 627
294.43. Creación de los Componentes 627
294.4.1 ProductCard.jsx 627
294.4.2 ProductList.jsx 628
294.54. Integración en App.jsx 629
294.65. Estilizado con Tailwind CSS 630
294.76. Ejecución del Proyecto 630
294.7.1 Crear un Build de Producción 630

295 Conclusiones **631**

296 CRUD en React con API REST **632**
296.1 Requisitos previos: 632

297 Conclusiones **639**

VII Ejercicios **640**

298 Desarrollo del Backend para un E-Commerce con Django Rest Framework **641**
298.11. Configuración Inicial del Proyecto 641
298.1.1 1.1. Crear un Proyecto Django 641
298.1.2 1.2 Crear una Aplicación Django 641
298.1.3 1.3 Instalar Django Rest Framework 641
298.1.4 1.4 Configurar el Proyecto 641
298.22. Definir el Modelo de Datos 642
298.2.1 2.1 Crear Modelos en products/models.py 642
298.2.2 2.2 Crear y Aplicar Migraciones 642
298.33. Crear Serializers 642
298.3.1 3.1 Definir Serializers en products/serializers.py 642
298.44. Crear Vistas y Rutas 643
298.4.1 4.1 Definir Vistas en products/views.py 643
298.54.2 Configurar Rutas en products/urls.py 644

298.64.3 Incluir las URLs en ecommerce_project/urls.py 644
298.75. Probar la API 644
298.7.1 5.1 Ejecutar el Servidor de Desarrollo 644
298.85.2 Probar los Endpoints 645

299 Extra 646
299.11. Instalar Django Rest Swagger 646
299.22. Configurar Django Rest Swagger 646
299.33. Configurar las URLs 647
299.44. Probar la Documentación 647

300 Ejercicios de Git y Github 648
300.0.1 Ejercicio 1 648
300.0.2 Ejercicio 2 648
300.0.3 Ejercicio 3 648
300.0.4 Ejercicio 4 649
300.0.5 Ejercicio 5 649

301 Ejercicios Python - Nivel 1 650
301.1 Ejercicio 1 650
301.2 Ejercicio 2 650
301.3 Ejercicio 3 650
301.4 Ejercicio 4 651
301.5 Ejercicio 5 651

302 Ejercicios Python - Nivel 2 652
302.1 Ejercicio 1 652
302.2 Ejercicio 2 652
302.3 Ejercicio 3 652
302.4 Ejercicio 4 653
302.5 Ejercicio 5 653

303 Ejercicios Python - Nivel 3 654
303.1 Ejercicio 1: 654
303.2 Ejercicio 2: 654
303.3 Ejercicio 3: 654
303.4 Ejercicio 4: 655
303.5 Ejercicio 5: 655

304 Ejercicios Python - Nivel 4 656
304.1 Ejercicio 1: 656
304.2 Ejercicio 2: 656
304.3 Ejercicio 3: 657
304.4 Ejercicio 4: 657
304.5 Ejercicio 5: 658

1 Bienvenido

¡Bienvenido al Curso Completo de Django y React!

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

1.1 ¿De qué trata este curso?

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

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

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

1.2 ¿Para quién es este curso?

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

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

1.3 ¿Cómo contribuir?

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

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

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

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

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

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

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

Part I

Unidad 1: Introducción a Python

2 Git y GitHub

Figure 2.1: Git and Github

2.1 ¿Qué es Git y GitHub?

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

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

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

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

2.2 ¿Quiénes utilizan Git?

Figure 2.2: Git

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

2.3 ¿Cómo se utiliza Git?

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

    Commit with error

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

    Adding common html code in a form

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

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

Figure 2.3: Git en Terminal

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

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

2.4 ¿Para qué sirve Git?

Figure 2.4: Seguimiento de Cambios con Git

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

2.5 ¿Por qué utilizar Git?

Figure 2.5: Ventajas de Git

Ofrece varias ventajas, como:

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

2.6 ¿Dónde puedo utilizar Git?

Figure 2.6: Git en Diferentes Sistemas Operativos

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

2.7 Pasos Básicos

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

2.8 Instalación de Visual Studio Code

Figure 2.7: Visual Studio Code

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

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

2.8.1 Descarga e Instalación de Git

Figure 2.8: Git

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

2.8.2 Configuración

Figure 2.9: Configuración de Git

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

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

2.8.3 Creación de un Repositorio "helloWorld" en Python

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

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

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

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

```
git init
```

- Añade el archivo al área de preparación con:

```
git add hello_world.py
```

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

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

2.8.4 Comandos Básicos de Git

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

2.8.5 Estados en Git

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

---

3 Tutorial: Moviendo Cambios entre Estados en Git

3.1 Introducción

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

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

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

3.2 Sección 1: Modificar Archivos en el Repositorio

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

3.3 Mover Cambios de Local a Staging:

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

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

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

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

3.4 Agregar Cambios de Local a Staging:

```
git add hello_world.py
```

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

3.5 Sección 2: Confirmar Cambios en un Commit

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

3.6 Mover Cambios de Staging a Commit:

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

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

3.7 Sección 3: Creación y Fusión de Ramas

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

3.8 Crear una Nueva Rama:

```
git branch feature
```

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

3.9 Implementar Funcionalidades en la Rama:

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

3.10 Fusionar Ramas con la Rama Principal:

```
git merge feature
```

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

3.11 Sección 4: Revertir Cambios en un Archivo

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

3.12 Revertir Cambios en un Archivo:

```
git reset hello_world.py
```

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

3.13 Conclusión

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

4 Asignación

Hello World!

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

La Asignación

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

El comando de ejecución del test es:

```
pytest test_hello.py
```

¡Mucha suerte!

5 GitHub Classroom

Figure 5.1: Github Classroom

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

5.1 ¿Qué es GitHub Classroom?

Figure 5.2: Github Classroom Windows

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

5.1.1 Funcionalidades Principales

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

criterios de evaluación.

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

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

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

5.2 Ejemplo Práctico

5.2.1 Creación de una Asignación en GitHub Classroom

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

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

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

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

5.3 Trabajo de los Estudiantes

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

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

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

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

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

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

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

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

```
pytest
```

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

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

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

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

Este Action lo que hace es evaluar los cambios realizados

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

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

Tip

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

6 Docker

Figure 6.1: Docker

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

7 Conceptos Básicos de Docker

7.1 Imagen

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

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

7.2 Contenedor

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

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

7.3 Dockerfile

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

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

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

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

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

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

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

7.4 Docker Compose

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

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

8 Uso de Docker

8.1 Definir un Dockerfile

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

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

# Establece el directorio de trabajo
WORKDIR /app

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

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

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

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

8.2 Construir la Imagen

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

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

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

8.3 Ejecutar un Contenedor

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

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

8.4 Gestionar Contenedores

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

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

8.5 Docker Compose

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

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

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

Part II

Unidad 2: Python Básico

9 Hola Mundo en Python

10 Introdución

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

10.0.1 Paso 1: Instalación de Python

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

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

10.0.2 Paso 2: Instalación de Python en Windows

1. **Descargar Python**

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

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

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

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

2. **Instalar Python**

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

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

3. **Comprobar que tenemos instalado Python**

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

```
python --version
```

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

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

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

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

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

1. **Instalar Visual Studio Code**

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

https://code.visualstudio.com/

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

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

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

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

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

3. Este código imprimirá "Hola Mundo" en la consola.

4. Ejecutar el programa

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

¡Felicidades!

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

11 Sintaxis Básica

Figure 11.1: Python

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

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

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

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

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

12 Comentarios

Los comentarios en python se escriben con el símbolo #.

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

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

13 Variables y Tipos de Datos

Tip

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

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

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

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

```
x = 5                                       ①
y = "Hola Mundo"                            ②
```

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

14 Tipos de Datos

En python, los tipos de datos más comunes son:

- **int**: Entero

Ejemplo:

```
x = 5 ①
```

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

- **float**: Flotante

Ejemplo:

```
y = 5.0 ①
```

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

- **str**: Cadena de texto

Ejemplo:

```
z = "Hola Mundo" ①
```

- **bool**: Booleano

Ejemplo:

```
a = True ①
```

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

- **list**: Lista

Ejemplo:

```
b = [1, 2, 3] ①
```

- **tuple**: Tupla

Ejemplo:

```
c = (1, 2, 3)  ①
```

- **dict**: Diccionario

Ejemplo:

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

- **set**: Conjunto

Ejemplo:

```
e = {1, 2, 3}  ①
```

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

- **None**: Nulo

Ejemplo:

```
f = None  ①
```

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

15 Operadores

En python, los operadores más comunes son:

- +: Suma

Ejemplo:

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

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

- -: Resta

Ejemplo:

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

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

- **“*“**: Multiplicación

Ejemplo:

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

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

- /: División

Ejemplo:

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

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

- //: División entera

Ejemplo:

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

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

- %: Módulo

Ejemplo:

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

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

- “”**: Potencia

Ejemplo:

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

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

- ==: Igualdad

Ejemplo:

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

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

- !=: Diferente

Ejemplo:

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

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

- >: Mayor que

Ejemplo:

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

- <: Menor que

Ejemplo:

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

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

- >=: Mayor

Ejemplo:

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

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

- <=: Menor

Ejemplo:

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

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

- **and**: Y

Ejemplo:

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

- **or**: O

Ejemplo:

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

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

- **not**: Negación

Ejemplo:

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

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

16 Estructura de Control

En python, las estructuras de control más comunes son:

- **if**: Si

Ejemplo:

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

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

- **elif**: Si no

Ejemplo:

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

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

- **else**: Si no

Ejemplo:

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

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

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

- **for**: Para

Ejemplo:

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

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

- **while**: Mientras

Ejemplo:

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

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

- **break**: Romper

Ejemplo:

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

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

- **continue**: Continuar

Ejemplo:

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

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

- **pass**: Pasar

Ejemplo:

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

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

- **return**: Retornar

Ejemplo:

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

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

- **def**: Definir

Ejemplo:

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

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

- **try**: Intentar

Ejemplo:

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

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

- **except**: Excepto

Ejemplo:

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

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

- **finally**: Finalmente

Ejemplo:

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

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

- **raise**: Lanzar

Ejemplo:

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

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

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

- **assert**: Afirmar

Ejemplo:

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

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

- **import**: Importar

Ejemplo:

```
import math  ①
```

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

- **from**: Desde

Ejemplo:

```
from math import sqrt  ①
```

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

- **as**: Como

Ejemplo:

```
import math as m  ①
```

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

17 Funciones

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

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

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

18 Llamada a Funciones

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

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

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

19 Parámetros y Argumentos

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

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

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

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

20 Retorno

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

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

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

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

21 Ejemplo

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

El funcionamiento del programa es el siguiente:

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

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

Ver Código

```
# Ejemplo práctico: Calculadora de propinas

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

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

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

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

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

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

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

22 Asignación

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

Ejercicios Python Básico

22.1 Objetivo

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

22.2 ¿Qué debes hacer?

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

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

22.3 Pruebas

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

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

22.4 Ejecución

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

Part III

Unidad 3: Python Intermedio

23 Listas

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

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

Ejemplo:

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

Ejercicio:

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

Solución

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

24 Tuplas

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

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

Ejemplo:

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

Ejercicio:

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

Solución

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

25 Manipulación de Listas y Tuplas

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

Ejemplo:

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

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

Ejercicio:

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

Solución

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

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

26 Funciones ingegradas para Listas y Tuplas

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

Ejemplo:

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

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

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

Ejercicio:

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

Solución

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

27 Listas Anidadas

Las listas anidadas son listas que contienen otras listas.

Ejemplo:

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

Ejercicio:

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

Solución

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

28 Listas y Tuplas como Argumentos de Funciones

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

Ejemplo:

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

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

Ejercicio:

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

Solución

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

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

29 Listas y Tuplas como Retorno de Funciones

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

Ejemplo:

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

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

Ejercicio:

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

Solución

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

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

30 Asignación

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

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

30.1 Descripción de la Asignación

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

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

30.2 Tarea Pendiente:

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

30.3 Cómo Ejecutar el Código

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

```
pytest -s
```

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

30.4 Ejemplo de salida:

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

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

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

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

31 Diccionarios

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

Ejemplo:

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

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

Ejercicio:

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

Respuesta

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

ciudad = mi_diccionario["ciudad"]

print(ciudad)  # Bogotá

print(mi_diccionario["nombre"])  # Juan

edad = mi_diccionario["edad"]

print(edad)  # 25
```

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

Respuesta

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

Ejemplo:

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

# nombre
# ciudad
# profesion
```

Ejercicio:

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

Respuesta

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

# nombre
# ciudad
# profesion
```

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

Ejemplo:

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

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

Ejercicio:

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

Respuesta

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

# nombre Juan
# ciudad Bogotá
# profesion Ingeniero

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

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

Ejemplo:

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

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

Ejercicio:

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

Respuesta

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

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

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

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

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

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

32 Conjuntos

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

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

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

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

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

Ejemplo:

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

Ejercicio:

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

Respuesta

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

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

Ejemplo:

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

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

```
# 10
# 11
```

Ejercicio:

- Recorrer el conjunto mi_conjunto e imprimir los elementos.

Respuesta

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

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

Ejemplo:

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

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

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

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

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

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

Ejercicio:

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

Respuesta

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

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

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

33 Operaciones con Diccionarios y Conjuntos

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

Ejercicio:

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

Respuesta

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

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

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

Ejercicio:

- Crear un diccionario con las siguientes claves y valores:

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

Respuesta

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

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

34 Asignación

https://classroom.github.com/a/93tJXiLB

34.1 Descripción

Esta asignación consiste en corregir y ejecutar un test unitario para un diccionario de frutas. Instrucciones

1. Abre el archivo **ejercicio.py**.
2. Corrige el diccionario frutas para que tenga las siguientes parejas **clave-valor**:

```
"manzana" - "roja"
"banana" - "amarilla"
"pera" - "verde"
"naranja" - "naranja"
```

3. Guarda y cierra el archivo **ejercicio.py**.
4. Ejecuta el test unitario test_ejercicio en tu terminal con el comando:

```
python -m unittest test_ejercicio.py
```

5. Si el **test unitario** se ejecuta sin errores, habrás **completado la asignación**.
6. Si el **test unitario** arroja errores, **corrige** el **diccionario frutas** en **ejercicio.py** y **vuelve** a **ejecutar el test unitario**.
7. Repite los pasos 4 a 6 hasta que el test unitario se ejecute sin errores.

34.2 Criterios de Evaluación

- El diccionario frutas en **ejercicio.py** tiene las parejas clave-valor correctas.
- El test unitario test_frutas en **test_ejercicio.py** se ejecuta sin errores.

> Tip
>
> Se sugiere revisar la sección de **Ejercicios Python - Nivel 4** para poder reforzar los conocimientos necesarios para completar esta sección.

Part IV

Unidad 4: Python Avanzado

35 Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que usa "objetos" para diseñar aplicaciones y programas informáticos.
Un objeto es una entidad que agrupa un estado (atributos) y un comportamiento (métodos). Por ejemplo, un objeto podría representar a una persona con atributos como nombre, edad, género, etc., y comportamientos como caminar, hablar, respirar, etc.
La programación orientada a objetos se basa en varios conceptos fundamentales:

- Clases
- Objetos
- Atributos
- Métodos
- Herencia
- Polimorfismo
- Abstracción
- Encapsulamiento

Estos conceptos se explican a continuación.

- Clases: Una clase es un plano para los objetos. Es un diseño que define un objeto. Una clase puede contener atributos y métodos. Por ejemplo, la clase "Perro" puede tener atributos como raza, color, edad, y métodos como ladrar, comer, dormir, etc.

Ejemlo:

```
class Perro:
    def __init__(self, raza, color, edad):
        self.raza = raza
        self.color = color
        self.edad = edad

    def ladrar(self):
        print("Guau! Guau!")
```

Ejericio:
Crear una clase llamada "Persona" con los atributos "nombre", "edad" y "género". La clase debe tener un método llamado "hablar" que imprime "Hola, mi nombre es [nombre]".
Ver respuesta

```python
class Persona:
    def __init__(self, nombre, edad, genero):
        self.nombre = nombre
        self.edad = edad
        self.genero = genero

    def hablar(self):
        print(f"Hola, mi nombre es {self.nombre}")
```

- Objetos: Un objeto es una instancia de una clase. Cuando se crea un objeto, se reserva memoria para el objeto y se inicializa. Un objeto puede tener atributos y métodos. Por ejemplo, si "Perro" es una clase, entonces "Perro Labrador" es un objeto de la clase "Perro".

Ejemplo:

```python
perro1 = Perro("Labrador", "Dorado", 5)
perro2 = Perro("Bulldog", "Blanco", 3)
```

Ejercicio:
Crear un objeto de la clase "Persona" con nombre "Juan", edad 25 y género "Masculino".
Ver respuesta

```python
juan = Persona("Juan", 25, "Masculino")
```

- Atributos: Los atributos son variables que pertenecen a un objeto. Los atributos definen las características de un objeto. Por ejemplo, "raza", "color" y "edad" son atributos de la clase "Perro".

Ejemplo:

```python
print(perro1.raza)  # Labrador
print(perro2.color)  # Blanco
```

Ejericio:
Imprimir el nombre de la persona "Juan".
Ver respuesta

```python
print(juan.nombre)  # Juan
```

- Métodos: Los métodos son funciones que pertenecen a un objeto. Los métodos definen el comportamiento de un objeto. Por ejemplo, "ladrar" y "comer" son métodos de la clase "Perro".

Ejemplo:

```
perro1.ladrar()  # Guau! Guau!
```

Ejercicio:
Llamar al método "hablar" del objeto "juan".
Ver respuesta

```
juan.hablar()  # Hola, mi nombre es Juan
```

- Herencia: La herencia es un mecanismo en el que una clase adquiere las propiedades y el comportamiento de otra clase. La clase que hereda se llama clase derivada o subclase, y la clase de la que se hereda se llama clase base o superclase. La herencia permite la reutilización del código y la creación de una jerarquía de clases.

Ejemplo:

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

    def comer(self):
        print("Comiendo...")

class Perro(Animal):
    def ladrar(self):
        print("Guau! Guau!")

perro = Perro("Firulais")
perro.comer()  # Comiendo...
perro.ladrar()  # Guau! Guau!
```

Ejercicio:
Crear una clase "Estudiante" que herede de la clase "Persona". La clase "Estudiante" debe tener un atributo adicional llamado "carrera" y un método llamado "estudiar" que imprime "Estudiando…".
Ver respuesta

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

    def estudiar(self):
        print("Estudiando...")
```

- Polimorfismo: El polimorfismo es la capacidad de un objeto para tomar muchas formas. En Python, el polimorfismo se logra mediante el uso de métodos con el mismo nombre en diferentes clases. El polimorfismo permite que un objeto se comporte de diferentes maneras según el contexto.

Ejemplo:

```
class Animal:
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        print("Guau! Guau!")

class Gato(Animal):
    def hablar(self):
        print("Miau! Miau!")

animales = [Perro(), Gato()]

for animal in animales:
    animal.hablar()
```

Ejercicio:
Crear una clase "Profesor" con un método "enseñar" que imprime "Enseñando...". Luego, crear una lista de objetos que contenga un objeto de la clase "Estudiante" y un objeto de la clase "Profesor". Llamar al método "hablar" de cada objeto en la lista.
Ver respuesta

```
class Profesor:
    def enseñar(self):
        print("Enseñando...")

profesor = Profesor()
estudiante = Estudiante("Ana", 20, "Femenino", "Ingeniería")

personas = [profesor, estudiante]

for persona in personas:
    persona.hablar()
```

- Abstracción: La abstracción es el proceso de ocultar los detalles de implementación y mostrar solo la funcionalidad al usuario. En Python, la abstracción se logra mediante el uso de clases y métodos. Los usuarios pueden interactuar con los objetos sin conocer los detalles internos de cómo funcionan.

Ejemplo:

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

    def arrancar(self):
        print("Arrancando...")
```

Ejercicio:
Crear una clase "Círculo" con un atributo "radio" y un método "calcular_area" que imprime el área del círculo. Luego, crear un objeto de la clase "Círculo" con radio 5 y llamar al método "calcular_area".
Ver respuesta

```python
import math

class Circulo:
    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        area = math.pi * self.radio ** 2
        print(f"El área del círculo es {area}")

circulo = Circulo(5)
circulo.calcular_area()
```

- Encapsulamiento: El encapsulamiento es el proceso de ocultar los detalles de implementación de un objeto y restringir el acceso a ciertos componentes. En Python, el encapsulamiento se logra mediante el uso de métodos y atributos privados. Los métodos y atributos privados no se pueden acceder directamente desde fuera de la clase.

Ejemplo:

```python
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.__velocidad = 0

    def acelerar(self):
        self.__velocidad += 10

    def frenar(self):
        self.__velocidad -= 10

    def get_velocidad(self):
        return self.__velocidad

coche = Coche("Toyota", "Corolla")
coche.acelerar()
print(coche.get_velocidad())  # 10
coche.frenar()

# Intentar acceder al atributo privado directamente
# print(coche.__velocidad)  # Error
```

Ejercicio:
Modificar la clase "Círculo" para que el atributo "radio" sea privado. Agregar métodos "get_radio" y "set_radio" para obtener y establecer el valor del radio. Luego, crear un objeto de la clase "Círculo" con radio 5, cambiar el radio a 10 y llamar al método "calcular_area".
Ver respuesta

```python
import math

class Circulo:
    def __init__(self, radio):
        self.__radio = radio

    def calcular_area(self):
        area = math.pi * self.__radio ** 2
        print(f"El área del círculo es {area}")

    def get_radio(self):
        return self.__radio

    def set_radio(self, radio):
        self.__radio = radio

circulo = Circulo(5)
circulo.set_radio(10)
circulo.calcular_area()
```

La programación orientada a objetos es un concepto fundamental en Python y en muchos otros lenguajes de programación. Al comprender los conceptos de clases, objetos, atributos, métodos, herencia, polimorfismo, abstracción y encapsulamiento, puedes diseñar y desarrollar aplicaciones y programas más eficientes y reutilizables.

35.1 Asignación

https://classroom.github.com/a/LVvqQCln
En esta asignación, aprenderás sobre los conceptos básicos de **Programación Orientada a Objetos (POO)** mediante la implementación de clases en Python.

35.2 Instrucciones

1. Lee cuidadosamente el contenido de este repositorio.
2. Implementa las clases solicitadas en el archivo main.py.
3. Realiza los commits y push necesarios para subir tus cambios a este repositorio.
4. Verifica que tus cambios pasen todas las pruebas.

35.3 Contenido del Repositorio

- **main.py**: Archivo principal donde implementarás tus clases.
- **test_main.py**: Archivo de pruebas unitarias.
- **README.md**: Este archivo con las instrucciones de la asignación.

35.4 Cómo Ejecutar las Pruebas

- Asegúrate de tener Python instalado en tu sistema.
- Abre una terminal y navega hasta la ubicación de este repositorio.
- Ejecuta **python -m unittest** para correr las pruebas.

36 Módulos

Un módulo es un archivo que contiene definiciones y declaraciones de Python. El archivo debe tener una extensión .py. Las definiciones de un módulo pueden ser importadas a otros módulos o al programa principal.
Ejemplo:

```
# modulo.py
def saludar():
    print("Hola, bienvenido a Python")
```

```
# programa.py
import modulo

modulo.saludar()
```

Ejercicio:
Crear un módulo llamado operaciones.py que contenga las siguientes funciones:

- suma(a, b): Retorna la suma de a y b
- resta(a, b): Retorna la resta de a y b
- multiplicacion(a, b): Retorna la multiplicación de a y b
- division(a, b): Retorna la división de a y b

Ver respuesta
operaciones.py

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

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

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

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

programa.py

```python
import operaciones

a = 10
b = 5

print(operaciones.suma(a, b))
print(operaciones.resta(a, b))
print(operaciones.multiplicacion(a, b))
print(operaciones.division(a, b))
```

37 Paquetes

Un paquete es un conjunto de módulos organizados en un directorio. Un paquete debe contener un archivo llamado **__init__.py**.py. Este archivo puede estar vacío o contener código de inicialización del paquete.
Ejemplo:

```
paquete/
    __init__.py
    modulo1.py
    modulo2.py
```

```python
# modulo1.py
def saludar():
    print("Hola, bienvenido a Python")
```

```python
# modulo2.py
def despedir():
    print("Adiós, hasta luego")
```

```python
# programa.py
from paquete import modulo1, modulo2

modulo1.saludar()
modulo2.despedir()
```

Ejercicio:
Crear un paquete llamado operaciones que contenga los módulos suma.py, resta.py, multiplicacion.py y division.py. Cada módulo debe contener una función que realice la operación correspondiente.
Ver respuesta
operaciones/ **init**.py suma.py resta.py multiplicacion.py division.py
suma.py

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

resta.py

```python
def resta(a, b):
    return a - b
```

multiplicacion.py

```python
def multiplicacion(a, b):
    return a * b
```

division.py

```python
def division(a, b):
    return a / b
```

programa.py

```python
from operaciones import suma, resta, multiplicacion, division

a = 10
b = 5

print(suma.suma(a, b))
print(resta.resta(a, b))
print(multiplicacion.multiplicacion(a, b))
print(division.division(a, b))
```

38 Creación y Uso de Módulos

> **Tip**
>
> La diferencia principal entre paquetes y módulos es que los paquetes son directorios que contienen módulos y un archivo _____init___.py, mientras que los módulos son archivos individuales que contienen funciones y variables.

38.1 Creación de Módulos

Para crear un módulo, simplemente se crea un archivo con extensión .py y se definen las funciones y variables que se desean exportar.
Ejemplo:

```
# modulo.py
def saludar():
    print("Hola, bienvenido a Python")
```

38.2 Uso de Módulos

Para usar un módulo, se utiliza la palabra reservada import seguida del nombre del módulo.
Ejemplo:

```
# programa.py
import modulo

modulo.saludar()
```

38.3 Importar Funciones Específicas

También es posible importar funciones específicas de un módulo.
Ejemplo:

```
# programa.py
from modulo import saludar

saludar()
```

38.4 Importar con Alias

Es posible importar un módulo o función con un alias.
Ejemplo:

```
# programa.py

import modulo as m

m.saludar()
```

38.5 Importar Todas las Funciones

También es posible importar todas las funciones de un módulo.
Ejemplo:

```
# programa.py

from modulo import *

saludar()
```

39 Creación y Uso de Paquetes

39.1 Creación de Paquetes

Para crear un paquete, se crea un directorio con el nombre del paquete y se agregan los módulos necesarios. Además, se debe crear un archivo _____init___.py en el directorio del paquete.
Ejemplo:

```
paquete/
    __init__.py
    modulo1.py
    modulo2.py
```

39.2 Uso de Paquetes

Para usar un paquete, se utiliza la palabra reservada import seguida del nombre del paquete y el nombre del módulo.
Ejemplo:

```python
# programa.py

from paquete import modulo1, modulo2

modulo1.saludar()
modulo2.despedir()
```

39.3 Importar con Alias

Es posible importar un paquete o módulo con un alias.
Ejemplo:

```python
from paquete import modulo1 as m1, modulo2 as m2

m1.saludar()
m2.despedir()
```

39.4 Importar Todas las Funciones

También es posible importar todas las funciones de un módulo.
Ejemplo:

```python
# programa.py

from paquete.modulo1 import *

saludar()
```

40 Asignación Calculadora Pythonica

https://classroom.github.com/a/Kyffdibl
En esta asignación, aprenderás sobre la creación y uso de módulos y paquetes en Python.

40.1 Instrucciones

1. Lee cuidadosamente el contenido de este repositorio.
2. Implementa las funciones solicitadas en el archivo **operaciones.py**.
3. Completa el programa principal en el archivo **programa.py**.
4. Realiza los commits y push necesarios para subir tus cambios a este repositorio.
5. Verifica que tus cambios funcionen correctamente.

40.2 Contenido del Repositorio

- **operaciones.py**: Archivo de módulo que contiene las funciones para realizar operaciones matemáticas.
- **programa.py**: Archivo principal donde se utiliza el módulo operaciones.py.
- **test_operaciones.py**: Archivo de pruebas unitarias para verificar las funciones del módulo **operaciones.py**.
- **.gitignore**: Archivo que indica a Git qué archivos y directorios debe ignorar al rastrear los cambios en el repositorio.
- **requirements.txt**: Archivo que especifica las dependencias del proyecto.

40.3 Ejercicio

Crear un módulo llamado operaciones.py que contenga las siguientes funciones:

1. **suma(a, b):** Retorna la **suma** de **a** y **b**.
2. **resta(a, b):** Retorna la **resta** de **a** y **b**.
3. **multiplicacion(a, b):** Retorna la **multiplicación** de **a** y **b**.
4. **division(a, b):** Retorna la **división** de **a** y **b**. Si **b** es **cero**, **retorna** un **mensaje de error**.

40.4 Cómo Ejecutar el Programa

- Asegúrate de tener Python instalado en tu sistema.
- Abre una terminal y navega hasta la ubicación de este repositorio.
- Instala las dependencias ejecutando pip install -r requirements.txt.
- Ejecuta python programa.py para ver los resultados de las operaciones.

40.5 Cómo Ejecutar las Pruebas

- Asegúrate de tener Python instalado en tu sistema.
- Abre una terminal y navega hasta la ubicación de este repositorio.
- Ejecuta **python -m unittest** para correr las pruebas.

41 Uso de Pypi

Pypi es un repositorio de software para Python. Es un lugar donde los desarrolladores pueden publicar y compartir sus paquetes de Python.

42 Utilizar algún paquete de Pypi

Para utilizar un paquete de Pypi, primero debes instalarlo usando **pip**. Por ejemplo, si quieres instalar el paquete **requests**, puedes hacerlo de la siguiente manera:

```
pip install requests
```

Una vez que hayas instalado el paquete, puedes importarlo en tu código de Python y utilizarlo. Por ejemplo:

```
import requests

response = requests.get('https://www.google.com')
print(response.status_code)
```

Otro ejemplo es por ejemplo el paquete emoji, que te permite utilizar emojis en tus programas de Python.

```
pip install emoji
```

```
import emoji

print(emoji.emojize('Python es :thumbs_up:'))
```

¡Y eso es todo! Ahora puedes utilizar cualquier paquete de Python disponible en Pypi en tus proyectos.
Algunos paquetes pueden ser muy importantes para tu proyecto, así que asegúrate de revisar Pypi para encontrar los paquetes que necesitas.

43 Conclusión

Pypi es un recurso valioso para los desarrolladores de Python. Te permite compartir tus paquetes con otros desarrolladores y utilizar los paquetes de otros desarrolladores en tus proyectos.

44 Públicar un paquete en Pypi

En este tutorial vamos a publicar un paquete llamado **status-checker-api** en Pypi. Si has creado un paquete de Python y quieres compartirlo con otros desarrolladores, puedes publicarlo en Pypi.
Para ello vamos a crear un repositorio en **GitHub** y subir nuestro paquete, esta práctica es importante para poder compartir nuestro paquete con otros desarrolladores.
En el caso de este tutorial, el repositorio se encuentra en status_checker_api
Lo más importante es tener el o los scripts que contienen el código que queremos convertir a paquete.
Para ello vamos a empezar creando un directorio con el nombre de nuestro paquete, por ejemplo **status_checker_api**.
A continuación se visualiza la estructura de nuestro paquete.

```
dist
    status_checker_api-0.5-py3-none-any.whl
    status_checker_api-0.5.tar.gz
img
    paste-5.png
LICENSE
README.md
setup.py
src
    status_checker_api
        __init__.py
        __main__.py
        __pycache__
            __init__.cpython-312.pyc
            __main__.cpython-312.pyc
    status_checker_api.egg-info
        dependency_links.txt
        entry_points.txt
        PKG-INFO
        requires.txt
        SOURCES.txt
        top_level.txt
    tests
        __init__.py
        __pycache__
            __init__.cpython-312.pyc
            test_status_checker_api.cpython-312-pytest-8.3.2.pyc
        test_status_checker_api.py
```

Dentro de este **status_checker_api** vamos a crear un directorio llamado **src** y dentro de este directorio vamos a crear un archivo llamado **__init__.py**, en este ejemplo tambien crearemos el archivo **__main__.py**.
Para poder publicar nuestro paquete en Pypi, necesitamos crear un archivo llamado **setup.py** en el directorio raíz de nuestro paquete. Este archivo contiene la información necesaria para empaquetar nuestro paquete y publicarlo en Pypi.

```python
from setuptools import setup, find_packages

with open('README.md', 'r', encoding="utf-8") as fh:
    long_description = fh.read()

setup(
    name='status_checker_api',
    version='0.5',
    packages=find_packages(where='src'),
    package_dir={'': 'src'},
    install_requires=[
        'requests',
    ],
    entry_points={
        'console_scripts': [
            'status-checker-api=status_checker_api.__main__:main',
        ],
    },
    author='Diego Saavedra',
    author_email='dsaavedra88@gmail.com',
    description='Un paquete para verificar el estado de una API.',
    long_description=long_description,
    long_description_content_type='text/markdown',
    url='https://github.com/statick88/status_checker_api',
    classifiers=[
        'Programming Language :: Python :: 3',
        'License :: OSI Approved :: MIT License',
        'Operating System :: OS Independent',
    ],
    options={
        'egg_info': {
            'egg_base': 'src'
        }
    },
    python_requires='>=3.12',
)
```

45 Creación del archivo README.md

```
# status_checker_api

Un paquete para verificar el estado de una API.

## Instalación

pip install status_checker_api

## Uso

api-status-checker

Ingrese la URL de la API: https://www.google.com

El status de la API es: 200

## Licencia

MIT License

## Autor

Diego Saavedra
```

El código del paquete se encuentra en el directorio **src**. Para poder ejecutar el paquete, necesitamos un archivo llamado **__init__.py** en el directorio **status_checker_api**.

```python
import requests
from urllib.parse import urlparse

def check_status(url):
    # Asegúrate de que la URL tenga un esquema (http o https)
    parsed_url = urlparse(url)
    if not parsed_url.scheme:
        url = 'https://' + url

    try:
        response = requests.get(url)
        return f"La URL está activa con código de estado: {response.status_code}"  # De
    except requests.exceptions.RequestException as e:
        return f"Error: {e}"  # Devuelve el mensaje de error con una x
```

Analizando el código anterior, podemos ver que el paquete **status_checker_api** contiene una función llamada **check_status** que verifica el estado de una API. La función toma una URL como argumento y devuelve un mensaje con el estado de la API.

46 Creación del archivo __main__.py

```python
from status_checker_api import check_status

def main():
    url = input('Ingrese la URL de la API: ')
    status = check_status(url)
    print(f'El status de la API es: {status}')

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

El archivo **__main__.py** contiene el código principal del paquete. Este archivo importa la función **check_status** del paquete **status_checker_api** y la utiliza para verificar el estado de una API.

47 Creación del archivo LICENSE

```
MIT License

Copyright (c) 2024 Diego Saavedra

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.
```

El archivo **LICENSE** contiene la licencia del paquete. En este caso, utilizamos la licencia MIT.

48 Creación del archivo .gitignore

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
dist/
build/
*.egg-info/
*.egg

# Virtual environments
venv/
env/
ENV/

# IDEs / Editors
.idea/
.vscode/
*.sublime-project
*.sublime-workspace

# Miscellaneous
*.swp
.DS_Store
```

El archivo **.gitignore** contiene los archivos y directorios que no queremos incluir en nuestro repositorio de Git. En este caso, ignoramos los archivos y directorios generados por Python y los entornos virtuales.

49 Creación de la cuenta en Pypi

Para poder publicar nuestro paquete en Pypi, necesitamos crear una cuenta en Pypi.

> Tip
>
> Una vez creada la cuenta en Pypi, necesitamos verificarla a través de un correo electrónico que nos enviarán. Adicional a ello es necesario configurar un factor de doble autenticación. Esto es indispensable para poder crear un token de acceso. El mismo que nos permitirá subir nuestro paquete a Pypi.

Una vez que hayamos creado la cuenta, necesitamos crear un archivo llamado **.pypirc** en nuestro directorio de usuario con la siguiente información:

```
[pypi]
  username = statick
  password = pypi-token
```

En el archivo **.pypirc**, reemplazamos **username** con nuestro nombre de usuario de Pypi y **password** con nuestro token de acceso de Pypi.

> Tip
>
> En sistemas operativos basados en Unix, el archivo **.pypirc** se encuentra en el directorio de usuario **.pypirc**.
> En sistemas operativos basados en Windows, el archivo **.pypirc** se encuentra en el directorio de usuario **C:** / Users / username / –> en este directorio se almacena el archivo **.pypirc**.

50 Publicar el paquete en Pypi

Para publicar nuestro paquete en Pypi, necesitamos instalar el paquete **twine**. Twine es una herramienta que nos permite subir paquetes de Python a Pypi.

```
pip install twine
```

Es recomandable que tengamos la última versión de **twine**.

```
pip install --upgrade twine
```

Una vez que hayamos instalado y actualizado **twine**, podemos publicar nuestro paquete en Pypi de la siguiente manera:

```
python -m pip install --upgrade build
```

El comando anterior instala el paquete **build** que necesitamos para construir nuestro paquete.

```
python -m build
```

El comando anterior crea un archivo **dist** en el directorio raíz de nuestro paquete. Este archivo contiene el paquete que vamos a publicar en Pypi. Es decir los archivos **.tar.gz** y **.whl**.
Estos archivos son los que vamos a subir a Pypi.

```
python -m twine upload --repository pypi dist/* --verbose
```

El comando anterior sube nuestro paquete a Pypi. El archivo **.pypirc** contiene la información de autenticación que necesitamos para subir nuestro paquete.

¡Y eso es todo! Ahora puedes compartir tu paquete de Python con otros desarrolladores en Pypi. En el caso de este paquete la url es status_checker_api.

51 Instalar el paquete

Para instalar el paquete que acabamos de publicar en Pypi, necesitamos usar **pip**.

```
Terminal
statick at fedora in ~
λ pip install api-status-checker
Defaulting to user installation because normal site-packages is not writeable
Collecting api-status-checker
  Using cached api_status_checker-0.1-py3-none-any.whl.metadata (659 bytes)
Requirement already satisfied: requests in /usr/lib/python3.12/site-packages (from api-status-checker) (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/lib/python3.12/site-packages (from requests->api-status-checker) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/lib/python3.12/site-packages (from requests->api-status-checker) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/lib/python3.12/site-packages (from requests->api-status-checker) (1.26.19)
Using cached api_status_checker-0.1-py3-none-any.whl (1.8 kB)
Installing collected packages: api-status-checker
  WARNING: The script api-status-checker is installed in '/home/statick/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed api-status-checker-0.1
statick at fedora in ~
λ
```

```
pip install status_checker_api
```

Una vez que hayamos instalado el paquete, podemos utilizarlo en nuestro código de Python.

52 Uso del paquete

```
api-status-checker
```

```
statick at fedora in ~
status-checker-api
Ingrese la URL de la API: www.google.com
El status de la API es: ✅ La URL está activa con código de estado: 200
statick at fedora in ~
```

Es necesario ingresar la URL de la API que queremos verificar.
Ingrese la URL de la API: https://www.google.com
El status de la API es: 200
¡Y eso es todo! Ahora puedes actualizar tu paquete de Python en Pypi.

> Tip
>
> No olvides cambiar la versión de tu paquete en el archivo **setup.py** antes de subirlo a Pypi si realizas alguna actualización.

Si decidimos actualizar el paquete en Pypi, necesitamos seguir los mismos pasos que hemos visto en este tutorial.
Sin embargo solo necesitaremos 2 comandos:

```
python -m build
```

```
python -m twine upload --repository pypi dist/* --verbose
```

> Tip
>
> En el directorio dist se generan los archivos **.tar.gz** y **.whl** que son los que vamos a subir a Pypi. Es necesario eliminar los archivos anteriores antes de subir los nuevos en este directorio, mi recomendación es eliminar el directorio **dist** y volver a ejecutar el comando **python -m build**.

53 Conclusión

En este tutorial aprendimos cómo publicar un paquete de Python en Pypi. Pudimos ver cómo crear un paquete de Python, subirlo a Pypi y compartirlo con otros desarrolladores.

54 Desarrollo de un Sistema de Gestión de Inventarios en Python

Este laboratorio tiene como objetivo guiarte en el desarrollo de un sistema de gestión de inventarios utilizando el lenguaje de programación Python. A través de esta actividad, aprenderás a implementar funcionalidades clave en un proyecto práctico que puedes utilizar como base para futuros desarrollos.

54.1 Objetivos

- **Diseño de la estructura de datos:** Aprenderás a diseñar y crear una estructura de datos para almacenar la información de los productos, incluyendo atributos como nombre, descripción, precio, y cantidad disponible.
- **Agregar productos:** Implementarás la funcionalidad para agregar nuevos productos al inventario.
- **Búsqueda y filtrado:** Aprenderás a implementar funciones de búsqueda y filtrado para encontrar productos específicos basados en diferentes criterios.
- **Actualización de inventario:** Desarrollarás funciones para manejar la compra y venta de productos, permitiendo ajustar la cantidad disponible en el inventario.
- **Generación de informes:** Crearás funciones para generar informes sobre el estado del inventario, tales como productos disponibles, productos más vendidos, y productos con bajo stock.

54.2 Entregables

- **Código fuente del proyecto:** Estructurado y organizado de manera coherente.
- **Documentación del proyecto:** Incluyendo instrucciones de instalación, uso, y una breve explicación del diseño de la solución (Esto se sugiere generar en el **Readme.md** del proyecto).
- **Pruebas unitarias:** Implementación de pruebas para verificar que las funcionalidades clave del sistema funcionan correctamente.

54.3 Instrucciones

54.3.1 1. Crear la Estructura de Datos

Diseña una clase Producto que contenga los atributos básicos como nombre, descripción, precio, y cantidad disponible. Implementa un método **str** para imprimir la

información del producto de manera legible.
Solución

```
class Producto:
    def __init__(self, nombre, descripcion, precio, cantidad):
        self.nombre = nombre
        self.descripcion = descripcion
        self.precio = precio
        self.cantidad = cantidad

    def __str__(self):
        return f"Producto: {self.nombre}, Precio: {self.precio}, Cantidad: {self.cantida
```

54.3.2 2. Agregar Productos

Implementa una función que permita agregar nuevos productos a una lista que actúe como el inventario.
Solución

```
inventario = []

def agregar_producto(producto):
    inventario.append(producto)
    print(f"{producto.nombre} ha sido añadido al inventario.")
```

54.3.3 3. Búsqueda y Filtrado

Crea funciones para buscar productos por nombre, categoría o rango de precios.
Solución

```
def buscar_producto_por_nombre(nombre):
    return [p for p in inventario if nombre.lower() in p.nombre.lower()]

def buscar_producto_por_precio(min_precio, max_precio):
    return [p for p in inventario if min_precio <= p.precio <= max_precio]
```

54.3.4 4. Actualización de Inventario

Implementa funciones para aumentar o disminuir la cantidad de productos en el inventario, simulando la compra o venta de productos.
Solución

```
def actualizar_cantidad(nombre, cantidad):
    for producto in inventario:
        if producto.nombre == nombre:
            producto.cantidad += cantidad
            print(f"Cantidad actualizada: {producto.nombre} ahora tiene {producto.canti
            return
    print("Producto no encontrado.")
```

54.3.5 5. Generación de Informes

Crea funciones para generar informes del estado del inventario.
Solución

```
def generar_informe_productos_disponibles():
    return [p for p in inventario if p.cantidad > 0]

def generar_informe_productos_bajo_stock(limite):
    return [p for p in inventario if p.cantidad <= limite]
```

54.3.6 6. Pruebas Unitarias

Escribe pruebas para cada una de las funciones clave utilizando unittest.
Solución

```
import unittest

class TestInventario(unittest.TestCase):
    def test_agregar_producto(self):
        producto = Producto("Test", "Descripcion", 10.0, 5)
        agregar_producto(producto)
        self.assertIn(producto, inventario)
```

54.3.7 7. Documentación y GitHub Classroom

Documenta el código fuente, incluyendo instrucciones sobre cómo ejecutar el programa y las pruebas.
Configura tu repositorio de GitHub Classroom y sube todo el código y documentación.

54.3.8 Evaluación

1. **Funcionalidad (40%):** El sistema implementa correctamente las funcionalidades solicitadas.
2. **Calidad del Código (30%):** El código es claro, bien estructurado, y sigue buenas prácticas de programación.
3. **Pruebas (20%):** Las pruebas cubren las funcionalidades clave y se ejecutan correctamente.

4. **Documentación (10%):** La documentación es clara y proporciona una guía adecuada para el usuario.

¡Buena suerte!

54.4 Asignación

https://classroom.github.com/a/OVCpAmrV
Aprenderás a desarrollar un proyecto de utilizando el lenguaje de programación Python.
Un sistema de gestión de inventarios es una herramienta que permite realizar un seguimiento y control de los productos o artículos almacenados en un negocio o empresa.
Aprenderás a utilizar diferentes conceptos y técnicas de programación para implementar las funcionalidades clave de este sistema.
Algunas de las funcionalidades que implementaremos incluyen:
Aprenderás a crear una estructura de datos para almacenar la información de los productos, como su nombre, descripción, precio, cantidad disponible, etc. También aprenderás a agregar nuevos productos al sistema.
Te enseñaré cómo implementar funciones de búsqueda y filtrado para encontrar productos específicos en base a diferentes criterios, como el nombre, la categoría o el precio.
Aprenderás a manejar las actualizaciones de inventario, como la compra o venta de productos. Implementaremos funciones que permitan aumentar o disminuir la cantidad disponible de un producto y mantener un registro de estas transacciones.
Te mostraré cómo generar informes sobre el estado del inventario, como la lista de productos disponibles, los productos más vendidos, los productos con bajo stock, etc. Utilizaremos técnicas de manipulación de datos y generación de informes para presentar esta información de manera clara y concisa.

Part V

Unidad 5: Django

55 Introducción a Django

Figure 55.1: Django Framework

Django es un framework web de alto nivel que fomenta el desarrollo rápido y limpio. Es un framework web de alto nivel que fomenta el desarrollo rápido y limpio. Diseñado por desarrolladores experimentados, Django se encarga de gran parte de la molestia del desarrollo web, por lo que puedes concentrarte en escribir tu aplicación sin necesidad de reinventar la rueda. Es gratuito y de código abierto, tiene una comunidad activa y amigable, y es utilizado por algunas de las mayores empresas del mundo.

55.1 Conceptos Importantes

> **Tip**
>
> Antes de iniciar con Django es necesario conocer el concepto de **Entornos Virtuales**.

55.1.1 Entornos Virtuales

Figure 55.2: Virtual Enviroment

Un entorno virtual es un **entorno de desarrollo aislado** que permite **instalar paquetes de Python sin afectar al sistema global**. Los entornos virtuales son útiles para gestionar las dependencias de un proyecto y para evitar conflictos entre diferentes versiones de los paquetes.

55.1.1.1 Crear un entorno virtual

Para crear un entorno virtual, se puede utilizar la herramienta **venv** de Python.

```
python -m venv env
```

Este comando creará un directorio llamado **env** en el directorio actual con el entorno virtual.

> **Tip**
>
> Tambien se puede utilizar virtualenv para crear entornos virtuales.

55.1.2 Modelo Template View (MTV)

Figure 55.3: Model View Controller

Figure 55.4: Model View Template

Django sigue el patrón de diseño Modelo Vista Template (MVT). Este patrón de diseño separa la lógica de la aplicación en tres componentes principales: Modelo, Vista y Template.

> **Tip**
>
> El archivo URLs.py es el encargado de mapear las URLs de la aplicación a las vistas correspondientes.

- **Modelo**: Es la representación de los datos de la aplicación y las reglas para manipular esos datos. Django utiliza un ORM (Object-Relational Mapping) para interactuar con la base de datos.
- **Vista**: Es la capa de presentación de la aplicación. Se encarga de mostrar los datos al usuario y de interpretar las acciones del usuario.
- **Template**: Es la capa de presentación de la aplicación. Define cómo se muestra la información al usuario. Django utiliza el motor de plantillas Jinja2 para renderizar los templates.

55.1.3 Formularios

Figure 55.5: Django Forms

Los formularios son una parte importante de cualquier aplicación web. Django proporciona una forma sencilla de crear y procesar formularios en las vistas.

55.1.4 Administrador de Django

Figure 55.6: Django Admin

El administrador de Django es una interfaz de administración que permite gestionar los datos de la aplicación de forma sencilla. Django genera automáticamente una interfaz de administración basada en los modelos de la aplicación.

55.1.5 Middleware

El middleware es una capa de procesamiento que se ejecuta antes y después de cada petición HTTP. Django proporciona un conjunto de middlewares que se pueden utilizar para añadir funcionalidades a la aplicación.

55.1.6 Autenticación y Autorización

Django proporciona un sistema de autenticación y autorización que permite gestionar los usuarios y los permisos de la aplicación de forma sencilla.

55.1.7 Internacionalización

Django proporciona soporte para la internacionalización de la aplicación. Permite traducir la aplicación a diferentes idiomas y gestionar las traducciones de forma sencilla.

55.1.8 Seguridad

Django proporciona un conjunto de medidas de seguridad para proteger la aplicación contra ataques comunes, como la inyección de SQL, la falsificación de solicitudes entre sitios (CSRF) y la inyección de código.

55.1.9 Testing

Django proporciona un conjunto de herramientas para realizar pruebas unitarias y de integración en la aplicación. Permite probar la lógica de la aplicación y asegurarse de que funciona correctamente.

55.1.10 Despliegue

Django proporciona un conjunto de herramientas para desplegar la aplicación en un servidor de producción. Permite configurar el entorno de producción y gestionar las actualizaciones de la aplicación de forma sencilla.

56 Configuración inicial de un proyecto.

56.1 1. Crear un entorno virtual

Figure 56.1: Creación de entorno Virtual

```
python3 -m venv env
```

El comando anterior creará un directorio llamado **env** en el directorio actual, que contendrá un entorno virtual de Python.

56.2 2. Activar el entorno virtual

Figure 56.2: Activación de entorno Virtual

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

El comando anterior activará el entorno virtual en sistemas Unix. En Windows, el comando es:

```
env\Scripts\activate
```

Este comando tambien se puede dividir en 2 partes:

```
cd env/Scripts/
activate
```

Para desactivar el entorno virtual, simplemente ejecute:

```
deactivate
```

56.3 3. Instalar Django

Figure 56.3: Instalación de Django

```
pip install django==4.2
```

El comando anterior instalará la última versión de Django en el entorno virtual.

56.4 4. Crear un proyecto de Django

Figure 56.4: Creación de un Proyecto en Django

```
django-admin startproject helloWorld .
```

El comando anterior creará un nuevo directorio llamado **helloWorld** en el directorio actual, que contendrá un proyecto de Django.

56.5 5. Crear una aplicación de Django

Figure 56.5: Creación de una App en Django

```
python manage.py startapp hello
```

El comando anterior creará un nuevo directorio llamado **hello** en el directorio actual, que contendrá una aplicación de Django.

> **Tip**
>
> Recerda que puedes abrir el editor de código Visual Studio Code con el comando **code .**

56.6 6. Crear una vista

Figure 56.6: Vistas en Django

```python
# hello/views.py

from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, World!")
```

56.7 7. Configurar las URL

Figure 56.7: URLs de la App en Django

```
# helloWorld/urls.py

from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("", include("hello.urls")),
    path("admin/", admin.site.urls),
]
```

56.8 8. Ejecutar el servidor de desarrollo

Figure 56.8: Servidor de Desarrollo en Django

```
python manage.py runserver
```

El comando anterior ejecutará el servidor de desarrollo de Django. Para acceder al servidor, abra un navegador web y vaya a la dirección **http://0.0.0.0:8000/**.

Figure 56.9: Visualizar el servidor corriendo desde el navegador

> **Tip**
>
> Para detener el servidor de desarrollo, presione **Ctrl + C** en la terminal.

56.9 9. Crear una migración

Figure 56.10: Preparación de las Migraciones en Django

```
python manage.py makemigrations
```

El comando anterior creará una migración para los cambios en los modelos de la base de datos.

56.10 10. Aplicar una migración

```
statick@lenovo [v] 3.12.3 ~/workspaces/course_of_django python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
statick@lenovo [v] 3.12.3 ~/workspaces/course_of_django
```

Figure 56.11: Preparación de las Migraciones en Django

```
python manage.py migrate
```

El comando anterior aplicará la migración a la base de datos.

56.11 12. Crear un superusuario

```
statick@lenovo [v] 3.12.3 ~/workspaces/course_of_django python manage.py createsuperuser
Username (leave blank to use 'statick'):
Email address:
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
statick@lenovo [v] 3.12.3 ~/workspaces/course_of_django
```

Figure 56.12: Creación de un Superusuario en Django

```
python manage.py createsuperuser
```

El comando anterior creará un superusuario para acceder al panel de administración de Django.

56.12 13. Acceder al panel de administración

Figure 56.13: Login Admin en Django

Para acceder al panel de administración de Django, abra un navegador web y vaya a la dirección **http://127.0.0.1:8000/admin/**. Inicie sesión con el superusuario creado en el paso anterior.

Figure 56.14: Admin en Django

57 Ejercicio

Crear un proyecto de Django llamado **helloWorld** con una aplicación llamada **hello** que muestre un mensaje de bienvenida en la página de inicio.

Ver solución

```
python3 -m venv env                                                    ①
source env/bin/activate                                                ②
pip install django==4.2                                                ③
django-admin startproject helloWorld .                                 ④
python manage.py startapp hello                                        ⑤
```

① Crear un entorno virtual.
② Activar el entorno virtual.
③ Instalar Django.
④ Crear un proyecto de Django.
⑤ Crear una aplicación de Django.

```
# hello/views.py

from django.http import HttpResponse                                   ①

def index(request):                                                    ②
    return HttpResponse("Hello, World!")                               ③
```

① Importar la clase **HttpResponse** de **django.http**.
② Crear una vista llamada **index**.
③ Devolver un mensaje de bienvenida.

```
# helloWorld/urls.py

from django.urls import path                                           ①
from hello import views                                                ②

urlpatterns = [                                                        ③
    path("", views.index),                                             ④
]
```

① Importar la función **path** de **django.urls**.
② Importar el módulo **views** de la aplicación **hello**.
③ Crear una lista de rutas.

④ Asociar la ruta raíz con la vista **index**.

```
# helloWorld/urls.py

from django.contrib import admin                                    ①
from django.urls import include, path                               ②

urlpatterns = [                                                     ③
    path("", include("hello.urls")),                                ④
    path("admin/", admin.site.urls),                                ⑤
]
```

① Importar el módulo **admin** de **django.contrib**.
② Importar la función **include** y la clase **path** de **django.urls**.
③ Crear una lista de rutas.
④ Incluir las rutas de la aplicación **hello**.
⑤ Incluir las rutas del panel de administración.

```
python manage.py runserver                                          ①
```

① Ejecutar el servidor de desarrollo.

58 Asignación

Desarrolla una aplicación web que muestre una lista de productos en la página de inicio. Cada producto debe tener un nombre, una descripción y un precio. Además, la aplicación debe tener un panel de administración donde se puedan agregar, editar y eliminar productos.

59 Estructura de archivos y carpetas

Django tiene una estructura de archivos y carpetas que se debe seguir para que el proyecto funcione correctamente. A continuación se muestra la estructura de archivos y carpetas de un proyecto Django:

> **Tip**
>
> Recuerda crear el entorno virtual y activarlo antes de ejecutar el comando.
>
> ```
> python -m venv venv
> source venv/bin/activate
> ```
>
> Creamos un directorio con el siguiente comando:
>
> ```
> mkdir myproject
> cd myproject
> ```
>
> Instalamos Django con el siguiente comando:
>
> ```
> pip install django==4.2.0
> ```
>
> Creamos el proyecto con el siguiente comando:
>
> ```
> django-admin startproject myproject .
> ```

```
manage.py # <1>
myproject # <2>
    asgi.py # <3>
    __init__.py # <4>
    settings.py # <5>
    urls.py # <6>
    wsgi.py # <7>
```

1.- Archivo de gestión del proyecto.

2.- Carpeta del proyecto.

3.- Archivo de configuración de ASGI.

4.- Archivo de inicialización del proyecto.

5.- Archivo de configuración del proyecto.

6.- Archivo de configuración de las rutas del proyecto.

7.- Archivo de configuración de WSGI.

60 Creación de una aplicación Django

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

```
python manage.py startapp myapp ①
```

① Nombre de la aplicación.

61 Configuración de la base de datos

Para configurar la base de datos se debe modificar el archivo `settings.py` del proyecto. A continuación se muestra un ejemplo de configuración de la base de datos:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',                    ①
        'NAME': BASE_DIR / 'db.sqlite3',                           ②
    }
}
```

① Motor de base de datos.
② Ruta del archivo de la base de datos.

Ejemplo

En este ejemplo crearemos una aplicación que muestre un mensaje en la página principal. Para ello, se deben seguir los siguientes pasos:

1. Crear una vista.
2. Crear una plantilla.
3. Configurar las rutas.

62 Crear una vista

Para crear una vista se debe modificar el archivo **views.py** de la aplicación. A continuación se muestra un ejemplo de vista:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world!")
```

63 Crear una plantilla

Para crear una plantilla se debe crear una carpeta llamada **templates** en la carpeta de la aplicación. A continuación se muestra un ejemplo de plantilla:

```
<!DOCTYPE html>
<html>
<head>
    <title>MyApp</title>
</head>
<body>
    <h1>Hello, world!</h1>
</body>
</html>
```

Para que Django pueda encontrar la plantilla, se debe configurar la ruta de la plantilla en el archivo **settings.py** del proyecto. A continuación se muestra un ejemplo de configuración de la ruta de la plantilla:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],                                  ①
        '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',
            ],
        },
    },
]
```

① Ruta de la plantilla.

64 Configurar las rutas

Para configurar las rutas se debe modificar el archivo **urls.py** de la aplicación. A continuación se muestra un ejemplo de configuración de las rutas:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

65 Correr el servidor de desarrollo

Para correr el servidor de desarrollo se debe ejecutar el siguiente comando:

```
python manage.py runserver
```

66 Acceder a la aplicación

Para acceder a la aplicación se debe abrir un navegador web y escribir la siguiente URL:

http://127.0.0.1:8000/

Posiblemente sea necesario preparar las migraciones y aplicarlas a la base de datos:

```
python manage.py makemigrations ①
python manage.py migrate ②
```

① Prepara las migraciones.
② Aplica las migraciones a la base de datos.

67 Acceder a la aplicación

Para acceder a la aplicación se debe abrir un navegador web y escribir la siguiente URL:

http://127.0.0.0.1:8000/

Muy bien hecho! Has creado tu primera aplicación Django. Ahora puedes seguir explorando la documentación oficial de Django para aprender más sobre el framework.

68 Asignación

Seguir cada uno de los pasos de esta sección para crear una aplicación Django que muestre un mensaje en la página principal. La aplicación debe tener los siguientes archivos y carpetas:

```
manage.py # <1>
myproject # <2>
    asgi.py # <3>
    __init__.py # <4>
    settings.py # <5>
    urls.py # <6>
    wsgi.py # <7>
myapp # <8>
    __init__.py # <9>
    admin.py # <10>
    apps.py # <11>
    migrations # <12>
        __init__.py # <13>
    models.py # <14>
    tests.py # <15>
    views.py # <16>
    templates # <17>
        index.html # <18>
```

1.- Archivo de gestión del proyecto.

2.- Carpeta del proyecto.

3.- Archivo de configuración de ASGI.

4.- Archivo de inicialización del proyecto.

5.- Archivo de configuración del proyecto.

6.- Archivo de configuración de las rutas del proyecto.

7.- Archivo de configuración de WSGI.

8.- Carpeta de la aplicación.

9.- Archivo de inicialización de la aplicación.

10.- Archivo de configuración del administrador de Django.

11.- Archivo de configuración de la aplicación.

12.- Carpeta de migraciones de la aplicación.

13.- Archivo de inicialización de las migraciones.

14.- Archivo de configuración de los modelos de la aplicación.

15.- Archivo de pruebas de la aplicación.

16.- Archivo de configuración de las vistas de la aplicación.

17.- Carpeta de plantillas de la aplicación.

18.- Archivo de la plantilla de la aplicación.

Recuerda que para que Django pueda encontrar la plantilla, se debe configurar la ruta de la plantilla en el archivo **settings.py** del proyecto. A continuación se muestra un ejemplo de configuración de la ruta de la plantilla:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],                                ①
        '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',
            ],
        },
    },
]
```

① Ruta de la plantilla.

69 Referencias

- Django

70 Modelos

Para entender este tema crearemos un sistema de gestión de inventario de productos. Para ello, crearemos una clase **Producto** que representará un producto en el inventario. Cada producto tendrá un nombre, un precio y una cantidad en inventario.

Empezaremos creando un entorno virtual e instalando Django.

```
python -m venv env
```

Ahora activaremos el entorno virtual.

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

crearemos un directorio llamado **inventario** y nos moveremos a ese directorio.

```
mkdir inventario
```

```
cd inventario
```

Una vez activado el entorno y creando el directorio **inventario**, instalaremos Django.

```
pip install django==4.2.0
```

> Tip
>
> **Tip:** Para crear un proyecto de Django, utilizamos el comando **django-admin startproject**. En este caso utilizamos la versión LTS, recuerda que puedes consultar la versión más reciente en la página oficial de Django. Sin embargo en este caso utilizaremos la versión 4.2.0.

Ahora crearemos un ***proyecto*** de Django llamado **inventario**

```
django-admin startproject inventario .
```

Luego crearemos una aplicación llamada **productos**

```
python manage.py startapp productos
```

Info: En Django, un proyecto es un conjunto de aplicaciones web y un proyecto puede contener múltiples aplicaciones.

Para que la app productos funcione, debemos registrarla en el archivo **settings.py** del proyecto **inventario**.

```
INSTALLED_APPS = [
    ...
    "productos",
]
```

Ahora crearemos la clase **Producto** en el archivo **models.py** de la aplicación **productos**.

```
from django.db import models

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

    def __str__(self):
        return self.nombre
```

Tip

Tip: La función **__str__** es una función especial que se llama cuando se convierte un objeto a una cadena de texto.
Los modelos en Django son clases que representan tablas en la base de datos. Cada atributo de la clase representa una columna en la tabla y cada instancia de la clase representa una fila en la tabla.

71 Registramos la aplicacion en admin.py

Para que Django reconozca la clase **Producto**, debemos registrarla en el archivo **admin.py** de la aplicación **productos**.

```
from django.contrib import admin
from .models import Producto                                    ①

admin.site.register(Producto)                                   ②
```

① Importamos la clase **Producto**.
② Registramos la clase **Producto** en el panel de administración de Django.

72 Vistas en Django

Ahora crearemos las vistas de nuestro sistema en Django. Para ello, crearemos una función para cada vista que renderizará una plantilla HTML. Por lo tanto nuestras vistas están basadas en funciones. Sin embargo tambien existen vistas basadas en clases.

> Tip
>
> En Django, una vista es una función que recibe una petición HTTP y devuelve una respuesta HTTP.

> Tip
>
> **¿Qué es un CRUD?**
> Un CRUD es un acrónimo que significa **Crear**, **Leer**, **Actualizar** y **Eliminar**. Es un conjunto de operaciones básicas que se pueden realizar en una base de datos o en un sistema de gestión de datos.

72.1 Listar productos

Para listar los productos en inventario, crearemos una función **listar_productos** que renderizará la plantilla **listar.html** con la lista de productos.

```
from pyexpat.errors import messages
from django.shortcuts import render, redirect, get_object_or_404
from .models import Producto
from django.urls import reverse

productos = []

def listar_productos(request):
    productos = Producto.objects.all()
    return render(request, 'listar.html', {'productos': productos})
```

72.2 Agregar producto

Para agregar un producto al inventario, crearemos una función **agregar_producto** que recibe los datos del producto a agregar y lo agrega a la lista de productos.

```
def agregar_producto(request):
    if request.method == "POST":
        nombre = request.POST.get("nombre")
        precio = request.POST.get("precio")
        cantidad = request.POST.get("cantidad")
        Producto.objects.create(nombre=nombre, precio=precio, cantidad=cantidad)
        return redirect('productos:listar_productos')
    return render(request, "agregar.html")
```

72.3 Actualizar producto

Para actualizar un producto en el inventario, crearemos una función **actualizar_producto** que recibe los datos del producto a actualizar y actualiza el precio y la cantidad del producto.

```
def actualizar_producto(request, id):
    producto = get_object_or_404(Producto, pk=id)
    if request.method == 'POST':
        nombre = request.POST.get('nombre')
        precio = request.POST.get('precio')
        cantidad = request.POST.get('cantidad')

        # Actualiza los campos del producto
        producto.nombre = nombre
        producto.precio = precio
        producto.cantidad = cantidad
        producto.save()

        return redirect('productos:listar_productos')
    else:
        return render(request, 'actualizar.html', {'producto': producto})
```

72.4 Eliminar producto

Para eliminar un producto del inventario, crearemos una función **eliminar_producto** que recibe el nombre del producto a eliminar y lo elimina de la lista de productos.

```
def eliminar_producto(request):
    if request.method == "POST":
        nombre = request.POST.get("nombre")
        try:
            producto = Producto.objects.get(nombre=nombre)
            producto.delete()
        except Producto.DoesNotExist:
```

```
            pass

        return redirect('productos:listar_productos')
    return render(request, "eliminar.html")
```

72.5 Buscar producto

Para buscar un producto en el inventario, crearemos una función **buscar_producto** que recibe el nombre del producto a buscar y renderiza la plantilla **buscar.html** con el producto encontrado.

```
def buscar_producto(request):
    if request.method == "POST":
        nombre = request.POST.get("nombre")
        try:
            producto = Producto.objects.get(nombre=nombre)
            return render(request, "buscar.html", {"producto": producto})
        except Producto.DoesNotExist:
            return render(request, "buscar.html", {"producto": None})
    return render(request, "buscar.html")
```

73 Templates

Django utiliza un sistema de herencia de plantillas llamado Jinja2. Adicional a ello uilizaremos un framework de CSS llamado Bootstrap.

> Tip
>
> **Jinja2** es un motor de plantillas para Python que se utiliza en Django para renderizar plantillas HTML.

> Tip
>
> **Bootstrap** es un framework de CSS que se utiliza para crear sitios web y aplicaciones web responsivos y móviles.

Para crear las plantillas de nuestro sistema, crearemos una carpeta llamada **templates** en el directorio de la aplicación **productos**.

Teniendo la siguiente estructura:

```
productos/
    migrations/
    templates/
        productos/
            base.html
            listar.html
            agregar.html
            actualizar.html
            eliminar.html
            buscar.html
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

73.1 Base

Crearemos un archivo **base.html** que contendrá la estructura base de todas las páginas de nuestro sistema.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>
        {% block title %}
        Inventario
        {% endblock %}
    </title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" 
</head>
<body>
    <div class="container">
        {% block content %}
        {% endblock %}
    </div>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js
</body>
</html>
```

73.2 Listar

Crearemos un archivo **listar.html** que contendrá la lista de productos en inventario.

```
{% extends "base.html" %}

{% block title %} Listar Productos {% endblock %}

{% block content%}
<h1>Listar Productos</h1>
<ul class="list-group">
    {% for producto in productos %}
    <li class="list-group-item">
        {{ producto.nombre }} - {{ producto.precio }} - {{ producto.cantidad }}
    </li>
    {% endfor %}
</ul>
{% endblock%}
```

73.3 Agregar

Crearemos un archivo **agregar.html** que contendrá un formulario para agregar un producto al inventario.

```
{% extends "base.html" %}

{% block title %}Agregar producto{% endblock %}

{% block content %}
<h1>Agregar producto</h1>

<form action="{% url 'productos:agregar_producto' %}" method="post">
    {% csrf_token %}
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" class="form-control" id="nombre" name="nombre">
    </div>
    <div class="mb-3">
        <label for="precio" class="form-label">Precio</label>
        <input type="number" class="form-control" id="precio" name="precio">
    </div>
    <div class="mb-3">
        <label for="cantidad" class="form-label">Cantidad</label>
        <input type="number" class="form-control" id="cantidad" name="cantidad">
    </div>
    <button type="submit" class="btn btn-primary">Agregar</button>
</form>
{% endblock %}
```

73.4 Actualizar

Crearemos un archivo **actualizar.html** que contendrá un formulario para actualizar un producto en el inventario.

```
{% extends "base.html" %}

{% block title %}Actualizar producto{% endblock %}

{% block content %}
<h1>Actualizar producto</h1>
<form action="{% url 'productos:actualizar_producto' producto.id %}" method="post">
    {% csrf_token %}
    <input type="hidden" name="nombre" value="{{ producto.nombre }}">
    <input type="hidden" name="id" value="{{ producto.id }}"> {# Agregamos un campo ocult
    <div class="mb-3">
        <label for="precio" class="form-label">Precio</label>
        <input type="number" class="form-control" id="precio" name="precio" value="{{ pro
    </div>
    <div class="mb-3">
        <label for="cantidad" class="form-label">Cantidad</label>
```

```
        <input type="number" class="form-control" id="cantidad" name="cantidad" value="{{
    </div>
    <button type="submit" class="btn btn-primary">Actualizar</button>
</form>
{% endblock %}
```

73.5 Eliminar

Crearemos un archivo **eliminar.html** que contendrá un formulario para eliminar un producto del inventario.

```
{% extends "base.html" %}

{% block title %}Eliminar producto{% endblock %}

{% block content %}
<h1>Eliminar producto</h1>
<form action="{% url 'productos:eliminar_producto' %}" method="post">
    {% csrf_token %}  <!-- Agrega el token CSRF aquí -->
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" class="form-control" id="nombre" name="nombre">
    </div>
    <button type="submit" class="btn btn-primary">Eliminar</button>
</form>
{% endblock %}
```

73.6 Buscar

Crearemos un archivo **buscar.html** que contendrá un formulario para buscar un producto en el inventario.

```
{% extends "base.html" %}

{% block title %}Buscar producto{% endblock %}

{% block content %}
<h1>Buscar producto</h1>
<form action="{% url 'productos:buscar_producto' %}" method="post">
    {% csrf_token %}
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" class="form-control" id="nombre" name="nombre">
    </div>
    <button type="submit" class="btn btn-primary">Buscar</button>
```

```
</form>

{% if producto %}
<div class="mt-3">
    <h2>Información del producto:</h2>
    <p>Nombre: {{ producto.nombre }}</p>
    <p>Precio: {{ producto.precio }}</p>
    <p>Cantidad: {{ producto.cantidad }}</p>
</div>
{% endif %}
{% endblock %}
```

74 URLs

Para que nuestro sistema funcione, necesitamos definir las URLs que se utilizarán para acceder a las diferentes vistas.

74.1 URLs en la aplicación y el proyecto

En el archivo **urls.py** de la aplicación **productos** definiremos las URLs de las vistas de nuestro sistema.

```
from django.urls import path
from . import views

app_name = 'productos'

urlpatterns = [
    path('', views.listar_productos, name='listar_productos'),
    path('agregar/', views.agregar_producto, name='agregar_producto'),
    path('actualizar/<int:id>/', views.actualizar_producto, name='actualizar_producto'),
    path('eliminar/', views.eliminar_producto, name='eliminar_producto'),
    path('buscar/', views.buscar_producto, name='buscar_producto'),
]
```

En el archivo **urls.py** del proyecto **inventario** incluiremos las URLs de la aplicación **productos**.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('productos.urls')),
]
```

Tip

Tip: Para acceder al panel de administración de Django, debemos crear un superusuario con el siguiente comando.

```
python manage.py createsuperuser
```

Tip

Si realizamos modificaciones en el modelo de datos, debemos aplicar las migraciones con el siguiente comando.

```
python manage.py makemigrations                    ①
python manage.py migrate                           ②
```

① Crea las migraciones.
② Aplica las migraciones.

Tip

Tip: Para acceder a las vistas de nuestro sistema, debemos definir las URLs en el archivo **urls.py** de la aplicación y el proyecto.
En Django, las URLs se definen en el archivo **urls.py** de la aplicación y el proyecto. Las URLs se utilizan para acceder a las vistas de nuestro sistema.

Es necesario realizar una modificación en el archivo **settings.py** del proyecto **inventario** para que Django pueda encontrar las plantillas de nuestro sistema.

```
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],      ①
        ...
    },
]
```

① Agregamos la ruta de la carpeta **templates** al directorio de plantillas.

75 Ejecutar el servidor

Para ejecutar el servidor de desarrollo de Django, utilizaremos el siguiente comando.

```
python manage.py runserver
```

Con esto hemos creado un sistema de gestión de inventario de productos con Django. Ahora podemos listar, agregar, actualizar, eliminar y buscar productos en el inventario y hemos utilizado el sistema de plantillas de Django para crear las vistas de nuestro sistema. Adicional a ello tambien hemos utilizado el framework de CSS Bootstrap para darle estilo a nuestro sistema.

76 Conclusiones

En este tema hemos aprendido a crear un sistema de gestión de inventario de productos con Django. Hemos creado un modelo de datos para representar los productos en el inventario y hemos creado vistas para listar, agregar, actualizar, eliminar y buscar productos en el inventario. Adicional a ello tambien hemos creado plantillas HTML para renderizar las vistas de nuestro sistema.

77 Django Rest Framework

Django Rest Framework más que una **librearía** es un **framework** que nos permite crear **APIs REST** de forma rápida y sencilla.

77.1 ¿Qué es una API REST?

Una **API REST** (Representational State Transfer) es una **interfaz de programación de aplicaciones** que utiliza los métodos HTTP para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en recursos.

77.2 Instalación

Para instalar Django Rest Framework, ejecutamos el siguiente comando:

```
pip install djangorestframework
```

Una vez instalado, añadimos **'rest_framework'** a la lista de aplicaciones instaladas en el archivo settings.py:

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

77.3 Actualizar el archivo requirements.txt

Es necesario **eliminar** el archivo **requirements.txt** y volver a crearlo con el siguiente comando:

```
pip freeze > requirements.txt
```

77.4 Serializers

Los serializadores nos permiten convertir los datos de nuestro modelo en un formato que pueda ser fácilmente consumido por una API REST.

Para crear un serializador, creamos un archivo serializers.py en la carpeta de nuestra aplicación y añadimos el siguiente código:

```
from rest_framework import serializers                          ①
from .models import Producto

class ProductoSerializer(serializers.ModelSerializer):          ②
    class Meta:                                                 ③
        model = Producto                                        ④
        fields = '__all__'                                      ⑤
```

① Importamos el módulo serializers de Django Rest Framework.
② Creamos un serializador ProductoSerializer que hereda de serializers.ModelSerializer.
③ Definimos la clase Meta para configurar el serializador.
④ Especificamos el modelo Producto que queremos serializar.
⑤ Indicamos que queremos serializar todos los campos del modelo Producto.

77.5 Views

Las vistas en Django Rest Framework son similares a las vistas en Django. En lugar de devolver una **respuesta HTML**, devuelven una **respuesta JSON** que puede ser consumida por una **API REST**.

Para crear una vista, creamos un archivo **views.py** en la carpeta de nuestra aplicación y añadimos el siguiente código:

```
from rest_framework import viewsets                             ①
from .serializers import ProductoSerializer                     ②
from .models import Producto

class ProductoViewSet(viewsets.ModelViewSet):                   ③
    queryset = Producto.objects.all()                           ④
    serializer_class = ProductoSerializer                       ⑤
```

① Importamos el módulo viewsets de Django Rest Framework.
② Importamos el serializador ProductoSerializer que creamos anteriormente.
③ Creamos una vista ProductoViewSet que hereda de viewsets.ModelViewSet.

77.6 URLs de la Aplicación

Para conectar nuestras vistas con las URLs de nuestra aplicación, creamos un archivo urls.py en la carpeta de nuestra aplicación y añadimos el siguiente código:

```
from django.urls import path
from .views import ProductoViewSet                                          ①

urlpatterns = [
    path('api/productos/', ProductoViewSet.as_view({'get': 'list', 'post': 'create'}), na
    path('api/productos/<int:pk>/', ProductoViewSet.as_view({'get': 'retrieve', 'put': 'u
]
```

① Importamos la vista ProductoViewSet que creamos anteriormente.
② Configuramos la URL '/api/productos/' para listar y crear productos.
③ Configuramos la URL '/api/productos/<int:pk>/' para ver, actualizar y eliminar un producto específico.

77.7 Configuración URLs del Proyecto

Para configurar nuestra API REST, añadimos las URLs de nuestra aplicación a las URLs del proyecto en el archivo urls.py de la carpeta del proyecto:

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers
from productos.views import ProductoViewSet

# Creamos un enrutador para las vistas de Django REST Framework
router = routers.DefaultRouter()
router.register(r'productos', ProductoViewSet)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('productos.urls')),
    path('api/', include(router.urls)),
]
```

77.8 Migraciones

Antes de utilizar nuestra API REST, debemos aplicar las migraciones necesarias para crear las tablas en la base de datos:

```
python manage.py makemigrations
python manage.py migrate
```

77.9 Instalación de setuptools

Para instalar setuptools, ejecutamos el siguiente comando:

```
pip install setuptools
```

Es necesario la instalación de setuptools para poder instalar las dependencias necesarias para el proyecto.

77.10 Ejecución

Una vez configurada nuestra API REST, podemos ejecutar el servidor de desarrollo de Django y acceder a la API a través de un navegador o una herramienta como Postman:

```
python manage.py runserver
```

En este caso, la API estará disponible en la ruta 'http://127.0.0.1:8000/api/productos/'.

78 Documentación de la API con drf-yasg

Primero instalamos el paquete de documentación de la API:

```
pip install drf-yasg
```

Luego, añadimos 'drf_yasg' a la lista de aplicaciones instaladas en el archivo settings.py:

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

Añadimos las URLs de la documentación de la API a las URLs del proyecto en el archivo urls.py de la carpeta del proyecto:

```
from django.urls import path, include
from rest_framework import routers
from productos.views import ProductoViewSet
from rest_framework.permissions import AllowAny
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

router = routers.DefaultRouter()
router.register(r'productos', ProductoViewSet)

schema_view = get_schema_view(
   openapi.Info(
      title="API de Productos",
      default_version='v1',
      description="Documentación de la API de Productos",
      terms_of_service="https://www.google.com/policies/terms/",
      contact=openapi.Contact(email="contact@example.com"),
      license=openapi.License(name="BSD License"),
   ),
   public=True,
   permission_classes=(AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('productos.urls')),
```

```
    path('api/', include(router.urls)),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagge
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

Una vez configurada la documentación de la API, podemos ejecutar el servidor de desarrollo de Django y acceder a la documentación de la API a través de un navegador:

```
python manage.py runserver
```

79 Documentación de la API con CoreAPI

CoreAPI es una biblioteca que facilita la creación y el consumo de APIs.

79.1 Primero instalamos CoreAPI:

```
pip install coreapi
```

Para generar la documentación automáticamente, agregamos la configuración en settings.py:

```
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema'
}
```

Luego, añadimos la vista de esquema en el archivo urls.py de la carpeta del proyecto:

```
from rest_framework.schemas import get_schema_view

schema_view = get_schema_view(title='API de Productos')

urlpatterns += [
    path('docs/', schema_view, name='api-docs'),
]
```

Ahora podemos ejecutar el servidor de desarrollo y acceder a la documentación generada por CoreAPI en la ruta '/docs/':

```
python manage.py runserver
```

Con estos pasos, hemos integrado Django Rest Framework, documentado nuestra API con drf-yasg y CoreAPI.

80 Bases de Dato en Django

En este capítulo, aprenderemos a trabajar con bases de datos en Django. Crearemos un modelo de base de datos para almacenar información sobre productos y luego lo expondremos a través de una API REST.

Utilizaremos tanto bases de datos relacionales como no relacionales para almacenar información sobre productos. Crearemos un modelo de base de datos para almacenar información sobre productos y luego lo expondremos a través de una API REST.

Utilizaremos un entorno docker para ejecutar una base de datos PostgreSQL y otra base de datos MongoDB. Luego, configuraremos Django para conectarse a estas bases de datos y almacenar información sobre productos.

Para ello iniciaremos un nuevo proyecto donde crearemos un sistema que permita administrar una biblioteca de libros. Crearemos dos modelos de base de datos: uno para almacenar información sobre los libros y otro para almacenar información sobre los autores.

80.1 Objetivos

- **Modelos de Base de Datos:** Crear modelos de base de datos para almacenar información sobre libros y autores.
- **API REST:** Exponer los modelos de base de datos a través de una API REST utilizando Django REST Framework.

- **Migraciones:** Aplicar migraciones para crear las tablas en la base de datos.
- **Documentación de la API:** Documentar la API utilizando drf-yasg.

81 Creación del Entorno Virtual

Para comenzar, crearemos un nuevo entorno virtual para nuestro proyecto. Utilizaremos **virtualenv** para crear un entorno virtual llamado **biblioteca**.

```
python -m venv env
```

Luego, activaremos el entorno virtual:

- En Windows:

```
env\Scripts\activate
```

- En macOS y Linux:

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

82 Instalación de Django

A continuación, instalaremos Django en nuestro entorno virtual:

```
pip install django==4.2.0
```

83 Creación del Proyecto

Crearemos un nuevo proyecto de Django llamado **biblioteca**:

```
django-admin startproject biblioteca .
```

Luego, crearemos una nueva aplicación llamada **libros**:

```
python manage.py startapp libros
```

84 Configuración de la Base de Datos

84.1 Base de Datos Relacional (PostgreSQL)

> **Tip**
>
> Para crear el contenedor de Docker con PostgreSQL, utilizaremos variables de entorno para configurar la base de datos. En este caso, configuraremos la base de datos con el nombre **biblioteca**, el usuario **admin**, y la contraseña **admin**.

Para trabajar con una base de datos relacional, utilizaremos PostgreSQL. Crearemos un contenedor de Docker con PostgreSQL y configuraremos Django para conectarse a esta base de datos.

Para trabajar con la base de datos no relacional, utilizaremos MongoDB. Crearemos un contenedor de Docker con MongoDB y configuraremos Django para conectarse a esta base de datos.

Primero, crearemos un archivo **docker-compose.yml** en la raíz del proyecto con la siguiente configuración:

```yaml
services:
  postgres:
    image: postgres:13
    environment:
      POSTGRES_DB: biblioteca
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  mongo:
    image: mongo:4.4
    environment:
      MONGO_INITDB_DATABASE: biblioteca
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin
    ports:
      - "27017:27017"
    volumes:
```

```
      - mongo_data:/data/db

volumes:
  postgres_data:
  mongo_data:
```

Con el comando anterior, creamos un contenedor de Docker con PostgreSQL y otro con MongoDB. Configuramos la base de datos con el nombre **biblioteca**, el usuario **admin**, y la contraseña **admin**.

85 Creación de las variables de entorno

Para cargar las variables de entorno desde un archivo, utilizaremos el paquete **python-dotenv**. A continuación, instalaremos **python-dotenv** en nuestro entorno virtual:

```
pip install python-dotenv
```

Luego, crearemos un archivo **.env** en la raíz del proyecto con las siguientes variables de entorno:

```
POSTGRES_DB=biblioteca
POSTGRES_USER=admin
POSTGRES_PASSWORD=admin

MONGO_INITDB_DATABASE=biblioteca
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=admin
```

Luego, iniciaremos el contenedor de Docker con PostgreSQL y MongoDB utilizando el siguiente comando:

```
doc ┌─env 🤵 statick at 💻 fedora in 📁 ~/.../practicas/practica_django_2024
└λ docker compose ps
NAME                              IMAGE          COMMAND                  SERVICE    CREATED          STATUS          PORTS
practica_django_2024-mongo-1      mongo:4.4      "docker-entrypoint.s…"   mongo      32 minutes ago   Up 32 minutes   0.0.0.0:27017->27017/
tcp, :::27017->27017/tcp
practica_django_2024-postgres-1   postgres:13    "docker-entrypoint.s…"   postgres   32 minutes ago   Up 32 minutes   0.0.0.0:5432->5432/tc
p, :::5432->5432/tcp
```

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

Para conectarnos a las bases de datos PostgreSQL y MongoDB, utilizaremos las siguientes credenciales:

- **PostgreSQL:**
  - **Usuario:** admin
  - **Contraseña:** admin
  - **Base de Datos:** biblioteca
- **MongoDB:**
  - **Usuario:** admin
  - **Contraseña:** admin
  - **Base de Datos:** biblioteca

Estos datos los utilizamos en el archivo **settings.py** para configurar la conexión a la base de datos.

85.1 Configuración de la Base de Datos en Django

Para configurar Django para conectarse a la base de datos PostgreSQL, y MongoDB, añadiremos las siguientes configuraciones al archivo **settings.py**:

```
import os
from dotenv import load_dotenv

load_dotenv()

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRES_DB'),
        'USER': os.getenv('POSTGRES_USER'),
        'PASSWORD': os.getenv('POSTGRES_PASSWORD'),
        'HOST': 'localhost',
        'PORT': '5432',
    },
    'mongodb': {
        'ENGINE': 'djongo',
        'NAME': os.getenv('MONGO_INITDB_DATABASE'),
        'ENFORCE_SCHEMA': False,
        'CLIENT': {
            'host': f"mongodb://{os.getenv('MONGO_INITDB_ROOT_USERNAME')}:{os.getenv('MON
        }
    }
}
```

86 Instalación de los drivers de PostgreSQL y MongoDB

Para conectarnos a la base de datos PostgreSQL y MongoDB, necesitamos instalar los drivers correspondientes. A continuación, instalaremos los drivers necesarios en nuestro entorno virtual:

```
pip install psycopg2-binary djongo
```

Con esta configuración, Django se conectará a la base de datos PostgreSQL y MongoDB utilizando las variables de entorno definidas en el archivo **.env**.

Finalmente, crearemos una nueva base de datos llamada **biblioteca**.

87 Modelos de Base de Datos

87.1 Modelo de Libro

Comenzaremos creando un modelo de base de datos para almacenar información sobre los libros. Abriremos el archivo **libros/models.py** y definiremos el modelo de libro de la siguiente manera:

```
from django.db import models

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autor = models.CharField(max_length=100)
    editorial = models.CharField(max_length=100)
    fecha_publicacion = models.DateField()

    def __str__(self):
        return self.titulo
```

Luego, registraremos el modelo en el archivo **libros/admin.py** para poder administrarlo a través del panel de administración de Django:

```
from django.contrib import admin
from .models import Libro

admin.site.register(Libro)
```

87.2 Modelo de Autor

A continuación, crearemos un modelo de base de datos para almacenar información sobre los autores. Abriremos el archivo **libros/models.py** y definiremos el modelo de autor de la siguiente manera:

```
class Autor (models.Model):
    nombre = models.CharField(max_length=100)
    apellido = models.CharField(max_length=100)
    fecha_nacimiento = models.DateField()

    def __str__(self):
        return self.nombre + ' ' + self.apellido
```

Luego, registraremos el modelo en el archivo **libros/admin.py**:

```
from .models import Autor

admin.site.register(Autor)
```

88 Migraciones

Antes de utilizar nuestra API REST, debemos aplicar las migraciones necesarias para crear las tablas en la base de datos:

```
python manage.py makemigrations
python manage.py migrate
```

89 Django REST Framework

A continuación, instalaremos Django REST Framework en nuestro entorno virtual:

```
pip install djangorestframework
```

Luego, añadiremos **rest_framework** a la lista de aplicaciones instaladas en el archivo **settings.py**:

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

90 Serializadores

Para exponer los modelos de base de datos a través de una API REST, crearemos serializadores para los modelos de libro y autor. Crearemos un archivo **libros/serializers.py** y definiremos los serializadores de la siguiente manera:

```
from rest_framework import serializers
from .models import Libro, Autor

class LibroSerializer(serializers.ModelSerializer):
    class Meta:
        model = Libro
        fields = '__all__'

class AutorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Autor
        fields = '__all__'
```

91 Vistas

A continuación, crearemos vistas utilizando Django REST Framework para exponer los modelos de libro y autor a través de una API REST. Crearemos un archivo **libros/views.py** y definiremos las vistas de la siguiente manera:

```
from rest_framework import viewsets
from .models import Libro, Autor
from .serializers import LibroSerializer, AutorSerializer

class LibroViewSet(viewsets.ModelViewSet):
    queryset = Libro.objects.all()
    serializer_class = LibroSerializer

class AutorViewSet(viewsets.ModelViewSet):
    queryset = Autor.objects.all()
    serializer_class = AutorSerializer
```

92 Rutas

Finalmente, crearemos rutas para acceder a los modelos de libro y autor a través de la API REST. Abriremos el archivo **biblioteca/urls.py** y definiremos las rutas de la siguiente manera:

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from libros.views import LibroViewSet, AutorViewSet
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="API de Biblioteca",
        default_version='v1',
        description="API para administrar una biblioteca de libros y autores.",
    ),
    public=True,
)

router = DefaultRouter()
router.register(r'libros', LibroViewSet)
router.register(r'autores', AutorViewSet)

urlpatterns = [
    path('', include(router.urls)),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagge
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

93 Documentación de la API

Para documentar la API REST, utilizaremos **drf-yasg**. A continuación, instalaremos **drf-yasg** en nuestro entorno virtual:

```
pip install drf-yasg
```

Luego, añadiremos **drf_yasg** a la lista de aplicaciones instaladas en el archivo **settings.py**:

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

Añadiremos las URLs de la documentación de la API a las URLs del proyecto en el archivo **biblioteca/urls.py**:

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from libros.views import LibroViewSet, AutorViewSet
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="API de Biblioteca",
        default_version='v1',
        description="API para administrar una biblioteca de libros y autores.",
    ),
    public=True,
)

router = DefaultRouter()
router.register(r'libros', LibroViewSet)
router.register(r'autores', AutorViewSet)

urlpatterns = [
     path('', include(router.urls)),
     path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagg
     path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

94 Instalación de setuptools

Para instalar el paquete de setuptools, ejecutaremos el siguiente comando:

```
pip install setuptools
```

95 Actualización de requirements.txt

Finalmente, actualizaremos el archivo **requirements.txt** con las dependencias necesarias para nuestro proyecto, recuerda que es necesario tener el archivo **requirements.txt** en la raíz del proyecto.:

```
pip freeze update > requirements.txt
```

96 Ejecución del Servidor

Finalmente, ejecutaremos el servidor de desarrollo de Django para probar nuestra API REST:

```
python manage.py runserver
```

Podremos acceder a la documentación de la API a través de un navegador:

- **Swagger:** http://127.0.0.1:8000/swagger/
- **Redoc:** http://127.0.0.1:8000/redoc/

Con esto, hemos creado un sistema para administrar una biblioteca de libros utilizando Django y bases de datos relacionales y no relacionales.

97 Probar la API con Thunder Client

Para probar la API REST, utilizaremos la extensión **Thunder Client** para Visual Studio Code. Crearemos una nueva solicitud para obtener una lista de libros:

- **Método:** GET
- **URL:** http://127.0.0.1:8000/libros/

Luego, enviaremos la solicitud y veremos la lista de libros en formato JSON.

- **Método:** POST
- **URL:** http://127.0.0.1:8000/libros/
- **Cuerpo:**

```
{
    "titulo": "El Quijote",
    "autor": "Miguel de Cervantes",
    "editorial": "Editorial Cervantes",
    "fecha_publicacion": "1605-01-01"
}
```

Con esto, hemos creado un sistema para administrar una biblioteca de libros utilizando Django y bases de datos relacionales y no relacionales.

- **Método:** Put
- **URL:** http://127.0.0.1:8000/libros/1/
- **Cuerpo:**

```
{
    "titulo": "El Quijote 1",
    "autor": "Miguel de Cervantes",
    "editorial": "Editorial Cervantes",
    "fecha_publicacion": "1605-01-01"
}
```

- **Método:** DELETE
- **URL:** http://127.0.0.1:8000/libros/1/

Con esto, hemos creado un sistema para administrar una biblioteca de libros utilizando Django y bases de datos relacionales y no relacionales.

98 Reto

1. **Modelos de Base de Datos:** Crea un modelo de base de datos para almacenar información sobre préstamos de libros en la biblioteca.
2. **API REST:** Expon los modelos de base de datos a través de una API REST utilizando Django REST Framework.
3. **Migraciones:** Aplica las migraciones necesarias para crear las tablas en la base de datos.
4. **Documentación de la API:** Documenta la API utilizando drf-yasg.

99 Conclusión

En este capítulo, aprendimos a trabajar con bases de datos en Django. Creamos un modelo de base de datos para almacenar información sobre libros y autores y lo expusimos a través de una API REST utilizando Django REST Framework. Utilizamos tanto bases de datos relacionales como no relacionales para almacenar información sobre productos. Configuramos Django para conectarse a bases de datos PostgreSQL y MongoDB y almacenar información sobre libros y autores. Finalmente, documentamos la API utilizando drf-yasg y probamos la API utilizando Thunder Client.

100 Testing

Vamos a empezar la parte de testing para ello vamos a crear en la ruta raiz el archivo pytest.ini

```
[pytest]

DJANGO_SETTINGS_MODULE = inventario.test_settings

python_files = tests.py test_*.py *_tests.py
```

Ahora vamos a crear el archivo test_settings.py en la carpeta inventario

```
from .settings import *

# Configuraciones adicionales para el entorno de prueba
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': ':memory:',
    }
}
```

Ahora vamos a crear el archivo test_productos.py en la carpeta inventario

```
import pytest
from productos.models import Producto

@pytest.mark.django_db
def test_crear_producto():
    producto = Producto.objects.create(
        nombre="Producto de prueba",
        precio=10.00,
        cantidad=5
    )
    assert producto.nombre == "Producto de prueba"
    assert producto.precio == 10.00
    assert producto.cantidad == 5

@pytest.mark.django_db
def test_str_producto():
    producto = Producto.objects.create(
```

```
        nombre="Producto de prueba",
        precio=10.00,
        cantidad=5
    )
    assert str(producto) == "Producto de prueba"
```

Ahora vamos a ejecutar el comando pytest

```
pytest
```

Si todo esta correcto deberiamos ver algo como esto

```
========================================================= test session starts ===============
platform linux -- Python 3.12.3, pytest-8.2.0, pluggy-1.5.0
django: version: 4.2, settings: inventario.test_settings (from ini)
rootdir: /home/statick/workspaces/Curso_django_and_react/inventario_django
configfile: pytest.ini
plugins: django-4.8.0
collected 2 items

productos/tests/test_productos.py ..

========================================================= warnings summary =================
```

Con esto podemos ver que los test han pasado correctamente.

Podemos ver que pytest.ini tiene la configuracion de DJANGO_SETTINGS_MODULE = inventario.test_settings, esto es para que pytest pueda encontrar las configuraciones de django.

101 Corrección de tests en Django Rest Framework

101.1 Introducción

En este documento se describen los pasos necesarios para corregir los tests de Django Rest Framework.

Actualmente tenemos el siguiente error en los tests:

```
=================================================== test session starts ===================
platform linux -- Python 3.12.3, pytest-8.2.0, pluggy-1.5.0
rootdir: /home/statick/workspaces/Curso_django_and_react/inventario_django
collected 0 items / 1 error

========================================================== ERRORS ==========================
_______________________________________ ERROR collecting productos/test_views.py _______
ImportError while importing test module '/home/statick/workspaces/Curso_django_and_react/
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib64/python3.12/importlib/__init__.py:90: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
productos/test_views.py:1: in <module>
    from django.urls import reverse
E   ModuleNotFoundError: No module named 'django'
================================================= short test summary info =================
ERROR productos/test_views.py
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!
===================================================== 1 error in 0.07s ====================
```

101.2 Pasos

101.2.1 1. Crear un entorno virtual

```
python3 -m venv venv
```

101.2.2 2. Activar el entorno virtual

```
source venv/bin/activate
```

101.2.3 3. Instalar las dependencias

```
pip install -r requirements.txt
```

101.2.4 4. Correr los tests

```
pytest
```

101.2.5 5. Corregir los tests

Para corregir los tests, se debe modificar el archivo `productos/test_views.py` y corregir el error de importación.

101.2.6 6. Correr los tests nuevamente

```
pytest
```

101.2.7 7. Desactivar el entorno virtual

```
deactivate
```

101.3 Conclusión

Una vez corregidos los tests, se debe hacer un pull request al repositorio original.

101.4 Referencias

- Django Rest Framework
- Django
- pytest
- Virtualenv
- Python
- Git
- GitHub
- Markdown
- Pip

Part VI

Unidad 6: Frontend

102 HTML

Este capítulo de HTML para desarrolladores web proporciona una sólida visión general para desarrolladores, desde principiantes hasta expertos en HTML.

Bienvenido a HTML!

HyperText Markup Language, o HTML, es la columna vertebral de la web, proporcionando el contenido, así como la estructura de ese contenido, que se muestra en tu navegador web.

A menos que estés leyendo un PDF o una versión impresa de esta página, este contenido está compuesto por varios elementos y texto HTML. HTML es la capa de contenido de la web. Los elementos HTML son los nodos que componen el Modelo de Objetos del Documento.

Las **Hojas de Estilo en Cascada** proporcionan el aspecto y la apariencia, o capa de presentación de la página. **JavaScript es la capa de comportamiento**, a menudo utilizado para manipular los objetos dentro de un documento. Los sitios web que se construyen con frameworks de JavaScript realmente están manipulando HTML. A su vez, es importante marcar tu HTML de una manera que los scripts puedan analizar fácilmente y que las tecnologías de asistencia puedan entender fácilmente. Esto significa escribir código HTML con estándares modernos.

103 Resumen

Este capítulo de HTML para desarrolladores web proporciona una sólida visión general para desarrolladores, desde principiantes hasta expertos en HTML. Si eres completamente nuevo en HTML, aprenderás cómo construir contenido estructuralmente sólido. Si has estado construyendo sitios web durante años, este curso puede llenar lagunas de conocimiento que ni siquiera sabías que tenías.

A lo largo de este viaje, estaremos construyendo la estructura para MachineLearningWorkshop.com. Ninguna máquina resultó dañada en la creación de esta serie.

Esto no es una referencia completa. Cada sección presenta el tema de la sección con explicaciones breves y ejemplos, brindándote la oportunidad de explorar más a fondo. Habrá enlaces a referencias de temas, como las especificaciones de MDN y WHATWG, y otros artículos de web.dev. Si bien este no es un curso de accesibilidad, cada sección incluirá las mejores prácticas de accesibilidad y problemas específicos, con enlaces a investigaciones más profundas sobre el tema. Cada sección tendrá una breve evaluación para ayudar a las personas a confirmar su comprensión.

103.1 Visión general de HTML

HTML es el lenguaje de marcado estándar para crear páginas web. HTML significa Lenguaje de Marcado de Hipertexto. Un lenguaje de marcado es un conjunto de etiquetas que se utilizan para definir la estructura de una página web. El contenido de una página web se define con HTML. Los elementos HTML son los bloques de construcción de las páginas HTML.

103.2 Una breve introducción a los conceptos clave en HTML.

HTML es un lenguaje de marcado que define la estructura de tu contenido, consta de una serie de elementos que rodean o envuelven el contenido para que se muestre o actúe de una manera particular.

Los elementos HTML son los bloques de construcción de las páginas HTML. Los elementos HTML se representan mediante etiquetas. Las etiquetas HTML etiquetan piezas de contenido como “encabezado”, “párrafo”, “tabla” y así sucesivamente. Browsers web renderizan el contenido HTML en una página web.

103.2.1 Encabezado de un documento HTML

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
El contenido de la página va aquí.
</body>
</html>
```

103.2.2 Párrafo

```
<p>Este es un párrafo.</p>
```

103.2.3 Tabla

```
<table>
  <tr>
    <th>Encabezado de la tabla</th>
  </tr>
  <tr>
    <td>Dato de la tabla</td>
  </tr>
</table>
```

103.3 Estructura del documento

103.3.1 Aprende cómo estructurar tus documentos HTML con una base sólida.

Un documento HTML consta de una serie de elementos HTML anidados. Un documento HTML comienza con un elemento raíz **

, seguido de un elemento

** y un elemento **

. El elemento

** contiene metadatos sobre el documento, como el título de la página. El elemento **

** contiene el contenido de la página.

```html
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
El contenido de la página va aquí.
</body>
</html>
```

103.4 Metadatos

103.4.1 Cómo utilizar etiquetas meta para proporcionar información sobre tus documentos.

Las etiquetas meta proporcionan información sobre el documento HTML. Los metadatos no se muestran en la página web, pero son importantes para los motores de búsqueda y otros servicios web.

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="description" content="Descripción de la página">
<meta name="keywords" content="Palabras clave, separadas por comas">
<meta name="author" content="Nombre del autor">
</head>
<body>
El contenido de la página va aquí.
</body>
</html>
```

En el ejemplo anterior se utilizan las etiquetas meta para proporcionar información sobre el documento HTML, como la codificación de caracteres, la descripción de la página, las palabras clave y el autor.

103.5 HTML semántico

103.5.1 Usar los elementos HTML correctos para describir el contenido de tu documento.

HTML semántico es un enfoque de escribir HTML que hace que el código sea más claro y legible tanto para los humanos como para las máquinas. Los elementos HTML semánticos describen el significado del contenido en lugar de su apariencia.

```html
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<header>
<h1>Encabezado de la página</h1>
</header>
<nav>
<ul>
<li><a href="#">Enlace 1</a></li>
<li><a href="#">Enlace 2</a></li>
<li><a href="#">Enlace 3</a></li>
</ul>
</nav>
<section>
<h2>Sección 1</h2>
<p>Contenido de la sección 1.</p>
</section>
<section>
<h2>Sección 2</h2>
<p>Contenido de la sección 2.</p>
</section>
<footer>
<p>Pie de página</p>
</footer>
</body>
</html>
```

En el ejemplo anterior se utilizan elementos HTML semánticos como **<header>**, **<nav>**, **<section>** y **<footer>** para describir el contenido de la página.

103.6 Encabezados y secciones

103.6.1 Cómo utilizar correctamente los elementos de sección para darle significado a tu contenido.

Los elementos de sección en HTML son elementos que definen secciones de contenido en un documento HTML. Los elementos de sección ayudan a organizar y estructurar el contenido de una página web.

```html
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
```

```
</head>
<body>
<h1>Encabezado de nivel 1</h1>
<section>
<h2>Sección 1</h2>
<p>Contenido de la sección 1.</p>
</section>
<section>
<h2>Sección 2</h2>
<p>Contenido de la sección 2.</p>
</section>
</body>
</html>
```

En el ejemplo anterior se utilizan elementos de sección como **<section>** y **<h2>** para organizar y estructurar el contenido de la página.

103.7 Atributos

103.7.1 Aprende sobre los diferentes atributos globales junto con los atributos específicos de elementos HTML particulares.

Los atributos HTML proporcionan información adicional sobre los elementos HTML. Los atributos pueden ser globales o específicos de un elemento.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<img src="imagen.jpg" alt="Texto alternativo">
<a href="https://www.ejemplo.com">Enlace</a>
</body>
</html>
```

En el ejemplo anterior se utilizan los atributos **src** y **alt** en el elemento **<img>** y el atributo **href** en el elemento **<a>**.

103.8 HTML particulares.

103.8.1 Aprende sobre los diferentes atributos globales junto con los atributos específicos de elementos HTML particulares.

Los atributos HTML proporcionan información adicional sobre los elementos HTML. Los atributos pueden ser globales o específicos de un elemento.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<img src="imagen.jpg" alt="Texto alternativo">
<a href="https://www.ejemplo.com">Enlace</a>
</body>
</html>
```

En el ejemplo anterior se utilizan los atributos **src** y **alt** en el elemento **<img>** y el atributo **href** en el elemento **<a>**.

103.9 Conceptos básicos de texto

103.9.1 Cómo formatear texto utilizando HTML.

HTML proporciona una serie de elementos para formatear texto, como ****, ****, **<u>**, **<s>**, **<sub>** y **<sup>**.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<p>Este es un <strong>párrafo</strong> con texto en negrita.</p>
<p>Este es un <em>párrafo</em> con texto en cursiva.</p>
<p>Este es un <u>párrafo</u> con texto subrayado.</p>
<p>Este es un <s>párrafo</s> con texto tachado.</p>
<p>Este es un <sub>párrafo</sub> con texto subíndice.</p>
<p>Este es un <sup>párrafo</sup> con texto superíndice.</p>
</body>
</html>
```

En el ejemplo anterior se utilizan elementos HTML como ****, ****, **<u>**, **<s>**, **<sub>** y **<sup>** para formatear texto.

103.10 Enlaces

103.10.1 Todo lo que necesitas saber sobre enlaces.

Los enlaces HTML se crean con el elemento **<a>**. El atributo **href** especifica la URL de la página a la que enlaza el enlace.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<a href="https://www.ejemplo.com">Enlace a ejemplo.com</a>
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<a>** con el atributo **href** para crear un enlace a una página web.

103.11 Listas

103.11.1 Cómo crear listas en HTML.

HTML proporciona elementos para crear listas ordenadas (**<ol>**), listas desordenadas (**<ul>**) y listas de definición (**<dl>**).

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<h2>Listas ordenadas</h2>
<ol>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ol>
<h2>Listas desordenadas</h2>
<ul>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ul>
<h2>Listas de definición</h2>
```

```
<dl>
<dt>Término 1</dt>
<dd>Definición 1</dd>
<dt>Término 2</dt>
<dd>Definición 2</dd>
</dl>
</body>
</html>
```

En el ejemplo anterior se utilizan elementos HTML como **<ol>**, **<ul>**, **<li>**, **<dl>**, **<dt>** y **<dd>** para crear listas.

103.12 Navegación

103.12.1 La navegación es un elemento clave de cualquier sitio o aplicación, y comienza con HTML.

La navegación en HTML se puede crear con elementos como **<nav>**, **<ul>** y **<li>**.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<nav>
<ul>
<li><a href="#">Inicio</a></li>
<li><a href="#">Acerca de</a></li>
<li><a href="#">Servicios</a></li>
<li><a href="#">Contacto</a></li>
</ul>
</nav>
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<nav>** con elementos **<ul>** y **<li>** para crear una barra de navegación.

103.13 Tablas

103.13.1 Comprender cómo utilizar tablas para marcar datos tabulares.

Las tablas en HTML se crean con elementos como **<table>**, **<tr>**, **<th>** y **<td>**.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<table>
<tr>
<th>Encabezado 1</th>
<th>Encabezado 2</th>
</tr>
<tr>
<td>Dato 1</td>
<td>Dato 2</td>
</tr>
<tr>
<td>Dato 3</td>
<td>Dato 4</td>
</tr>
</table>
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<table>** con elementos **<tr>**, **<th>** y **<td>** para crear una tabla.

103.14 Formularios

103.14.1 Una visión general de los formularios en HTML.

Los formularios en HTML se crean con elementos como **<form>**, **<input>**, **<textarea>** y **<button>**.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<form>
<label for="nombre">Nombre:</label>
<input type="text" id="nombre" name="nombre">
<label for="correo">Correo:</label>
<input type="email" id="correo" name="correo">
<label for="mensaje">Mensaje:</label>
<textarea id="mensaje" name="mensaje"></textarea>
```

```
<button type="submit">Enviar</button>
</form>
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<form>** con elementos **<input>**, **<textarea>** y **<button>** para crear un formulario.

103.15 Imágenes

103.15.1 Una visión general de las imágenes en HTML.

Las imágenes en HTML se crean con el elemento **<img>**. El atributo **src** especifica la URL de la imagen.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<img src="imagen.jpg" alt="Texto alternativo">
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<img>** con el atributo **src** para mostrar una imagen en la página.

103.16 Audio y video

103.16.1 Descubre cómo trabajar con medios HTML como audio y video.

Los elementos de audio y video en HTML se crean con los elementos **<audio>** y **<video>**. Los atributos **src** y **controls** son comunes a ambos elementos.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<audio src="audio.mp3" controls></audio>
<video src="video.mp4" controls></video>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<audio>** y **<video>** con los atributos **src** y **controls** para reproducir audio y video en la página.

103.17 Plantilla, ranura y sombra

103.17.1 Una explicación de plantilla, ranura y sombra.

Los elementos de plantilla, ranura y sombra en HTML se utilizan para crear contenido reutilizable y encapsulado.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<template>
<p>Este es un contenido de plantilla.</p>
</template>
<slot></slot>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<template>** y **<slot>** para crear contenido reutilizable y encapsulado.

103.18 APIs de HTML

103.18.1 Aprende cómo se puede exponer y manipular información HTML utilizando JavaScript.

Las APIs de HTML proporcionan una forma de interactuar con el contenido HTML utilizando JavaScript.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<p id="parrafo">Este es un párrafo.</p>
<script>
document.getElementById("parrafo").innerHTML = "Este es un nuevo párrafo.";
</script>
</body>
</html>
```

En el ejemplo anterior se utiliza el método **getElementById** para seleccionar un elemento HTML y cambiar su contenido.

103.19 Enfoque

103.19.1 Cómo gestionar el orden de enfoque en tus documentos HTML.

El enfoque en HTML se puede gestionar con el atributo **tabindex** y el método **focus()** en JavaScript.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<input type="text" id="campo1" tabindex="1">
<input type="text" id="campo2" tabindex="2">
<input type="text" id="campo3" tabindex="3">
<script>
document.getElementById("campo2").focus();
</script>
</body>
</html>
```

En el ejemplo anterior se utiliza el atributo **tabindex** y el método **focus()** para gestionar el orden de enfoque en los campos de texto.

103.20 Otros elementos de texto en línea

103.20.1 Una introducción a la variedad de elementos utilizados para marcar texto.

HTML proporciona una variedad de elementos para marcar texto, como **<a>**, ****, ****, **<u>**, **<s>**, **<sub>** y **<sup>**.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<p>Este es un <a href="#">enlace</a>.</p>
<p>Este es un <strong>texto en negrita</strong>.</p>
<p>Este es un <em>texto en cursiva</em>.</p>
```

```
<p>Este es un <u>texto subrayado</u>.</p>
<p>Este es un <s>texto tachado</s>.</p>
<p>Este es un <sub>texto subíndice</sub>.</p>
<p>Este es un <sup>texto superíndice</sup>.</p>
</body>
</html>
```

En el ejemplo anterior se utilizan elementos HTML como **<a>**, ****, ****, **<u>**, **<s>**, **<sub>** y **<sup>** para marcar texto.

103.21 Detalles y resumen

103.21.1 Descubre cómo funcionan los elementos de detalles y resumen, muy útiles, y dónde usarlos.

Los elementos de detalles y resumen en HTML se utilizan para crear un widget de detalles que se puede mostrar u ocultar.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<details>
<summary>Detalles</summary>
<p>Este es un contenido de detalles.</p>
</details>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<details>** y **<summary>** para crear un widget de detalles.

103.22 Diálogo

103.22.1 El elemento es un elemento útil para representar cualquier tipo de diálogo en HTML, descubre cómo funciona.

El elemento **<dialog>** en HTML se utiliza para representar cualquier tipo de diálogo, como una ventana emergente o un cuadro de diálogo.

```html
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<dialog open>
<p>Este es un contenido de diálogo.</p>
<button>Botón de cierre</button>
</dialog>
</body>
</html>
```

En el ejemplo anterior se utiliza el elemento **<dialog>** para representar un diálogo con un botón de cierre.

103.23 Elementos de agrupación

103.23.1 Descubre cómo utilizar los elementos de agrupación en HTML.

Los elementos de agrupación en HTML se utilizan para agrupar elementos relacionados juntos.

```html
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<fieldset>
<legend>Grupo de campos</legend>
<input type="text" id="campo1">
<input type="text" id="campo2">
<input type="text" id="campo3">
</fieldset>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<fieldset>** y **<legend>** para agrupar campos de formulario relacionados juntos.

103.24 Elementos de contenido incrustado

103.24.1 Descubre cómo incrustar contenido en tu documento HTML.

Los elementos de contenido incrustado en HTML se utilizan para incrustar contenido de otros documentos o fuentes.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<iframe src="https://www.ejemplo.com"></iframe>
<object data="archivo.pdf" type="application/pdf"></object>
<embed src="archivo.swf" type="application/x-shockwave-flash">
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<iframe>**, **<object>** y **<embed>** para incrustar contenido en la página.

103.25 Elementos de formulario

103.25.1 Descubre cómo utilizar los elementos de formulario en HTML.

Los elementos de formulario en HTML se utilizan para crear formularios interactivos en una página web.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<form>
<label for="nombre">Nombre:</label>
<input type="text" id="nombre" name="nombre">
<label for="correo">Correo:</label>
<input type="email" id="correo" name="correo">
<label for="mensaje">Mensaje:</label>
<textarea id="mensaje" name="mensaje"></textarea>
<button type="submit">Enviar</button>
</form>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<form>**, **<input>**, **<textarea>** y para crear un formulario interactivo.

103.26 Elementos de marco

103.26.1 Descubre cómo utilizar los elementos de marco en HTML.

Los elementos de marco en HTML se utilizan para dividir una página en secciones independientes.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<frameset cols="25%, 75%">
<frame src="menu.html">
<frame src="contenido.html">
</frameset>
</html>
```

En el ejemplo anterior se utilizan los elementos **<frameset>** y **<frame>** para dividir una página en secciones independientes.

103.27 Elementos de interacción

103.27.1 Descubre cómo utilizar los elementos de interacción en HTML.

Los elementos de interacción en HTML se utilizan para crear elementos interactivos en una página web.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<button>Botón</button>
<input type="text">
<select>
<option>Elemento 1</option>
<option>Elemento 2</option>
<option>Elemento 3</option>
</select>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<button>**, **<input>** y **<select>** para crear elementos interactivos en la página.

103.28 Elementos de lista

103.28.1 Descubre cómo utilizar los elementos de lista en HTML.

Los elementos de lista en HTML se utilizan para crear listas de elementos.

```
<!DOCTYPE html>
<html>
<head>
<title>Título de la página</title>
</head>
<body>
<ul>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ul>
<ol>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ol>
</body>
</html>
```

En el ejemplo anterior se utilizan los elementos **<ul>** y **<ol>** para crear listas de elementos.

104 Reto

Crea una página web con los siguientes elementos:

1. Un encabezado con el título de la página.
2. Un párrafo con el contenido de la página.
3. Una lista ordenada con al menos tres elementos.
4. Una lista desordenada con al menos tres elementos.
5. Una tabla con al menos tres filas y dos columnas.
6. Un formulario con al menos tres campos de entrada y un botón de envío.
7. Una imagen incrustada en la página.
8. Un enlace a una página web externa.

¡Buena suerte!

Ver solución

```
<!DOCTYPE html>
<html>
<head>
<title>Página de ejemplo</title>
</head>
<body>
<h1>Encabezado de la página</h1>
<p>Este es un párrafo de ejemplo.</p>
<h2>Lista ordenada</h2>
<ol>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ol>
<h2>Lista desordenada</h2>
<ul>
<li>Elemento 1</li>
<li>Elemento 2</li>
<li>Elemento 3</li>
</ul>
<h2>Tabla</h2>
<table>
<tr>
<th>Encabezado 1</th>
<th>Encabezado 2</th>
</tr>
```

```html
<tr>
<td>Dato 1</td>
<td>Dato 2</td>
</tr>
<tr>
<td>Dato 3</td>
<td>Dato 4</td>
</tr>
</table>
<h2>Formulario</h2>
<form>
<label for="nombre">Nombre:</label>
<input type="text" id="nombre" name="nombre">
<label for="correo">Correo:</label>
<input type="email" id="correo" name="correo">
<label for="mensaje">Mensaje:</label>
<textarea id="mensaje" name="mensaje"></textarea>
<button type="submit">Enviar</button>
</form>
<h2>Imagen</h2>
<img src="imagen.jpg" alt="Texto alternativo">
<h2>Enlace</h2>
<a href="https://www.ejemplo.com">Enlace a ejemplo.com</a>
</body>
</html>
```

105 Conclusiones

HTML es un lenguaje de marcado que define la estructura de tu contenido. Los elementos HTML son los bloques de construcción de las páginas HTML. HTML proporciona una serie de elementos para crear contenido estructurado, como encabezados, párrafos, listas, tablas, formularios, imágenes, audio y video. HTML también proporciona elementos para formatear texto, crear enlaces, navegar por el contenido, agrupar elementos y crear contenido incrustado. HTML es un lenguaje poderoso que te permite crear páginas web interactivas y atractivas.

:::

106 CSS

CSS significa Cascading Style Sheets, es un lenguaje de diseño gráfico para definir la presentación de documentos HTML. CSS describe cómo los elementos HTML deben ser mostrados en la pantalla, en papel, o en otras formas de medios. CSS ahorra mucho trabajo. Puede controlar el diseño de varias páginas a la vez. CSS puede ser almacenado en archivos externos, lo que permite actualizar un sitio web entero simplemente actualizando un archivo. CSS hace que el sitio web sea más accesible para los visitantes con discapacidades. CSS es fácil de aprender y entender, pero también es poderoso.

Para utilizar CSS en un documento HTML, se puede hacer de tres maneras:

1. **CSS Externo**: Se crea un archivo con extensión **.css** y se enlaza al documento HTML con la etiqueta **<link>**.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

En el archivo **styles.css** se pueden definir los estilos de los elementos HTML.

106.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```css
h1 {
    color: blue;
}
p {
    color: red;
}
```

2. **CSS Interno**: Se utiliza la etiqueta **<style>** dentro del documento HTML.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
    <style>
        h1 {
            color: blue;
        }
        p {
            color: red;
        }
    </style>
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

106.2 Actividad

- Definir los estilos de los elementos HTML dentro del documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <style>
        h1 {
            color: blue;
        }
        p {
            color: red;
        }
    </style>
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

3. **CSS en línea**: Se utiliza el atributo **style** en los elementos HTML.

Ejemplo:

```
<!DOCTYPE html>
<html>
<body>
    <h1 style="color: blue;">Este es un título</h1>
    <p style="color: red;">Este es un párrafo</p>
</body>
</html>
```

106.3 Actividad

- Definir los estilos de los elementos HTML utilizando el atributo **style**.

Ver respuesta

```
<!DOCTYPE html>
<html>
<body>
    <h1 style="color: blue;">Este es un título</h1>
    <p style="color: red;">Este es un párrafo</p>
</body>
</html>
```

En este capítulo se utilizará CSS Externo para definir los estilos de los elementos HTML.

Ahora que hemos aprendido cómo se puede utilizar CSS en un documento HTML, vamos a ver cómo se pueden definir los estilos de los elementos HTML.

Para comprender mejor el tema de CSS vamos a utilizar varios ejemplos que permitirán conocer cada uno de los conceptos necesarios para poder aplicar estilos a los elementos HTML.

107 Selectores

Los selectores son patrones que se utilizan para seleccionar los elementos a los que se les aplicarán los estilos. Existen varios tipos de selectores, los más comunes son:

1. Selector de tipo
2. Selector de clase
3. Selector de ID
4. Selector de atributo
5. Selector universal

A continuación se describen brevemente cada uno de los selectores:

1. **Selector de tipo**: Selecciona todos los elementos de un tipo específico.

Ejemplo:

```
p {
    color: red;
}
```

2. **Selector de clase**: Selecciona todos los elementos que tienen un atributo **class** específico.

Ejemplo:

```
.texto-rojo {
    color: red;
}
```

3. **Selector de ID**: Selecciona un único elemento que tiene un atributo **id** específico.

Ejemplo:

```
#titulo {
    color: blue;
}
```

4. **Selector de atributo**: Selecciona todos los elementos que tienen un atributo específico.

Ejemplo:

```
[title] {
    color: green;
}
```

5. **Selector universal**: Selecciona todos los elementos de la página.

Ejemplo:

```
* {
    color: black;
}
```

107.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando los selectores de tipo, clase, ID, atributo y universal.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1 id="titulo">Este es un título</h1>
    <p class="texto-rojo">Este es un párrafo</p>
    <p title="Este es un párrafo con título">Este es un párrafo con título</p>
    <p>Este es un párrafo sin título</p>
</body>
</html>
```

```
h1 {
    color: blue;
}

.texto-rojo {
    color: red;
}

#titulo {
    color: green;
}
```

```
[title] {
    color: purple;
}

* {
    color: black;
}
```

En los siguientes ejemplos se verán más detalles sobre cómo se pueden utilizar los selectores para aplicar estilos a los elementos HTML.

108 Comentarios

Para añadir comentarios en un archivo CSS se utiliza la siguiente sintaxis:

```
/* Este es un comentario */
```

Los comentarios en CSS se utilizan para explicar el código y hacerlo más fácil de entender.

108.1 Actividad

- Añadir comentarios en el archivo **styles.css**.

Ver respuesta

```
/* Este es un comentario */

h1 {
    color: blue; /* Este es un comentario */
}

.texto-rojo {
    color: red; /* Este es un comentario */
}

#titulo {
    color: green; /* Este es un comentario */
}

[title] {
    color: purple; /* Este es un comentario */
}

* {
    color: black; /* Este es un comentario */
}
```

Los comentarios son útiles para explicar el código y hacerlo más fácil de entender.

109 Colores, valores y convenciones

Par entender acerca de este tema crearemos un archivo **styles.css** y lo enlazaremos a un archivo **index.html**.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

En el archivo **styles.css** se pueden definir los estilos de los elementos HTML.

Ejemplo:

```css
h1 {
    color: blue;
}
p {
    color: red;
}
```

En este ejemplo, el título se mostrará en color azul y el párrafo en color rojo.

109.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```css
h1 {
    color: blue;
}

p {
    color: red;
}
```

En CSS se pueden utilizar varios tipos de valores para definir los estilos de los elementos HTML. Algunos de los valores más comunes son:

1. **Colores**: Se pueden utilizar nombres de colores, códigos hexadecimales, códigos RGB, códigos RGBA, códigos HSL y códigos HSLA.

Ejemplo:

```css
h1 {
    color: blue; /* Nombre de color */
    color: #ff0000; /* Código hexadecimal */
    color: rgb(255, 0, 0); /* Código RGB */
    color: rgba(255, 0, 0, 0.5); /* Código RGBA */
    color: hsl(0, 100%, 50%); /* Código HSL */
    color: hsla(0, 100%, 50%, 0.5); /* Código HSLA */
}
```

2. **Tamaños**: Se pueden utilizar valores en píxeles, porcentajes, ems, rems, etc.

Ejemplo:

```css
h1 {
    font-size: 24px; /* Tamaño en píxeles */
    font-size: 150%; /* Tamaño en porcentaje */
    font-size: 1.5em; /* Tamaño en ems */
    font-size: 1.5rem; /* Tamaño en rems */
}
```

3. **Unidades de medida**: Se pueden utilizar diferentes unidades de medida como píxeles, porcentajes, ems, rems, etc.

Ejemplo:

```
h1 {
    margin: 10px; /* Margen en píxeles */
    margin: 10%; /* Margen en porcentaje */
    margin: 1em; /* Margen en ems */
    margin: 1rem; /* Margen en rems */
}
```

4. **Fuentes**: Se pueden utilizar diferentes fuentes para los textos.

Ejemplo:

```
h1 {
    font-family: Arial, sans-serif; /* Fuente Arial */
}
```

110 Border

5. **Bordes**: Se pueden utilizar diferentes estilos de bordes.

Ejemplo:

```
h1 {
    border: 1px solid black; /* Borde sólido de 1 píxel de ancho y color */
}
```

110.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando colores, tamaños, unidades de medida y fuentes.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```
h1 {
    color: blue; /* Nombre de color */
    font-size: 24px; /* Tamaño en píxeles */
    margin: 10px; /* Margen en píxeles */
    font-family: Arial, sans-serif; /* Fuente Arial */
}

p {
    color: red; /* Nombre de color */
    font-size: 18px; /* Tamaño en píxeles */
```

```
    margin: 5px; /* Margen en píxeles */
    font-family: Verdana, sans-serif; /* Fuente Verdana */
}
```

En los siguientes ejemplos se verán más detalles sobre cómo se pueden utilizar los valores y convenciones para aplicar estilos a los elementos HTML.

111 Border en profundidad

Los bordes son una parte importante de la presentación de un sitio web. Se pueden utilizar para resaltar elementos, crear separaciones entre elementos, etc.

En CSS se pueden definir los bordes de un elemento utilizando la propiedad **border**. La propiedad **border** tiene tres sub-propiedades que se pueden utilizar para definir el ancho, el estilo y el color del borde.

Ejemplo:

```
h1 {
    border: 1px solid black;
}
```

En este ejemplo, se define un borde de 1 píxel de ancho, de estilo sólido y black.

La propiedad **border** también se puede dividir en tres sub-propiedades: **border-width**, **border-style** y **border-color**.

Ejemplo:

```
h1 {
    border-width: 1px;
    border-style: solid;
    border-color: black;
}
```

En este ejemplo, se define un borde de 1 píxel de ancho, de estilo sólido y black.

La propiedad **border** también se puede dividir en cuatro sub-propiedades: **border-top**, **border-right**, **border-bottom** y **border-left**.

Ejemplo:

```
h1 {
    border-top: 1px solid black;
    border-right: 2px dashed red;
    border-bottom: 3px dotted blue;
    border-left: 4px double green;
}
```

En este ejemplo, se definen bordes diferentes para cada lado del elemento.

111.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando bordes.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```css
h1 {
    border-top: 1px solid black;
    border-right: 2px dashed red;
    border-bottom: 3px dotted blue;
    border-left: 4px double green;
}

p {
    border-top: 1px solid black;
    border-right: 2px dashed red;
    border-bottom: 3px dotted blue;
    border-left: 4px double green;
}
```

112 Unidades de medida

Las unidades de medida son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir tamaños, márgenes, rellenos, etc.

En CSS se pueden utilizar diferentes unidades de medida como píxeles, porcentajes, ems, rems, etc.

Ejemplo:

```
h1 {
    font-size: 24px; /* Tamaño en píxeles */
    margin: 10px; /* Margen en píxeles */
    padding: 5%; /* Relleno en porcentaje */
    width: 50em; /* Ancho en ems */
    height: 100%; /* Altura en porcentaje */
}
```

112.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando diferentes unidades de medida.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```
h1 {
    font-size: 24px; /* Tamaño en píxeles */
    margin: 10px; /* Margen en píxeles */
    padding: 5%; /* Relleno en porcentaje */
    width: 50em; /* Ancho en ems */
    height: 100%; /* Altura en porcentaje */
}

p {
    font-size: 18px; /* Tamaño en píxeles */
    margin: 5px; /* Margen en píxeles */
    padding: 2%; /* Relleno en porcentaje */
    width: 30em; /* Ancho en ems */
    height: 50%; /* Altura en porcentaje */
}
```

En este ejemplo, se utilizan diferentes unidades de medida para definir el tamaño de la fuente, el margen, el relleno, el ancho y la altura de un elemento.

113 Background

El Background es una parte importante de la presentación de un sitio web. Se puede utilizar para definir el color de fondo, una imagen de fondo, etc.

En CSS se pueden definir el color de fondo de un elemento utilizando la propiedad **background-color**.

Ejemplo:

```
h1 {
    background-color: yellow;
}
```

En este ejemplo, se define el color de fondo del elemento h1 como amarillo.

113.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando el color de fondo.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```
h1 {
    background-color: yellow;
}

p {
```

```
    background-color: lightblue;
}
```

114 Box model, margin, padding, overflow

El Box Model es una parte importante de la presentación de un sitio web. Se puede utilizar para definir el tamaño, el margen, el relleno, etc. de un elemento.

En CSS se pueden definir el tamaño de un elemento utilizando las propiedades **width** y **height**.

Ejemplo:

```
h1 {
    width: 200px;
    height: 100px;
}
```

En este ejemplo, se define el ancho del elemento h1 como 200 píxeles y la altura como 100 píxeles.

También se pueden definir el margen y el relleno de un elemento utilizando las propiedades **margin** y **padding**.

Ejemplo:

```
h1 {
    margin: 10px;
    padding: 5px;
}
```

En este ejemplo, se define un margen de 10 píxeles y un relleno de 5 píxeles para el elemento h1.

La propiedad **overflow** se utiliza para definir cómo se comporta un elemento cuando su contenido es más grande que su tamaño.

Ejemplo:

```
h1 {
    width: 200px;
    height: 100px;
    overflow: hidden;
}
```

En este ejemplo, se define que el contenido del elemento h1 se ocultará si es más grande que su tamaño.

114.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando el Box Model, el margen, el relleno y el overflow.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
</body>
</html>
```

```css
h1 {
    width: 200px;
    height: 100px;
    margin: 10px;
    padding: 5px;
    overflow: hidden;
}

p {
    width: 300px;
    height: 150px;
    margin: 20px;
    padding: 10px;
    overflow: auto;
}
```

115 Outline, text-align, text-decoration, text-shadow

Outline es una parte importante de la presentación de un sitio web. Se puede utilizar para definir un contorno alrededor de un elemento.

En CSS se puede definir un contorno alrededor de un elemento utilizando la propiedad **outline**.

Ejemplo:

```
h1 {
    outline: 1px solid black;
}
```

En este ejemplo, se define un contorno de 1 píxel de ancho

Otros estilos de contorno son **dotted**, **dashed**, **double**, **groove**, **ridge**, **inset**, **outset**.

Ejemplo:

```
h1 {
    outline: 1px dotted black;
}

h2 {
    outline: 2px dashed red;
}

h3 {
    outline: 3px double blue;
}

h4 {
    outline: 4px groove green;
}

h5 {
    outline: 5px ridge orange;
}

h6 {
    outline: 6px inset purple;
```

```
}

p {
    outline: 7px outset pink;
}
```

En este ejemplo, se definen diferentes estilos de contorno para los elementos h1, h2, h3, h4, h5, h6 y p.

115.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando el contorno.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <h2>Este es un subtítulo</h2>
    <h3>Este es un subtítulo</h3>
    <h4>Este es un subtítulo</h4>
    <h5>Este es un subtítulo</h5>
    <h6>Este es un subtítulo</h6>
    <p>Este es un párrafo</p>
</body>
</html>
```

```
h1 {
    outline: 1px dotted black;
}

h2 {
    outline: 2px dashed red;
}

h3 {
    outline: 3px double blue;
}
```

```
h4 {
    outline: 4px groove green;
}

h5 {
    outline: 5px ridge orange;
}

h6 {
    outline: 6px inset purple;
}

p {
    outline: 7px outset pink;
}
```

116 Fuentes custom

Las fuentes personalizadas son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir una fuente personalizada para los textos.

En CSS se puede definir una fuente personalizada para los textos utilizando la propiedad **font-family**.

Ejemplo:

```
h1 {
    font-family: Arial, sans-serif;
}
```

En este ejemplo, se define la fuente Arial para el elemento h1.

También se pueden utilizar fuentes personalizadas de Google Fonts.

Ejemplo:

```
h1 {
    font-family: 'Roboto', sans-serif;
}
```

En este ejemplo, se define la fuente Roboto de Google Fonts para el elemento h1.

116.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando fuentes personalizadas.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
```

```
</body>
</html>
```

```
h1 {
    font-family: Arial, sans-serif;
}

p {
    font-family: 'Roboto', sans-serif;
}
```

117 Links y sus estados

Los Links son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir el estilo de los enlaces.

En CSS se pueden definir los estilos de los enlaces utilizando los selectores **a**, **a:link**, **a:visited**, **a:hover** y **a:active**.

Ejemplo:

```
a {
    color: blue;
    text-decoration: none;
}

a:link {
    color: blue;
    text-decoration: none;
}

a:visited {
    color: purple;
    text-decoration: none;
}

a:hover {
    color: red;
    text-decoration: underline;
}

a:active {
    color: green;
    text-decoration: underline;
}
```

En este ejemplo, se definen los estilos de los enlaces, los enlaces visitados, los enlaces al pasar el ratón por encima y los enlaces activos.

117.1 Actividad

- Crear un archivo **styles.css**.

- Definir los estilos de los enlaces utilizando los selectores **a**, **a:link**, **a:visited**, **a:hover** y **a:active**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <a href="#">Este es un enlace</a>
</body>
</html>
```

```css
a {
    color: blue;
    text-decoration: none;
}

a:link {
    color: blue;
    text-decoration: none;
}

a:visited {
    color: purple;
    text-decoration: none;
}

a:hover {
    color: red;
    text-decoration: underline;
}

a:active {
    color: green;
    text-decoration: underline;
}
```

118 Listas

Las Listas son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir el estilo de las listas.

En CSS se pueden definir los estilos de las listas utilizando los selectores **ul**, **ol** y **li**.

Ejemplo:

```
ul {
    list-style-type: disc;
}

ol {
    list-style-type: decimal;
}

li {
    color: blue;
}

li:first-child {
    color: red;
}

li:last-child {
    color: green;
}

li:nth-child(2) {
    color: orange;
}
```

En este ejemplo, se definen los estilos de las listas no ordenadas, las listas ordenadas y los elementos de la lista.

118.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de las listas utilizando los selectores **ul**, **ol** y **li**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <ul>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <ol>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ol>
</body>
</html>
```

```css
ul {
    list-style-type: disc;
}

ol {
    list-style-type: decimal;
}

li {
    color: blue;
}

li:first-child {
    color: red;
}

li:last-child {
    color: green;
}

li:nth-child(2) {
    color: orange;
}
```

119 Tablas

Las Tablas son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir el estilo de las tablas.

En CSS se pueden definir los estilos de las tablas utilizando los selectores **table**, **tr**, **th** y **td**.

Ejemplo:

```
table {
    border-collapse: collapse;
}

th, td {
    border: 1px solid black;
}

th {
    background-color: lightgray;
}

td {
    background-color: lightblue;
}
```

En este ejemplo, se definen los estilos de las tablas, las filas, las celdas de encabezado y las celdas de datos.

119.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de las tablas utilizando los selectores **table**, **tr**, **th** y **td**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <table>
        <tr>
            <th>Encabezado 1</th>
            <th>Encabezado 2</th>
        </tr>
        <tr>
            <td>Dato 1</td>
            <td>Dato 2</td>
        </tr>
    </table>
</body>
</html>
```

```css
table {
    border-collapse: collapse;
}

th, td {
    border: 1px solid black;
}

th {
    background-color: lightgray;
}

td {
    background-color: lightblue;
}
```

120 Display, max-width y position

Display es una parte importante de la presentación de un sitio web. Se puede utilizar para definir el tipo de visualización de un elemento.

En CSS se puede definir el tipo de visualización de un elemento utilizando la propiedad **display**.

Ejemplo:

```
h1 {
    display: block;
}

p {
    display: inline;
}

div {
    display: inline-block;
}

span {
    display: none;
}

img {
    display: none;
}

a {
    display: none;
}

ul {
    display: none;
}

ol {
    display: none;
}

li {
```

```
    display: none;
}

table {
    display: none;
}

tr {
    display: none;
}

th {
    display: none;
}

td {
    display: none;
}

form {
    display: none;
}

input {
    display: none;
}

button {
    display: none;
}
```

En este ejemplo, se definen los tipos de visualización de los elementos h1, p, div, span, img, a, ul, ol, li, table, tr, th, td, form, input y button.

La propiedad **max-width** se utiliza para definir el ancho máximo de un elemento.

Ejemplo:

```
h1 {
    max-width: 200px;
}

p {
    max-width: 300px;
}

div {
    max-width: 400px;
}
```

```
span {
    max-width: 500px;
}

img {
    max-width: 600px;
}

a {
    max-width: 700px;
}

ul {
    max-width: 800px;
}

ol {
    max-width: 900px;
}

li {
    max-width: 1000px;
}

table {
    max-width: 1100px;
}

tr {
    max-width: 1200px;
}

th {
    max-width: 1300px;
}

td {
    max-width: 1400px;
}

form {
    max-width: 1500px;
}

input {
    max-width: 1600px;
}
```

```
button {
    max-width: 1700px;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}

center ol {
```

```
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}
```

La propiedad **position** se utiliza para definir la posición de un elemento.

Ejemplo:

```
h1 {
    position: static;
}

p {
    position: relative;
}

div {
    position: absolute;
}

span {
    position: fixed;
}

img {
    position: sticky;
}

a {
    position: static;
}

ul {
```

```
    position: relative;
}

ol {
    position: absolute;
}

li {
    position: fixed;
}

table {
    position: sticky;
}

tr {
    position: static;
}

th {
    position: relative;
}

td {
    position: absolute;
}

form {
    position: fixed;
}

input {
    position: sticky;
}

button {
    position: static;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
```

```
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}

center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}
```

```
center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}

center input {
    display: inline-block;
}

center button {
    display: inline-block;
}
```

En este ejemplo, se definen las posiciones de los elementos h1, p, div, span, img, a, ul, ol, li, table, tr, th, td, form, input y button.

120.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando la propiedad **display**, **max-width** y **position**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Este es un título</h1>
    <p>Este es un párrafo</p>
    <div>Este es un div</div>
    <span>Este es un span</span>
    <img src="https://via.placeholder.com/150" alt="Imagen">
    <a href="#">Este es un enlace</a>
    <ul>
```

```html
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <ol>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ol>
    <table>
        <tr>
            <th>Encabezado 1</th>
            <th>Encabezado 2</th>
        </tr>
        <tr>
            <td>Dato 1</td>
            <td>Dato 2</td>
        </tr>
    </table>
    <form>
        <input type="text" placeholder="Texto">
        <button>Enviar</button>
    </form>
</body>
</html>
```

```css
h1 {
    display: block;
    max-width: 200px;
    position: static;
}

p {
    display: inline;
    max-width: 300px;
    position: relative;
}

div {
    display: inline-block;
    max-width: 400px;
    position: absolute;
}

span {
    display: none;
    max-width: 500px;
    position: fixed;
```

```
}

img {
    display: none;
    max-width: 600px;
    position: sticky;
}

a {
    display: none;
    max-width: 700px;
    position: static;
}

ul {
    display: none;
    max-width: 800px;
    position: relative;
}

ol {
    display: none;
    max-width: 900px;
    position: absolute;
}

li {
    display: none;
    max-width: 1000px;
    position: fixed;
}

table {
    display: none;
    max-width: 1100px;
    position: sticky;
}

tr {
    display: none;
    max-width: 1200px;
    position: static;
}

th {
    display: none;
    max-width: 1300px;
    position: relative;
```

```
}

td {
    display: none;
    max-width: 1400px;
    position: absolute;
}

form {
    display: none;
    max-width: 1500px;
    position: fixed;
}

input {
    display: none;
    max-width: 1600px;
    position: sticky;
}

button {
    display: none;
    max-width: 1700px;
    position: static;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}
```

```
center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}

center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}
```

```
center input {
    display: inline-block;
}

center button {
    display: inline-block;
}
```

121 Float

Float es una parte importante de la presentación de un sitio web. Se puede utilizar para definir la posición de un elemento.

En CSS se puede definir la posición de un elemento utilizando la propiedad **float**.

Ejemplo:

```
img {
    float: left;
}

p {
    float: right;
}

div {
    float: none;
}

span {
    float: none;
}

a {
    float: none;
}

ul {
    float: none;
}

ol {
    float: none;
}

li {
    float: none;
}

table {
    float: none;
```

```
}

tr {
    float: none;
}

th {
    float: none;
}

td {
    float: none;
}

form {
    float: none;
}

input {
    float: none;
}

button {
    float: none;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}
```

En este ejemplo, se definen las posiciones de los elementos img, p, div, span, a, ul, ol, li, table, tr, th, td, form, input y button.

121.1 Actividad

- Crear un archivo **styles.css**.

- Definir los estilos de los elementos HTML utilizando la propiedad **float**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <img src="https://via.placeholder.com/150" alt="Imagen">
    <p>Este es un párrafo</p>
    <div>Este es un div</div>
    <span>Este es un span</span>
    <a href="#">Este es un enlace</a>
    <ul>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <ol>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ol>
    <table>
        <tr>
            <th>Encabezado 1</th>
            <th>Encabezado 2</th>
        </tr>
        <tr>
            <td>Dato 1</td>
            <td>Dato 2</td>
        </tr>
    </table>
    <form>
        <input type="text" placeholder="Texto">
        <button>Enviar</button>
    </form>
</body>
</html>
```

```css
img {
    float: left;
}

p {
```

```
    float: right;
}

div {
    float: none;
}

span {
    float: none;
}

a {
    float: none;
}

ul {
    float: none;
}

ol {
    float: none;
}

li {
    float: none;
}

table {
    float: none;
}

tr {
    float: none;
}

th {
    float: none;
}

td {
    float: none;
}

form {
    float: none;
}

input {
```

```
    float: none;
}

button {
    float: none;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}
```

```
center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}

center input {
    display: inline-block;
}

center button {
    display: inline-block;
}
```

122 inline-block

Inline-block es una parte importante de la presentación de un sitio web. Se puede utilizar para definir la posición de un elemento.

En CSS se puede definir la posición de un elemento utilizando la propiedad **display** con el valor **inline-block**.

Ejemplo:

```
img {
    display: inline-block;
}

p {
    display: inline-block;
}

div {
    display: inline-block;
}

span {
    display: inline-block;
}

a {
    display: inline-block;
}

ul {
    display: inline-block;
}

ol {
    display: inline-block;
}

li {
    display: inline-block;
}

table {
```

```
    display: inline-block;
}

tr {
    display: inline-block;
}

th {
    display: inline-block;
}

td {
    display: inline-block;
}

form {
    display: inline-block;
}

input {
    display: inline-block;
}

button {
    display: inline-block;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}
```

En este ejemplo, se definen las posiciones de los elementos img, p, div, span, a, ul, ol, li, table, tr, th, td, form, input y button.

122.1 Actividad

- Crear un archivo **styles.css**.

- Definir los estilos de los elementos HTML utilizando la propiedad **display** con el valor **inline-block**.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <img src="https://via.placeholder.com/150" alt="Imagen">
    <p>Este es un párrafo</p>
    <div>Este es un div</div>
    <span>Este es un span</span>
    <a href="#">Este es un enlace</a>
    <ul>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <ol>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ol>
    <table>
        <tr>
            <th>Encabezado 1</th>
            <th>Encabezado 2</th>
        </tr>
        <tr>
            <td>Dato 1</td>
            <td>Dato 2</td>
        </tr>
    </table>
    <form>
        <input type="text" placeholder="Texto">
        <button>Enviar</button>
    </form>
</body>
</html>
```

```css
img {
    display: inline-block;
}
```

```
p {
    display: inline-block;
}

div {
    display: inline-block;
}

span {
    display: inline-block;
}

a {
    display: inline-block;
}

ul {
    display: inline-block;
}

ol {
    display: inline-block;
}

li {
    display: inline-block;
}

table {
    display: inline-block;
}

tr {
    display: inline-block;
}

th {
    display: inline-block;
}

td {
    display: inline-block;
}

form {
    display: inline-block;
}
```

```
input {
    display: inline-block;
}

button {
    display: inline-block;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
```

```
}

center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}

center input {
    display: inline-block;
}

center button {
    display: inline-block;
}
```

123 Centrar un elemento

Centrar un elemento es una parte importante de la presentación de un sitio web. Se puede utilizar para definir la posición de un elemento.

En CSS se puede definir la posición de un elemento utilizando la propiedad **margin** con los valores **auto** y **0**.

Ejemplo:

```
img {
    display: block;
    margin: 0 auto;
}

p {
    display: block;
    margin: 0 auto;
}

div {
    display: block;
    margin: 0 auto;
}

span {
    display: block;
    margin: 0 auto;
}

a {
    display: block;
    margin: 0 auto;
}

ul {
    display: block;
    margin: 0 auto;
}

ol {
    display: block;
    margin: 0 auto;
```

```
}

li {
    display: block;
    margin: 0 auto;
}

table {
    display: block;
    margin: 0 auto;
}

tr {
    display: block;
    margin: 0 auto;
}

th {
    display: block;
    margin: 0 auto;
}

td {
    display: block;
    margin: 0 auto;
}

form {
    display: block;
    margin: 0 auto;
}

input {
    display: block;
    margin: 0 auto;
}

button {
    display: block;
    margin: 0 auto;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
```

```
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}

center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}
```

```
center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}

center input {
    display: inline-block;
}

center button {
    display: inline-block;
}

center clear {
    clear: both;
}

center clearfix {
    overflow: hidden;
}

center clearfix::after {
    content: "";
    display: table;
    clear: both;
}
```

En este ejemplo, se definen las posiciones de los elementos img, p, div, span, a, ul, ol, li, table, tr, th, td, form, input y button.

123.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML utilizando la propiedad **margin** con los valores **auto** y **0**.

- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <img src="https://via.placeholder.com/150" alt="Imagen">
    <p>Este es un párrafo</p>
    <div>Este es un div</div>
    <span>Este es un span</span>
    <a href="#">Este es un enlace</a>
    <ul>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <ol>
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ol>
    <table>
        <tr>
            <th>Encabezado 1</th>
            <th>Encabezado 2</th>
        </tr>
        <tr>
            <td>Dato 1</td>
            <td>Dato 2</td>
        </tr>
    </table>
    <form>
        <input type="text" placeholder="Texto">
        <button>Enviar</button>
    </form>
</body>
</html>
```

```css
img {
    display: block;
    margin: 0 auto;
}

p {
```

```
    display: block;
    margin: 0 auto;
}

div {
    display: block;
    margin: 0 auto;
}

span {
    display: block;
    margin: 0 auto;
}

a {
    display: block;
    margin: 0 auto;
}

ul {
    display: block;
    margin: 0 auto;
}

ol {
    display: block;
    margin: 0 auto;
}

li {
    display: block;
    margin: 0 auto;
}

table {
    display: block;
    margin: 0 auto;
}

tr {
    display: block;
    margin: 0 auto;
}

th {
    display: block;
    margin: 0 auto;
}
```

```
td {
    display: block;
    margin: 0 auto;
}

form {
    display: block;
    margin: 0 auto;
}

input {
    display: block;
    margin: 0 auto;
}

button {
    display: block;
    margin: 0 auto;
}

clear {
    clear: both;
}

clearfix {
    overflow: hidden;
}

clearfix::after {
    content: "";
    display: table;
    clear: both;
}

center {
    text-align: center;
}

center img {
    display: inline-block;
}

center p {
    display: inline-block;
}

center div {
```

```
    display: inline-block;
}

center span {
    display: inline-block;
}

center a {
    display: inline-block;
}

center ul {
    display: inline-block;
}

center ol {
    display: inline-block;
}

center li {
    display: inline-block;
}

center table {
    display: inline-block;
}

center tr {
    display: inline-block;
}

center th {
    display: inline-block;
}

center td {
    display: inline-block;
}

center form {
    display: inline-block;
}

center input {
    display: inline-block;
}

center button {
```

```
    display: inline-block;
}

center clear {
    clear: both;
}

center clearfix {
    overflow: hidden;
}

center clearfix::after {
    content: "";
    display: table;
    clear: both;
}
```

124 Proyecto: Danto estilo a un portafolio personal de HTML

En el presente proyecto se aplicarán los conocimientos adquiridos en este capítulo para dar estilo a un portafolio personal de HTML.

El portafolio personal de HTML se compone de las siguientes secciones:

1. **Header**: Contiene el nombre del autor y una imagen de perfil.
2. **About**: Contiene una breve descripción del autor.
3. **Skills**: Contiene una lista de habilidades del autor.
4. **Projects**: Contiene una lista de proyectos del autor.
5. **Contact**: Contiene un formulario de contacto.

En el archivo **index.html** se encuentra la estructura del portafolio personal de HTML.

Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
```

```html
<body>
    <header>
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil">
    </header>
    <section id="about">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje"></textarea>
            <button type="submit">Enviar</button>
        </form>
    </section>
</body>
</html>
```

En el archivo **styles.css** se deben definir los estilos de los elementos HTML.

Ejemplo:

```css
header {
    text-align: center;
}
```

```
header h1 {
    color: blue;
}

header img {
    border-radius: 50%;
}

section {
    margin: 20px;
}

section h2 {
    color: red;
}

section p {
    color: green;
}

section ul {
    list-style-type: disc;
}

section li {
    color: orange;
}

form {
    margin: 20px;
}

form label {
    color: purple;
}

form input, form textarea {
    width: 100%;
    margin: 10px 0;
}

form button {
    background-color: pink;
    color: white;
    padding: 10px;
    border: none;
    cursor: pointer;
}
```

En este ejemplo, se definen los estilos de los elementos HTML del portafolio personal de HTML.

124.1 Actividad

- Crear un archivo **styles.css**.
- Definir los estilos de los elementos HTML del portafolio personal de HTML.
- Enlazar el archivo **styles.css** al documento HTML.

Ver respuesta

```html
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <header>
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil">
    </header>
    <section id="about">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre">
```

```
            <label for="email">Email:</label>
            <input type="email" id="email" name="email">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje"></textarea>
            <button type="submit">Enviar</button>
        </form>
    </section>
</body>
</html>
```

```
header {
    text-align: center;
}

header h1 {
    color: blue;
}

header img {
    border-radius: 50%;
}

section {
    margin: 20px;
}

section h2 {
    color: red;
}

section p {
    color: green;
}

section ul {
    list-style-type: disc;
}

section li {
    color: orange;
}

form {
    margin: 20px;
}

form label {
    color: purple;
```

```
}

form input, form textarea {
    width: 100%;
    margin: 10px 0;
}

form button {
    background-color: pink;
    color: white;
    padding: 10px;
    border: none;
    cursor: pointer;
}
```

125 Frameworks de CSS

Los Frameworks de CSS son una parte importante de la presentación de un sitio web. Se pueden utilizar para definir los estilos de los elementos HTML de forma rápida y sencilla.

Algunos de los Frameworks de CSS más populares son:

1. **Bootstrap**: Es un Framework de CSS que permite crear sitios web responsivos y móviles de forma rápida y sencilla.
2. **Tailwind CSS**: Es un Framework de CSS que permite crear sitios web responsivos y móviles de forma rápida y sencilla.
3. **Bulma**: Es un Framework de CSS que permite crear sitios web responsivos y móviles de forma rápida y sencilla.

126 Proyecto: Danto estilo a un portafolio personal de HTML con Bootstrap,

En el presente proyecto se aplicarán los conocimientos adquiridos en este capítulo para dar estilo a un portafolio personal de HTML con Bootstrap.

El portafolio personal de HTML con Bootstrap se compone de las siguientes secciones:

1. **Header**: Contiene el nombre del autor y una imagen de perfil.
2. **About**: Contiene una breve descripción del autor.
3. **Skills**: Contiene una lista de habilidades del autor.
4. **Projects**: Contiene una lista de proyectos del autor.
5. **Contact**: Contiene un formulario de contacto.

En el archivo **index.html** se encuentra la estructura del portafolio personal de HTML con Bootstrap.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
</head>
<body>
    <header class="text-center">
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-circle">
    </header>
    <section id="about" class="container">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="form-control">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" class="form-control">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="form-control"></textarea>
            <button type="submit" class="btn btn-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

En este ejemplo, se utiliza Bootstrap para dar estilo al portafolio personal de HTML.

126.1 Actividad

- Crear un archivo **index.html**.
- Definir la estructura del portafolio personal de HTML con Bootstrap.
- Enlazar el archivo **index.html** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" 
</head>
<body>
    <header class="text-center">
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-circle">
    </header>
    <section id="about" class="container">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="form-control">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" class="form-control">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="form-control"></textarea>
```

```
            <button type="submit" class="btn btn-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

127 Proyecto: Danto estilo a un portafolio personal de HTML con Tailwind CSS

En el presente proyecto se aplicarán los conocimientos adquiridos en este capítulo para dar estilo a un portafolio personal de HTML con Tailwind CSS.

El portafolio personal de HTML con Tailwind CSS se compone de las siguientes secciones:

1. **Header**: Contiene el nombre del autor y una imagen de perfil.
2. **About**: Contiene una breve descripción del autor.
3. **Skills**: Contiene una lista de habilidades del autor.
4. **Projects**: Contiene una lista de proyectos del autor.
5. **Contact**: Contiene un formulario de contacto.

En el archivo **index.html** se encuentra la estructura del portafolio personal de HTML con Tailwind CSS.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
      <script src="https://cdn.tailwindcss.com"></script>
</head>
<body>
    <header class="text-center">
        <h1 class="text-2xl font-bold">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-full w-24 h-24">
    </header>
    <section id="about" class="container mx-auto">
        <h2 class="text-xl font-bold">Acerca de mí</h2>
        <p class="text-lg">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container mx-auto">
        <h2 class="text-xl font-bold">Habilidades</h2>
        <ul class="list-disc">
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container mx-auto">
        <h2 class="text-xl font-bold">Proyectos</h2>
        <ul class="list-disc">
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container mx-auto">
        <h2 class="text-xl font-bold">Contacto</h2>
        <form>
            <label for="nombre" class="block">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="w-full p-2 border border-
            <label for="email" class="block">Email:</label>
            <input type="email" id="email" name="email" class="w-full p-2 border border-g
            <label for="mensaje" class="block">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="w-full p-2 border border-gray-30
            <button type="submit" class="w-full p-2 bg-blue-500 text-white rounded">Envia
        </form>
    </section>
</body>
</html>
```

En este ejemplo, se utiliza Tailwind CSS para dar estilo al portafolio personal de HTML.

127.1 Actividad

- Crear un archivo **index.html**.
- Definir la estructura del portafolio personal de HTML con Tailwind CSS.
- Enlazar el archivo **index.html** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
      <script src="https://cdn.tailwindcss.com"></script>
</head>
<body>
    <header class="text-center">
        <h1 class="text-2xl font-bold">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-full w-24 h-24">
    </header>
    <section id="about" class="container mx-auto">
        <h2 class="text-xl font-bold">Acerca de mí</h2>
        <p class="text-lg">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container mx-auto">
        <h2 class="text-xl font-bold">Habilidades</h2>
        <ul class="list-disc">
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container mx-auto">
        <h2 class="text-xl font-bold">Proyectos</h2>
        <ul class="list-disc">
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container mx-auto">
        <h2 class="text-xl font-bold">Contacto</h2>
        <form>
            <label for="nombre" class="block">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="w-full p-2 border border-
            <label for="email" class="block">Email:</label>
            <input type="email" id="email" name="email" class="w-full p-2 border border-g
            <label for="mensaje" class="block">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="w-full p-2 border border-gray-30
```

```
            <button type="submit" class="w-full p-2 bg-blue-500 text-white rounded">Envia
        </form>
    </section>
</body>
</html>
```

128 Proyecto: Danto estilo a un portafolio personal de HTML con Bulma.

En el presente proyecto se aplicarán los conocimientos adquiridos en este capítulo para dar estilo a un portafolio personal de HTML con Bulma.

El portafolio personal de HTML con Bulma se compone de las siguientes secciones:

1. **Header**: Contiene el nombre del autor y una imagen de perfil.
2. **About**: Contiene una breve descripción del autor.
3. **Skills**: Contiene una lista de habilidades del autor.
4. **Projects**: Contiene una lista de proyectos del autor.
5. **Contact**: Contiene un formulario de contacto.

En el archivo **index.html** se encuentra la estructura del portafolio personal de HTML con Bulma.

Ejemplo:

```html
<!DOCTYPE html>
<html>
<head>
    <link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bulma@1.0.2/css/bulma.min.css"
>
</head>
<body>
    <header class="has-text-centered">
        <h1 class="title is-2">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="is-rounded is-128x128">
    </header>
    <section id="about" class="container">
        <h2 class="title is-4">Acerca de mí</h2>
        <p class="subtitle is-5">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2 class="title is-4">Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container">
        <h2 class="title is-4">Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2 class="title is-4">Contacto</h2>
        <form>
            <label for="nombre" class="label">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="input">
            <label for="email" class="label">Email:</label>
            <input type="email" id="email" name="email" class="input">
            <label for="mensaje" class="label">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="textarea"></textarea>
            <button type="submit" class="button is-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

En este ejemplo, se utiliza Bulma para dar estilo al portafolio personal de HTML.

128.1 Actividad

- Crear un archivo **index.html**.
- Definir la estructura del portafolio personal de HTML con Bulma.
- Enlazar el archivo **index.html** al documento HTML.

Ver respuesta

```
<!DOCTYPE html>
<html>
<head>
    <link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bulma@1.0.2/css/bulma.min.css"
>
</head>
<body>
    <header class="has-text-centered">
        <h1 class="title is-2">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="is-rounded is-128x128">
    </header>
    <section id="about" class="container">
        <h2 class="title is-4">Acerca de mí</h2>
        <p class="subtitle is-5">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2 class="title is-4">Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container">
        <h2 class="title is-4">Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2 class="title is-4">Contacto</h2>
        <form>
```

```
            <label for="nombre" class="label">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="input">
            <label for="email" class="label">Email:</label>
            <input type="email" id="email" name="email" class="input">
            <label for="mensaje" class="label">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="textarea"></textarea>
            <button type="submit" class="button is-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

129 Reto

Crea un portafolio personal de HTML con CSS utilizando los conocimientos adquiridos en este capítulo, puede ser con CSS de forma nativa o con un Framework de CSS como Bootstrap, Tailwind CSS o Bulma.

El portafolio personal de HTML debe contener las siguientes secciones:

1. **Header**: Contiene el nombre del autor y una imagen de perfil.
2. **About**: Contiene una breve descripción del autor.
3. **Skills**: Contiene una lista de habilidades del autor.
4. **Projects**: Contiene una lista de proyectos del autor.
5. **Contact**: Contiene un formulario de contacto.

Ver respuesta

- CSS Nativo

```
<!DOCTYPE html>
<html>
<head>
    <style>
        header {
            text-align: center;
        }

        header h1 {
            color: blue;
        }

        header img {
            border-radius: 50%;
        }

        section {
            margin: 20px;
        }

        section h2 {
            color: red;
        }
```

```
        section p {
            color: green;
        }

        section ul {
            list-style-type: disc;
        }

        section li {
            color: orange;
        }

        form {
            margin: 20px;
        }

        form label {
            color: purple;
        }

        form input, form textarea {
            width: 100%;
            margin: 10px 0;
        }

        form button {
            background-color: pink;
            color: white;
            padding: 10px;
            border: none;
            cursor: pointer;
        }
    </style>
</head>
<body>
    <header>
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil">
    </header>
    <section id="about">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
```

```
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje"></textarea>
            <button type="submit">Enviar</button>
        </form>
    </section>
</body>
</html>
```

- Bootstrap

```
<!DOCTYPE html>
<html>
<head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
</head>
<body>
    <header class="text-center">
        <h1>Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-circle">
    </header>
    <section id="about" class="container">
        <h2>Acerca de mí</h2>
        <p>Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2>Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
```

```
        </ul>
    </section>
    <section id="projects" class="container">
        <h2>Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2>Contacto</h2>
        <form>
            <label for="nombre">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="form-control">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" class="form-control">
            <label for="mensaje">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="form-control"></textarea>
            <button type="submit" class="btn btn-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

- Tailwind CSS

```
<!DOCTYPE html>
<html>
<head>
      <script src="https://cdn.tailwindcss.com"></script>
</head>
<body>
    <header class="text-center">
        <h1 class="text-2xl font-bold">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="rounded-full w-24 h-24">
    </header>
    <section id="about" class="container mx-auto">
        <h2 class="text-xl font-bold">Acerca de mí</h2>
        <p class="text-lg">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container mx-auto">
        <h2 class="text-xl font-bold">Habilidades</h2>
        <ul class="list-disc">
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
            <li>Habilidad 3</li>
        </ul>
```

```
    </section>
    <section id="projects" class="container mx-auto">
        <h2 class="text-xl font-bold">Proyectos</h2>
        <ul class="list-disc">
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container mx-auto">
        <h2 class="text-xl font-bold">Contacto</h2>
        <form>
            <label for="nombre" class="block">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="w-full p-2 border border-
            <label for="email" class="block">Email:</label>
            <input type="email" id="email" name="email" class="w-full p-2 border border-g
            <label for="mensaje" class="block">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="w-full p-2 border border-gray-30
            <button type="submit" class="w-full p-2 bg-blue-500 text-white rounded">Envia
        </form>
    </section>
</body>
</html>
```

- Bulma

```
<!DOCTYPE html>
<html>
<head>
    <link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bulma@1.0.2/css/bulma.min.css"
>
</head>
<body>
    <header class="has-text-centered">
        <h1 class="title is-2">Nombre del autor</h1>
        <img src="imagen.jpg" alt="Imagen de perfil" class="is-rounded is-128x128">
    </header>
    <section id="about" class="container">
        <h2 class="title is-4">Acerca de mí</h2>
        <p class="subtitle is-5">Breve descripción del autor</p>
    </section>
    <section id="skills" class="container">
        <h2 class="title is-4">Habilidades</h2>
        <ul>
            <li>Habilidad 1</li>
            <li>Habilidad 2</li>
```

```html
            <li>Habilidad 3</li>
        </ul>
    </section>
    <section id="projects" class="container">
        <h2 class="title is-4">Proyectos</h2>
        <ul>
            <li>Proyecto 1</li>
            <li>Proyecto 2</li>
            <li>Proyecto 3</li>
        </ul>
    </section>
    <section id="contact" class="container">
        <h2 class="title is-4">Contacto</h2>
        <form>
            <label for="nombre" class="label">Nombre:</label>
            <input type="text" id="nombre" name="nombre" class="input">
            <label for="email" class="label">Email:</label>
            <input type="email" id="email" name="email" class="input">
            <label for="mensaje" class="label">Mensaje:</label>
            <textarea id="mensaje" name="mensaje" class="textarea"></textarea>
            <button type="submit" class="button is-primary">Enviar</button>
        </form>
    </section>
</body>
</html>
```

130 Conclusiones

En este capítulo se aprendió sobre CSS, que es un lenguaje de estilo utilizado para dar estilo a los elementos HTML de un sitio web.

Se aprendió sobre las propiedades de CSS, que se utilizan para definir el estilo de los elementos HTML, como el color de fondo, el tamaño, el margen, el relleno, etc.

Se aprendió sobre el Box Model, que es una parte importante de la presentación de un sitio web, y se puede utilizar para definir el tamaño, el margen, el relleno, etc. de un elemento.

Se aprendió sobre el Outline, que se utiliza para definir un contorno alrededor de un elemento.

Se aprendió sobre las fuentes personalizadas, que se pueden utilizar para definir una fuente personalizada para los textos.

Se aprendió sobre los Links y sus estados, que se pueden utilizar para definir el estilo de los enlaces.

Se aprendió sobre las Listas, que se pueden utilizar para definir el estilo de las listas.

Se aprendió sobre las Tablas, que se pueden utilizar para definir el estilo de las tablas.

Se aprendió sobre el Display, que se puede utilizar para definir el tipo de visualización de un elemento.

Se aprendió sobre el Float, que se puede utilizar para definir la posición de un elemento.

Se aprendió sobre el Inline-block, que se puede utilizar para definir la posición de un elemento.

Se aprendió sobre cómo centrar un elemento, que se puede utilizar para definir la posición de un elemento.

Se aprendió sobre los Frameworks de CSS, que se pueden utilizar para definir los estilos de los elementos HTML de forma rápida y sencilla.

Se aplicaron los conocimientos adquiridos en este capítulo para dar estilo a un portafolio personal de HTML con CSS, Bootstrap, Tailwind CSS y Bulma.

131 Recursos

- CSS Tutorial
- CSS Reference
- Bootstrap
- Tailwind CSS
- Bulma

132 JavaScript

Esté capítulo está basado en el Curso de **"Aprende JavaScript"** en la url Aprende-JavaScript.dev creado por Miguel Ángel Midudev, el cual es un desarrollador FullStack y Creador de Contenido, sin más que agregar comencemos.

132.1 ¿Qué es JavaScript?

JavaScript es uno de los lenguajes de programación más usados y populares del mundo. Nació en 1995 para darle interactividad a las páginas web y desde entonces ha evolucionado

hasta convertirse en un lenguaje de programación de propósito general. Dicho de otra forma: se puede usar casi para cualquier cosa.

132.2 ¿Qué es programar?

Es el acto de construir un programa o conjunto de instrucciones para decirle a una computadora qué y cómo queremos que haga algo. No es diferente a cuando "programamos" la lavadora, sólo que en vez de pulsar un botón, vamos a usar texto. A este texto se le conoce como "código".

132.3 ¿Por qué aprender JavaScript?

JavaScript es, a día de hoy, el único lenguaje de programación que todos los navegadores web entienden sin necesidad de realizar ningún paso previo. Esto hace que casi cualquier página web que visitas tiene alguna línea de JavaScript en su interior.

Su curva de aprendizaje para iniciarte es muy corta ya que en muy poco tiempo puedes empezar a hacer cosas interesantes. Para alcanzar a ser un buen programador en JavaScript necesitarás años de práctica, pero para empezar a hacer cosas interesantes bastará con poco tiempo.

Por si fuera poco, JavaScript es uno de los lenguajes de programación más demandados en el mercado laboral. Es normal, ya que es un lenguaje muy versátil y que se puede usar para casi cualquier cosa.

Además, con JavaScript vas a poder desarrollar casi cualquier cosa que te propongas. Desde aplicaciones web, móviles y de escritorio a backend, videojuegos, inteligencia artificial, Internet de las cosas. Todo un mundo de posibilidades con un sólo lenguaje.

132.4 Quiz

¿JavaScript sólo se puede usar para crear páginas web?

Verdadero

Falso

¿Qué es programar en el mundo del software?

Crear páginas web y que el usuario pueda usarlas

Es el acto de construir un programa o conjunto de instrucciones para decirle a una comptuadora qué y cómo queremos que haga algo

Cualquier cosa que hagamos en un ordenador

133 La consola del navegador

Ahora es necesario utilizar un navegador web, ya que vamos a empezar a escribir código JavaScript.

Todos los navegadores tienen una herramienta llamada **consola**. La consola nos permite ejecutar código JavaScript en tiempo real y ver el resultado. También ahí podemos ver los errores, advertencias y trazas que se producen en nuestro código.

Aunque podemos ejecutar código JavaScript de otras formas, para empezar, vamos a usar esta herramienta que nos ofrece el navegador.

> Tip
>
> Recuerda que llamamos código a las instrucciones que le damos a la computadora para que haga algo.

133.1 Cómo abrir la consola

Para abrir la consola del navegador debes hacer lo siguiente:

1. Abre tu navegador web favorito, por ejemplo, Google Chrome, Firefox, Safari o Edge.
2. Ve a la página **about:blank** en la barra de direcciones. Así evitaremos que la página web que hemos cargado nos moleste.
3. Haz **click derecho** en cualquier parte de la página y selecciona la opción **Inspeccionar Elemento** o **Inspect**.

Aquí podrás ejecutar tu código JavaScript y ver el resultado. Es lo que usaremos en un inicio. Más adelante pasaremos a usar un editor como **VSCode**.

En **Google Chrome** o **Brave**, este es el aspecto de la consola

En **Firefox**, este es el aspecto de la consola

133.2 Quiz

¿Para qué sirve la consola del navegador?

La consola sirve para poder jugar con JavaScript. Es la única forma de poder ejecutar nuestro código y por eso todos los navegadores lo incorporan.

La consola nos permite ejecutar código JavaScript en tiempo real y ver el resultado. También ahí podemos ver los errores, advertencias y trazas que se producen en nuestro código.

134 Editores y entornos de desarrollo

En la actualidad existen muchos editores de código y entornos de desarrollo que nos permiten escribir código de forma más cómoda y eficiente. Algunos de los más populares son:

- **Visual Studio Code**: Es uno de los editores de código más populares y es el que usaremos en este curso.

- **Runjs**: Un entorno de desarrollo online que nos permite escribir código JavaScript y ver el resultado en tiempo real.

- **Zed Code Editor**: Un editor de código muy sencillo y minimalista.

135 Tipos de Datos

En JavaScript, como cualquier otro lenguaje de programación, vamos a querer trasladar la información del mundo real a nuestro código. Para ello, necesitamos entender qué tipos de datos existen en el lenguaje para poder representar la información que necesitamos.

En JavaScript tenemos 9 tipos de datos que se dividen en dos grandes grupos: **primitivos** o **no primitivos**.

135.1 Tipos Primitivos.

Dentro de los tipos primitivos tenemos 7 tipos de datos:

- number
- string
- boolean
- null
- undefined
- symbol
- bigint

No te preocupes, no tienes ni recordarlos ni entenderlos todos ahora. En esta lección vamos a ver los tres primeros que son, sin ninguna duda, los que más vamos a usar al inicio del curso.

135.2 Números

Los números (tipo number) son los datos más básicos que podemos representar en JavaScript. En JavaScript, no hay una diferencia entre números enteros y números decimales, todos los números son de tipo number:

```
7
3.14
19.95
2.998e8
-1
```

135.3 Operadores aritméticos

Con los números, puedes usar los operadores aritméticos para realizar operaciones matemáticas. En JavaScript tenemos los siguientes operadores aritméticos:

- +: suma
- -: resta
- *: multiplicación
- /: división
- %: módulo (resto de la división)
- **: exponente

Al usar los operadores aritméticos, el resultado siempre será un número. Por ejemplo:

```
2 + 2 // 4
4 - 2 // 2
3 * 2 // 6
2 / 2 // 1
2 % 2 // 0
3 ** 3 // 27
```

135.4 ¿Qué significa el // que ves en los ejemplos?

Es un comentario. En JavaScript, los comentarios se escriben con // y todo lo que escribas después de'// será ignorado por el navegador. Los comentarios son muy útiles para explicar qué hace nuestro código.

Al igual que las matemáticas, las operaciones siguen un orden de precedencia. Por ejemplo, si queremos calcular el resultado de **2 + 2 * 3**, primero se multiplicará **2 * 3** y luego se sumará **2 + 6**. El resultado será **8**.

También puedes usar paréntesis para cambiar el orden de las operaciones.

```
2 + 2 * 3 // 8
(2 + 2) * 3 // 12
```

135.5 Cadenas de texto

La cadena de texto (tipo string) es otro tipo de dato muy común. En JavaScript, las cadenas de texto se representan entre **comillas simples**, **dobles** o **acentos graves**:

```
'Estás aprendiendo JavaScript'

"JavaScript te va a gustar"

`Esto es una cadena de texto`
```

Las comillas simples y dobles funcionan igual, pero al usar **acentos graves** podemos escribir cadenas de texto que ocupen varias líneas:

```
`Esto es una cadena de texto
que ocupa varias líneas. Puedes escribir
tantas líneas como quieras`
```

135.6 Concatenación

Para unir dos cadenas de texto, podemos usar el operador +:

```
'Estás aprendiendo ' + 'JavaScript' // 'Estás aprendiendo JavaScript'
```

Como ves, el operador + de concatenación de cadenas de texto es visualmente el mismo que el operador + de **suma de números**. El operador + funciona de forma diferente dependiendo del tipo de dato que estemos usando.

135.7 Booleanos

Los booleanos representa sólo dos valores: true (verdadero) o false (falso). Por ejemplo:

- ¿La luz está encendida (true) o apagada (false)?
- ¿Está lloviendo (true) o no está lloviendo (false)?
- ¿Está el usuario logueado (true) o no está logueado (false)?

Estos son ejemplos de preguntas que podemos responder con un valor booleano.

- true
- false

135.8 Quiz

¿Cuántos tipos de datos existen en JavaScript?

Sólo existen 2 tipos de datos

9 tipos de datos que se dividen en dos grandes grupos

¿Cuántos valores se pueden representar en el tipo de dato Boolean?

true y false

Cualquier valor que se te ocurra

Números, cadenas de texto y verdadero/falso.

El operador + se utiliza para...

Sólo sirve para sumar dos números

Concatenar dos textos

Depende del tipo de dato que se esté utilizando

136 Los operadores de comparación

Los operadores de comparación en JavaScript nos permiten comparar dos valores. Siempre devuelven un valor booleano (true o false).

Por ejemplo, podemos comparar si un número es mayor que otro con el operador >, o si un número es menor que otro con el operador <.

```
5 > 3 // true
5 < 3 // false
```

También tenemos los operadores >= y <= que nos permiten comparar si un número es mayor o igual que otro, o si un número es menor o igual que otro.

```
5 >= 3 // true
5 >= 5 // true
5 <= 3 // false
5 <= 5 // true
```

Para saber si dos valores son iguales podemos usar el operador === o, para saber si son diferentes, el operador !==.

```
5 === 5 // true
5 !== 5 // false
```

136.1 Actividad

1. Escribe un código que compruebe si 10 es mayor o igual que 9
2. Comprueba que 0 es igual a 0

Respuesta

R1.

```
10 >= 9 // true
```

R2.

```
0 === 0 // true
```

136.2 Comparando cadenas de texto

No sólo podemos usar los comparadores para comparar números, también podemos usarlos para comparar cadenas de texto y otros tipos de datos.

```
'JavaScript' === 'JavaScript' // true
'JavaScript' === 'Java' // false
"JavaScript" !== 'PHP' // true
`Estoy Aprendiendo JavaScript` === 'Estoy Aprendiendo JavaScript' // true
```

> **Tip**
>
> Fíjate que puedes comparar cadenas de texto que usan comillas simples, dobles o acentos graves. Al final, siguen siendo cadenas de texto y lo importante es que sean iguales.

136.3 ¿Y si usamos el operador > con cadenas de texto?

Aunque no es muy común, podemos usar los operadores > , >=, < y <= para comparar cadenas de texto.

JavaScript comparará las cadenas de texto según el valor de su código Unicode.

Por ejemplo, la letra **A** tiene un valor de **65** y la letra **B** tiene un valor de **66**. Por lo tanto, la letra **A** es menor que la letra **B**. Pero ten cuidado, ya que las letras mayúsculas tienen un valor menor que las letras minúsculas.

```
'Alfa' > 'Beta' // false
'Omega' > 'Beta' // true
'alfa' > 'Alfa' // true
```

136.4 Comparando booleanos

También podemos comparar booleanos con los operadores de comparación.

```
true === true // true
true === false // false
false !== false // false
```

Por si te preguntas cómo se comportan los operadores de mayor o menor que (> y <) con booleanos te diré que, aunque no tiene sentido, debes saber que true es mayor que false.

```
true > false // true
false < true // true
true > true // false
false < false // false
```

136.5 Comparando valores de diferentes tipos

Por ahora hemos visto cómo comparar valores de un mismo tipo, pero ¿Qué pasa si queremos comparar valores de diferentes tipos? En JavaScript es algo que es posible pero no suele ser recomendable.

En clases posteriores veremos cómo funciona este tipo de mecanismo y explicaremos el operador de igualdad débil == y el operador de desigualdad débil **!=** pero, por ahora, nos limitaremos a utilizar siempre el operador de igualdad estricta === y el operador de desigualdad estricta **!==**.

136.6 Quiz

¿Cuál es el resultado de 24 > 12?

true

false

Los operadores de compración solo se pueden usar con números

Sí, sólo ahí tiene sentido

No, se pueden usar con otros tipos de datos como cadenas de textos o boleanos

137 Operadores lógicos en JavaScript

Los operadores lógicos en JavaScript (y en muchos otros lenguajes de programación) se utilizan para evaluar expresiones lógicas.

En JavaScript, hay tres operadores lógicos: AND (&&), OR (||) y NOT (!).

137.1 Operador lógico AND &&

El operador lógico AND se indica con &&. Devuelve true cuando ambos valores que conecta son true.

```
true && true // → true
true && false // → false
false && false // → false
```

Un ejemplo en la vida real sería preguntar. **¿Están todas las luces encendidas?** Se comprueba cada luz y si todas están encendidas (true), entonces la respuesta es **true**. Si alguna luz está apagada (false), entonces la respuesta es **false**.

137.2 Operador lógico OR ||

El operador lógico OR se indica con || y devuelve true cuando cualquiera de los valores que conecta es true.

```
true || true // → true
true || false // → true
false || false // → false
```

Un ejemplo en la vida real sería preguntar. **¿Hay alguna luz encendida?** Se comprueba cada luz y si alguna está encendida (true), entonces la respuesta es **true**. Sólo si todas las luces están apagadas (false), entonces la respuesta es **false**.

137.3 Operador lógico NOT !

El operador lógico NOT se indica con ! e invierte el valor de un valor booleano. Se pone delante del valor que queremos invertir.

```
!true // → false
!false // → true
```

Un ejemplo en la vida real sería pulsar el interruptor de la luz. Si la luz está encendida (true), entonces pulsando el interruptor la apagamos (false). Si la luz está apagada (false), entonces pulsando el interruptor la encendemos (true).

137.4 Combinando operadores lógicos, aritméticos y de comparación

Los operadores lógicos y los operadores de comparación se pueden combinar para crear expresiones más complejas. Por ejemplo, podemos preguntar si un número está entre dos valores.

```
2 < 3 && 3 < 4 // → true
```

En este caso, la expresión se evalúa como **true** porque $\mathbf{2 < 3}$ es **true** y $\mathbf{3 < 4}$ es **true**. Si cualquiera de las dos expresiones fuera **false**, entonces la expresión sería **false**.

También puedes usar paréntesis para agrupar operaciones y usar operadores lógicos y aritméticos.

```
(2 + 2) < 3 && (10 < (8 * 2)) // → false
```

Igualmente, es importante que sepas que las operaciones aritméticas tienen precedencia sobre las operaciones de comparación.

```
2 + 2 < 3 && 10 < 8 * 2
// Primero se hacen las operaciones aritméticas:
// → 4 < 3 && 10 < 16
// Ahora las comparaciones:
// → false && true
// Finalmente:
// → false
```

137.5 Actividad

1. Comprueba si 7 es mayor que 8 y menor que 10
2. Tenemos un producto en una tienda. Cuesta 1500 y tenemos un descuento del 25%. Tengo $1150 en mi billetera. Escribe un código que me diga si puedo comprarlo. No utilizar parentesis.

Respuesta

R1.

```
7 > 8 && 7 < 10 // false
```

R2.

```
1150 >= 1500 * 0.75 // false
```

138 Dos o más operandos

Aunque todos los ejemplos que has visto hasta ahora usan dos operandos, los operadores lógicos también pueden usarse con más de dos operandos.

```
true && true && true // → true
```

También puedes mezclar operadores lógicos:

```
true && true || false // → true
!true && !true // → false
false && true || !true // → false
```

Ahora te puede costar un poco entender qué está pasando, pero con la práctica te irá resultando más fácil. Prueba a hacer los ejercicios y a jugar con los operadores lógicos para que te vayan quedando más claros.

138.1 Quiz

¿Qué operador lógico se utiliza para comprobar si dos o más condiciones son verdaderas?

OR (||)

AND (&&)

¿Qué operador lógico se utiliza para invertir el valor de un booleano?

OR (||)

NOT (!)

¿Qué valor devuelve la expresión false || true?

false

true

139 Variables

A la hora de crear programas, es vital poder almacenar la información para poder utilizarla en un futuro. En JavaScript, usamos variables para conseguirlo.

Para crear una variable podemos usar la palabra reservada **let** y le damos un nombre a la variable. Por ejemplo:

```
let numero
```

Tenemos una variable llamada numero pero no le hemos asignado ningún valor. Para asignarle un valor, usamos el operador de asignación =:

```
let numero = 1
```

Ahora, podemos referirnos a la variable numero y obtener el valor que le hemos asignado.

```
numero + 1 // -> 1 + 1
```

También podemos reasignarle un valor a la variable:

```
numero = 5
numero + 1 // -> 5 + 1
```

> Tip
>
> ¡Ojo! Si no guardas el valor de esta nueva operación, el valor de la variable **numero** seguirá siendo **5**.
>
> ```
> numero = 5
> numero + 1 // -> 5 + 1
> numero + 1 // -> 5 + 1
> ```

Ten en cuenta que el valor de la variable no tiene porque ser un número. Puede ser cualquier tipo de dato, como un texto o un booleano.

```
let welcomeText = 'Hola'
let isCool = true
```

139.1 Actividad

1. Crea una variable llamada mensaje y asígnale el valor “Hola JavaScript”
2. Crea una variable llamada resultado y asígnale la suma de 2 y 3

Respuesta

R1.

```
let mensaje = 'Hola JavaScript'
```

R2.

```
let resultado = 2 + 3
```

139.2 Constantes const

Las constantes son variables que no pueden ser reasignadas. Para crear una constante, usamos la palabra reservada const:

```
const PI = 3.1415
```

Si intentas reasignar el valor de una constante, obtendrás un error:

```
PI = 3 // -> TypeError: Assignment to constant variable.
```

Como no se pueden reasignar, las constantes siempre deben ser inicializadas con algún valor. Esto es otra diferencia respecto a let, que no es necesario inicializarla con un valor.

```
let numero //
const RADIUS //  SyntaxError: Missing initializer in const declaration
```

Son muy útiles para almacenar valores que no van a cambiar. Siempre que puedas, procura usar constantes para que tu código sea más predecible.

139.3 Actividad

1. Crea una constante llamada IS_DISABLED y asígnale el booleano true

Respuesta

```
const IS_DISABLED = true
```

139.4 Variables var

En JavaScript, también podemos crear variables usando la palabra reservada **var**. Es la forma más antigua y es normal que encuentres muchos tutoriales que lo usen. Sin embargo, a día de hoy, no es recomendable usar **var** ya que tiene comportamientos extraños que pueden causar errores en tu código.

En una clase posterior te explicaré cuál es la diferencia entre **let**, **const** y **var** además de por qué, siempre que puedas, deberías evitar var.

139.5 El nombre de las variables

En JavaScript, los nombres de las variables pueden contener letras, números y el guión bajo (_). Además, el primer carácter del nombre de la variable no puede ser un número.

Es importante tener en cuenta que los nombres de las variables son sensibles a las mayúsculas y minúsculas, lo que significa que **miVariable** y **mivariable** son dos variables diferentes en JavaScript.

```
let miVariable = 1
let mivariable = 2
miVariable + mivariable // -> 1 + 2
```

También es importante que los nombres de las variables sean descriptivos. Por ejemplo, si queremos almacenar el nombre de un usuario, podemos llamar a la variable userName. De esta forma, cuando leamos el código, sabremos que la variable contiene el nombre de un usuario.

```
let n = 'Pepe' //  Mal, no es descriptivo
let userName = 'Juan' //  Bien, se entiende
```

139.6 Convenciones y nomenclaturas

En JavaScript, existen diferentes nomenclaturas para nombrar las variables: **camelCase**, **snake_case** y **SCREAMING_CASE**.

camelCase es la forma más común de nombrar las variables en JavaScript. Consiste en escribir la primera palabra en minúsculas y las siguientes palabras con su primera letra en mayúsculas. Por ejemplo:

```
let camelCase = 1
let camelCaseIsCool = 2
let userName = 'statick'
```

snake_case es una forma de nombrar que consiste en escribir todas las palabras en minúsculas y separarlas con guiones bajos. Por ejemplo:

```
let snake_case = 1
let snake_case_is_cool = 2
let user_name = 'statick'
```

En algunos lenguajes de programación es muy común usar snake_case para nombrar las variables. En JavaScript no lo es tanto, pero todavía puedes encontrar código que lo use.

Lo más habitual, y es buena idea, es usarlo en los nombres de archivos. Por ejemplo, **mi_archivo.js**. Esto es porque algunos sistemas operativos distinguen entre mayúsculas y minúsculas y, por tanto, **mi_archivo.js** y **Mi_archivo.js** son dos archivos diferentes.

> Tip
>
> También existe kebab-case, que es una forma de nombrar que consiste en escribir todas las palabras en minúsculas y separarlas con guiones. Por ejemplo: mi-archivo.js. Es muy similar a snake_case pero con guiones en vez de guiones bajos. No se puede usar para nombrar variables pero sí es común usarlo en los nombres de archivos.

SCREAMING_CASE es una forma de nombrar que consiste en escribir todas las palabras en mayúsculas y separarlas con guiones bajos. Por ejemplo:

```
const SCREAMING_CASE = 1
const SCREAMING_CASE_IS_COOL = 2
const USER_NAME = 'statick'
```

Para las constantes, con valores que no van a cambiar, es muy común usar **SCREAMING_CASE**. Así se puede distinguir fácilmente de las variables que sí cambian de valor. Por eso, no debes usarla para nombrar variables con **let**.

139.7 Quiz

¿Qué es una variable?

En JavaScript, una variable es un contenedor de información que se utiliza para almacenar datos en un programa.

Una variable es un tipo de dato que se utiliza para almacenar información en un programa.

¿Qué peculiaridad tienen las variables const en JavaScript?

No se pueden reasignar

Sólo pueden guardar números

Siempre hay que asignar un valor a una variable

Sí, siempre

No, nunca

Sólo si es una variable const

139.8 null y undefined

¿Recuerdas que te he comentado que existen diferentes tipos de datos en JavaScript? Hemos visto números, cadenas de texto y booleanos.

En esta clase vas a aprender otros dos tipos de datos que, aunque son similares, tienen ligeras diferencias. Son **null** y **undefined**.

La particularidad de estos dos tipos de datos es que cada uno sólo tiene un valor. El tipo **null** sólo puede tener el valor **null** y el tipo **undefined** sólo puede tener el valor undefined.

> Tip
>
> Es como el tipo booleano que podía ser **true** y **false** pero, en este caso, sólo tiene un valor.

139.9 La diferencia entre null y undefined

Mientras que null es un valor que significa que algo no tiene valor, undefined significa que algo no ha sido definido. Por ejemplo, si creamos una variable sin asignarle ningún valor, su valor será undefined:

```
let rolloDePapel // -> undefined
```

También podemos asignar directamente el valor undefined a una variable:

```
let rolloDePapel = undefined // -> undefined
```

En cambio, para que una variable tenga el valor null, sólo podemos conseguirlo asignándole explícitamente ese valor:

```
let rolloDePapel = null
```

Un caso bastante ilustrativo para entender la diferencia entre null y undefined es el siguiente:

139.10 Actividad

1. Crea una variable con let llamada capacidad y asígnale un valor null
2. Crea una variable con let llamada dinero y asegúrate que tenga un valor de undefined

Respuesta

R1.

```
let capacidad = null
```

R2.

```
let dinero
```

139.11 Comparaciones con null y undefined

Al usar la igualdad estricta que hemos visto en la clase anterior, null y undefined son considerados diferentes entre sí:

```
null === undefined // -\> false
```

Sólo cuando comparamos null con null o undefined con undefined obtenemos true:

```
null === null // -\> true undefined === undefined // -\> true
```

139.12 Quiz

null y undefined son dos valores que significan lo mismo.

Sí, ambos indican la ausencia de valor.

No, null es un valor que indica algo vacío, mientras que undefined indica algo que no está definido todavía.

¿Qué valor tiene una variable a la que no se le ha asignado nada al declararla?

Un valor null

Un valor undefined

No se puede declarar a una variable sin asignar un valor

140 Operador typeof

El operador **typeof** devuelve una cadena de texto que indica el tipo de un operando. Puede ser usado con cualquier tipo de operando, incluyendo variables y literales.

```
const MAGIC_NUMBER = 7
typeof MAGIC_NUMBER // "number"
```

También puedes usarlo directamente con los valores que quieras comprobar:

```
typeof undefined // "undefined"
typeof true // "boolean"
typeof 42 // "number"
typeof "Hola mundo" // "string"
```

140.1 Actividad

1. Tengo una variable llamada userName. Excribe el código necesario para ver su tipo.

Respuesta

```
let userName
typeof userName // "undefined"
```

Existe, sin embargo, un valor especial en JavaScript, **null**, que es considerado un bug en el lenguaje. El operador **typeof** devuelve **"object"** cuando se usa con **null**:

```
typeof null // "object"
```

Lo correcto sería que **typeof null** devolviera **"null"**, pero es un error histórico que no se puede corregir sin romper el código existente.

Por eso, si quieres comprobar si una variable es **null**, debes usar la comparación estricta ===:

```
const foo = null
foo === null // true
```

Otra pregunta es… ¿Qué es ese **object**? Es un tipo de dato que está en el centro de JavaScript y que veremos en detalle más adelante.

140.2 Usando con operadores de comparación

El operador **typeof** es muy útil cuando se usa con operadores de comparación. Por ejemplo, para comprobar si una variable es del tipo que esperamos:

```
const age = 42
typeof age === "number" // true
```

Una vez que tenemos expresiones lógicas, podemos empezar a encadenar operadores lógicos para comprobar múltiples condiciones:

```
const age = 42
typeof age === "number" && age > 18 // true
```

140.3 Actividad

1. Tengo una variable llamada dogId pero no tengo claro si es una cadena de texto. Escribe el código necesario para asegurarte.

Respuesta

```
typeof dogId === "string" // false
```

140.4 Quiz

¿Para qué sirve el operador typeof en JavaScript?

Nos permite leer el valor de una variable

Nos da el tipo de dato de una variable

El operador typeof siempre devuelve una cadena de texto con el tipo del operando.

Si, siempre devuelve una cadena de texto

Depende del tipo de dato que le pasemos, Si le pasamos un número, nos devolverá un number

140.5 Comentarios

En JavaScript, los comentarios son **una forma de agregar explicaciones al código que se ignora al ejecutar el programa**.

Los comentarios son útiles para explicar el por qué del código, documentar los cambios realizados en el código y hacer que el código sea más fácil de entender para otros desarrolladores.

Hay **dos tipos de comentarios en JavaScript**: los comentarios de una sola línea y los comentarios de varias líneas.

140.6 Comentarios de una sola línea //

Los comentarios de una sola línea comienzan con // y se utilizan para agregar una explicación en una sola línea de código. Por ejemplo:

```
// Sólo usamos 6 decimales
const PI = 3.141592

// Iniciamos el radio por 10, pero puede cambiar
let radio = 10
```

También puedes añadir un comentario de una sola línea al final de una línea de código. Por ejemplo:

```
const PI = 3.141592 // Sólo usamos 6 decimales
```

140.7 Comentarios de varias líneas /* */

Los comentarios de varias líneas comienzan con /* y terminan con */. Se utilizan para agregar notas explicativas que ocupan varias líneas de código. Por ejemplo:

```
/*
  Este es un comentario de varias líneas.
  Se utiliza para agregar notas explicativas que ocupan varias líneas de código.
*/
```

Ten en cuenta que también puedes lograr varias líneas usando el tipo de comentario //, sólo que en este caso, cada línea de código debe comenzar con //. Por ejemplo:

```
// Este es un comentario de varias líneas.
// Se utiliza para agregar notas explicativas que ocupan varias líneas de código.
```

Es importante tener en cuenta que **los comentarios no afectan el funcionamiento del código**. Es decir, si se eliminan los comentarios, el código seguirá funcionando de la misma manera.

Es recomendable utilizar comentarios con moderación y de manera efectiva para hacer que el código sea más fácil de entender y mantener. **Los comentarios deben ser claros y concisos, y deben explicar lo que hace el código sin repetir lo que ya es obvio**.

Es mejor que tu código sea lo suficientemente claro como para no necesitar comentarios, pero si es necesario, **utiliza comentarios para explicar el por qué del código, no el qué**.

140.8 Quiz

¿Para qué sirven los comentarios en JavaScript?

Sirven para documentar nuestro código y explicar qué hace cada parte de él

Para cambiar el comportamiento de nuestro código

¿Cómo se escriben los comentarios en JavaScript?

Puedes usar # para escribir comentarios de una línea o varias

Puedes usar // para escribir comentarios de una línea o /* */ para escribir comentarios de varias líneas

140.9 console.log()

console.log() es una función integrada en *JavaScript* que se utiliza para imprimir mensajes en la consola del navegador o del editor de código. Se utiliza principalmente para depurar el código y para **imprimir valores de variables y mensajes para ayudar en el proceso de desarrollo**.

> Tip
>
> En programación, una función es un conjunto de instrucciones que se pueden usar una y otra vez para hacer una tarea específica. Muchas veces, las funciones se utilizan para evitar repetir código y son parametrizables. Más adelante tendremos una sección sólo para ellas.

140.10 Sintaxis

Para poder mostrar estos mensajes en consola, debes escribir **console.log()** y dentro de los paréntesis, el mensaje que quieres mostrar.

```
console.log('Hola, JavaScript')
// -> 'Hola, JavaScript'
```

También puedes averiguar el valor de una variable, escribiendo el nombre de la variable dentro de los paréntesis.

```
const nombre = 'JavaScript'
console.log(nombre)
// -> 'JavaScript'
```

Como ya sabes concatenar cadenas de texto, puedes mostrar un mensaje y el valor de una variable en el mismo **console.log()**.

```
const nombre = 'JavaScript'
console.log('Hola, ' + nombre)
// -> 'Hola, JavaScript'
```

Además, puedes mostrar varios mensajes y valores de variables en el mismo **console.log()** separándolos por comas.

```
const nombre = 'JavaScript'
const version = 2023
console.log(nombre, version)
// -> 'JavaScript 2023'
```

140.11 Más métodos de console

Además de **console.log()**, existen otros métodos que puedes utilizar para imprimir mensajes en la consola. Algunos de ellos son:

> Tip
>
> - **console.error():** Imprime un mensaje de error en la consola.
> - **console.warn():** Imprime un mensaje de advertencia en la consola.
> - **console.info():** Imprime un mensaje de información en la consola.

Como ves, la sintaxis es la misma que **console.log()**, sólo cambia el nombre del método.

> Tip
>
> Aunque puedes usar **console.log()** para imprimir cualquier tipo de mensaje, es recomendable utilizar los métodos que acabamos de ver para imprimir mensajes de error, advertencia e información ya que tienen un formato especial que los hace más fáciles de identificar.

```
console.error('Error')
//   Error
console.warn('Advertencia')
//   Advertencia
console.info('Información')
//   Información
```

Prueba estos métodos en la consola del navegador y observa los resultados.

140.12 Quiz

¿Cuál es la sintaxis adecuada para imprimir un mensaje en la consola del navegador?

console("mensaje")

console.log("mensaje")

console.write("mensaje")

¿Cómo podrías usar console.log() para imprimir múltiples valores o variables en una sola línea de código?

Utilizando una sintaxis especial que formatea los valores o variables en una sola línea de código

No es posible imprimir múltiples valores o variables en una sola línea de código con console.log()

Separando cada valor o variable con comas dentro de los paréntesis de console.log()

¿Cuál es la diferencia entre console.log() y console.error() en JavaScript?

console.log() se utiliza para imprimir mensajes en la consola del navegador, mientras que console.error() se utiliza para imprimir mensajes de error.

console.log() se utiliza para imprimir mensajes de error en la consola del navegador, mientras que console.error() se utiliza para imprimir mensajes normales

No hay diferencia entre console.log() y console.error() en JavaScript

141 Código Condicional con if

El código condicional es un bloque de código que se ejecuta sólo si se cumple una condición. En JavaScript usamos la palabra reservada if para crear un bloque condicional, así:

```
if (condición) {
  // código que se ejecuta si la condición es verdadera
}
```

Como ves, ponemos la condición entre paréntesis y el código se ejecuta si la condición entre llaves es **true**. Si la condición es **false**, el código no se ejecuta.

Imagina que quieres mostrar un mensaje si la edad de un usuario es mayor o igual a 18 años. Podrías hacerlo así:

```
const edad = 18

if (edad >= 18) {
  console.log('Eres mayor de edad')
}
```

141.1 else

Es posible utilizar la palabra clave **else** para ejecutar un bloque de código diferente si la condición es falsa:

```
const edad = 17

if (edad >= 18) {
  console.log('Eres mayor de edad')
} else {
  console.log('Eres menor de edad')
}
```

Esto es útil para ejecutar un bloque de código u otro dependiendo de si se cumple o no una condición. else if

También podemos utilizar la palabra clave **else if** para comprobar más de una condición:

```
const edad = 17

if (edad >= 18) {
  console.log('Eres mayor de edad')
} else if (edad >= 16) {
  console.log('Eres casi mayor de edad')
} else {
  console.log('Eres menor de edad')
}
```

El programa comprueba la primera condición. Si es **true**, ejecuta el código dentro del bloque **if**. Si es **false**, comprueba la siguiente condición. Si es **true**, ejecuta el código dentro del bloque **else if**. Si es false, ejecuta el código dentro del bloque **else**.

Dicho de otra forma, entrará en el primer bloque que cumpla la condición y no entrará en los demás. Si no cumple ninguna, entonces entrará en el bloque **else**.

141.2 Anidación de condicionales

Es posible anidar condicionales dentro de otros condicionales. Por ejemplo:

```
const edad = 17
const tieneCarnet = true

if (edad >= 18) {
  if (tieneCarnet) {
    console.log('Puedes conducir')
  } else {
    console.log('No puedes conducir')
  }
} else {
  console.log('No puedes conducir')
}
```

En muchas ocasiones vas a querer evitar la anidación innecesaria de condicionales ya que se hacen difíciles de leer y mantener. En estos casos es mejor utilizar operadores lógicos para crear la condición:

```
const edad = 17
const tieneCarnet = true

// si es mayor de edad y tiene carnet entonces...
if (edad >= 18 && tieneCarnet) {
  console.log('Puedes conducir')
} else {
  console.log('No puedes conducir')
}
```

Otra técnica muy interesante es la de guardar el resultado de la condición en una variable, para que tus condiciones sean mucho más legibles:

```
const edad = 17
const tieneCarnet = true
const puedeConducir = edad >= 18 && tieneCarnet

if (puedeConducir) {
  console.log('Puedes conducir')
} else {
  console.log('No puedes conducir')
}
```

> **Tip**
>
> ¡Wow! ¿Has visto cómo hemos mejorado la legibilidad de nuestro código? ¡Es mucho más fácil de leer y entender! A este tipo de técnica se le llama *refactorización* y consiste en mejorar el código sin cambiar su comportamiento.

141.3 La importancia de las llaves

Es importante que sepas que las llaves **{ }** no siempre son obligatorios. Si el bloque de código sólo tiene una línea, puedes omitir las llaves:

```
const edad = 17

if (edad >= 18)
  console.log('Eres mayor de edad')
else
  console.log('Eres menor de edad')
```

También lo puedes escribir en la misma línea:

```
const edad = 18

if (edad >= 18) console.log('Eres mayor de edad')
else console.log('Eres menor de edad')
```

Sin embargo, **te recomiendo que mientras estés aprendiendo siempre escribas las llaves**. Esto te ayudará a evitar errores y a que tu código sea más legible.

141.4 Quiz

¿Qué es un bloque condicional en JavaScript?

Un bloque de código que se ejecuta sólo si se cumple una condición

Un bloque de código que se ejecuta siempre

¿Qué palabra clave se utiliza en JavaScript para crear un bloque condicional?

if

else

then

¿Qué palabra clave se utiliza en JavaScript para ejecutar un bloque de código si la condición de un bloque condicional es falsa?

if

then

else

141.5 Actividad

1. Crea una variable llamada **temperatura** y asígnale un valor numérico. Escribe un bloque condicional que imprima en la consola si la temperatura es mayor o igual a 25 grados.
2. Crea una variable llamada **esVerano** y asígnale un valor booleano. Escribe un bloque condicional que imprima en la consola si es verano o no.

Respuesta

R1.

```
const temperatura = 25

if (temperatura >= 25) {
  console.log('Hace calor')
}
```

R2.

```
const esVerano = true

if (esVerano) {
  console.log('Es verano')
}
```

142 Bucles con while

Un bucle es una **estructura de control** que permite repetir un bloque de instrucciones. Vamos, **repetir una tarea tantas veces como queramos**.

En JavaScript, existen varias formas de lograrlo, y una de ellas es el bucle con **while**. El bucle **while** es una estructura de control de flujo que ejecuta una sección de código mientras se cumple una determinada condición.

En esta clase, vamos a explicar cómo funciona el bucle **while** en JavaScript y cómo podemos utilizarlo en nuestros programas.

142.1 Sintaxis

La sintaxis del bucle **while** es similar a la de un condicional **if**. La única diferencia es que, en lugar de ejecutar el código una sola vez, se ejecuta mientras se cumpla la condición.

```
while (condición) {
  // código a ejecutar mientras se cumpla la condición
}
```

El bucle comienza evaluando la condición dentro de los paréntesis. Si la condición es **true**, se ejecuta el código dentro de las llaves.

Después de ejecutar el código, la condición se evalúa de nuevo, y si sigue siendo verdadera, el código dentro de las llaves se ejecuta de nuevo. **Este proceso se repite hasta que la condición se evalúa como falsa**.

Ten en cuenta que, si la condición es falsa desde el principio, el código dentro de las llaves nunca se ejecutará.

> Tip
>
> A cada vuelta del bucle se le llama **iteración**. Una iteración es la repetición de un proceso o acción un número determinado de veces, de manera ordenada y sistemática.

142.2 Ejemplo de uso de while

Vamos a crear la cuenta atrás de un cohete. Creamos una variable **cuentaAtras** que contenga el número de segundos que faltan para el lanzamiento. En este caso, vamos a empezar con 10 segundos.

```
let cuentaAtras = 10
```

Para quitarle un segundo a la cuenta atrás, vamos a utilizar el operador de resta (-) y el operador de asignación (=).

```
let cuentaAtras = 10
cuentaAtras = cuentaAtras - 1
console.log(cuentaAtras) // -> 9
```

Sabiendo esto y cómo funciona el bucle while, podemos crear la cuenta atrás del cohete.

```
// iniciamos la variable fuera del bucle
let cuentaAtras = 10

// mientras la cuenta atrás sea mayor que 0
while (cuentaAtras > 0) {
  // mostramos el valor de la cuenta atrás en cada iteración
  console.log(cuentaAtras)
  // restamos 1 a la cuenta atrás
  cuentaAtras = cuentaAtras - 1
}

console.log('¡Despegue!  ')
```

Si ejecutas este código en consola, deberías ver los números del 10 al 1, y después el mensaje de despegue.

142.3 Cuidado con los bucles infinitos

Los bucles **while** son muy potentes, pero también pueden ser peligrosos. Si la condición nunca se evalúa como falsa, el bucle se ejecutará infinitamente.

```
while (true) {
  console.log('¡Hola hasta el infinito!')
}
```

Esto evaluará la condición **true** como verdadera, y ejecutará el código dentro de las llaves una y otra vez.

142.4 Saliendo de un bucle con break

Podemos controlar cuándo queremos salir de un bucle utilizando la palabra reservada **break**. Cuando el intérprete de JavaScript encuentra la palabra **break**, sale del bucle y continúa ejecutando el código que haya después.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  console.log(cuentaAtras)
  cuentaAtras = cuentaAtras - 1

  // si la cuenta atrás es 5, salimos del bucle
  if (cuentaAtras === 5) {
    break // <---- salimos del bucle
  }
}
```

¿Cuál es el valor de **cuentaAtras** en este código? Veamos, el bucle estaba haciendo una cuenta atrás… pero le hemos dicho que cuando tuviese el valor **5** saliese del bucle. Por lo tanto, el valor de **cuentaAtras** es **5**.

Usar **break** puede ser útil en bucles si queremos salir de ellos por alguna condición en concreto o para evitar justamente los bucles infinitos.

142.5 Saltando una iteración con continue

Igual que tenemos la posibilidad de "romper" el bucle con **break**, también podemos saltarnos una iteración con **continue**. Cuando el intérprete de JavaScript encuentra la palabra **continue**, salta a la siguiente iteración del bucle.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  cuentaAtras = cuentaAtras - 1

  // si la cuenta atrás es un número par...
  if (cuentaAtras % 2 === 0) {
    continue // <---- saltamos a la siguiente iteración
  }

  console.log(cuentaAtras)
}
```

¿Qué aparece en la salida de la consola? El bucle está haciendo una cuenta atrás… pero le hemos dicho que si el número es par, se salte esa iteración y deje de ejecutar el código que le sigue.

Por ello, los números pares no aparecen en la consola.

142.6 Anidación de bucles

Podemos anidar bucles dentro de otros bucles. Imagina que en nuestra cuenta atrás para el cohete, tenemos que revisar que 3 cosas están en sus parámetros: el oxígeno, el combustible y la temperatura.

```
const NUMERO_REVISIONES = 3
let cuentaAtras = 10

// mientras la cuenta atrás sea mayor que 0
while (cuentaAtras > 0) {
  // mostramos el valor de la cuenta atrás
  console.log(cuentaAtras)

  // creamos una variable para contar las revisiones realizadas
  // y la inicializamos a cero
  let revisionesRealizadas = 0

  // hasta que no hayamos realizado las 3 revisiones...
  while (revisionesRealizadas < NUMERO_REVISIONES) {
    // y sumamos 1 a las revisiones realizadas
    revisionesRealizadas = revisionesRealizadas + 1
    console.log(revisionesRealizadas + ' revisiones realizadas...')
  }

  // ahora podemos restar 1 a la cuenta atrás
  cuentaAtras = cuentaAtras - 1
}
```

> **Tip**
>
> ¿Por qué la constante la hemos puesto toda en mayúsculas? Como ya explicamos en la clase de introducción a JavaScript, es una convención para indicar que es una constante y que no va a cambiar.

Además del bucle anidado, hay algo también muy interesante en el código anterior y es la creación de la variable **let revisionesRealizadas**.

Ten en cuenta que esa variable se creará y se inicializará a **0** en cada iteración del bucle.

Las variables creadas con **let** y **const** que se crean dentro de un bucle, solo existen dentro de ese bucle. Cuando el bucle termina, la variable desaparece. De hecho si intentas acceder a ella fuera del bucle, te dará un error.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  let revisionesRealizadas = 3
  console.log(revisionesRealizadas)
```

```
  cuentaAtras = cuentaAtras - 1
}

console.log(revisionesRealizadas) // -> ERROR: ReferenceError
```

Esto también pasa con otras estructuras de control. Eso es porque el alcance de las variables creadas con **let** y **const** es el bloque entre **{ }** en el que se crean. Lo iremos viendo más adelante para que vayas practicando, no te preocupes.

142.7 Quiz

¿Cuál es la sintaxis correcta para utilizar un bucle while en JavaScript?

```
while (condition){
  // código a ejecutar mientras se cumpla la condición
}
```

```
while {
  // código a ejecutar mientras se cumpla la condición
} (condition)
```

¿Cuál es la palabra reservada que se utiliza para salir de un bucle en JavaScript?

break

continue

exit

¿Cuántas veces saldrá en consola la palabra 'Café'?

```
let drinkCoffee = 0

while (drinkCoffee < 10){
    drinkCoffee++;

    if (drinkCoffee === 8){
        break;
    }

    if (drinkCoffee === 5){
        continue;
    }
    console.log('Café');
}
```

6

4

5

143 Bucles con do while

Aunque no es muy famoso ni muy utilizado, es interesante que sepas que existe en JavaScript un bucle que se ejecuta al menos una vez, y luego se repite mientras se cumpla una condición. Este bucle se llama **do while** y tiene la siguiente sintaxis:

```
do {
  // código que se ejecuta al menos una vez
} while (condición);
```

Vamos a aprovechar para hacer más interesante la clase. Por eso te voy a presentar una función integradas en JavaScript que nos van a dar mucho juego: **confirm**.

144 confirm

La función **confirm** muestra un cuadro de diálogo con dos botones: "Aceptar" y "Cancelar". Si el usuario pulsa "Aceptar", la función devuelve **true**. Si pulsa "Cancelar", devuelve **false**.

```
confirm("¿Te gusta JavaScript?");
```

Llamar a la función **confirm** es una expresión y, por lo tanto, produce un valor que podemos guardar en una variable.

```
let respuesta = confirm("¿Te gusta JavaScript?")
console.log(respuesta) // -> true o false
```

145 Usando do while

Vamos a hacer un programa que saldrá de un bucle **do while** cuando el usuario pulse "Cancelar" en el cuadro de diálogo que muestra la función **confirm**.

```
let respuesta

do {
  respuesta = confirm("¿Te gusta JavaScript?");
} while (respuesta)
```

> Tip
>
> ¿Por qué hay que poner la variable **respuesta** fuera? Porque si no, no podría ser usada en la condición del bucle. Ya vas viendo lo que comentámos antes del ámbito de las variables.

Si el usuario pulsa "Aceptar", la variable **respuesta** valdrá **true** y el bucle se repetirá. Si el usuario pulsa "Cancelar", la variable **respuesta** valdrá **false** y el bucle se detendrá.

145.1 Quiz

¿Cuántas veces como mínimo se ejecutará el código dentro de un bucle do{ ... } while?

Ninguna vez

Depende de la condición

Una vez

146 Expresiones y declaraciones

En JavaScript, existen dos tipos de elementos fundamentales para escribir código: **expresiones** y **declaraciones**. Aunque estos términos pueden parecer confusos al principio, son conceptos clave para comprender cómo funciona JavaScript.

147 Declaraciones

Las declaraciones en JavaScript son sentencias que definen la creación de una variable, función o clase. **Podríamos decir que las declaraciones son como las instrucciones que le damos a JavaScript para que haga algo**.

> Tip
>
> Las funciones y las clases las veremos más adelante en el curso, no te preocupes.

Por ejemplo, una declaración de variable es una sentencia que le da un nombre y un valor a una variable. El siguiente código es un ejemplo de una declaración de variable:

```
let nombre = "Juan"
```

Este código no lo puedes usar con el método **console.log**, ya que no produce ningún valor. Si lo intentas, obtendrás un error:

```
console.log(let nombre = "Juan") // SyntaxError
```

148 Expresiones

Las expresiones en JavaScript son sentencias que producen un valor. Las expresiones pueden ser tan simples como un solo número o una cadena de texto, o tan complejas como el cálculo de una operación matemática, la evaluación de diferentes valores o la llamada a una función.

Por ejemplo, una expresión numérica es una sentencia que produce un número:

```
2 + 3 // -> 5
```

De hecho, lo que guardamos en las variables son expresiones. Por ejemplo, en el siguiente código, la expresión **2 + 3** se evalúa y el resultado se guarda en la variable **resultado**:

```
let resultado = 2 + 3
```

En ese código tenemos la declaración que queremos guardar en la variable resultado el **resultado** de la expresión **2 + 3**.

149 ¿Por qué es importante la diferencia?

La diferencia entre declaraciones y expresiones es importante ya que no podemos usar una declaración donde se espera una expresión y viceversa.

Por ejemplo, ya hemos conocido las estructuras de control **if** y **while**. Ambas esperan una expresión que se evalúa a un valor booleano. Por lo tanto, no podemos usar una declaración en su lugar:

```
//  Ambos códigos están mal y sirven para
// ilustrar que no debes usar declaraciones
// cuando espera expresiones

if (let nombre = "Juan") { //  SyntaxError
  console.log("Hola, Juan")
}

while (let i = 0) { //  SyntaxError
  console.log("Iteración")
  i = i + 1
}
```

150 Quiz

¿Qué son las declaraciones en JavaScript y para qué se utilizan?

Las declaraciones en JavaScript son sentencias que definen la creación de una variable, función o clase para que puedan ser usadas posteriormente en el código.

Las declaraciones en JavaScript son sentencias que producen un valor. Se utilizan para calcular un valor y devolverlo en una expresión.

Las declaraciones en JavaScript son sentencias que se utilizan para realizar operaciones matemáticas complejas.

¿Qué son las expresiones en JavaScript y para qué se utilizan?

Las expresiones en JavaScript son sentencias que definen la creación de una variable

Las expresiones en JavaScript son sentencias que producen un valor.

Las expresiones sólo se usan como condiciones de los condicionales if y los bucles.

151 Bucles con for

La estructura de control **for** en JavaScript es muy útil para ejecutar una serie de instrucciones un número determinado de veces. A diferencia de **while** que usa una condición para determinar si se ejecuta o no el bloque de código, **for** usa un contador que se incrementa en cada iteración hasta que se cumple una condición.

La sintaxis de **for** es la siguiente:

```
for (inicialización; condición; actualización) {
  // código a ejecutar
}
```

Importante, fíjate que **for** tiene tres partes separadas por ;:

- La *inicialización* se realiza antes de que se inicie el bucle y se utiliza para declarar variables y asignar valores iniciales.
- La *condición* es una expresión *booleana* que se evalúa antes de cada iteración del bucle. Si la expresión se evalúa como **true**, se ejecuta el bloque de código dentro del bucle. Si la expresión se evalúa como **false**, el bucle termina.
- La actualización se utiliza para actualizar el valor de la variable de control del bucle después de cada iteración. Normalmente, se incrementa o decrementa el valor de la variable de control del bucle.

152 Ejemplo de uso de for

El siguiente ejemplo muestra cómo se puede usar **for** para imprimir los números del 1 al 10:

```
for (let number = 1; number <= 10; number++) {
  console.log(number)
}
```

- La inicialización es la declaración de la variable **number** y la asignación del valor **1**.
- La condición es que mientras **number <= 10**, se itera el bucle.
- La actualización es **number++** que incrementa el valor de **number** en 1 después de cada iteración.

Oh, espera, **¿qué es eso de number++? Te lo explico**.

153 Incrementando números con ++

El operador ++ nos permite incrementar el valor de una variable número en 1. Por ejemplo, si tenemos la variable **i** con el valor **5**, **i++** incrementaría el valor de **i** en 1, es decir, **i** pasaría a valer **6**.

```
let i = 5
i++
console.log(i) // -> 6
```

Es una forma abreviada de escribir **i = i + 1**. Y, como ves, es la forma más típica de usar en bucles como **for** para no tener que escribir **i = i + 1** cada vez.

154 Iterando al revés

El método **for** en JavaScript también se puede utilizar para iterar al revés Para ello, se utiliza la sintaxis **i–** en la sección de actualización del bucle en lugar de **i++**.

La sintaxis del método for para iterar en reversa es la siguiente:

```
for (let i = 10; i >= 0; i--) {
  if (i === 0) {
    console.log('¡Despegue !')
  } else {
    console.log('Faltan ' + i + ' segundos')
  }
}
```

155 Llevando más allá el bucle for

Aunque siempre los ejemplos con **for** son los mismos, ten en cuenta que puedes hacer cualquier cosa en la inicialización, condición y actualización. Podrías, por ejemplo, usar dos variables a la vez.

```
for (let i = 0, j = 5; i < 5; i++, j--) {
  console.log(i, j);
}
```

La coma **,** nos permite declarar dos variables en la inicialización y actualizarlas en la actualización. En este caso la **,** es un **operador que evalua cada uno de sus operandos** (de izquierda a derecha) y retorna el valor del último operando.

No es muy común su uso, pero es bueno saber que existe.

156 continue y break

Al igual que en **while**, **for** también tiene las palabras reservadas **continue** y **break** para controlar el flujo de ejecución del bucle.

Recuerda que **continue** se usa para saltar a la siguiente iteración del bucle y **break** para salir del bucle.

Ejemplo:

```
for (let i = 0; i < 10; i++) {
  const esPar = i % 2 === 0
  if (esPar) {
    continue
  }

  // Solo mostramos este consola.log si es impar
  console.log(i)

  // Salimos del bucle al llegar al 7
  if (i === 7) {
    break
  }
}
```

157 Bucles anidados con for

Imagina que quieres crear **la tabla de multiplicar del 1 al 10**. Para ello, necesitas un bucle que itere del 1 al 10 y, dentro de ese bucle, otro bucle que itere del 1 al 10. Esto se puede hacer con dos bucles **for** anidados (uno dentro del otro).

```
for (let i = 1; i <= 10; i++) {
  for (let j = 1; j <= 10; j++) {
    const resultado = i * j
    console.log(i + ' x ' + j + ' = ' + resultado)
  }
}
```

158 Quiz

¿Cuál es la sintaxis correcta para crear un bucle for en JavaScript?

```
for (let i = 0, i < Array.length; i++) {
    // código a ejecutar en cada iteracción
}
```

```
for (let i = 0; i < array.length; i++) {
    // código a ejecutar en cada iteracción
}
```

¿Cuántas veces veremos el mensaje "Hola" en consola?

```
for (let i = 0; i < 10 && i !== 3; i++) {
  console.log('Hola')
}
```

10 veces

4 veces

3 veces

159 Switch

En JavaScript, la sentencia **switch** es una estructura de control que nos permite ejecutar diferentes bloques de código dependiendo del valor de una **expresión**. Esta estructura es útil cuando queremos realizar diferentes acciones basadas en una única variable.

160 Sintaxis

La sentencia switch evalúa una expresión, comparando el valor con los diferentes casos que le hemos definido. Si hay coincidencia ejecuta el bloque de código asociado. Para ello, se utiliza la sentencia **break** para separar cada caso y evitar que se sigan evaluando el resto de casos.

```
switch (expresión) {
  case valor1:
    // código a ejecutar si la expresión coincide con valor1
    break

  case valor2:
    // código a ejecutar si la expresión coincide con valor2
    break
  default:
    // código a ejecutar si la expresión no coincide con ningún valor
    break
}
```

> Tip
>
> La condición de **default** es opcional. No es obligatorio que la uses si no la necesitas. Es como el **else** de las condiciones **if**.

Por ejemplo si queremos mostrar un mensaje diferente dependiendo del día de la semana:

```
const dia = "lunes"

switch (dia) {
  case "lunes":
    console.log("¡Hoy es lunes!  ")
    break

  default:
    console.log("No es lunes, YAY!  ")
    break
}
```

Vamos a hacer esto un poco más interesante. En JavaScript podemos recuperar la información de la hora y la fecha usando el objeto **Date**. Este objeto tiene un método llamado **getDay()** que nos devuelve el día de la semana en formato numérico, siendo 0 el domingo y 6 el sábado.

Tip

Igual que **console** es un objeto. También **Date** lo es. Un objeto en JavaScript no es muy diferente a un objeto en la vida real. Tiene propiedades y funciones que podemos usar. En este caso, **getDay()** es una función que nos devuelve el día de la semana.

```
const dia = new Date().getDay()

// segun el dia de la semana, mostramos un mensaje diferente
switch (dia) {
  case 0:
    console.log("¡Hoy es domingo!  ")
    break
  case 1:
    console.log("¡Nooo, es lunes!  ")
    break
  case 2:
    console.log("¡Hoy es martes!  ")
    break
  case 3:
    console.log("¡Hoy es miércoles!  ")
    break
  default:
    console.log("Se acerca el fin de!  ")
    break
}
```

161 switch vs if

Muchas veces verás que puedes escribir el mismo código usando **switch** o **if**. El ejemplo anterior con un **if**:

```
const dia = new Date().getDay()

if (dia === 0) {
  console.log("¡Hoy es domingo!  ")
} else if (dia === 1) {
  console.log("¡Nooo, es lunes!  ")
} else if (dia === 2) {
  console.log("¡Hoy es martes!  ")
} else if (dia === 3) {
  console.log("¡Hoy es miércoles!  ")
} else {
  console.log("Se acerca el fin de!  ")
}
```

A veces es más fácil de leer con **switch** y otras con **if**. ¡Depende de ti! Aunque más adelante, en el curso, veremos alternativas a **switch** usando algunas estructuras de datos.

162 Agrupando cases

En ocasiones, queremos que varios casos ejecuten el mismo código. En lugar de repetir el mismo código en cada caso, podemos agruparlos usando el mismo **case** para cada uno de ellos.

```
const dia = new Date().getDay()

switch (dia) {
  case 0:
  case 6:
    console.log("¡Hoy es fin de semana!  ")
    break
  case 1:
  case 2:
  case 3:
  case 4:
    console.log("¡Nooo, a trabajar!  ")
    break
  case 5:
    console.log("¡Hoy es viernes!  ")
    break
}
```

Como ves, en este caso, tampoco estamos usando el **default**. Como te comentaba antes, es totalmente opcional, no tienes por qué usarlo siempre.

163 ¡No olvides el break!

Es muy importante que no te olvides de poner el **break** al final de cada caso. Si no lo pones, se ejecutarán todos los casos que vengan después del que coincide. Por ejemplo:

```
//  Este código no es correcto
// Lo muestro para que veas qué pasa
// si no pones break en tus casos
const dia = new Date().getDay()

switch (dia) {
  case 0:
    console.log("¡Hoy es domingo!  ")
  case 1:
    console.log("¡Nooo, es lunes!  ")
  case 2:
    console.log("¡Hoy es martes!  ")
  case 3:
    console.log("¡Hoy es miércoles!  ")
  default:
    console.log("Se acerca el fin de!  ")
}
```

Si hoy es martes (**2**), se ejecutarán todos los casos después de ese, ya que no hay ningún **break** que lo impida. Por lo tanto, el resultado en consola sería:

```
¡Hoy es martes!
¡Hoy es miércoles!
Se acerca el fin de!
```

164 El patrón switch(true)

Existe un patrón que se usa a veces con **switch** que es **switch(true)**. Esto es, en lugar de evaluar una expresión, evalúa una condición ignorando por completo el valor de la expresión.

Por ejemplo:

```
let edad = 25;

switch(true) {
  case (edad >= 18 && edad < 25):
    console.log("Eres mayor de edad y eres joven");
    break;
  case (edad >= 25 && edad < 40):
    console.log("Eres mayor de edad y estás en plena madurez");
    break;
  case (edad >= 40):
    console.log("Eres mayor de edad y estás en la mejor edad");
    break;
  default:
    console.log("Eres menor de edad");
}
```

Es un patrón algo más avanzado y es una alternativa a usar **if** y **else if**. En mi opinión, es más difícil de leer, ya que puede crear un poco de confusión. Pero es bueno que lo conozcas por si te lo encuentras en algún sitio.

165 Quiz

¿Cuál es la función de la palabra reservada default en una declaración switch en JavaScript?

Es el caso que se ejecuta cuando se produce un error en el switch

Es el caso que se ejecuta cuando ninguno de los casos coincide con la expresión

Es un caso opcional que se puede utilizar para realizar acciones adicionales

¿Qué problema tiene este código?

```
switch (day) {
  case 0:
    console.log('Hoy es Domingo')
  case 1:
    console.log('Hoy es Lunes')
  case 2:
    console.log('Hoy es Martes')
  default:
    console.log('Es otro día')
}
```

No tiene ningún problema

Se repite mucho código

Falta usar break en cada caso

¿Se puede hacer esto con switch?

```
let x = 5

switch (true) {
  case (x < 0):
    console.log('x es negativo')
    break
  case (x > 0):
    console.log('x es positivo')
    break
  default:
    console.log('x es cero')
}
```

No, porque no se puede usar una expresión booleana en el switch

No, porque no se puede saber el valor de x dentro

Sí, no hay ningún problema

166 Tu primera función

Las *funciones* son uno de los conceptos más importantes en *JavaScript* y casi cualquier lenguaje de programación.

Una función es un bloque de código que realiza una tarea específica cuando se llama. Puedes pensar en una función como en un microondas: le das algo para cocinar, le pasas algunos parámetros (como el tiempo y la potencia) y luego hace su trabajo y te devuelve el resultado.

En JavaScript, las funciones se pueden definir de varias maneras, pero la forma más común y básica es mediante la palabra clave **function**. Esta es tu primera función:

```
function saludar() {
  console.log('Hola Diego')
}
```

Como ves, declarar una función consiste de:

- La palabra reservada **function** que indica que estamos definiendo una función.
- El nombre de la función, en este caso **saludar**. Normalmente usamos verbos para nombrar funciones, ya que son acciones que realizan.
- Un par de paréntesis **( )** después del nombre. Pueden contener parámetros. En este caso, no tenemos ninguno.
- Un bloque de código entre llaves **{ }**. En este caso, solo tenemos una línea de código, pero podríamos tener muchas más.

Nuestra función ahora mismo no devuelve nada pero cada vez que la llamemos, imprimirá **Hola** en la consola.

> Tip
>
> Las funciones pueden devolver un resultado (un número, una cadena de texto, un booleano…) o puede no devolver nada. En ese caso, la función devuelve **undefined**.

saludar() // -> Hola Diego saludar() // -> Hola Diego saludar() // -> Hola Diego

167 Devolviendo un resultado

Las funciones pueden devolver un resultado. Para ello, utilizamos la palabra reservada **return** y después el valor que queremos devolver:

```
function sumar() {
  return 1 + 1
}
```

Ahora, cada vez que llamemos a la función **sumar**, nos devolverá el resultado de la suma:

```
// podemos guardar el resultado en una variable
const resultado = sumar()

// o ver en consola directamente el resultado
console.log(sumar()) // -> 2
```

> Tip
>
> En la siguiente clase verás cómo puedes pasar parámetros a una función para hacerlas todavía más interesantes y reutilizables.

> Tip
>
> Recuerda, si no utilizamos **return**, la función devolverá **undefined**.

168 Una función realmente útil

La función **saludar** y **sumar** no parecen muy útiles, pero con lo poco que sabemos ya podemos hacer cosas más productivas. Por ejemplo, podemos crear una función que nos devuelva un número aleatorio del 1 al 10.

Para ello vamos a descubrir **Math**, un objeto incorporado en JavaScript que tiene propiedades y métodos para constantes y funciones matemáticas. Vamos a ver dos métodos que nos van a ser muy útiles:

- **Math.random():** devuelve un número aleatorio entre 0 y 1, con decimales.
- **Math.floor():** redondea un número hacia abajo.

> Tip
>
> **Math.random** es parecido a **console.log**, en el sentido que son métodos que JavaScript incorpora de serie y que podemos utilizar en cualquier punto de nuestro programa.

Sabiendo esto, podríamos crear una **función que nos devuelva un número aleatorio del 1 al 10**:

```
function getRandomNumber() {
  // recuperamos un número aleatorio entre 0 y 1
  const random = Math.random() // por ejemplo: 0.6803487380457318

  // lo multiplicamos por 10 para que esté entre 0 y 10
  const multiplied = random * 10 // -> 6.803487380457318

  // redondeamos hacia abajo para que esté entre 0 y 9
  const rounded = Math.floor(multiplied) // -> 6

  // le sumamos uno para que esté entre 1 y 10
  const result = rounded + 1 // -> 7

  // devolvemos el resultado
  return result
}
```

¡Bueno! Hay que ver cuantas cosas has puesto en práctica en muy pocas líneas de código: *constantes*, *variables*, *operadores*, comentarios… ¡y todo en una función!

¿Sabes lo mejor? Que **esa función va a acompañarte durante toda tu carrera**. Es muy común necesitar crear un número aleatorio en un programa, así que es muy útil tener una función que lo haga por nosotros.

> **Tip**
>
> En la función hemos utilizado varias variables para ir guardando los resultados intermedios. Podríamos haberlo hecho todo en una sola línea, pero así es más fácil de entender.

169 Quiz

¿Qué es una función en JavaScript?

Algo que se puede hacer con JavaScript de base

Un bloque de código que realiza una tarea específica cuando se llama

Una variable que almacena información

¿Cuál es la sintaxis básica de una función en JavaScript?

function nombreFuncion() { … }

const function() { … }

fn nombreFuncion() { … }

¿Qué devuelve una función declarada con function si no tiene una declaración de retorno explícita?

null

undefined

0

170 Parámetros

Hasta ahora hemos visto como crear funciones y cómo llamarlas. Pero para poder desbloquear todo el potencial de las funciones, **necesitamos que puedan recibir parámetros**. Esto es, que su comportamiento pueda cambiar según los datos que le pasemos.

Imagina que queremos que nuestra función de **saludar**, pueda saludar a cualquier persona, no solo a Miguel. Para ello, podemos hacer que la función acepte un parámetro, que será el nombre de la persona a la que queremos saludar.

```
function saludar(nombre) {
  console.log("Hola " + nombre)
}
```

Ahora, cada vez que llamemos a la función, podemos pasarle una cadena de texto:

```
saludar('Diego') // -> Hola Diego
saludar('Itziar') // -> Hola Itziar
saludar('Carmen') // -> Hola Carmen
```

Ya ves como algo que parece tan sencillo, tiene un poder enorme. **Las funciones son la base de la programación y nos permiten crear programas complejos a partir de pequeñas piezas**.

Por ejemplo, para crear una calculadora, deberíamos empezar a crear funciones que realicen las operaciones básicas. En este caso necesitaremos parametrizar la función para que acepte los dos números para hacer la operación.

```
function sumar(a, b) {
  return a + b
}

function restar(a, b) {
  return a - b
}
```

Y ya podríamos usarlas:

```
const resultadoSuma = sumar(2, 3)
console.log(resultadoSuma) // -> 5

const resultadoResta = restar(5, 2)
console.log(resultadoResta) // -> 3
```

171 Parámetros y argumentos

En el ejemplo anterior, hemos visto que podemos pasarle valores a la función cuando la llamamos. A estos valores se les llama **argumentos**. Y a los valores que recibe la función, se les llama **parámetros**.

```
// la función suma tiene dos parámetros: a y b
function sumar(a, b) {
  return a + b
}

// cuando llamamos a la función, le pasamos dos argumentos: 2 y 3
const resultadoSuma = sumar(2, 3)
```

Normalmente mucha gente confunde los términos y usa parámetro para referirse a los argumentos. Pero es importante que sepas diferenciarlos, porque son cosas distintas.

172 El orden de los parámetros importa

Algo muy importante, y un error muy común al empezar a crear funciones, es olvidar que el orden de los parámetros importa.

Vamos a crear una función que simule un microondas. Le pasaremos el plato, tiempo y potencia. Y nos devolverá el resultado del plato cocinado.

```
function cocinarMicroondas(plato, tiempo, potencia) {
  if (plato === ' ' && tiempo === 1 && potencia === 5) {
    return ' '
  }

  if (plato === ' ' && tiempo === 2 && potencia === 3) {
    return ' '
  }

  return '  Plato no soportado'
}
```

> Tip
>
> Sí, puedes hacer un **return** dentro de un condicional. Esto hace que la función se detenga y devuelva el valor indicado en el **return**.

Si usamos bien los parámetros, la función funciona correctamente:

```
const resultado = cocinarMicroondas(' ', 1, 5)
console.log(resultado) // ->
```

Pero si cambiamos el orden de los parámetros, la función no funciona:

```
const resultado = cocinarMicroondas(1, 5, ' ')
console.log(resultado) // -> '  Plato no soportado'
```

Esto es porque la función espera que el primer parámetro sea el plato, el segundo el tiempo y el tercero la potencia. **Si cambiamos el orden, podemos tener comportamientos inesperados o errores**.

Más adelante veremos una forma en la que podemos evitar este tipo de errores.

173 Quiz

¿Qué son los parámetros de una función en JavaScript?

Valores que espera recibir una función al llamarla para usarlos dentro de ella

Valores que devuelve una función

Valores que se declaran dentro de una función

¿Qué son los argumentos de una función en JavaScript?

Valores que espera recibir una función

Valores que se pasan a una función al llamarla

Lo mismo que los parámetros

¿Qué pasa si cambiamos el orden de los parámetros al llamar una función?

La función se detiene y no devuelve nada

La función devuelve undefined

Podemos tener comportamientos inesperados o errores

174 Function Expression

Hasta ahora hemos visto cómo podemos declarar funciones usando la palabra clave **function**.

Sin embargo, existen dos formas más de declarar funciones en JavaScript: function expression y arrow function. En este caso, hablemos de la primera.

175 ¿Qué es una function expression?

Una **function expression** es una función que se asigna a una variable. Por ejemplo:

```
// esto es una function expression
const sum = function (a, b) { return a + b }

// esto es una declaración de función
function sum(a, b) { return a + b }
```

> Tip
>
> ¡Fíjate que la **function** no tiene nombre en la **function expression**! Cuando una función no tiene nombre se le llama función anónima. Aunque en este caso, la función está asignada a una variable que sí tiene nombre y por eso podremos utilizarla más adelante.

Con la function expression, a la función se asigna a la variable **sum**. Esto significa que podemos llamar a la función usando el nombre de la variable:

```
sum(1, 2) // 3
```

El comportamiento es muy similar al de una función declarada con la palabra clave **function**. Sin embargo, hay una diferencia muy importante entre ambas que debes conocer: el hoisting.

176 Hoisting

El hoisting es un término que se usa para describir cómo JavaScript parece que mueve las declaraciones funciones al principio del código, de forma que las puedes usar incluso antes de declararlas. Por ejemplo:

```
sum(1, 2) // 3

function sum(a, b) {
  return a + b
}
```

Aquí primero usamos la función y la declaramos luego. ¡Y funciona! ¿Cómo puede ser? **¿No debería darnos un error ya que la función todavía no existe?**

Lo que está pasando es que JavaScript asigna en memoria durante la fase de compilación las declaraciones de funciones y por eso al ejecutarse el código tiene acceso a esa posición de memoria a la que se refiere la función.

Eso sería la explicación técnica del hoisting, pero si te sirve de ayuda, puedes pensar que es como si JavaScript moviese las declaraciones de funciones al principio del código.

177 ¿Y qué pasa con las function expression?

Pues que no se aplica el hoisting. Por ejemplo:

```
sum(1, 2) //  ReferenceError: sum is not defined

const sum = function (a, b) {
  return a + b
}
```

178 Quiz

¿Qué es una function expression en JavaScript?

Una función que se define dentro de otra función

Una función que se declara con la palabra clave “function”

Una función que se asigna a una variable

¿Qué diferencia hay entre una declaración de función y una función de expresión en JavaScript?

Sólo la forma en que se declaran

El hoisting y cómo se crean

El comportamiento al llamarlas

¿Se aplica el hoisting a las funciones de expresión en JavaScript?

Sí

No

179 Funciones flecha

En esta clase vamos a hablar sobre las *funciones flecha*, también conocidas como **arrow functions** en inglés.

Las funciones flecha son una forma más concisa de crear funciones en *JavaScript*, y se han vuelto muy populares en los últimos años debido a su sintaxis simplificada.

La sintaxis básica de una función flecha es la siguiente:

```
const miFuncionFlecha = () => {
  // código a ejecutar
}
```

> Tip
>
> Las funciones flecha son siempre **funciones anónimas** y **function expressions**. Esto significa que no tienen nombre y que se asignan a una variable.

En lugar de la palabra clave **function**, utilizamos una flecha => para definir la función. También podemos omitir los paréntesis alrededor de los parámetros si solo tenemos uno:

```
const saludar = nombre => {
  console.log("Hola " + nombre)
}
```

180 Ventajas de las funciones flecha

Las funciones flecha tienen varias **ventajas sobre las funciones regulares en JavaScript**. Algunas son:

- **Sintaxis más concisa:** la sintaxis de las funciones flecha es más corta y más fácil de leer que la sintaxis de las funciones regulares, especialmente cuando se trabaja con funciones de una sola línea.
- **Return implícito:** las funciones flecha puede devolver el valor de la expresión sin usar la palabra clave **return** cuando son de una sola línea. Esto hace que las funciones flecha sean aún más cortas y más fáciles de leer.
- **Funciones anónimas más legibles:** las funciones flecha son una forma más legible y concisa de crear funciones anónimas en JavaScript, lo cual puede hacer que nuestro código sea más fácil de entender.

> Tip
>
> Recuerda, las funciones anónimas son funciones que no tienen nombre. Son como las que vimos en la clase anterior.

181 Return implícito

Cuando una función flecha tiene una sola expresión, podemos omitir las llaves **{ }** y la palabra clave return para hacerla aún más corta. Esto se conoce como **return implícito**. Vamos a pasar una función regular a una función flecha y vamos a ver cómo se ve finalmente con return implícito:

```
// Declaración de función regular
function sumar(a, b) {
  return a + b
}

// Función flecha
const sumarFlecha = (a, b) => {
  return a + b
}

// Función flecha con return implícito
const sumarFlecha = (a, b) => a + b
```

Como podemos ver, la función flecha con return implícito es mucho más corta y fácil de leer que la función regular. Esto es especialmente útil cuando estamos trabajando con funciones de una sola línea.

182 Quiz

¿Qué son las funciones flecha en JavaScript?

Una forma más concisa de crear funciones en JavaScript

Una forma más compleja de crear funciones en JavaScript

Una técnica para evitar el uso de funciones en JavaScript

¿Cuál es una de las ventajas de las funciones flecha?

Return implícito

Son más rápidas

Funciones anónimas

¿Qué devuelve la siguiente función flecha?

```
const sum = (a, b) => { a + b }
sum(2, 3)
```

5

undefined

Error

183 Recursividad

La recursividad es una técnica de programación que consiste en que una función se llame a sí misma.

Pero claro, si una función se llama a sí misma, ¿cuándo termina? ¿cómo se evita que se llame infinitamente? Con una condición base.

184 Ejemplo de recursividad

Vamos a crear una función que cuente desde un número hasta cero.

```
function cuentaAtras(numero) {
  // Condición base: Si el número que recibe es
  // menor de 0 entonces salimos de la función
  if (numero < 0) { return }

  // Si el número era mayor o igual a 0, lo mostramos
  console.log(numero)

  // Y llamamos a la función con el número anterior
  cuentaAtras(numero - 1)
}
```

Si llamamos a la función con el número 3, el resultado será:

```
cuentaAtras(3)
// -> 3
// -> 2
// -> 1
// -> 0
```

La ejecución la veríamos así:

```
cuentaAtras(3) -> (muestra 3)
               \
          cuentaAtras(2) -> (muestra 2)
                      \
                 cuentaAtras(1) -> (muestra 1)
                             \
                        cuentaAtras(0) -> (muestra 0)
                                     \
                                cuentaAtras(-1) -> salida
```

> **Tip**
>
> ¡Cuidado! Si no ponemos la condición base, la función se llamará infinitamente y el navegador se quedará bloqueado. Cuando hacemos recursividad SIEMPRE hay que tener una condición que haga que la función salga de sí misma.

185 Usando recursividad y devolviendo un valor

La recursividad se usa muchas veces para solucionar algoritmos. Por ejemplo, vamos a crear una función que calcule el factorial de un número.

> **Tip**
>
> El factorial de un número es el resultado de multiplicar ese número por todos los anteriores hasta llegar a 1. Por ejemplo, el factorial de 5 es 5 * 4 * 3 * 2 * 1 = 120

“‘ javascriptfunction factorial(n) { // Condición base: Si el número es 0 o 1, devolvemos 1 // y no llamamos a la función de nuevo if (n === 0 || n === 1) { return 1 } else { // Si el número es mayor que 1, llamamos a la función return n * factorial(n - 1) } }

console.log(factorial(5)) // Resultado: 120 console.log(factorial(3)) // Resultado: 6

```javascript
factorial(3) --------------------------> 6
        \
  3 * factorial(2) ---------------> 6
          \
    2 * factorial(1) -----------> 2
            \
      1 * factorial(0) -------> 1
```

La recursividad es un concepto que cuesta entender al principio. ¿Una función que se llama a sí misma? A mucha gente le vuela la cabeza.

¿Qué te recomiendo para aprenderlo? Primero, entiende los ejemplos que hemos hecho y dibuja en un papel cómo se ejecutan las funciones.

Después, haz estos ejercicios de recursividad:

Escribe una función que calcule la suma de los primeros n números enteros de forma recursiva. Por ejemplo: **suma(3) -> 1 + 2 + 3 = 6**

Escribe una función que calcule la sucesión de Fibonacci de forma recursiva. La sucesión de Fibonacci es una serie de números que empieza por 0 y 1 y cada número es la suma de los dos anteriores. Por ejemplo: **fibonacci(6) -> 8 (0, 1, 1, 2, 3, 5, 8)**

186 Quiz

¿Qué es la recursividad en programación?

Una técnica de programación que consiste en que una función se llame a sí misma

Un tipo de condición que permite evitar que una función se llame infinitamente

Un error que ocurre cuando una función se llama a sí misma

¿Qué es la condición base en la recursividad?

Una condición que permite evitar que una función se llame infinitamente

Una condición que permite que una función se llame infinitamente

Una condición que permite que una función se llame infinitamente

¿Qué sucede si no se define una condición base en una función recursiva?

La función se puede llamar infinitamente y el programa se bloquea

La función deja de llamarse después de un cierto número de iteraciones

La función devuelve un valor incorrecto

¿Cuál es la salida de este código al pasar como argumento el número 3?

```
function recursive(n) {
  if (n === 0) {
    return 0
  } else {
    return n + recursive(n - 1)
  }
}
```

0

6

4

187 Arrays: colecciones de elementos

Hasta ahora hemos visto algunos tipos de datos. Cadenas de texto, números, booleanos… Pero… **¿Qué pasa si queremos tener una colección de ellos?** En *JavaScript* tenemos los **arrays**.

188 Declaración y asignación de arrays

Para declarar un array usamos los corchetes [ ] y dentro los elementos de la colección separados por comas **,**.

Por ejemplo, para crear una colección de números del 1 al 5:

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

Los elementos de un array pueden ser de cualquier tipo, incluso otros arrays.

```
[1, 2, 3, 4, [5, 6, 7, 8, 9]]
```

Y, aunque no siempre sea recomendable, puedes mezclar tipos de datos dentro:

```
['uno', 2, true, null, undefined]
```

Para asignar un array a una variable, lo hacemos igual que con los otros tipos de datos:

```
const numbers = [1, 2, 3, 4, 5]
let names = ['Dani', 'Miguel', 'Maria']
```

189 Acceso a los elementos de un array

Para acceder a los elementos de un array usamos los corchetes [ ] y dentro el índice del elemento que queremos acceder. **Los índices empiezan en 0**.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers[0]) // 1
console.log(numbers[2]) // 3
```

> Tip
>
> El índice es la posición del elemento dentro del array. El primer elemento tiene índice 0, el segundo índice 1, el tercero índice 2…

Si intentamos acceder a un elemento que no existe, nos devolverá **undefined**.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers[10]) // undefined
```

Puedes usar variables para acceder a los elementos de un array.

```
const numbers = [1, 2, 3, 4, 5]
let index = 2

console.log(numbers[index]) // 3
```

190 Modificar elementos de un array

Igual que podemos acceder a los elementos de un array, podemos modificarlos.

```
const numbers = [1, 2, 3, 4, 5]

numbers[0] = 10
numbers[2] = 30

console.log(numbers) // [10, 2, 30, 4, 5]
```

> **Tip**
>
> ¿Cómo es que pese a que hemos indicado que es una constante **const** podemos modificar el Array? Aunque hay una explicación mucho más larga, que veremos más adelante, la respuesta corta es que **const** sólo impide que se reasigne el valor de la variable, no que se modifique el valor en sí.

191 Quiz

¿Qué se imprimirá en la consola al ejecutar el siguiente código:

```
const numbers = [1, 2, 3, 4, 5]
numbers[0] = 10
numbers[2] = 30
console.log(numbers)
```

[1, 2, 3, 4, 5]

[10, 2, 30, 4, 5]

Error de asignación a una constante

192 Métodos y propiedades de Array

Los arrays de JavaScript tienen una serie de métodos y propiedades que nos permiten trabajar con ellos de forma sencilla.

> **Tip**
>
> Un método es una función que se ejecuta sobre un objeto. Una propiedad es una variable que pertenece a un objeto. En este caso, el objeto es un array.

193 La longitud de un array

Puedes conocer la longitud de una colección de elementos usando la propiedad **.length**:

```
const frutas = ["manzana", "pera", "plátano", "fresa"]
console.log(frutas.length) // 4
```

También puedes cortar su longitud asignando un nuevo valor a la propiedad **.length**:

```
const frutas = ["manzana", "pera", "plátano", "fresa"]
frutas.length = 2

console.log(frutas) // ["manzana", "pera"]
console.log(frutas.length) // 2
```

194 Métodos de arrays

Cuando trabajamos con colecciones de elementos, vamos a querer hacer cosas con ellos. Por ejemplo: añadir un elemento, eliminarlo, buscarlo, etc. Para ello, los arrays tienen una serie de métodos que nos permiten hacer estas operaciones:

195 .push()

El método **.push()** nos permite añadir un elemento al final de un array:

```
const frutas = ["plátano", "fresa"]
frutas.push("naranja")
console.log(frutas) // ["plátano", "fresa", "naranja"]
```

Además, el método **.push()** devuelve la nueva longitud del array:

```
const frutas = ["plátano", "fresa"]
console.log(frutas.length) // 2

const newLength = frutas.push("naranja")
console.log(newLength) // 3
console.log(frutas) // ["plátano", "fresa", "naranja"]
```

196 .pop()

El método **.pop()** elimina y devuelve el último elemento de un array:

```
const frutas = ["plátano", "fresa", "naranja"]
const ultimaFruta = frutas.pop()

console.log(frutas) // ["plátano", "fresa"]
console.log(ultimaFruta) // "naranja"
```

197 .shift()

El método **.shift()** elimina y devuelve el primer elemento de un array. Es lo mismo que .pop(), pero con el primer elemento en lugar del último:

```
const frutas = ["plátano", "fresa", "naranja"]
const primeraFruta = frutas.shift()

console.log(frutas) // ["fresa", "naranja"]
console.log(primeraFruta) // "plátano"
```

198 .unshift()

El método **.unshift()** añade un elemento al principio de un array. Es lo mismo que **.push()**, pero con el primer elemento en lugar del último:

```
const frutas = ["plátano", "fresa", "naranja"]
frutas.unshift("manzana")

console.log(frutas) // ["manzana", "plátano", "fresa", "naranja"]
```

199 Concatenar arrays

Podemos concatenar dos arrays usando el método **concat()**.

```
const numbers = [1, 2, 3]
const numbers2 = [4, 5]

const allNumbers = numbers.concat(numbers2)

console.log(allNumbers) // [1, 2, 3, 4, 5]
```

Otra forma de concatenar arrays es usando el operador **...** (spread operator). Este operador propaga los elementos de un array. Así que podríamos hacer lo siguiente:

```
const numbers = [1, 2, 3]
const numbers2 = [4, 5]

//                    1, 2, 3        4, 5
const allNumbers = [...numbers, ...numbers2]

console.log(allNumbers) // [1, 2, 3, 4, 5]
```

Más adelante hablaremos más de este operador y sus diferentes utilidades.

200 Actividad Práctica

En un restaurante se reciben pedidos de comida a domicilio. Vamos a escribir una función **procesarPedido** que recibe un pedido, que es un array de platos. Lo que debemos hacer es:

- El primer elemento lo sacamos del array, ya que es el nombre del cliente.
- Añadimos al principio del array la cadena de texto "bebida", ya que es una promoción que tenemos.
- Después añadimos al final del array el nombre del usuario que sacamos antes.

Recuerda que debes devolver el array modificado:

```
function procesarPedido(pedidos) {
  // tu código aquí
}
```

Ver solución

```
function procesarPedido(pedidos) {
  const nombreCliente = pedidos.shift()
  pedidos.unshift("bebida")
  pedidos.push(nombreCliente)

  return pedidos
}
```

201 Iteración de Arrays en JavaScript

Ya hemos visto en clases anteriores cómo podemos crear bucles con **for** y **while**. Con estas estructuras de control, y una nueva que veremos en este clase, podemos también recorrer cada uno de los elementos de un array.

Además, veremos como los propios arrays tienen un método **.forEach** que nos permite ejecutar una función para cada uno de los elementos del array.

202 Bucle while

El bucle **while** vimos que permitía ejecutar un bloque de código mientras una condición era verdadera. En el caso de la iteración de arrays, la condición generalmente se basa en el índice del elemento.

Podemos, por ejemplo, crear una variable con **let** para guardar un puntero al índice del elemento que estamos iterando. En cada iteración, podemos incrementar el valor de la variable en 1, para que en la siguiente iteración se imprima el siguiente elemento.

```
let frutas = [' ', ' ', ' ']
let i = 0 // lo usaremos como índice

while (i < frutas.length) {
  console.log(frutas[i]) // imprime el elemento en la posición i
  i++ // incrementamos el índice en 1 para la siguiente iteración
}
```

203 Bucle for

El bucle **for** nos permitía ejecutar un bloque de código un número determinado de veces. En el caso de la iteración de arrays, podemos usarlo para recorrer cada uno de los elementos del array, usando la longitud del array como condición.

```
let frutas = [' ', ' ', ' ']

for (let i = 0; i < frutas.length; i++) {
  console.log(frutas[i]) // imprime el elemento en la posición i
}
```

También podrías recorrer el array en orden inverso, empezando desde el último elemento hasta el primero, usando **i–** en lugar de **i++**.

```
let frutas = [' ', ' ', ' ']

for (let i = frutas.length - 1; i >= 0; i--) {
  console.log(frutas[i]) // imprime el elemento en la posición i
}
```

204 Bucle for...of

Además de **while** y **for**, existe otra estructura de control que nos permite iterar sobre los elementos de un array: **for...of**. Esta estructura de control es más simple que **for**, ya que no necesitamos crear una variable para guardar el índice del elemento que estamos iterando.

Es mucho más sencilla y fácil de entender:

```
let frutas = [' ', ' ', ' ']

for (let fruta of frutas) {
  console.log(fruta) // imprime el elemento en la posición i
}
```

> Tip
>
> Como ves, hay algunas limitaciones en el uso de **for...of**. Por ejemplo, no podemos usarlo para recorrer un array en orden inverso y tampoco tenemos acceso al índice del elemento que estamos iterando.

205 El método array.forEach()

Ya hemos visto en la clase anterior que los arrays tienen métodos que nos permiten realizar operaciones sobre ellos, como añadir elementos o eliminarlos. Pero tiene muchos más métodos que iremos viendo que nos ayuda a trabajar con ellos.

Uno de esos métodos es **array.forEach()**, que nos permite ejecutar una **function** para cada uno de los elementos del array. Esta función recibe como parámetros el elemento que se está iterando en ese momento, el índice del elemento y el propio array.

```
let frutas = [' ', ' ', ' ']

frutas.forEach(function (fruta, index, originalArray) {
  console.log(fruta) // imprime el elemento en la posición i
})
```

Usando una **arrow function** e indicando sólo los parámetros que necesitamos de nuestra función podemos simplificarlo aún más:

```
let frutas = [' ', ' ', ' ']

frutas.forEach((fruta) => {
  console.log(fruta) // imprime el elemento en la posición i
})
```

> **Tip**
>
> Algo importante es que **array.forEach()** no devuelve nada, por lo que no podemos asignar el resultado a una variable.

¿Cuál usar? Depende...

Método	Acceso al Índice	Acceso al Valor	Puede Usar Break	Control Manual del Índice
for				
while				
for...of				
forEach				

> **Tip**
>
> Aunque técnicamente no puedes utilizar la sentencia **break** para salir de forEach, puedes simular esto lanzando una excepción y capturándola en un bloque **try/catch** externo. Obviamente es una mala práctica que no te recomiendo.

Cada uno de estos métodos tiene sus propios casos de uso. **for** y **while** son más generales y te permiten controlar el índice manualmente, lo que es útil si necesitas saltar a índices específicos, cambiar el orden de iteración o salir del bucle antes de que se complete.

for...of es más fácil de usar cuando solo necesitas los valores de un array y no te importa el índice. Es especialmente útil también cuando estás trabajando con iterables que no son arrays, como las cadenas de caracteres o los conjuntos (sets).

Finalmente, **forEach** es una forma muy eficiente y legible de iterar sobre un array si no necesitas **break** y si no necesitas controlar manualmente el índice.

206 Actividad Práctica

Dada una lista de números, escribe una función en JavaScript que devuelva la suma de todos los números pares en la lista. La función deberá iterar sobre cada número en la lista, comprobar si el número es par y, si es así, añadirlo a la suma total. Usa el bucle que quieras para solucionarlo.

```
function sumarPares(numeros) {
  let suma = 0
  // tu código aquí
  return suma
}
```

Ver solución

```
function sumarPares(numeros) {
  let suma = 0

  for (let numero of numeros) {
    if (numero % 2 === 0) {
      suma += numero
    }
  }

  return suma
}
```

207 Búsqueda en Arrays con sus métodos

Existen varios métodos en **JavaScript** que te permiten, específicamente, buscar dentro de un Array. Estos métodos incluyen **indexOf**, **some**, **every**, **find**, **findIndex** e **includes**.

Cada uno de estos métodos tiene un propósito específico y una funcionalidad única. Vamos a explorar cada uno de ellos utilizando un Array de emojis como ejemplo.

```
const emojis = [' ', ' ', ' ']
```

208 indexOf: ¿En qué posición está el elemento?

El método **indexOf** te permite encontrar la posición de un elemento dentro de un **Array**. Si el elemento no existe, entonces retorna **-1**.

```
const emojis = [' ', ' ', ' ']

const posicionCorazon = emojis.indexOf(' ')

console.log(posicionCorazon) // -> 2
```

209 includes: ¿El elemento existe en el Array?

El método includes determina si un Array incluye un determinado elemento, devolviendo **true** o **false** según corresponda.

```
const emojis = [' ', ' ', ' ']

const tieneCorazon = emojis.includes(' ')

console.log(tieneCorazon) // -> true
```

Es la forma más sencilla y corta de buscar un elemento específico dentro de un **Array**. Sin embargo, si queremos revisar si un **Array** contiene un elemento que cumpla con una condición, entonces tenemos que utilizar otros métodos…

> Tip
>
> El método **.includes()** también funciona con las cadenas de texto. Puedes utilizarlo para buscar una subcadena dentro de una cadena de texto: **'Hola mundo'.includes('Hola') // -> true**

210 some: ¿Alguno de los elementos cumple con la condición?

El método **some** te permite verificar si al menos uno de los elementos de un **Array** cumple con una condición.

Para utilizarlo, le pasas una función como argumento. Esta función recibe como argumento cada uno de los elementos del **Array** y debe retornar un valor **booleano**.

Si al menos uno de los elementos retorna **true**, entonces el método **some** retorna **true**. Si ninguno de los elementos retorna **true**, entonces el método **some** retorna **false**.

```
const emojis = [' ', ' ', ' ']

const tieneCorazon = emojis.some(emoji => emoji === ' ')
console.log(tieneCorazon) // -> true
```

¡Pero si esto ya lo podíamos hacer con **includes**! Sí, pero **some** es mucho más potente…

Podemos crear funciones más complejas para pasarle al método **some**. Por ejemplo, podemos crear una función que verifique si un **Array** contiene un elemento que sea un **string** de más de 3 caracteres.

```
const names = ['Leo', 'Isa', 'Ían', 'Lea']

const tieneNombreLargo = names.some(name => name.length > 3)
console.log(tieneNombreLargo) // -> false
```

Algo importante a tener en cuenta es que el método **some** deja de iterar sobre el **Array** en cuanto encuentra un elemento que cumple con la condición. Por ejemplo, si tenemos un **Array** de 10 elementos y el elemento número 3 cumple con la condición, el método **some** no va a iterar sobre los 7 elementos restantes:

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

const tieneNumeroMayorA5 = numbers.some(number => {
  console.log(`Estoy iterando sobre el número ${number}`) // -> Imprime hasta el número 6
  return number > 5
})

console.log(tieneNumeroMayorA5) // -> true
```

211 every: ¿Todos los elementos cumplen con la condición?

El método **every** te permite verificar si todos los elementos de un **Array** cumplen con una condición. Es similar a **some**, pero en lugar de verificar si al menos uno de los elementos cumple con la condición, los verifica todos.

Para utilizarlo, le pasas una función como argumento. Esta función recibe como argumento el elemento del array que está iterando en ese momento y debe retornar un valor **booleano** para saber si el elemento cumple con la condición.

Si todos los elementos retornan **true**, entonces el método **every** retorna true. Si al menos uno de los elementos retorna **false**, entonces el método **every** retorna **false**.

```
// ¿Todos los emojis son felices?
const emojis = [' ', ' ', ' ', ' ', ' ', ' ']
const todosSonFelices = emojis.every(emoji => emoji === ' ')
console.log(todosSonFelices) // -> false

// ¿Todos los números son pares?
const numbers = [2, 4, 7, 10, 12]
const todosSonPares = numbers.every(number => number % 2 === 0)
console.log(todosSonPares) // -> false

// ¿Todos los strings son mayores a 3 caracteres?
const names = ['Miguel', 'Juan', 'Itziar', 'Isabel']
const todosLosNombresSonLargos = names.every(name => name.length > 3)
console.log(todosLosNombresSonLargos) // -> true
```

Al igual que **some**, el método **every** deja de iterar sobre el **Array** en cuanto encuentra un elemento que no cumple con la condición.

212 find: Devuelve el primer elemento que cumple con la condición

El método **find** te permite encontrar el primer elemento que cumple con una condición. Lo interesante es que este método te devuelve el elemento en sí, no un valor **booleano** como **some** y **every**. Aunque el funcionamiento es igual: hay que pasarle una función como argumento que retorne un valor **booleano**.

```
const numbers = \[13, 27, 44, -10, 81\]
// encuentra el primer número negativo
const firstNegativeNumber = numbers.find(number =\> number \< 0)

console.log(firstNegativeNumber) // -\> -10
```

Si no encuentra ningún elemento que cumpla con la condición, el método **find** retorna **undefined**.

```
const numbers = [13, 27, 44, 81]
// encuentra el primer número negativo
const firstNegativeNumber = numbers.find(number => number < 0)

console.log(firstNegativeNumber) // -> undefined
```

De nuevo, igual que **some** y **every**, el método **find** deja de iterar sobre el **Array** en cuanto encuentra un elemento que cumple con la condición.

213 findIndex: Devuelve el índice del primer elemento que cumple con la condición

El método **findIndex** es similar a **find**, pero en lugar de devolver el elemento que cumple con la condición, devuelve el índice de ese elemento.

```
const numbers = [13, 27, 44, -10, 81]

// encuentra el índice del primer número negativo
const firstNegativeNumberIndex = numbers.findIndex(number => number < 0)

console.log(firstNegativeNumberIndex) // -> 3

// ahora puedes usar el índice para acceder al elemento
console.log(numbers[firstNegativeNumberIndex]) // -> -10
```

Si no encuentra ningún elemento que cumpla con la condición, el método **findIndex** retorna **-1**.

```
const numbers = [13, 27, 44, 81]

// encuentra el índice del primer número negativo
const firstNegativeNumberIndex = numbers.findIndex(number => number < 0)

console.log(firstNegativeNumberIndex) // -> -1
```

214 Actividad Práctica

Crear una función que reciba un array de palabras y devuelva **true** si todas las palabras terminan con la letra "a" y **false** si al menos una palabra no termina con la letra "a".

Usa el método endsWith() de string para resolverlo.

```
function acabanEnA(words) {
  // tu código aquí
}
```

Ver solución

```
function acabanEnA(words) {
  return words.every(word => word.endsWith('a'))
}
```

215 Ordenamiento de Arrays en JavaScript

JavaScript proporciona un método incorporado en arrays llamado **.sort()** para ordenar sus elementos. Por defecto no es muy útil pero puedes personalizar su comportamiento para que ordene los elementos de la manera que necesites.

216 Ordenamiento básico con sort()

Por defecto, **.sort()** hace una ordenación un poco extraña:

```
let numeros = [5, 10, 2, 25, 7]
numeros.sort()
console.log(numeros) // [10, 2, 25, 5, 7]
```

¿Qué ha pasado? **Este comportamiento puede resultar confuso cuando se trabaja con números**, ya que **.sort()** ordenará los números en función de su valor como cadena de texto, no de su valor numérico.

Por suerte, podemos arreglarlo…

217 Ordenamiento personalizado con sort()

Para personalizar cómo **sort()** ordena los elementos, puedes pasar una función de comparación como argumento. La función de comparación debe devolver:

- Un **valor negativo** si el primer argumento debe aparecer antes que el segundo.
- Un **valor positivo** si el segundo argumento debe aparecer antes que el primero.
- **Cero** si ambos argumentos son iguales.

Por ejemplo, **para ordenar los números de menor a mayor**, puedes usar la siguiente función de comparación:

```
let numeros = [5, 10, 2, 25, 7]

numeros.sort(function(a, b) {
  return a - b
})

console.log(numeros) // [2, 5, 7, 10, 25]
```

Para ordenarlos de manera descendente, de mayor a menor, deberías cambiar el orden del **return**:

```
let numeros = [5, 10, 2, 25, 7]

numeros.sort(function(a, b) {
  return b - a
})

console.log(numeros) // [25, 10, 7, 5, 2]
```

Puedes usar una **arrow function** para simplificar el código:

```
let numeros = [5, 10, 2, 25, 7]

numeros.sort((a, b) => a - b)
```

218 sort() y toSorted()

Como ves, **.sort()** modifica el array original. Si quieres obtener un array ordenado sin modificar el original, puedes usar el método **.toSorted()**. Sólo ten en cuenta que, ahora mismo, su soporte en navegadores es limitado.

```
let numeros = [5, 10, 2, 25, 7]

let numerosOrdenados = numeros.toSorted((a, b) => {
  return a - b
})

console.log(numerosOrdenados) // [2, 5, 7, 10, 25]
console.log(numeros) // [5, 10, 2, 25, 7]
```

También podrías usar el operador de propagación **(...)** para crear una copia del array original y ordenarla:

```
let numeros = [5, 10, 2, 25, 7]

const copiaNumeros = [...numeros]
// ordenamos la copia y no el original
copiaNumeros.sort((a, b) => a - b)

console.log(copiaNumeros) // [2, 5, 7, 10, 25]
console.log(numeros) // [5, 10, 2, 25, 7]
```

219 Actividad Práctica

Recibes una lista de números. Debes ordenar los números de menor a mayor según su valor absoluto. Eso quiere decir que los números negativos pierden el signo y se ordenan como si fueran positivos.

Por ejemplo, si recibes [**5, -10, -2, -25, -7**] deberías devolver [**-2, 5, -7, -10, -25**].

Puedes usar el método **Math.abs(num)** para obtener el valor absoluto de un número.

```
function sortAbsoluteNumbers(numbers) {
  // tu código aquí
  }
```

Ver solución

```
function sortAbsoluteNumbers(numbers) {
  return numbers.sort((a, b) => Math.abs(a) - Math.abs(b))
}
```

220 Transformación de Arrays en JavaScript

Ya hemos visto cómo podemos crear arrays, modificarlos y buscar en ellos. Ahora vamos a ver cómo podemos transformarlos. Esto es, cómo podemos crear nuevos arrays a partir de los que ya tenemos.

221 filter: Un nuevo Array con los elementos que cumplan una condición

El método filter crea un nuevo array con todos los elementos que devuelven **true** al ejecutar una función que le pasamos como parámetro.

Por ejemplo, podríamos filtrar en un nuevo array todos los números pares de un array de números:

```
const numbers = [1, 2, 3, 4, 5, 6, 7]

const evenNumbers = numbers.filter(function (number) {
  return number % 2 === 0
})

console.log(evenNumbers) // [2, 4, 6]
```

O quedarnos con todas las cadenas de texto que tengan la letra **a**:

```
const strings = ['hola', 'adiós', 'casa', 'coche', 'perro', 'gato']

// en las cadenas de texto podemos usar el método `includes`
// para saber si una cadena contiene otra:
const stringsWithA = strings.filter(string => string.includes('a'))

console.log(stringsWithA) // ['hola', 'adiós', 'casa', 'gato']
```

222 map: Un nuevo Array con los elementos transformados

El método **map** crea un nuevo array de la misma longitud que el original, pero con los elementos transformados por una función que le pasamos como parámetro. Así que la función recibirá cada elemento del array y tendremos que devolver el elemento transformado.

Por ejemplo, podríamos crear un array con el doble de cada número de un array de números:

```
const numbers = [1, 2, 3]

const doubleNumbers = numbers.map((number) => {
  return number * 2
})

console.log(doubleNumbers) // [2, 4, 6]
```

O podríamos crear un array con la longitud de cada cadena de texto de un array de cadenas:

```
const strings = ['hola', 'javascript', 'midu', 'programación']

const stringsLength = strings.map((string) => string.length)

console.log(stringsLength) // [4, 10, 4, 12]
```

223 map + filter: Un nuevo Array con los elementos transformados y filtrados

En JavaScript podemos crear un encadenamiento de métodos. De esta forma, si un método devuelve un array, podemos llamar a otro método sobre ese array sin necesidad de guardarlo en una variable.

Imagina que queremos crear un array con el doble de los números y sólo quedarnos con los que sean mayores que 5. Podríamos hacerlo así:

```
const numbers = [1, 2, 3, 4, 5, 6, 7]

const numsGreaterThanFive = numbers
  .map(number => number * 2) // [2, 4, 6, 8, 10, 12, 14]
  .filter(number => number > 5) // [6, 8, 10, 12, 14]

console.log(numsGreaterThanFive) // [6, 8, 10, 12, 14]
```

También podríamos hacerlo al revés, primero filtrando y luego transformando. Por ejemplo, filtrar los números pares y luego multiplicarlos por 2:

```
const numbers = [1, 2, 3, 4, 5, 6, 7]

const doubleEvenNumbers = numbers
  .filter(number => number % 2 === 0) // [2, 4, 6]
  .map(number => number * 2) // [4, 8, 12]

console.log(doubleEvenNumbers) // [4, 8, 12]
```

224 reduce: Un único valor a partir de un array

El método **reduce** es un poco más complejo que los anteriores. Este método te permite crear un único valor a partir de un Array.

Recibe dos parámetros: una función que se ejecutará por cada elemento y un valor inicial, opcional, que será donde podremos acumular los valores.

El primer parámetro de **reduce**, que es la función que se ejecutará por cada elemento, recibe tres parámetros:

- **El acumulador:** el valor que se va a ir acumulando en cada iteración
- **El elemento actual:** el elemento del Array que estamos iterando en ese momento.
- **El índice:** la posición del elemento actual en el Array.

Y debe devolver el valor que se va a acumular en cada iteración.

Un caso de uso muy típico de **reduce** es sumar todos los elementos de un Array:

```
const numbers = [1, 2, 3]

const sum = numbers.reduce((accumulator, currentNumber) => {
  return accumulator + currentNumber
}, 0) // <- el 0 es el valor inicial

console.log(sum) // 6
```

Vamos a ver qué está pasando en cada iteración:

- **Iteración 1:** el acumulador vale 0 (ya que 0 es el valor inicial) y el elemento actual vale 1. Así que devolvemos $0 + 1 = 1$
- **Iteración 2:** el acumulador vale 1 (el valor que devolvimos en la iteración anterior) y el elemento actual vale 2. Así que devolvemos $1 + 2 = 3$
- **Iteración 3:** el acumulador vale 3 (el valor que devolvimos en la iteración anterior) y el elemento actual vale 3. Así que devolvemos $3 + 3 = 6$

Por lo tanto, el valor final de **sum** es 6.

¿Recuerdas que hemos visto antes cómo usar **map** y **filter** juntos? Pues podrías usar **reduce** para tener que evitar crear un array intermedio. Por ejemplo, imagina que queremos doblar los números pares y sólo quedarnos con los mayores de 5 (como hicimos en el ejemplo anterior):

```
const numbers = [1, 2, 3, 4, 5, 6, 7]

const doubleEvenNumbers = numbers.reduce((accumulator, currentNumber) => {
  const isEven = currentNumber % 2 === 0
  const doubled = currentNumber * 2
  const isGreaterThanFive = doubled > 5

  // si es par y mayor que 5, lo añadimos al array
  if (isEven && isGreaterThanFive) {
    // para ello devolvemos un nuevo array con el valor actual
    return accumulator.concat(doubled)
  } else {
    // si no, devolvemos lo que ya teníamos
    return accumulator
  }
}, []) // <- el array vacío es el valor inicial

console.log(doubleEvenNumbers) // [8, 12]
```

Lo interesante en este caso es que no hemos tenido que crear un array intermedio con **map** y **filter**. Hemos ido acumulando los valores que nos interesaban en cada iteración y los que no nos interesaban los hemos descartado devolviendo lo que ya teníamos en el acumulador.

> **Tip**
>
> Ten cuidado con reduce. Como puedes ver, su carga cognitiva es mayor que la de map y filter. Así que si puedes usar map y filter para hacer lo mismo, hazlo. Sólo usa reduce cuando no te quede más remedio.

225 Actividad Práctica

Recibes dos parámetros: una lista de palabras **words** y una palabra **word**. Primero, busca el índice de la palabra en la lista. Después, usa ese índice (que será un número) y devuelve todas las palabras de **words** que sean más largas (**length**) que el número del índice.

Ten en cuenta que la palabra **word** siempre existirá en el array, por lo que no es necesario comprobar si existe o no.

```
function buscaPalabras(words, word) {
  // tu código aquí
}
```

Ver solución

```
function buscaPalabras(words, word) {
  const index = words.indexOf(word)
  return words.filter(word => word.length > index)
}
```

226 Matrices

En matemáticas y en programación, una matriz se refiere a una colección de elementos dispuestos en filas y columnas. En términos más sencillos, y en JavaScript, podemos representar una matriz como un array de arrays.

226.1 Creación de Matrices

Para empezar, aquí se muestra cómo se puede crear una matriz simple de 2x3 en JavaScript:

```
const matriz = [
  [1, 2, 3],
  [4, 5, 6]
]
```

> **Tip**
>
> Las matrices no tienen por qué ser cuadradas. Pueden tener cualquier número de filas y columnas.

227 Acceso a los Elementos de una Matriz

Para acceder a los elementos de una matriz, necesitaremos utilizar dos índices: uno para la *fila* y otro para la *columna*.

Por ejemplo, si queremos acceder al número **6** en la matriz anterior, haríamos lo siguiente:

```
let numero = matriz[1][2]
console.log(numero) // -> 6
```

Aquí, el primer índice, [**1**], se refiere a la segunda fila (recuerda que los arrays en JavaScript tienen un índice base cero), y el segundo índice, [**2**], se refiere a la tercera columna.

228 Iteración sobre Matrices

Podemos utilizar bucles anidados para iterar sobre todos los elementos de una matriz. Aquí hay un ejemplo de cómo hacerlo utilizando un bucle for:

```
for (let i = 0; i < matriz.length; i++) { // {1}
  for (let j = 0; j < matriz[i].length; j++) { // {2}
    console.log(matriz[i][j])
  }
}
```

En este ejemplo, el bucle externo recorre **{1}** cada fila de la matriz, mientras que el bucle interno **{2}** recorre cada elemento dentro de una fila específica.

Puedes usar otros bucles para iterar sobre una matriz, como un bucle while o un bucle for…of. También puedes utilizar el método **forEach()** de los arrays para iterar sobre una matriz, pero tendrás que anidar dos llamadas a **forEach()** para iterar sobre todos los elementos de la matriz.

```
matriz.forEach((fila, filaIndex) => {
  fila.forEach((elemento, columnaIndex) => {
    console.log('Fila ' + filaIndex)
    console.log('Columna ' + columnaIndex)
    console.log('Elemento ', elemento)
    console.log('')
  })
})
```

229 Ejemplo práctico de Matrices

Un ejemplo sencillo de entender pero muy útil es el juego del 3 en raya. En este juego, los jugadores se turnan para colocar sus fichas en una cuadrícula de **3x3**.

El objetivo del juego es colocar tres fichas en una fila, columna o diagonal.

Para representar el tablero de juego, podemos utilizar una matriz de **3x3**. Cada elemento de la matriz puede ser una cadena que represente una ficha:

```
const tablero = [
  ['X', 'O', 'X'],
  ['O', 'X', 'O'],
  ['O', 'O', 'X']
]
```

Ahora, para **buscar si hay un ganador en el tablero**, podemos utilizar un bucle **for** para iterar sobre cada fila y columna del tablero.

Si encontramos tres fichas iguales en una fila, columna o diagonal, entonces tenemos un ganador.

```
// Verifica las líneas horizontales
for (let i = 0; i < 3; i++) {
  if (
    tablero[i][0] === tablero[i][1] &&
    tablero[i][1] === tablero[i][2]
  ) {
    console.log(`El ganador es: ${tablero[i][0]}`);
  }
}

// Verifica las líneas verticales
for (let i = 0; i < 3; i++) {
  if (
    tablero[0][i] === tablero[1][i] &&
    tablero[1][i] === tablero[2][i]
  ) {
    console.log(`El ganador es: ${tablero[0][i]}`);
  }
}

// Verifica la diagonal de arriba izquierda a derecha abajo
```

```
if (
  tablero[0][0] === tablero[1][1] &&
  tablero[1][1] === tablero[2][2]
) {
  console.log(`El ganador es: ${tablero[0][0]}`);
}

// Verifica la diagonal de derecha arriba a izquierda abajo
if (
  tablero[0][2] === tablero[1][1] &&
  tablero[1][1] === tablero[2][0]
) {
  console.log(`El ganador es: ${tablero[0][2]}`);
}
```

Las matrices son muy útiles y necesarias para muchos juegos (ajedrez, damas, buscaminas, etc.) y aplicaciones (hojas de cálculo, bases de datos, etc.). Así que es importante que te familiarices con ellas.

230 Actividad Práctica

Recibimos una matriz de cadenas de texto. Tenemos que localizar la posición de la palabra **"JavaScript"** en la matriz y devolver su posición como un **array de dos elementos**: el índice de la fila y el índice de la columna.

```
const matrix = [
  ['HTML', 'CSS', 'JavaScript'],
  ['Java', 'C++', 'Python'],
  ['Ruby', 'Go', 'Swift']
]

const position = findJavaScript(matrix)
console.log(position) // -> [0, 2]
```

Si no encuentra la palabra debe devolver **[-1, -1]**. Usa el siguiente código como punto de partida:

```
function findJavaScript(matrix) {
  // tu código aquí
}
```

Ver solución

```
function findJavaScript(matrix) {
  for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
      if (matrix[i][j] === 'JavaScript') {
        return [i, j]
      }
    }
  }

  return [-1, -1]
}
```

231 Algoritmos con Arrays

Los *algoritmos* son una serie de **pasos que se realizan para resolver un problema**. En este caso, vamos a ver algunos *algoritmos* que se pueden realizar con arrays donde se pueden aplicar los conceptos de complejidad algorítmica (cuánto tarda un algoritmo en ejecutarse).

232 Algoritmo para encontrar el número mayor

El primer algoritmo que vamos a ver es el de encontrar el número mayor de un array. Para ello, vamos a crear una función que reciba un array de números y devuelva el número mayor.

```
function findMaxAlgorithm(array) {
  let max = array[0] // recuperamos el primer elemento del array

  // recorremos el array desde el segundo elemento
  for (let i = 1; i < array.length; i++) {
    // ¿es el elemento actual mayor que el máximo?
    if (array[i] > max) {
      // si es así, lo guardamos como nuevo máximo
      max = array[i]
    }
  }

  // devolvemos el máximo número que hemos encontrado
  return max;
}
```

> **Tip**
>
> En JavaScript podríamos usar el método **Math.max** para encontrar el número mayor de un array. Se usaría así: **Math.max(...array)**. Pero por motivos didácticos, hemos creado nuestro propio algoritmo. Siempre que puedas, te recomiendo usar **Math.max** en su lugar.

233 Complejidad algorítmica

Para calcular la complejidad algorítmica del algoritmo anterior, tenemos que tener en cuenta el número de operaciones que se realizan.

En nuestro ejemplo, para encontrar el mayor número de un array tenemos que recorrer todos los elementos del array. Si el array tiene 10 elementos, tendremos que recorrerlo 10 veces. Si tiene 100 elementos, tendremos que recorrerlo 100 veces. Es decir, **el número de operaciones que se realizan es igual al número de elementos del array**.

En programación se usa la notación O(n) para indicar que el número de operaciones que se realizan es igual al número de elementos del array. En este caso, n es el número de elementos del array.

234 Búsqueda binaria

Imagina que tienes un array de números ordenados de menor a mayor. Por ejemplo, [**11, 18, 23, 25, 48, 55, 88, 111, 120**]. Si te pido que busques el número **55** en el array, ¿cómo lo harías?.

Una forma de hacerlo sería recorrer el array desde el principio hasta el final y comprobar si el número que estamos buscando es igual al número actual. Si es así, hemos encontrado el número. Si no, seguimos buscando.

```
function findNumber(array, number) {
  for (let i = 0; i < array.length; i++) {
    if (array[i] === number) {
      return true
    }
  }

  return false
}
```

> Tip
>
> Obviamente, podríamos usar el método **.includes()** para comprobar si un array contiene un elemento. Pero por motivos didácticos, hemos creado nuestro propio algoritmo. Siempre que puedas, te recomiendo usar **.includes()** en su lugar.

La complejidad algorítmica de este algoritmo es O(n). Es decir, el número de operaciones que se realizan es igual al número de elementos del array. Pero podemos hacerlo mejor.

Existe un algoritmo muy popular llamado búsqueda binaria que nos permite encontrar un elemento en un array ordenado en O(log n). Es decir, el número de operaciones que se realizan es igual al logaritmo del número de elementos del array. ¿Qué significa esto?.

Imagina que tenemos un array de 10 elementos. Si aplicamos la búsqueda binaria, el número de operaciones que se realizan es igual a 3. Si tenemos un array de 100 elementos, el número de operaciones que se realizan es igual a 6. ¿Ves la diferencia? ¡Es mucho más rápido!

> Tip
>
> Para aplicar la búsqueda binaria, tenemos que tener en cuenta que el array tiene que estar ordenado de menor a mayor. Si no, no funcionará.

235 Implementación de la búsqueda binaria

La idea de la búsqueda binaria es ir dividiendo el array en dos mitades. Si el elemento que estamos buscando es mayor que el elemento de la mitad del array, tenemos que buscar en la mitad derecha. Si el elemento que estamos buscando es menor que el elemento de la mitad del array, tenemos que buscar en la mitad izquierda. Y así sucesivamente hasta encontrar el elemento que estamos buscando.

```
function busquedaBinaria(array, elemento) {
  let index = 0 // primer elemento del array
  let final = array.length - 1 // último elemento del array a buscar

  // mientras el índice sea menor o igual que el final
  // seguiremos buscando
  while (index <= final) {
    // calculamos la mitad del array
    // como puede ser impar, usamos Math.floor para redondear hacia abajo
    const mitad = Math.floor((index + final) / 2)

    // si el elemento de la mitad es igual al elemento que estamos buscando
    // entonces hemos encontrado el elemento
    if (array[mitad] === elemento) {
      return mitad
    } else if (array[mitad] < elemento) {
      // si el elemento de la mitad es menor que
      // el elemento que estamos buscando
      // entonces tenemos que buscar en la mitad derecha
      index = mitad + 1
    } else {
      // si el elemento de la mitad es mayor que
      // el elemento que estamos buscando
      // entonces tenemos que buscar en la mitad izquierda
      final = mitad - 1
    }
  }

  // si llegamos hasta aquí, es que no hemos encontrado el elemento
  // devolvemos -1, para indicar que no se ha encontrado
  return -1
}
```

Tip

En JavaScript podríamos usar el método **Array.prototype.indexOf** para encontrar el índice de un elemento en un array. Se usaría así: **array.indexOf(elemento)**. Pero por motivos didácticos, hemos creado nuestro propio algoritmo. Siempre que puedas, te recomiendo usar **Array.prototype.indexOf** en su lugar.

236 Actividad Práctica

En una biblioteca queremos saber qué libro es el que menos páginas tiene y el que más páginas. Por suerte, no hay dos libros con el mismo número de páginas.

Necesitamos que la función reciba un array de números, sin ordenar, y que devuelva un array de dos posiciones con el índice del libro con menos páginas y el índice del libro con más páginas.

```
function minAndMaxWord(words) {
  // tu código aquí
}
```

Ver solución

```
function minAndMaxWord(words) {
  let min = 0
  let max = 0

  for (let i = 1; i < words.length; i++) {
    if (words[i] < words[min]) {
      min = i
    }

    if (words[i] > words[max]) {
      max = i
    }
  }

  return [min, max]
}
```

237 Objetos

Los objetos en JavaScript son una colección de propiedades y métodos que definen un elemento a través de una clave y un valor.

Por ejemplo, si pensamos en una persona, podemos definir sus propiedades como su nombre, su edad, su altura, su peso, etc. Y sus métodos como caminar, correr, saltar, etc. Declaración y asignación de objetos

238 Declaración y asignación de objetos

Para declarar un objeto usamos las llaves **{}** y dentro las propiedades y métodos separados por comas **,**. Cada propiedad o método se define con una clave y un valor separados por dos puntos **:**.

Por ejemplo, vamos a crear un objeto que represente a una persona:

```
const persona = {
  name: 'Dani',
  age: 30,
  isWorking: true
}
```

Las propiedades y métodos de un objeto pueden ser de cualquier tipo de JavaScript, incluso otros objetos o arrays.

```
const persona = {
  name: 'Dani',
  age: 30,
  isWorking: true,
  family: ['Miguel', 'Maria'], // array
  address: { // otro objeto
    street: 'Calle de la piruleta',
    number: 13,
    city: 'Barcelona'
  }
}
```

Y, claro, como hemos comentado antes, también podemos tener funciones. **Cuando una función es parte de un objeto se le llama método**.

```
const persona = {
  name: 'Dani',
  age: 30,
  isWorking: true,
  family: ['Miguel', 'Maria'],
  address: {
    street: 'Calle de la piruleta',
    number: 13,
    city: 'Barcelona'
  },
```

```
  walk: function () { // <- método
    console.log('Estoy caminando')
  }
}
```

239 Acceder a propiedades y métodos de un objeto

Para acceder a las propiedades y métodos de un objeto usamos el punto . y el nombre de la propiedad o método.

```
const persona = { name: 'Dani' }
console.log(persona.name) // Dani
```

Si intentamos acceder a una propiedad o método que no existe, nos devolverá **undefined**.

```
const persona = { name: 'Dani' }
console.log(persona.fullName) // -> undefined
```

Puedes usar variables para acceder a las propiedades y métodos de un objeto. Para ello, tienes que usar la notificación por corchetes [].

```
const persona = { name: 'Dani' }
let property = 'name'

console.log(persona[property]) // -> Dani
```

También necesitas usar los corchetes si la clave de la propiedad o método tiene espacios o caracteres especiales.

```
const persona = { 'full name': 'Dani' }

console.log(persona['full name']) // -> Dani

//   no puedes hacer esto
// persona.full name
```

> **Tip**
>
> Siempre que puedas, evita usar espacios o caracteres especiales en las claves de las propiedades y métodos de un objeto. Aunque se puede, es más complicado de leer y de escribir.

Igual que las propiedades, puedes acceder a los métodos de un objeto usando cualquiera de las dos formas anteriormente comentadas:

```
const persona = {
  name: 'Dani',
  walk: function () {
    console.log('Estoy caminando')
  }
}

persona.walk() // -> Estoy caminando

let method = 'walk'
persona[method]() // -> Estoy caminando
```

> **Tip**
>
> **persona[method]()** parece un poco raro, pero si lo separas parte por parte, verás que tiene sentido. **persona** es el objeto. Accedemos a la propiedad **method** del objeto persona con **persona[method]**. Y, finalmente, ejecutamos la función con **persona[method]()**. Es decir, estamos ejecutando **persona.walk()**.

240 Añadir y modificar propiedades de un objeto

Igual que podemos acceder a las propiedades de un objeto, podemos añadir nuevas propiedades o modificar las existentes.

```
const persona = { name: 'Dani' }

persona.age = 30

console.log(persona) // -> { name: 'Dani', age: 30 }
```

En el código estamos modificando el objeto **persona** añadiendo la propiedad **age** con el valor **30**.

> Tip
>
> Fíjate que, al igual que en los Arrays, podemos añadir propiedades a un objeto aunque sea una constante. Esto es porque no estamos reasignando la variable **persona**, si no que estamos modificando una propiedad interna del objeto.

241 Eliminar propiedades de un objeto

Para eliminar una propiedad de un objeto usamos la palabra reservada **delete**.

```
const persona = { name: 'Dani', age: 18 }

delete persona.age

console.log(persona) // -> { name: 'Dani' }
```

242 Actividad Práctica

Tenemos una función que recibe dos parámetros. **name** y **subs**. Haz que la función devuelva un objeto con la siguiente información:

- **name** con el valor del parámetro **name**
- **subscribers** con el valor del parámetro **subs**
- **hash**, con el valor de la longitud del string **name** multiplicado por el parámetro **subs**
- Un método **getStatus** que devuelva el texto **El canal de <name> tiene <subs> suscriptores**. Por ejemplo, para name = * 'Dani' y subs = 100, el método **getStatus** devolvería **El canal de Dani tiene 100 suscriptores**.

¡Ojo! El método **getStatus** debe devolver el texto, NO imprimirlo por consola.

```
function createObject(name, subs) {
  // tu código aquí
}
```

Ver solución

```
function createObject(name, subs) {
  return {
    name: name,
    subscribers: subs,
    hash: name.length * subs,
    getStatus: function () {
      return `El canal de ${this.name} tiene ${this.subscribers} suscriptores`
    }
  }
}
```

243 Atajos para trabajar con objetos

Trabajar con objetos es muy común en JavaScript. En esta clase vamos a ver algunos atajos que nos permiten trabajar con ellos de forma más cómoda, tanto a la hora de crearlos como a la hora de recuperar sus propiedades.

243.1 Atajo al crear un objeto

Imagina que quieres crear un objeto y que algunas de sus propiedades usen como valor algunas variables que ya tenemos.

```
const name = 'Spidey'
const universe = 42

const spiderman = {
  name: name,
  universe: universe,
  powers: ['web', 'invisibility', 'spider-sense']
}
```

En este caso, podemos usar un atajo para crear el objeto. **Si la propiedad y la variable tienen el mismo nombre**, podemos omitir el valor y dejar solo el nombre de la propiedad.

```
const name = 'Spidey'
const universe = 42

const spiderman = {
  name, // <- automáticamente asigna el valor de la variable name
  universe, // <- automáticamente asigna el valor de la variable universe
  powers: ['web', 'invisibility', 'spider-sense']
}
```

Fíjate que si tuvieramos **powers** fuera del objeto, podríamos crear el objeto en la misma línea:

```
const name = 'Spidey'
const universe = 42
const powers = ['web', 'invisibility', 'spider-sense']

const spiderman = { name, universe, powers }
```

244 Destructuración: el atajo al recuperar propiedades

En la anterior clase hemos visto que para recuperar una propiedad de un objeto podemos usar la notación de corchetes o la notación de punto:

```
const spiderman = {
  name: 'Spidey',
  universe: 42,
  powers: ['web', 'invisibility', 'spider-sense']
}

console.log(spiderman['name']) // Spidey
console.log(spiderman.name) // Spidey
```

A veces queremos recuperar el valor de la propiedad y guardarlo en una variable, para usarlo más adelante. Haríamos algo así:

```
const universe = spiderman.universe
console.log(universe) // 42

const powers = spiderman['powers'][1]
console.log(powers) // 'invisibility'
```

> Tip
>
> **spiderman['powers'][1]** de nuevo puede parecer un poco confuso, pero hay que separarlo en partes. **spiderman** es el objeto. Accedemos a la propiedad **powers** y como es un array, podemos acceder a su segundo elemento con **[1]**. Y ahí está el valor que queremos: **invisibility**.

A la hora de acceder a las propiedades existe un atajo llamado **desestructuración** (o destructuring en inglés) muy interesante que nos permite recuperar el valor de una propiedad y guardarlo en una variable de una.

Lo importante es que tengamos en cuenta que **la variable que se creará tendrá el mismo nombre que la propiedad que estamos recuperando**.

```
const { universe } = spiderman
console.log(universe) // 42
```

Además, puedes recuperar tantas propiedades como quieras, separándolas por comas.

```
const { universe, name, powers } = spiderman
console.log(universe) // 42
console.log(name) // 'Spidey'
console.log(powers) // ['web', 'invisibility', 'spider-sense']
```

245 Renombrar variables y valores por defecto

Si quieres que la variable que se crea tenga un **nombre diferente al de la propiedad**, puedes hacerlo así:

```
const { universe: universeNumber } = spiderman
console.log(universeNumber) // 42
```

Lo que estamos haciendo es recuperar la propiedad **universe** pero crear la constante con el nombre **universeNumber**.

Si la propiedad no existe, podemos asignarle un valor por defecto:

```
const { name, isAvenger = false } = spiderman
console.log(name) // 'Spidey'
console.log(isAvenger) // false
```

El objeto que hemos creado antes no tenía la propiedad **isAvenger**, pero al recuperarla le hemos asignado un valor por defecto de **false**. Si no le hubiéramos asignado un valor por defecto, la variable **isAvenger** sería **undefined**.

Así que si recuperamos una propiedad que no existe, la variable que se creará tendrá el valor por defecto que le hayamos asignado. Y lo hacemos usando el operador =.

> Tip
>
> Esto es muy útil, por ejemplo, para recuperar opciones o configuración del usuario que quizás no ha definido todos los valores pero tenemos claro qué comportamiento queremos que tenga nuestra aplicación por defecto.

246 Objetos anidados y atajo

Ya sabemos que podemos tener un objeto dentro de un objeto:

```
const spiderman = {
  name: 'Spidey',
  universe: 42,
  powers: ['web', 'invisibility', 'spider-sense'],
  partner: {
    name: 'Mary Jane',
    universe: 42,
    powers: ['red hair', 'blue eyes']
  }
}
```

Para acceder a las propiedades de un objeto anidado, podemos usar la notación de corchetes o la notación de punto:

```
console.log(spiderman.partner.name) // 'Mary Jane'
console.log(spiderman['partner']['name']) // 'Mary Jane'
```

Si queremos recuperar la propiedad **name** del objeto **partner** y guardarla en una variable, podemos hacerlo así:

```
const { partner } = spiderman
const { name } = partner
console.log(name) // 'Mary Jane'
```

Pero también podemos hacerlo en una sola línea:

```
const { partner: { name } } = spiderman
console.log(name) // 'Mary Jane'
```

Lo que estamos haciendo es: del objeto **spiderman** extrae la propiedad **partner**, y de esta, extrae la propiedad **name**. Como resultado sólo crearemos la variable **name** con el valor **'Mary Jane'**.

247 Quiz

¿Cómo se llama el atajo que nos permite recuperar el valor de una propiedad de un objeto?

dot notation o notación de punto

desestructuración o destructuring

extracting o extracción

¿Qué se imprimirá en la consola al ejecutar el siguiente código?

```
const person = {
  name: 'John',
  age: 30,
  hobbies: ['reading', 'music', 'sports']
}

const { age: personAge } = person
console.log(age)
```

30

ReferenceError: age is not defined

undefined

¿Qué se imprimirá en la consola al ejecutar el siguiente código?

```
const person = {
  name: 'John',
  age: 30,
  hobbies: ['reading', 'music', 'sports']
}

const {
  weight = 70,
  height = 180,
  } = person

console.log(weight, height)
```

70 180

ReferenceError: weight is not defined

undefined undefined

248 Iterando Objetos en JavaScript

El manejo de objetos es uno de los pilares en JavaScript y es crucial conocer cómo podemos iterar o recorrer los mismos. Existen varias formas de hacerlo y en esta clase vamos a revisar algunas de las más comunes: el bucle **for...in** y los métodos **Object.keys()**, **Object.values()** y **Object.entries()**.

248.1 Iterando con for...in

La estructura de control **for...in** nos permite crear un bucle que itera sobre todas las propiedades enumerables de un objeto, en un orden arbitrario.

```
const spiderman = {
  name: 'Spidey',
  universe: 42,
  powers: ['web', 'invisibility', 'spider-sense']
}

for (const property in spiderman) {
  console.log(`${property}: ${spiderman[property]}`);
}

// -> name: Spidey
// -> universe: 42
// -> powers: web,invisibility,spider-sensibles
```

> **Tip**
>
> No confundas el bucle **for...in** con el de **for...of** que vimos en la sección de arrays. El primero itera sobre las propiedades enumerables de un objeto, mientras que el segundo itera sobre los elementos de un objeto iterable (como los elementos de un array, por ejemplo).

249 Transformar un objeto en un array

A veces queremos transformar un objeto en un array para poder iterar sobre él. Para esto podemos utilizar el método **Object.keys(), Object.values()** o **Object.entries()**.

Cada uno de estos métodos retorna un array, pero cada uno de ellos contiene elementos diferentes. Vamos a verlos uno por uno.

249.1 Iterar con Object.keys()

El método **Object.keys()** retorna un array con las propiedades enumerables de un objeto.

```
const spiderman = {
  name: 'Spidey',
  universe: 42,
  powers: ['web', 'invisibility', 'spider-sense']
}

const properties = Object.keys(spiderman)

console.log(properties.length) // 3

properties.forEach(property => {
  console.log(property)
})

// -> name
// -> universe
// -> powers
```

Este código hace algo muy similar al anterior, pero utilizando **Object.keys()** para obtener las propiedades del objeto.

> Tip
>
> **¿Qué son las propiedades enumerables?** Por defecto, cuando añadimos propiedades a un objeto, estas son enumerables, es decir, que se pueden iterar sobre ellas. Sin embargo, como veremos más adelante, podemos crear propiedades que, por lo que sea, nos interesa que no sean enumerables. A veces estas son más de uso interno, como métodos auxiliares que el propio lenguaje JavaScript tiene, y no

queremos que aparezcan cuando iteramos sobre un objeto.

249.2 Iterando con Object.values()

El método **Object.values()** retorna un array con los valores correspondientes a las propiedades enumerables de un objeto.

```
const values = Object.values(spiderman)

values.forEach(value => {
  console.log(value)
})

// -> Spidey
// -> 42
// -> [ 'web', 'invisibility', 'spider-sense' ]
```

Como ves, la diferencia con **Object.values()** es que este método retorna los valores y **Object.keys()** retorna las propiedades.

249.3 Iterando con Object.entries()

El método **Object.entries()** retorna un array de arrays, donde cada subarray es un par [propiedad, valor] correspondiente a las propiedades enumerables de un objeto.

```
const entries = Object.entries(spiderman)

entries.forEach(entry => {
  console.log(entry)
})

// -> [ 'name', 'Spidey' ]
// -> [ 'universe', 42 ]
// -> [ 'powers', [ 'web', 'invisibility', 'spider-sense' ] ]
```

Lo interesante es que este método nos da acceso tanto a las propiedades como a los valores. Lo puedes ver más claro en el siguiente ejemplo:

```
const entries = Object.entries(spiderman)

entries.forEach(entry => {
  const property = entry[0]
  const value = entry[1]
```

```
  console.log(`${property}: ${value}`)
})
```

250 Actividad Práctica

Tienes una función que recibe un objeto como parámetro. La función debe retornar un array con el nombre de las propiedades que su tipo sea **boolean**.

Por ejemplo, para el objeto **{ a: true, b: 42, c: false }** la función debe retornar [**'a', 'c'**] ya que son las dos propiedades que tienen valores booleanos.

```
function getKeysOfBooleanValues(obj) {
    // tu código aquí
  }
```

Ver solución

```
function getKeysOfBooleanValues(obj) {
  return Object.keys(obj).filter(key => typeof obj[key] === 'boolean')
}
```

251 Operador de Encadenamiento Opcional

Si intentamos acceder a una propiedad de un objeto que no existe… **¡se romperá nuestra aplicación!**

Por suerte, existen formas de evitar que esto ocurra. En esta clase veremos algunas de ellas y presentaremos **el operador de Encadenamiento Opcional**, también llamado *Optional Chaining*.

Este operador es especialmente útil cuando trabajamos con objetos que podrían tener estructuras complejas o variables que podrían no estar definidas en determinados momentos de la ejecución de nuestro código.

252 El error fatal de intentar leer una propiedad de undefined

Uno de los errores más comunes en JavaScript, y que estoy seguro que verás tarde o temprano, es el que dice **que no puedes leer una propiedad de undefined**.

Este error ocurre cuando intentamos acceder a una propiedad de un objeto que no existe. Por ejemplo:

```
const gamesystem = {
  name: 'PS5',
  price: 550,
  specs: {
    cpu: 'AMD Ryzen Zen 2',
    gpu: 'AMD Radeon RDNA 2',
  }
}

console.log(gamesystem.name) // -> PS5

console.log(gamesystem.specifications) // -> undefined

console.log(gamesystem.specifications.ram)
//  Uncaught TypeError: Cannot read property 'ram' of undefined
```

El error parece inocente pero fíjate. En lugar de usar la propiedad **specs** hemos usado **specifications**. Pero esto es suficiente para que JavaScript nos diga que no puede leer la propiedad **ram** de **undefined**.

253 Evitar el error con if

Una forma de evitar este error es usando un **if** para comprobar si la propiedad existe antes de intentar acceder a ella:

```
if (typeof gamesystem.specifications === 'object') {
  console.log(gamesystem.specifications.ram)
} else {
  console.log('No hay especificaciones')
}
```

> Tip
>
> ¡Cuidado con esto! Al usar el operador **typeof** con un valor **null** nos dice que es **object**. Esto es un error histórico del lenguaje. Así que si la propiedad **specifications** es **null**, el **if** se ejecutará y obtendremos un error al intentar acceder a **ram**. Para evitar esto, podemos añadir una comprobación extra: **gamesystem.specifications !== null**.

Estamos usando el operador **typeof** para comprobar si la propiedad **specifications** es un objeto. Ya lo vimos en la primera sección del curso y nos ayuda a asegurarnos que **specifications** existe y es un objeto, antes de acceder a su propiedad **ram**.

253.1 El operador in para comprobar si una propiedad existe

Otra forma de comprobar si una propiedad existe es usando el operador **in**. Este operador comprueba si una propiedad existe en un objeto y devuelve **true** o **false**:

```
console.log('name' in gamesystem) // -> true
console.log('specifications' in gamesystem) // -> false
console.log('specs' in gamesystem) // -> true

if ('specifications' in gamesystem) {
  console.log(gamesystem.specifications.ram)
} else {
  console.log('No hay especificaciones')
}
```

¡Pero ojo! ¿Qué pasa si la propiedad existe pero su valor es **undefined**?

```
const gamesystem = {
  name: 'PS5',
  price: 550,
  specifications: undefined,
}

console.log('specifications' in gamesystem) // -> true

gamesystem.specifications.ram
//  Uncaught TypeError: Cannot read property 'ram' of undefined
```

El operador **in** puede ser interesante para comprobar si una propiedad existe en un objeto, pero no nos sirve para comprobar si el valor de esa propiedad es **undefined** y habría que hacer comprobaciones extra para evitar el error.

```
if (
  'specifications' in gamesystem &&
  gamesystem.specifications !== undefined &&
  gamesystem.specifications !== null) {
  console.log(gamesystem.specifications.ram)
} else {
  console.log('No hay especificaciones')
}
```

Mucho código para algo tan simple. Por suerte, existe una forma más sencilla de hacerlo.

253.2 El operador de Encadenamiento Opcional, ?.

El Operador de Encadenamiento Opcional **?.** permite leer el valor de una propiedad ubicada profundamente dentro de una cadena de objetos conectados, sin tener que validar expresamente que cada referencia en la cadena es válida.

```
const gamesystem = {
  name: 'PS5',
  price: 550,
  specs: {
    cpu: 'AMD Ryzen Zen 2',
    gpu: 'AMD Radeon RDNA 2',
  }
}

console.log(gamesystem.specifications?.cpu)
// -> undefined
console.log(gamesystem.specs?.cpu)
// -> AMD Ryzen Zen 2
```

Si la propiedad **specifications** no existe, el operador **?.** devuelve **undefined** y no se produce ningún error. Si la propiedad existe, el operador **?.** devuelve el valor de la propiedad.

253.3 Beneficios del Operador de Encadenamiento Opcional

Uno de los principales beneficios del operador de encadenamiento opcional es que simplifica nuestro código y lo hace más legible. En lugar de tener que hacer varias comprobaciones para asegurarnos de que un objeto y sus propiedades existen, podemos intentar acceder directamente a la propiedad que nos interesa, y JavaScript no se quejará si no existe.

Este operador es especialmente útil en aplicaciones con datos dinámicos, donde no siempre podemos garantizar la estructura exacta de nuestros objetos.

```
const user = {
  name: 'Peter',
  settings: {
    theme: 'dark',
    notifications: {
      email: true,
      push: false,
      marketing: undefined
    }
  }
}

// la forma clásica de acceder a una propiedad anidada
// de forma segura
let email = undefined
if (user && user.settings &&
  user.settings.notifications &&
  user.settings.notifications.email) {
  email = user.settings.notifications.email
}

console.log(email) // -> true

// con Optional Chaining Operator
const email = user?.settings?.notifications?.email
console.log(email) // -> true
```

El operador de encadenamiento opcional es una excelente herramienta para trabajar con objetos en JavaScript, especialmente cuando no podemos garantizar la estructura o presencia de ciertos datos. **Nos ayuda a escribir un código más limpio, legible y resistente a errores**.

254 Actividad Práctica

Somos un equipo de submarinistas. Estamos explorando el fondo marino pero no sabemos qué nos vamos a encontrar porque vamos casi a ciegas. Vamos con un pequeño robot y funciona con un programa que recibe tres parámetros:

- Un objeto con la información del fondo marino
- Una cadena de texto con el nombre del lugar que queremos explorar
- Una cadena de texto con el nombre del objeto que queremos encontrar

Por ejemplo. Con el objeto **ocean** que nos llega como primer parámetro, podemos explorar el lugar **deep** y buscar el objeto **treasure**. Así que intentaríamos acceder a **ocean.deep.treasure**.

Si lo encontramos el robot nos devuelve **true** y si no lo encuentra nos devuelve **false**. Sólo necesitamos saber si lo hemos encontrado o no, no necesitamos saber su valor.

Vamos a necesitar asegurarnos que el robot no se rompa, que cuesta mucho dinero, en el caso que no pueda encontrar el lugar o el objeto que le hemos pedido. Así que vamos a usar el operador de encadenamiento opcional para evitar errores.

Pista: ¡Recuerda que en una clase anterior hemos visto cómo podemos acceder a una propiedad de un objeto usando una variable! Por ejemplo, si tenemos un objeto **person** y una variable **key** con el valor **name**, podemos acceder a la propiedad **name** del objeto **person** usando **person[key]** y sería como hacer **person.name**.

```
function searchInOcean(ocean, section, item) {
    // tu código aquí
  }
```

Ver solución

```
function searchInOcean(ocean, section, item) {
  return ocean?.[section]?.[item] !== undefined
}
```

255 Rest Operator

El operador de propagación o **rest operator** es una característica de JavaScript que nos permite capturar un número indefinido de argumentos en una función. Es muy útil cuando no sabemos cuántos argumentos vamos a recibir o queremos trabajar con un número variable de argumentos.

255.1 Sintaxis del operador de propagación

El operador de propagación se representa con tres puntos suspensivos **...** seguidos de un nombre de variable. Por convención, se suele usar el nombre **rest**.

```
function sum(...rest) {
  console.log(rest)
}

sum(1, 2, 3, 4, 5) // -> [1, 2, 3, 4, 5]

sum('a', 'b', 'c') // -> ['a', 'b', 'c']

sum() // -> []
```

En este ejemplo, la función **sum** recibe un número indefinido de argumentos y los guarda en un array llamado **rest**. Luego, simplemente imprimimos el array **rest** por consola.

255.2 Usos comunes del operador de propagación

El operador de propagación es muy versátil y se puede utilizar en muchas situaciones diferentes. Algunos de los usos más comunes son:

- **Capturar argumentos de una función**: como hemos visto en el ejemplo anterior, el operador de propagación nos permite capturar un número indefinido de argumentos en una función.
- **Concatenar arrays**: podemos usar el operador de propagación para concatenar dos o más arrays en uno solo.

```
const arr1 = [1, 2, 3]
const arr2 = [4, 5, 6]
const arr3 = [...arr1, ...arr2]

console.log(arr3) // -> [1, 2, 3, 4, 5, 6]
```

- **Copiar arrays**: también podemos usar el operador de propagación para copiar un array.

```
const arr1 = [1, 2, 3]
const arr2 = [...arr1]

console.log(arr2) // -> [1, 2, 3]
```

- **Crear objetos**: podemos usar el operador de propagación para crear un nuevo objeto a partir de uno existente.

```
const obj1 = { a: 1, b: 2 }
const obj2 = { ...obj1, c: 3 }

console.log(obj2) // -> { a: 1, b: 2, c: 3 }
```

- **Eliminar propiedades de un objeto**: también podemos usar el operador de propagación para eliminar propiedades de un objeto.

```
const obj1 = { a: 1, b: 2, c: 3 }
const { c, ...obj2 } = obj1

console.log(obj2) // -> { a: 1, b: 2 }
```

255.3 Rest Operator vs Spread Operator

Es importante tener en cuenta que el operador de propagación o **rest operator** es diferente del operador de propagación o **spread operator**. Aunque ambos utilizan la misma sintaxis de tres puntos suspensivos **...**, tienen diferentes propósitos y se utilizan en diferentes contextos.

- **Rest Operator**: se utiliza para capturar un número indefinido de argumentos en una función o para desestructurar un objeto o array en variables.
- **Spread Operator**: se utiliza para expandir un objeto o array en otro objeto o array, para concatenar arrays o para copiar un objeto o array.

256 Actividad Práctica

Tenemos una función que recibe un número indefinido de argumentos. La función debe devolver un array con los argumentos que sean de tipo **string**.

Por ejemplo, si llamamos a la función con los argumentos **'a', 1, 'b', 2, 'c'**, la función debería devolver [**'a', 'b', 'c'**].

```
function getStrings(...rest) {
    // tu código aquí
  }
```

Ver solución

```
function getStrings(...rest) {
  return rest.filter(arg => typeof arg === 'string')
}
```

257 Promesas y async/await

Las promesas son un objeto de JavaScript que representa la eventual finalización (o falla) de una operación asíncrona, y su resultado.

Las promesas son una forma de trabajar con código asíncrono en JavaScript. Nos permiten ejecutar tareas en segundo plano y manejar el resultado de esas tareas de forma más sencilla y legible.

257.1 Crear una promesa

Para crear una promesa, usamos la clase **Promise** y pasamos una función con dos parámetros: **resolve** y **reject**.

- **resolve**: se llama cuando la promesa se resuelve satisfactoriamente y se le pasa el valor resultante.
- **reject**: se llama cuando la promesa falla y se le pasa el motivo del fallo.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('¡La promesa se ha resuelto!')
  }, 2000)
})

promise.then((result) => {
  console.log(result) // -> ¡La promesa se ha resuelto!
})

// -> ¡La promesa se ha resuelto!
```

En este ejemplo, creamos una promesa que se resuelve después de 2 segundos y luego imprimimos el resultado por consola.

257.2 Encadenar promesas

Una de las características más poderosas de las promesas es que podemos encadenarlas para ejecutar tareas en secuencia.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('¡La primera promesa se ha resuelto!')
  }, 2000)
})

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('¡La segunda promesa se ha resuelto!')
  }, 2000)
})

promise1
  .then((result) => {
    console
    console.log(result) // -> ¡La primera promesa se ha resuelto!
    return promise2
  })
  .then((result) => {
    console.log(result) // -> ¡La segunda promesa se ha resuelto!
  })

// -> ¡La primera promesa se ha resuelto!
// -> ¡La segunda promesa se ha resuelto!
```

En este ejemplo, creamos dos promesas y las encadenamos para que se ejecuten en secuencia. Primero se resuelve la **promise1** y luego la **promise2**.

257.3 async/await

El operador **async** se utiliza para declarar una función asíncrona, es decir, una función que devuelve una promesa. El operador **await** se utiliza para esperar a que una promesa se resuelva antes de continuar con la ejecución de la función.

```
async function fetchData() {
  const response = await fetch('https://api.example.com/data')
  const data = await response.json()
  return data
}

fetchData()
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.error(error)
```

```
  })

// -> { key: 'value' }
```

En este ejemplo, creamos una función **fetchData** que utiliza el operador **await** para esperar a que la promesa devuelta por **fetch** se resuelva antes de continuar con la ejecución de la función.

257.4 Beneficios de async/await

El uso de **async/await** tiene varios beneficios:

- **Código más legible**: el uso de **async/await** hace que el código sea más fácil de leer y entender, ya que se parece más a la forma en que escribimos código síncrono.
- **Manejo de errores**: podemos usar **try/catch** para manejar los errores de forma más sencilla y clara.
- **Encadenamiento de promesas**: podemos encadenar promesas de forma más sencilla y legible.
- **Mejor manejo de la lógica asíncrona**: **async/await** nos permite escribir código asíncrono de forma más natural y estructurada.

258 Actividad Práctica

Tenemos una función que recibe un número y devuelve una promesa que se resuelve con el doble del número recibido.

```
function doubleNumber(number) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(number * 2)
    }, 2000)
  })
}

doubleNumber(5)
  .then((result) => {
    console.log(result) // -> 10
  })

// -> 10
```

Tu tarea es modificar la función **doubleNumber** para que utilice **async/await** en lugar de crear una promesa manualmente.

```
async function doubleNumber(number) {
  // tu código aquí
}

doubleNumber(5)
  .then((result) => {
    console.log(result) // -> 10
  })

// -> 10
```

Ver solución

```
async function doubleNumber(number) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(number * 2)
    }, 2000)
  })
```

```
}

doubleNumber(5)
  .then((result) => {
    console.log(result) // -> 10
  })

// -> 10
```

259 Quiz

¿Qué es una promesa en JavaScript?

Un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca.

Un objeto que representa un valor que puede cambiar en cualquier momento.

Un objeto que representa la eventual finalización (o falla) de una operación asíncrona, y su resultado.

¿Qué hace el operador await en una función asíncrona?

Detiene la ejecución de la función hasta que se resuelva una promesa.

Continúa la ejecución de la función sin esperar a que se resuelva una promesa.

Espera a que una promesa se resuelva antes de continuar con la ejecución de la función.

¿Qué hace el operador async en una función?

Declara una función que devuelve una promesa.

Declara una función que se ejecuta de forma asíncrona.

Declara una función asíncrona que devuelve una promesa.

260 Módulos

Los módulos son una forma de organizar y reutilizar código en JavaScript. Nos permiten dividir nuestro código en archivos separados y exportar e importar funciones, variables y clases entre ellos.

260.1 Exportar e importar

Para exportar una función, variable o clase de un módulo, usamos la palabra clave **export** seguida del nombre de la función, variable o clase que queremos exportar.

```
// math.js
export function sum(a, b) {
  return a + b
}

export const PI = 3.14159
```

Para importar una función, variable o clase de un módulo, usamos la palabra clave **import** seguida del nombre de la función, variable o clase que queremos importar y el nombre del módulo entre llaves.

```
// app.js
import { sum, PI } from './math.js'

console.log(sum(1, 2)) // -> 3
console.log(PI) // -> 3.14159
```

260.2 Exportar e importar por defecto

También podemos exportar e importar por defecto una función, variable o clase de un módulo. Para exportar por defecto, usamos la palabra clave **export default** seguida del nombre de la función, variable o clase que queremos exportar por defecto.

```
// math.js
export default function sum(a, b) {
  return a + b
}

export const PI = 3.14159
```

Para importar por defecto una función, variable o clase de un módulo, usamos la palabra clave **import** seguida del nombre de la función, variable o clase que queremos importar y el nombre del módulo entre llaves.

```
// app.js
import sum, { PI } from './math.js'

console.log(sum(1, 2)) // -> 3
console.log(PI) // -> 3.14159
```

260.3 Exportar e importar todo

También podemos exportar e importar todo un módulo. Para exportar todo, usamos la palabra clave **export ***.

```
// math.js
export function sum(a, b) {
  return a + b
}

export const PI = 3.14159
```

Para importar todo un módulo, usamos la palabra clave **import ***.

```
// app.js
import * as math from './math.js'

console.log(math.sum(1, 2)) // -> 3
console.log(math.PI) // -> 3.14159
```

260.4 Módulos en el navegador

Los módulos también se pueden utilizar en el navegador. Para importar un módulo en un archivo HTML, usamos la etiqueta **script** con el atributo **type="module"** y la ruta del archivo que queremos importar en el atributo **src**.

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
```

```
</head>
<body>
  <script type="module" src="app.js"></script>
</body>
</html>
```

En el archivo **app.js**, podemos importar un módulo de la misma forma que en Node.js.

```
// app.js
import { sum, PI } from './math.js'

console.log(sum(1, 2)) // -> 3
console.log(PI) // -> 3.14159
```

260.5 Módulos en Node.js

En Node.js, los módulos se pueden utilizar de la misma forma que en el navegador. Para exportar e importar un módulo, usamos la palabra clave **export** y **import**.

```
// math.js
export function sum(a, b) {
  return a + b
}

export const PI = 3.14159
```

```
// app.js
import { sum, PI } from './math.js'

console.log(sum(1, 2)) // -> 3
console.log(PI) // -> 3.14159
```

260.6 Módulos en CommonJS

En CommonJS, los módulos se pueden utilizar de la misma forma que en Node.js. Para exportar e importar un módulo, usamos la palabra clave **module.exports** y **require**.

```
// math.js
function sum(a, b) {
  return a + b
}

const PI = 3.14159

module.exports = { sum, PI }
```

```
// app.js
const { sum, PI } = require('./math.js')

console.log(sum(1, 2)) // -> 3
console.log(PI) // -> 3.14159
```

261 Actividad Práctica

Tenemos un módulo que exporta una función que recibe un número y devuelve el doble del número recibido.

```
// math.js
export function doubleNumber(number) {
  return number * 2
}
```

Tu tarea es importar la función **doubleNumber** en un archivo **app.js** y usarla para duplicar el número **5**.

```
// app.js
// tu código aquí
```

Ver solución

```
// app.js
import { doubleNumber } from './math.js'

console.log(doubleNumber(5)) // -> 10
```

262 Quiz

¿Qué es un módulo en JavaScript?

Un objeto que representa la eventual finalización (o falla) de una operación asíncrona, y su resultado.

Un archivo que contiene código JavaScript que se puede exportar e importar en otros archivos.

Un archivo que contiene código JavaScript que se puede exportar e importar en otros archivos.

263 Nodejs

Figure 263.1: Nodejs

264 1. Introducción a Node.js

Node.js es un entorno de ejecución de JavaScript que permite ejecutar código JavaScript en el lado del servidor. Fue creado por Ryan Dahl en 2009 y está basado en el motor V8 de Google Chrome. Node.js es una herramienta muy poderosa para el desarrollo de aplicaciones web y APIs RESTful, ya que permite realizar operaciones de entrada/salida de forma asíncrona y no bloqueante.

264.1 ¿Por qué es Node.js?

Node.js permite manejar múltiples solicitudes simultáneamente sin bloquear el hilo principal, lo que lo hace muy eficiente para aplicaciones en tiempo real y de alta concurrencia.

264.2 Historia y Evolución de Node.js

- **2009:** Creación de Node.js por Ryan Dahl.
- **2010:** Lanzamiento de la primera versión estable.
- **2011:** Lanzamiento de npm (Node Package Manager).
- **2012-2021:** Mejora continua con nuevas versiones y características.

264.3 Características de Node.js

- Asíncrono y no bloqueante
- Eventos y callbacks
- Módulos y paquetes
- NPM (Node Package Manager)
- Rendimiento y escalabilidad
- Comunidad activa

265 2. Manejadores de Paquetes en Node.js

Los manejadores de paquetes son herramientas esenciales para la gestión de dependencias en proyectos Node.js.

265.1 npm (Node Package Manager)

- **Descripción:** El manejador de paquetes por defecto para Node.js.
- **Uso Básico:**

```
npm install <paquete>: Instala un paquete.
npm install --save-dev <paquete>: Instala un paquete para desarrollo.
npm update: Actualiza paquetes instalados.
```

265.2 yarn

- **Descripción:** Alternativa a npm creada por Facebook, ofrece una instalación más rápida y una mejor gestión de dependencias.
- **Uso Básico:**

```
yarn add <paquete>: Instala un paquete.
yarn add --dev <paquete>: Instala un paquete para desarrollo.
yarn upgrade: Actualiza paquetes instalados.
```

265.3 pnpm

- **Descripción:** Un manejador de paquetes más rápido y eficiente que almacena un único conjunto de paquetes en lugar de duplicar dependencias.
- **Uso Básico:**

```
pnpm install <paquete>: Instala un paquete.
pnpm install --save-dev <paquete>: Instala un paquete para desarrollo.
pnpm update: Actualiza paquetes instalados.
```

266 3. Manejadores de Versiones de Node.js

Los manejadores de versiones permiten gestionar múltiples versiones de Node.js en un solo sistema.

266.1 nvm (Node Version Manager)

- **Descripción:** Herramienta para instalar y gestionar múltiples versiones de Node.js.
- **Uso Básico:**

```
nvm install <version>: Instala una versión específica de Node.js.
nvm use <version>: Cambia a una versión específica.
nvm ls: Lista las versiones instaladas.
```

266.2 fnm (Fast Node Manager)

- **Descripción:** Alternativa más rápida a nvm, también permite gestionar versiones de Node.js.
- **Uso Básico:**

```
fnm install <version>: Instala una versión específica de Node.js.
fnm use <version>: Cambia a una versión específica.
fnm ls: Lista las versiones instaladas.
```

267 4. Entornos de Ejecución de JavaScript

Node.js no es el único entorno de ejecución para JavaScript. Aquí se presentan otras opciones que están ganando popularidad.

268 Node.js

- **Descripción:** Entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome, diseñado para ejecutar código JavaScript en el lado del servidor.
- **Características:** Asíncrono, no bloqueante, amplio ecosistema de paquetes.

268.1 Deno

- **Descripción:** Entorno de ejecución para JavaScript y TypeScript creado por Ryan Dahl (creador de Node.js) como una alternativa moderna a Node.js.
- **Características:**

```
*   Soporte nativo para TypeScript.
*   Seguridad mejorada con permisos explícitos.
*   No utiliza npm, usa deno.land/x para paquetes.
```

268.2 Bun

- **Descripción:** Un entorno de ejecución de JavaScript relativamente nuevo que se enfoca en el rendimiento y la simplicidad.
- **Características:**

```
*   Enfocado en la velocidad y la compatibilidad.
*   Incluye un empaquetador, un administrador de paquetes y un motor de ejecución en un s
*   Compatible con Node.js en muchos aspectos, pero aún en desarrollo y evolución.
```

269 5. Fundamentos del Asincronismo en Node.js

Node.js es conocido por su modelo de programación asíncrona no bloqueante. Esto permite manejar múltiples solicitudes de entrada/salida (I/O) simultáneamente sin bloquear el hilo principal, haciendo que Node.js sea muy eficiente en operaciones que requieren acceder a archivos, bases de datos o servicios externos.

- **Sincronía vs Asincronía**

```
*   **Sincronía:** Las operaciones se ejecutan secuencialmente.
*   **Asincronía:** Las operaciones no bloquean el hilo principal, utilizando callbacks, 
```

270 6. Callbacks

Un callback es una función que se pasa como argumento a otra función y se ejecuta después de que la operación asíncrona ha finalizado.

270.1 Ejemplo de Callback:

```
const fs = require('fs');

// Leer un archivo de manera asíncrona usando un callback
fs.readFile('archivo.txt', 'utf8', (err, data) => {
  if (err) {
    console.error("Error leyendo el archivo:", err);
    return;
  }
  console.log("Contenido del archivo:", data);
});
```

Problema: Callback Hell A medida que anidamos callbacks, el código se vuelve difícil de leer y mantener.

271 7. Promesas

Las promesas son una forma más clara y manejable de trabajar con operaciones asíncronas.

271.1 Ejemplo de Promesa:

```
const fs = require('fs').promises;

fs.readFile('archivo.txt', 'utf8')
  .then(data => {
    console.log("Contenido del archivo:", data);
  })
  .catch(err => {
    console.error("Error leyendo el archivo:", err);
  });
```

272 8. Async/Await

Async/Await es una sintaxis moderna para trabajar con promesas que permite escribir código asíncrono de manera más legible.

272.1 Ejemplo de async/await:

```
const fs = require('fs').promises;

async function leerArchivo() {
  try {
    const data = await fs.readFile('archivo.txt', 'utf8');
    console.log("Contenido del archivo:", data);
  } catch (err) {
    console.error("Error leyendo el archivo:", err);
  }
}

leerArchivo();
```

273 9. Introducción a Node.js en la práctica

Vamos a crear una pequeña aplicación que lea archivos de manera asíncrona y envíe una respuesta utilizando un servidor básico de Node.js.

273.1 Crear un Servidor HTTP Asíncrono

```
const http = require('http');
const fs = require('fs').promises;

const server = http.createServer(async (req, res) => {
  try {
    const data = await fs.readFile('archivo.txt', 'utf8');
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(data);
  } catch (err) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end("Error leyendo el archivo");
  }
});

server.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

274 10. Gestión de Paquetes con NPM

Node.js utiliza NPM (Node Package Manager) para instalar y gestionar dependencias.

274.1 Comandos básicos de NPM:

- **npm init**: Inicia un nuevo proyecto y crea un archivo package.json.
- **npm install <paquete>**: Instala un paquete.
- **npm install –save-dev <paquete>**: Instala un paquete para desarrollo.
- **npm update**: Actualiza paquetes instalados.

11. Preparando el Entorno para React

Para conectar lo aprendido en Node.js con React en la próxima clase, es fundamental instalar y configurar herramientas adicionales.

274.2 Instalar create-react-app:

275 Tutorial de Fetch

En este tutorial vamos a aprender a hacer peticiones Fetch con JavaScript puro. Para ello vamos a hacer una aplicación que nos permita buscar pokemon y ver sus detalles.

275.1 Requisitos

Para este tutorial vamos a utilizar tecnologias como **html**, **css** y **javascript**. Además, vamos a necesitar un navegador web y un editor de código, se recomienda leer acerca de peticiones con fetch antes de empezar este tutorial.

275.2 Instrucciones

1. Creamos un archivo **index.html** y agregamos el siguiente código:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Pokemon</title>
```

```
</head>
<body>
    <h1>Pokemon</h1>
    <input type="text" id="search" placeholder="Buscar pokemon">
    <button id="search-button">Buscar</button>
    <div id="pokemon"></div>
    <script src="app.js"></script>
</body>
</html>
```

En esta sección hemos creado la **estructura básica de nuestro proyecto**. Hemos **agregado** un **título**, un **input** para **buscar pokemon**, un **botón** para **buscar pokemon**, un div para **mostrar** la **información** del **pokemon** y un **script** para nuestro archivo **app.js**.

2. Creamos un archivo **app.js** y agregamos el siguiente código:

```
document.getElementById('search-button').addEventListener('click', () => {
    const search = document.getElementById('search').value;
    fetch(`https://pokeapi.co/api/v2/pokemon/${search}`)
        .then(response => response.json())
        .then(data => {
            const pokemon = document.getElementById('pokemon');
            pokemon.innerHTML = `
                <h2>${data.name}</h2>
                <img src="${data.sprites.front_default}" alt="${data.name}">
                <p>Altura: ${data.height}</p>
                <p>Peso: ${data.weight}</p>
            `;
        })
        .catch(error => {
            console.error(error);
        });
});
```

En esta sección hemos **agregado** un **evento** al **botón** de **buscar pokemon**. Cuando se haga **click** en el **botón**, se va a hacer una **petición** a la **API** de PokeAPI. Si la petición es exitosa, se va a **mostrar la información** del **pokemon** en el **div pokemon**. Si la petición falla, se va a **mostrar** un **mensaje** de **error** en la consola.

Para más información sobre la API de PokeAPI, se puede consultar la documentación.

5. Mediante el plugin de Live Server, abrimos el archivo **index.html** en el navegador y probamos la aplicación.

6. Vamos a mejorar la presentación de nuestro proyecto agregando estilos CSS. Creamos un archivo **styles.css** y agregamos el siguiente código:

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    display: flex;
    flex-direction: column;
    align-items: center;
}

h1 {
    margin: 1rem 0;
}

input {
    padding: 0.5rem;
    margin: 0.5rem 0;
}

button {
    padding: 0.5rem;
    margin: 0.5rem 0;
    background-color: #007bff;
    color: white;
    border: none;
    cursor: pointer;
}

#pokemon {
    display: flex;
    flex-direction: column;
    align-items: center;
    margin: 1rem 0;
}

img {
    width: 100px;
    height: 100px;
    margin: 0.5rem 0;
}

p {
    margin: 0.5rem 0;
}
```

En esta sección hemos agregado **estilos CSS** para **mejorar** la presentación de nuestro proyecto, como **la fuente**, **el color de fondo**, **el color del texto**, **el tamaño de la letra**, **el espaciado**, **el color del botón**, **el cursor**, **el tamaño de la imagen**, **el margen de la imagen**, **el margen del párrafo**, etc.

7. Agregamos el archivo **styles.css** al archivo **index.html**:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Pokemon</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Pokemon</h1>
    <input type="text" id="search" placeholder="Buscar pokemon">
    <button id="search-button">Buscar</button>
    <div id="pokemon"></div>
    <script src="app.js"></script>
</body>
</html>
```

En esta sección hemos agregado el archivo **styles.css** al archivo **index.html**.

8. Mediante el plugin de **Live Server**, abrimos el archivo **index.html** en el navegador y probamos la aplicación.

276 Deploy

Se sugiere hacer el deploy de la aplicación en **Github Pages**, **Vercel**, **Netlify**, etc.

277 Reto

Ahora es necesario mejorar la aplicación. Se sugiere agregar las siguientes funcionalidades:

1. Agregar un mensaje de error si el pokemon no existe.
2. Agregar un mensaje de error si la petición falla.
3. Agregar un spinner mientras se hace la petición.
4. Agregar un botón para limpiar el div **pokemon**.
5. Agregar un botón para buscar un pokemon aleatorio.
6. Agregar un botón para buscar un pokemon por id.
7. Agregar un botón para buscar un pokemon por nombre.
8. Agregar un botón para buscar un pokemon por tipo.
9. Agregar un botón para buscar un pokemon por habilidad.

278 Fetch con Fake Store API

Para crear una aplicación que muestre productos de una tienda, se puede utilizar la Fake Store API.

1. Crear un archivo **index.html** y agregar el siguiente código:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Productos</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Productos</h1>
    <div id="products"></div>
    <script src="app.js"></script>
</body>
</html>
```

2. Crear un archivo **app.js** y agregar el siguiente código:

```
fetch('https://fakestoreapi.com/products')
    .then(response => response.json())
    .then(data => {
        const products = document.getElementById('products');
        data.forEach(product => {
            products.innerHTML += `
                <div class="product">
                    <img src="${product.image}" alt="${product.title}">
                    <h2>${product.title}</h2>
                    <p>${product.description}</p>
                    <p>${product.price}</p>
                </div>
            `;
        });
    })
    .catch(error => {
        console.error(error);
    });
```

3. Crear un archivo **styles.css** y agregar el siguiente código:

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    display: flex;
    flex-direction: column;
    align-items: center;
}

h1 {
    margin: 1rem 0;
}

#products {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    gap: 1rem;
    margin: 1rem 0;
}

.product {
    border: 1px solid #ccc;
    padding: 1rem;
    text-align: center;
}

img {
```

```
    width: 100px;
    height: 100px;
    margin: 0.5rem 0;
}

h2 {
    margin: 0.5rem 0;
}

p {
    margin: 0.5rem 0;
}
```

4. Mediante el plugin de Live Server, abrimos el archivo **index.html** en el navegador y probamos la aplicación.
5. Se puede mejorar la aplicación agregando más funcionalidades, como un buscador, un filtro por categoría, un ordenamiento por precio, etc.

279 Reto

Ahora es necesario mejorar la aplicación. Se sugiere agregar las siguientes funcionalidades:

1. Agregar un mensaje de error si la petición falla.
2. Agregar un spinner mientras se hace la petición.
3. Agregar un botón para limpiar el div **products**.
4. Agregar un botón para buscar un producto por id.
5. Agregar un botón para buscar un producto por título.
6. Agregar un botón para buscar un producto por categoría.
7. Agregar un botón para buscar un producto por precio.
8. Agregar un botón para buscar un producto por rating.
9. Agregar un botón para buscar un producto por cantidad.
10. Agregar un botón para buscar un producto por descripción.

280 Fetch con Rick and Morty API.

Para crear una aplicación que muestre personajes de la serie Rick and Morty, se puede utilizar la Rick and Morty API.

1. Crear un archivo **index.html** y agregar el siguiente código:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Personajes</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Personajes</h1>
    <div id="characters"></div>
    <script src="app.js"></script>
</body>
</html>
```

2. Crear un archivo **app.js** y agregar el siguiente código:

```
fetch('https://rickandmortyapi.com/api/character')
    .then(response => response.json())
    .then(data => {
        const characters = document.getElementById('characters');
        data.results.forEach(character => {
            characters.innerHTML += `
                <div class="character">
                    <img src="${character.image}" alt="${character.name}">
                    <h2>${character.name}</h2>
                    <p>${character.species}</p>
                    <p>${character.status}</p>
                </div>
            `;
        });
    })
    .catch(error => {
        console.error(error);
    });
```

3. Crear un archivo **styles.css** y agregar el siguiente código:

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    display: flex;
    flex-direction: column;
    align-items: center;
}

h1 {
    margin: 1rem 0;
}

#characters {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    gap: 1rem;
    margin: 1rem 0;
}

.character {
    border: 1px solid #ccc;
    padding: 1rem;
    text-align: center;
}

img {
```

```
    width: 100px;
    height: 100px;
    margin: 0.5rem 0;
}

h2 {
    margin: 0.5rem 0;
}

p {
    margin: 0.5rem 0;
}
```

4. Mediante el plugin de Live Server, abrimos el archivo **index.html** en el navegador y probamos la aplicación.
5. Se puede mejorar la aplicación agregando más funcionalidades, como un buscador, un filtro por especie, un ordenamiento por nombre, etc.

281 Reto

Ahora es necesario mejorar la aplicación. Se sugiere agregar las siguientes funcionalidades:

1. Agregar un mensaje de error si la petición falla.
2. Agregar un spinner mientras se hace la petición.
3. Agregar un botón para limpiar el div **characters**.
4. Agregar un botón para buscar un personaje por id.
5. Agregar un botón para buscar un personaje por nombre.
6. Agregar un botón para buscar un personaje por especie.
7. Agregar un botón para buscar un personaje por estado.
8. Agregar un botón para buscar un personaje por género.
9. Agregar un botón para buscar un personaje por origen.
10. Agregar un botón para buscar un personaje por ubicación.

282 Fetch con otras APIs

Para crear aplicaciones con Fetch, se pueden utilizar otras APIs como:

- The Dog API
- The Cat API
- The Joke API
- The Weather API
- The News API
- The Movie API
- The Music API
- The Recipe API
- The Book API

Solo para tener un ejemplo crearemos una aplicación que muestre chistes de la JokeAPI.

1. Crear un archivo **index.html** y agregar el siguiente código:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chistes</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Chistes</h1>
    <div id="jokes"></div>
    <button id="joke-button">Nuevo chiste</button>
    <script src="app.js"></script>
</body>
</html>
```

2. Crear un archivo **app.js** y agregar el siguiente código:

```javascript
document.getElementById('joke-button').addEventListener('click', () => {
    fetch('https://v2.jokeapi.dev/joke/Any')
        .then(response => response.json())
        .then(data => {
            const jokes = document.getElementById('jokes');
            jokes.innerHTML = `
                <h2>${data.setup}</h2>
                <p>${data.delivery}</p>
            `;
        })
        .catch(error => {
            console.error(error);
        });
});
```

3. Crear un archivo **styles.css** y agregar el siguiente código:

```css
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    display: flex;
    flex-direction: column;
    align-items: center;
}

h1 {
    margin: 1rem 0;
}
```

```
#jokes {
    margin: 1rem 0;
    text-align: center;
}

button {
    padding: 0.5rem;
    margin: 0.5rem 0;
    background-color: #007bff;
    color: white;
    border: none;
    cursor: pointer;
}

h2 {
    margin: 0.5rem 0;
}

p {
    margin: 0.5rem 0;
}
```

4. Mediante el plugin de Live Server, abrimos el archivo **index.html** en el navegador y probamos la aplicación.
5. Se puede mejorar la aplicación agregando más funcionalidades, como un buscador, un filtro por categoría, un ordenamiento por fecha, etc.

283 Reto

Ahora es necesario mejorar la aplicación. Se sugiere agregar las siguientes funcionalidades:

1. Agregar un mensaje de error si la petición falla.
2. Agregar un spinner mientras se hace la petición.
3. Agregar un botón para limpiar el div **jokes**.
4. Agregar un botón para buscar un chiste por categoría.
5. Agregar un botón para buscar un chiste por tipo.
6. Agregar un botón para buscar un chiste por bandera.
7. Agregar un botón para buscar un chiste por idioma.
8. Agregar un botón para buscar un chiste por cantidad.
9. Agregar un botón para buscar un chiste por filtro.
10. Agregar un botón para buscar un chiste por blacklist.

283.1 Resumen

En este tutorial hemos aprendido a hacer peticiones Fetch con JavaScript puro. Hemos creado una aplicación que nos permite buscar pokemon y ver sus detalles. También hemos aprendido a mejorar la presentación de nuestro proyecto agregando estilos CSS. Además, hemos aprendido a hacer peticiones Fetch con la Fake Store API y la Rick and Morty API. Se sugiere seguir practicando y experimentando con Fetch para mejorar nuestras habilidades de programación.

283.2 Conclusión

En este tutorial hemos aprendido a hacer peticiones Fetch con JavaScript puro. Hemos creado una aplicación que nos permite buscar pokemon y ver sus detalles. También hemos aprendido a mejorar la presentación de nuestro proyecto agregando estilos CSS. Además, hemos aprendido a hacer peticiones Fetch con la Fake Store API y la Rick and Morty API. Se sugiere seguir practicando y experimentando con Fetch para mejorar nuestras habilidades de programación.

283.3 Referencias

- Fetch API
- PokéAPI
- Fake Store API
- Rick and Morty API
- JokeAPI
- The Dog API
- The Cat API
- The Weather API
- The News API
- The Movie API
- The Music API
- The Recipe API
- The Book API
- Github Pages
- Vercel
- Netlify
- Live Server
- Visual Studio Code
- HTML
- CSS
- JavaScript
- API
- JSON
- DOM
- Node.js
- NPM
- Async/Await
- Promesas

284 Introducción a React

Figure 284.1: React Logo

285 1. ¿Qué es React?

React es una biblioteca de JavaScript desarrollada por Facebook para construir interfaces de usuario. Se enfoca en la creación de componentes reutilizables y eficientes que permiten el desarrollo de aplicaciones web rápidas y escalables.

285.1 Características clave de React:

- **Componentes:** React basa su arquitectura en componentes. Cada componente es una pieza reutilizable de código que puede manejar su propio estado y lógica.
- **Virtual DOM:** React utiliza un DOM virtual para actualizar solo las partes que cambian, lo que mejora significativamente el rendimiento.
- **Unidirectional Data Flow (Flujo de Datos Unidireccional):** En React, los datos fluyen en una sola dirección, desde el componente padre hacia los componentes hijos.
- **JSX:** Un lenguaje similar a HTML que permite combinar la estructura del componente con lógica JavaScript.

286 2. Evolución de React: De create-react-app a Vite

286.1 React con create-react-app (Antes).

Antes de la aparición de herramientas como Vite, React se inicializaba principalmente usando create-react-app (CRA). CRA es un generador de proyectos que proporciona una configuración completa para desarrollar aplicaciones React sin necesidad de configuraciones manuales con Webpack o Babel.

```
npxnpx create-react-app my-app
8 vulnerabilities (2 moderate, 6 high)

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.

Created git commit.

Success! Created my-app at /home/statick/prueba/my-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd my-app
  npm start

Happy hacking!
```

```
npx create-react-app my-app
cd my-app
npm start
```

Con estos comandos, podías crear un proyecto React con CRA y ejecutarlo en un servidor de desarrollo. Sin embargo, CRA tiene algunas desventajas que han llevado a la adopción de herramientas más modernas como Vite.

```
~/prueba (0.048s)
cd cd my-app/
/home/statick/prueba/my-app

^[i statick at fedora in ~/prueba/my-app on (master ✓)
λ

~/prueba/my-app git:(master)
npm npm start
Compiled successfully!

You can now view my-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.100.139:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

286.2 Desventajas de create-react-app:

Lentitud: A medida que las aplicaciones crecen, CRA puede volverse lento durante el tiempo de compilación y recarga en caliente.

Sobrecarga de configuración: Aunque CRA oculta la configuración compleja, esta puede ser difícil de personalizar cuando el proyecto lo requiere.

286.3 React con Vite (Ahora)

Vite es una herramienta de desarrollo moderna que mejora significativamente los tiempos de desarrollo y la experiencia del programador, superando a CRA en términos de velocidad y simplicidad.

286.3.1 Ventajas de Vite:

- **Velocidad:** Vite es extremadamente rápido, incluso en proyectos grandes, debido a su compilación basada en ES Modules (módulos de ECMAScript).
- **Recarga en caliente instantánea:** Vite ofrece Hot Module Replacement (HMR), que recarga solo los módulos que cambian, proporcionando un entorno de desarrollo mucho más ágil.

- **Configuración ligera:** La configuración es mínima y fácil de personalizar cuando es necesario.

286.4 Crear un proyecto React con Vite:

Hoy en día, los desarrolladores de React utilizan Vite para crear proyectos mucho más rápido. Aquí te mostramos cómo hacerlo.

```
npm create vite@latest .
```

Con esta sencilla línea de comandos, puedes crear un proyecto React con Vite y comenzar a desarrollar aplicaciones web modernas de forma rápida y eficiente.

```
~/prueba/react-project (7.791s)
npmnpm create vite@latest .
Need to install the following packages:
  create-vite@5.5.2
Ok to proceed? (y) y
✔ Select a framework: › React
✔ Select a variant: › JavaScript

Scaffolding project in /home/statick/prueba/react-project...

Done. Now run:

  npm install
  npm run dev

statick at fedora in ~/prueba/react-project

~/prueba/react-project
```

> **Tip**
>
> **Nota:** Al ejecutar el comando anterior en tu terminal, se abrirá un asistente para configurar tu proyecto con Vite. Puedes elegir entre diferentes plantillas, incluyendo React, Vue, Preact y más. De la misma forma la forma en la que puedes programar ya que es posible utilizar javascrit o typescript.

```
npm install
npm run dev
```

Con los comandos anteriores, puedes instalar las dependencias y ejecutar el servidor de desarrollo de Vite para comenzar a trabajar en tu proyecto React.

287 3. Conceptos Básicos de React

Antes de iniciar con React es necesario conocer un poco la estructura básica de un proyecto en React, los componentes, JSX, props, estado y hooks.

Por ahora lo más importante es mantener un orden que quizás al principio no se entienda pero con el tiempo se irá aclarando.

Para ello vamos a crear una directorio llamado **components** y dentro del directorio **src** vamos a crear un archivo llamado **Gretting.jsx**.

> Tip
>
> **JSX:** Es una extensión de JavaScript que permite escribir HTML en archivos JavaScript.

```
function Greeting() {
  return <h1>¡Hola, mundo!</h1>;
}

export default Greeting;
```

287.1 3.1. Componentes en React

Los componentes son bloques reutilizables que contienen lógica, estructura y estilo. En React, un componente puede ser:

Funcional: Es el enfoque moderno en React. Son componentes escritos como funciones de JavaScript.

De clase: Son el enfoque anterior, basados en la programación orientada a objetos. Aún se usan en proyectos más antiguos.

En la sección anterior, creamos un componente funcional llamado Greeting. Ahora, vamos a importar y usar este componente en el archivo **App.js**.

```
import Greeting from './components/Greeting';

function App() {
  return (
    <div>
      <Greeting />
    </div>
  );
}

export default App;
```

Para correr el proyecto ejecutamos el comando:

```
npm run dev
```

287.2 3.2. JSX: JavaScript + HTML

JSX es una extensión de JavaScript que permite escribir HTML dentro de archivos JavaScript. Esto facilita la creación de componentes y mejora la legibilidad del código.

```
<h1>¡Hola, mundo!</h1>
```

JSX se compila a llamadas de funciones de React que crean elementos de React.

Por ejemplo, el código JSX anterior se compila a:

```
React.createElement('h1', null, '¡Hola, mundo!');
```

Sin embargo no es necesario preocuparse por esto ya que React se encarga de hacerlo por nosotros.

> **Tip**
>
> **Nota:** Los elementos JSX deben tener un solo contenedor raíz. Si necesitas devolver varios elementos, puedes envolverlos en un contenedor div o usar Fragment.

Ejemplo de Fragment:

```
function App() {
  return (
    <>
      <h1>¡Hola, mundo!</h1>
      <p>Bienvenido a mi aplicación.</p>
    </>
  );
}
```

En el ejemplo anterior, usamos Fragment para envolver múltiples elementos sin necesidad de un contenedor adicional.

Para poder probar este ejemplo es necesario modificar el archivo **App.js**. y correr el servidor de desarrollo.

```
npm run dev
```

287.3 3.3. Props y Estado

Props: Son los "argumentos" que los componentes reciben para configurar su contenido o comportamiento.

Para entender mejor cómo funcionan las props, vamos a modificar el componente Greeting para que reciba un prop llamado name.

En el directorio **components**, modificamos el archivo llamado **Greeting.jsx** con el siguiente contenido:

```
export default function Greeting(props) {
  return <h1>¡Hola, {props.name}!</h1>;
}
```

Muy bien ahora vamos a modificar el archivo **App.js** para que el componente Greeting reciba el prop name.

```
import Greeting from './components/Greeting';

function App() {
  return (
    <>
      <Greeting name="Diego" />
    </>
  );
}

export default App;
```

En el ejemplo anterior, pasamos el prop name con el valor “Diego” al componente Greeting. Esto permite personalizar el saludo que se muestra en la pantalla.

287.4 3.4. Hooks

Los hooks fueron introducidos en **React 16.8** y son funciones que permiten a los componentes funcionales gestionar el estado y otros efectos del ciclo de vida, funcionalidades que anteriormente solo estaban disponibles en los componentes de clase.

useState: Gestiona el estado dentro de un componente funcional.

useEffect: Maneja efectos secundarios, como llamadas a APIs o actualizaciones del DOM.

287.5 Ejemplo con useState y useEffect:

```
import { useState, useEffect } from 'react';

function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Has hecho clic ${count} veces`;
  }, [count]); // Se ejecuta cada vez que 'count' cambia

  return (
    <div>
      <p>Has hecho clic {count} veces</p>
      <button onClick={() => setCount(count + 1)}>Haz clic</button>
    </div>
  );
}

export default App;
```

En el código anterior, usamos **useState** para **inicializar** el **estado count** en **0** y **setCount** para **actualizarlo**. También usamos **useEffect** para **actualizar** el **título de la página** cada vez que count cambia.

Para entender mejor este concepto vamo a crear un nuevo componente llamado Clasificador de Edades.

En el directorio **components**, creamos un archivo llamado **AgeClassifier.jsx** con el siguiente contenido:

```
import { useState } from 'react';

export default function AgeClassifier() {
  const [age, setAge] = useState(0);
  const [classification, setClassification] = useState('');

  function classifyAge() {
    if (age < 18) {
      setClassification('Menor de edad');
    } else {
      setClassification('Mayor de edad');
    }
  }

  return (
    <div>
      <input
        type="number"
        value={age}
        onChange={(e) => setAge(parseInt(e.target.value))}
      />
      <button onClick={classifyAge}>Clasificar Edad</button>
      <p>{classification}</p>
    </div>
  );
}
```

En el código anterior, usamos **useState** para inicializar el estado **age** en **0** y **classification** en una cadena vacía. Luego, definimos una función **classifyAge** que clasifica la edad en "Menor de edad" o "Mayor de edad" según el valor de age. Finalmente, mostramos un campo de entrada para la edad, un botón para clasificarla y un mensaje con la clasificación.

Para probar este componente, vamos a importarlo y usarlo en el archivo **App.js**.

```
import AgeClassifier from './components/AgeClassifier';

function App() {
  return (
    <>
      <AgeClassifier />
    </>
  );
}

export default App;
```

Ahora, al correr el servidor de desarrollo, podrás ver el componente AgeClassifier en acción.

```
npm run dev
```

Con este ejercicio hemos aprendido a usar los **hooks useState** y **useEffect** en un **componente funcional de React**. Estos **hooks** son **fundamentales** para **gestionar el estado** y los **efectos secundarios** en aplicaciones React modernas.

287.6 4. Diferencias entre el Uso Antiguo y el Actual de React

Antes: Componentes de Clase

En versiones anteriores, los desarrolladores usaban componentes de clase para manejar el estado y el ciclo de vida de los componentes.

```
class Greeting extends React.Component {
  render() {
    return <h1>¡Hola, {this.props.name}!</h1>;
  }
}
```

Ahora: Componentes Funcionales con Hooks

Hoy en día, la norma es usar componentes funcionales con hooks, ya que son más concisos y fáciles de trabajar.

```
function Greeting({ name }) {
  return <h1>¡Hola, {name}!</h1>;
}
```

287.7 Ventajas del enfoque moderno con Hooks:

Menos código: Los componentes funcionales con hooks son más cortos y claros.

Manejo de estado más simple: useState y otros hooks ofrecen una forma directa y sencilla de gestionar el estado y el ciclo de vida.

Mejor rendimiento: React puede optimizar mejor los componentes funcionales.

288 Microproyecto con React

Para finalizar este módulo vamos a crear un microproyecto con React y Vite. En este proyecto vamos a crear una aplicación de lista de tareas (To-Do List) que permita agregar, eliminar y marcar tareas como completadas.

288.1 1. Crear un Proyecto con Vite

Para comenzar, vamos a crear un nuevo proyecto React con Vite. Abre tu terminal y ejecuta el siguiente comando:

```
npm create vite@latest todo-list
cd todo-list
npm install
```

Este comando creará un nuevo proyecto React con Vite llamado todo-list y lo instalará en tu directorio actual.

288.2 2. Crear Componentes

En el directorio **src/components**, crea un nuevo archivo llamado **TodoList.jsx** con el siguiente contenido:

```
import { useState } from 'react';

export default function TodoList() {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState('');

  function addTask() {
    if (newTask.trim() !== '') {
      setTasks([...tasks, { id: tasks.length + 1, text: newTask, completed: false }]);
      setNewTask('');
    }
  }

  function deleteTask(id) {
    setTasks(tasks.filter((task) => task.id !== id));
  }

  function toggleTask(id) {
    setTasks(tasks.map((task) => {
      if (task.id === id) {
        return { ...task, completed: !task.completed };
      }
      return task;
    }));
  }
```

```
  return (
    <div>
      <h1>Lista de Tareas</h1>
      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Nueva tarea"
      />
      <button onClick={addTask}>Agregar Tarea</button>
      <ul>
        {tasks.map((task) => (
          <li key={task.id}>
            <input
              type="checkbox"
              checked={task.completed}
              onChange={() => toggleTask(task.id)}
            />
            <span style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>
              {task.text}
            </span>
            <button onClick={() => deleteTask(task.id)}>Eliminar</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

En el código anterior, creamos un componente funcional TodoList que gestiona una lista de tareas. Usamos el hook useState para mantener el estado de las tareas y la nueva tarea que se está escribiendo. También definimos funciones para agregar, eliminar y marcar tareas como completadas.

288.3 3. Usar el Componente TodoList

En el archivo **src/App.js**, importa y usa el componente TodoList que acabamos de crear:

```
import TodoList from './components/TodoList';

function App() {
  return (
    <div>
      <TodoList />
    </div>
```

```
  );
}

export default App;
```

288.4 4. Ejecutar el Proyecto

Finalmente, ejecuta el servidor de desarrollo para ver tu aplicación de lista de tareas en acción:

```
npm run dev
```

Con estos pasos, has creado una aplicación de lista de tareas funcional con React y Vite. Puedes agregar, eliminar y marcar tareas como completadas, demostrando cómo los componentes, el estado y los hooks de React trabajan juntos para crear aplicaciones web interactivas y dinámicas.

289 Reto

Mejorar la aplicación de lista de tareas agregando las siguientes funcionalidades:

- **Editar Tareas:** Permite editar el texto de una tarea existente.
- **Filtrar Tareas:** Agrega botones para filtrar las tareas por estado (completadas, pendientes, todas).
- **Guardar Tareas:** Implementa la capacidad de guardar las tareas en el almacenamiento local del navegador para que persistan entre sesiones.
- **Estilos Personalizados:** Añade estilos CSS personalizados para mejorar la apariencia de la aplicación.

Solución

Para implementar estas funcionalidades, puedes seguir los siguientes pasos:

1. **Editar Tareas:** Agrega un botón de edición a cada tarea que permita cambiar su texto.

```
function editTask(id, text) {
  const updatedText = prompt('Editar tarea:', text);
  if (updatedText !== null) {
    setTasks(tasks.map((task) => {
      if (task.id === id) {
```

```
        return { ...task, text: updatedText };
      }
      return task;
    }));
  }
}
```

2. **Filtrar Tareas:** Agrega botones para filtrar las tareas por estado.

```
function filterTasks(status) {
  switch (status) {
    case 'completed':
      return tasks.filter((task) => task.completed);
    case 'pending':
      return tasks.filter((task) => !task.completed);
    default:
      return tasks;
  }
}
```

3. **Guardar Tareas:** Utiliza el almacenamiento local del navegador para guardar las tareas.

```
useEffect(() => {
  const savedTasks = JSON.parse(localStorage.getItem('tasks'));
  if (savedTasks) {
    setTasks(savedTasks);
  }
}, []);

useEffect(() => {
  localStorage.setItem('tasks', JSON.stringify(tasks));
}, [tasks]);
```

El archivo **TodoList.jsx** completo con estas mejoras se vería así:

```
import { useState, useEffect } from 'react';

export default function TodoList() {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState('');

  useEffect(() => {
    const savedTasks = JSON.parse(localStorage.getItem('tasks'));
    if (savedTasks) {
      setTasks(savedTasks);
    }
```

```
}, []);

useEffect(() => {
  localStorage.setItem('tasks', JSON.stringify(tasks));
}, [tasks]);

function addTask() {
  if (newTask.trim() !== '') {
    setTasks([...tasks, { id: tasks.length + 1, text: newTask, completed: false }]);
    setNewTask('');
  }
}

function deleteTask(id) {
  setTasks(tasks.filter((task) => task.id !== id));
}

function toggleTask(id) {
  setTasks(tasks.map((task) => {
    if (task.id === id) {
      return { ...task, completed: !task.completed };
    }
    return task;
  }));
}

function editTask(id, text) {
  const updatedText = prompt('Editar tarea:', text);
  if (updatedText !== null) {
    setTasks(tasks.map((task) => {
      if (task.id === id) {
        return { ...task, text: updatedText };
      }
      return task;
    }));
  }
}

function filterTasks(status) {
  switch (status) {
    case 'completed':
      return tasks.filter((task) => task.completed);
    case 'pending':
      return tasks.filter((task) => !task.completed);
    default:
      return tasks;
  }
}
```

```
  return (
    <div>
      <h1>Lista de Tareas</h1>
      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Nueva tarea"
      />
      <button onClick={addTask}>Agregar Tarea</button>
      <ul>
        {filterTasks('all').map((task) => (
          <li key={task.id}>
            <input
              type="checkbox"
              checked={task.completed}
              onChange={() => toggleTask(task.id)}
            />
            <span style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>
              {task.text}
            </span>
            <button onClick={() => editTask(task.id, task.text)}>
              Editar
            </button>
            <button onClick={() => deleteTask(task.id)}>
              Eliminar
            </button>
          </li>
        ))}
      </ul>
      <div>
        <button onClick={() => setTasks(filterTasks('all'))}>
          Todas
        </button>
        <button onClick={() => setTasks(filterTasks('completed'))}>
          Completadas
        </button>
        <button onClick={() => setTasks(filterTasks('pending'))}>
          Pendientes
        </button>
      </div>
    </div>
  );
}
```

4. **Estilos Personalizados:** Crea un archivo CSS en el directorio **src** y agrega estilos personalizados.

```
body {
  font-family: Arial, sans-serif;
}

button {
  padding: 0.5rem 1rem;
  margin: 0.5rem;
  cursor: pointer;
}

li {
  display: flex;
  align-items: center;
  margin: 0.5rem 0;
}

input[type="checkbox"] {
  margin-right: 1rem;
}

span {
  flex: 1;
}

button {
  background-color: #f44336;
  color: white;
  border: none;
  border-radius: 4px;
}

button:hover {
  background-color: #d32f2f;
}

button:active {
  background-color: #b71c1c;
}

input[type="text"] {
  padding: 0.5rem;
  margin-right: 1rem;
}

input[type="checkbox"] {
  transform: scale(1.5);
}
```

```
ul {
  list-style-type: none;
  padding: 0;
}

h1 {
  text-align: center;
}

button.add {
  background-color: #4caf50;
}

button.add:hover {
  background-color: #388e3c;
}

button.add:active {
  background-color: #2e7d32;
}

button.edit {
  background-color: #2196f3;
}

button.edit:hover {
  background-color: #1e88e5;
}

button.edit:active {
  background-color: #1976d2;
}

button.delete {
  background-color: #f44336;
}

button.delete:hover {
  background-color: #d32f2f;
}

button.delete:active {
  background-color: #b71c1c;
}

button.filter {
  background-color: #9e9e9e;
}
```

```
button.filter:hover {
  background-color: #757575;
}

button.filter:active {
  background-color: #424242;
}

button.clear {
  background-color: #f44336;
}

button.clear:hover {
  background-color: #d32f2f;
}

button.clear:active {
  background-color: #b71c1c;
}

button.save {
  background-color: #4caf50;
}

button.save:hover {
  background-color: #388e3c;
}

button.save:active {
  background-color: #2e7d32;
}

button.cancel {
  background-color: #f44336;
}

button.cancel:hover {
  background-color: #d32f2f;
}

button.cancel:active {
  background-color: #b71c1c;
}

button.toggle {
  background-color: #4caf50;
}
```

```
button.toggle:hover {
  background-color: #388e3c;
}

button.toggle:active {
  background-color: #2e7d32;
}

button.toggle-all {
  background-color: #9e9e9e;
}

button.toggle-all:hover {
  background-color: #757575;
}

button.toggle-all:active {
  background-color: #424242;
}

button.clear-completed {
  background-color: #f44336;
}

button.clear-completed:hover {
  background-color: #d32f2f;
}

button.clear-completed:active {
  background-color: #b71c1c;
}

button.clear-all {
  background-color: #f44336;
}

button.clear-all:hover {
  background-color: #d32f2f;
}

button.clear-all:active {
  background-color: #b71c1c;
}

button.save-all {
  background-color: #4caf50;
}
```

```
button.save-all:hover {
  background-color: #388e3c;
}

button.save-all:active {
  background-color: #2e7d32;
}

button.cancel-all {
  background-color: #f44336;
}

button.cancel-all:hover {
  background-color: #d32f2f;
}

button.cancel-all:active {
  background-color: #b71c1c;
}

button.filter-all {
  background-color: #9e9e9e;
}

button.filter-all:hover {
  background-color: #757575;
}

button.filter-all:active {
  background-color: #424242;
}

button.filter-completed {
  background-color: #9e9e9e;
}

button.filter-completed:hover {
  background-color: #757575;
}

button.filter-completed:active {
  background-color: #424242;
}

button.filter-pending {
  background-color: #9e9e9e;
}
```

```
button.filter-pending:hover {
  background-color: #757575;
}

button.filter-pending:active {
  background-color: #424242;
}

button.filter-clear {
  background-color: #f44336;
}

button.filter-clear:hover {
  background-color: #d32f2f;
}

button.filter-clear:active {
  background-color: #b71c1c;
}

button.filter-save {
  background-color: #4caf50;
}

button.filter-save:hover {
  background-color: #388e3c;
}

button.filter-save:active {
  background-color: #2e7d32;
}

button.filter-cancel {
  background-color: #f44336;
}

button.filter-cancel:hover {
  background-color: #d32f2f;
}

button.filter-cancel:active {
  background-color: #b71c1c;
}

button.filter-toggle {
  background-color: #4caf50;
}
```

```
button.filter-toggle:hover {
  background-color: #388e3c;
}

button.filter-toggle:active {
  background-color: #2e7d32;
}

button.filter-toggle-all {
  background-color: #9e9e9e;
}

button.filter-toggle-all:hover {
  background-color: #757575;
}

button.filter-toggle-all:active {
  background-color: #424242;
}

button.filter-clear-completed {
  background-color: #f44336;
}

button.filter-clear-completed:hover {
  background-color: #d32f2f;
}

button.filter-clear-completed:active {
  background-color: #b71c1c;
}

button.filter-clear-all {
  background-color: #f44336;
}

button.filter-clear-all:hover {
  background-color: #d32f2f;
}

button.filter-clear-all:active {
  background-color: #b71c1c;
}

button.filter-save-all {
  background-color: #4caf50;
}
```

```
button.filter-save-all:hover {
  background-color: #388e3c;
}

button.filter-save-all:active {
  background-color: #2e7d32;
}

button.filter-cancel-all {
  background-color: #f44336;
}

button.filter-cancel-all:hover {
  background-color: #d32f2f;
}

button.filter-cancel-all:active {
  background-color: #b71c1c;
}

button.filter-all-all {
  background-color: #9e9e9e;
}

button.filter-all-all:hover {
  background-color: #757575;
}

button.filter-all-all:active {
  background-color: #424242;
}

button.filter-completed-completed {
  background-color: #9e9e9e;
}

button.filter-completed-completed:hover {
  background-color: #757575;
}

button.filter-completed-completed:active {
  background-color: #424242;
}

button.filter-pending-pending {
  background-color: #9e9e9e;
}
```

```
button.filter-pending-pending:hover {
  background-color: #757575;
}

button.filter-pending-pending:active {
  background-color: #424242;
}

button.filter-clear-clear {
  background-color: #f44336;
}
```

Con estas mejoras, tu aplicación de lista de tareas será más interactiva, funcional y atractiva visualmente. ¡Sigue experimentando y mejorando tus habilidades con React!

290 Conclusión: React Antes y Ahora con Vite

React ha evolucionado significativamente desde sus inicios, pasando de componentes de clase y callbacks a componentes funcionales y hooks. La introducción de Vite ha mejorado aún más la experiencia de desarrollo al proporcionar una configuración ligera y un rendimiento excepcional.

291 Consumir una API con React con Fetch.

291.1 1. Crear un Proyecto React con Vite y Tailwind CSS

Vamos a crear un proyecto React utilizando Vite y configuraremos Tailwind CSS para el estilizado.

291.1.1 Pasos:

291.1.1.1 Crear un nuevo proyecto con Vite:

Abre tu terminal y ejecuta:

```
mkdir my-fake-store
cd my-fake-store
npm create vite@latest .
```

291.1.1.2 Instalar las dependencias:

Instala las dependencias necesarias para el proyecto:

```
npm install
```

291.1.1.3 Instalar Tailwind CSS:

Ejecuta los siguientes comandos para instalar Tailwind CSS y sus dependencias:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init
```

291.1.1.4 Configurar Tailwind CSS:

Agrega las rutas a los archivos de tu proyecto en el archivo tailwind.config.cjs:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './index.html',
    './src/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

291.1.1.5 Añadir directivas de Tailwind CSS:

Crea un archivo src/index.css (si no existe) y añade las directivas de Tailwind CSS:

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

291.1.1.6 Iniciar el servidor de desarrollo:

Ejecuta el siguiente comando para iniciar el servidor de desarrollo:

```
npm run dev
```

Abre tu navegador en http://localhost:5173 para ver la aplicación en funcionamiento.

291.2 2. Estructura del Proyecto

Separaremos la lógica de la aplicación en componentes más pequeños:

- **ProductCard:** Componente que mostrará cada producto individual.
- **ProductList:** Componente que gestionará la lista de productos.

La estructura de directorios será:

```
src/

   components/
      ProductCard.jsx
      ProductList.jsx
   App.jsx
```

291.3 3. Creación de los Componentes

291.3.1 ProductCard.jsx

Este componente mostrará la información de cada producto de manera estilizada con Tailwind CSS.

```
import React from 'react';

function ProductCard({ product }) {
  return (
    <div className="border border-gray-200 p-4 rounded-lg shadow-sm hover:shadow-lg trans
      <img
        className="w-32 h-32 object-contain mx-auto"
        src={product.image}
        alt={product.title}
```

```
      />
      <h3 className="text-lg font-semibold mt-4">{product.title}</h3>
      <p className="text-gray-500 mt-2">{product.category}</p>
      <p className="text-green-600 font-bold mt-2">{product.price} USD</p>
    </div>
  );
}

export default ProductCard;
```

291.3.2 ProductList.jsx

Este componente gestionará la lógica de la API y renderizará los productos usando el componente ProductCard.

```
import React, { useState, useEffect } from 'react';
import ProductCard from './ProductCard';

function ProductList() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  const fetchProducts = async () => {
    try {
      const response = await fetch('https://fakestoreapi.com/products');
      if (!response.ok) {
        throw new Error('Algo salió mal al obtener los productos');
      }
      const data = await response.json();
      setProducts(data);
      setLoading(false);
    } catch (error) {
      setError(error.message);
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchProducts();
  }, []);

  if (loading) return <p className="text-center text-gray-500">Cargando productos...</p>;
  if (error) return <p className="text-center text-red-500">Error: {error}</p>;

  return (
    <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">
```

```
      {products.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}

export default ProductList;
```

291.4 4. Integración en App.jsx

Finalmente, integra el componente ProductList dentro de tu archivo App.jsx.

```
import React from 'react';
import ProductList from './components/ProductList';

function App() {
  return (
    <div className="container mx-auto px-4 py-8">
      <h1 className="text-3xl font-bold text-center mb-8">Productos de Fake Store</h1>
      <ProductList />
    </div>
  );
}

export default App;
```

291.5 5. Estilizado con Tailwind CSS

Tailwind CSS te permite estilizar la aplicación de manera eficiente. Aquí hay un desglose de algunas clases utilizadas:

- **grid-cols-1 sm:grid-cols-2 lg:grid-cols-4:** Define la cuadrícula con 1 columna por defecto, 2 columnas en pantallas medianas, y 4 columnas en pantallas grandes.
- **border, rounded-lg, shadow-sm:** Añaden bordes, esquinas redondeadas y sombras suaves.
- **transition, hover:shadow-lg:** Crea un efecto de transición en el hover.

291.6 6. Ejecución del Proyecto

Con todo el código en su lugar, puedes ejecutar tu aplicación:

```
npm run dev
```

Visita http://localhost:5173 en tu navegador para ver la lista de productos de la Fake Store API estilizados con Tailwind CSS.

292 Extra

292.1 Agregar el Cambio de Tema de Oscuro a Claro

Para implementar la funcionalidad de cambio de tema entre oscuro y claro, sigue estos pasos:

292.1.1 Actualiza App.jsx:

Utiliza el siguiente código en tu archivo App.jsx para manejar el cambio de tema y aplicar estilos condicionales:

```
import React, { useState } from 'react';
import ProductList from './components/ProductList';

function App() {
  const [darkMode, setDarkMode] = useState(false);

  return (
    <div className={`min-h-screen ${darkMode ? 'dark' : ''}`}>
      <div className="container mx-auto px-4 py-8">
        <div className="flex justify-end mb-4">
          <button
            className="px-4 py-2 bg-gray-200 text-gray-800 rounded-lg dark:bg-gray-800 da
            onClick={() => setDarkMode(!darkMode)}
          >
            {darkMode ? 'Modo Claro' : 'Modo Oscuro'}
          </button>
        </div>
        <h1 className="text-3xl font-bold text-center mb-8 text-gray-900 dark:text-gray-1
          Productos de Fake Store
        </h1>
        <ProductList />
      </div>
    </div>
  );
}

export default App;
```

292.1.2 Cambios importantes:

- Se añadió **min-h-screen** para asegurar que el contenedor principal ocupe toda la altura de la pantalla.
- Se **actualizaron** las **clases** del **botón** y del **encabezado** para manejar el cambio de estilo en modo oscuro.

292.1.3 Asegúrate de Configurar Tailwind para el Modo Oscuro:

Verifica que tu archivo tailwind.config.cjs esté configurado correctamente para soportar el modo oscuro:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  darkMode: 'class', // Habilita el modo oscuro utilizando la clase 'dark'
  content: [
    './index.html',
    './src/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Esta configuración permite que el modo oscuro se active añadiendo la clase dark al contenedor principal.

292.1.4 Actualiza los Estilos en los Componentes:

Asegúrate de que todos los componentes respondan al modo oscuro. Por ejemplo, en ProductCard.jsx:

```
import React from 'react';

function ProductCard({ product }) {
  return (
    <div className="border border-gray-200 dark:border-gray-700 p-4 rounded-lg shadow-sm
      <img
        className="w-32 h-32 object-contain mx-auto"
        src={product.image}
        alt={product.title}
      />
      <h3 className="text-lg font-semibold mt-4 text-gray-900 dark:text-gray-100">
        {product.title}
      </h3>
```

```
      <p className="text-gray-500 mt-2 dark:text-gray-400">{product.category}</p>
      <p className="text-green-600 font-bold mt-2">{product.price} USD</p>
    </div>
  );
}

export default ProductCard;
```

292.1.5 Cambios importantes:

- Se actualizaron las clases para manejar el estilo en modo oscuro.

292.2 Crear un Build de Producción

Para crear un build optimizado para producción, ejecuta el siguiente comando:

```
npm run build
```

Este comando generará una carpeta dist con los archivos optimizados para producción.

> **Tip**
>
> Para probar el build de producción, puedes usar npx para ejecutar el servidor de pruebas sin necesidad de instalación global.
> Ejecuta el siguiente comando:
>
> ```
> npx serve -s dist
> ```
>
> Luego, abre tu navegador en http://localhost:5000 para ver la aplicación en funcionamiento.

293 Conclusiones

En este tutorial, aprendiste a consumir una API con React y a estilizar la aplicación con Tailwind CSS. También implementaste la funcionalidad de cambio de tema entre oscuro y claro. Ahora puedes aplicar estos conceptos en tus propios proyectos para crear aplicaciones web modernas y atractivas.

294 Consumir una API con React con Axios.

294.1 1. Crear un Proyecto React con Vite y Tailwind CSS

Vamos a crear un proyecto React utilizando Vite y configuraremos Tailwind CSS para el estilizado.

294.2 Pasos:

- Crear un nuevo proyecto con Vite:

Abre tu terminal y ejecuta:

```
mkdir my-fake-store-axios
cd my-fake-store-axios
npm create vite@latest .
```

- Instalar las dependencias:

Instala las dependencias necesarias para el proyecto:

```
npm install
```

- Instalar Tailwind CSS:

Ejecuta los siguientes comandos para instalar Tailwind CSS y sus dependencias:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init
```

- Configurar Tailwind CSS:

Agrega las rutas a los archivos de tu proyecto en el archivo tailwind.config.cjs:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  darkMode: 'class', // Habilita el modo oscuro utilizando la clase 'dark'
  content: [
    './index.html',
    './src/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

- Crear y configurar **postcss.config.cjs**:

Crea un archivo **postcss.config.cjs** en la raíz de tu proyecto y añade el siguiente contenido:

```
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
}
```

- Añadir directivas de Tailwind CSS:

Crea un archivo src/index.css (si no existe) y añade las directivas de Tailwind CSS:

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

- Iniciar el servidor de desarrollo:

Ejecuta el siguiente comando para iniciar el servidor de desarrollo:

```
npm run dev
```

Abre tu navegador en http://localhost:5173 para ver la aplicación en funcionamiento.

294.3 2. Estructura del Proyecto

Separaremos la lógica de la aplicación en componentes más pequeños:

- **ProductCard:** Componente que mostrará cada producto individual.
- **ProductList:** Componente que gestionará la lista de productos.

La estructura de directorios será:

```
src/

   components/
      ProductCard.jsx
      ProductList.jsx
   App.jsx
```

294.4 3. Creación de los Componentes

294.4.1 ProductCard.jsx

Este componente mostrará la información de cada producto de manera estilizada con Tailwind CSS.

```
import React from 'react';

function ProductCard({ product }) {
  return (
    <div className="border border-gray-200 dark:border-gray-700 p-4 rounded-lg shadow-sm
      <img
        className="w-32 h-32 object-contain mx-auto"
```

```
        src={product.image}
        alt={product.title}
      />
      <h3 className="text-lg font-semibold mt-4 text-gray-900 dark:text-gray-100">
        {product.title}
      </h3>
      <p className="text-gray-500 mt-2 dark:text-gray-400">{product.category}</p>
      <p className="text-green-600 font-bold mt-2">{product.price} USD</p>
    </div>
  );
}

export default ProductCard;
```

294.4.2 ProductList.jsx

Este componente gestionará la lógica de la API y renderizará los productos usando el componente ProductCard.

```
import React, { useState, useEffect } from 'react';
import ProductCard from './ProductCard';

function ProductList() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  const fetchProducts = async () => {
    try {
      const response = await fetch('https://fakestoreapi.com/products');
      if (!response.ok) {
        throw new Error('Algo salió mal al obtener los productos');
      }
      const data = await response.json();
      setProducts(data);
      setLoading(false);
    } catch (error) {
      setError(error.message);
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchProducts();
  }, []);

  if (loading) return <p className="text-center text-gray-500">Cargando productos...</p>;
```

```
  if (error) return <p className="text-center text-red-500">Error: {error}</p>;

  return (
    <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">
      {products.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}

export default ProductList;
```

294.5 4. Integración en App.jsx

Finalmente, integra el componente ProductList dentro de tu archivo App.jsx y añade la funcionalidad de cambio de tema entre oscuro y claro.

```
import React, { useState } from 'react';
import ProductList from './components/ProductList';

function App() {
  const [darkMode, setDarkMode] = useState(false);

  return (
    <div className={`min-h-screen ${darkMode ? 'dark' : ''}`}>
      <div className="container mx-auto px-4 py-8">
        <div className="flex justify-end mb-4">
          <button
            className="px-4 py-2 bg-gray-200 text-gray-800 rounded-lg dark:bg-gray-800 da
            onClick={() => setDarkMode(!darkMode)}
          >
            {darkMode ? 'Modo Claro' : 'Modo Oscuro'}
          </button>
        </div>
        <h1 className="text-3xl font-bold text-center mb-8 text-gray-900 dark:text-gray-1
          Productos de Fake Store
        </h1>
        <ProductList />
      </div>
    </div>
  );
}

export default App;
```

294.6 5. Estilizado con Tailwind CSS

Tailwind CSS te permite estilizar la aplicación de manera eficiente. Aquí hay un desglose de algunas clases utilizadas:

- min-h-screen: Asegura que el contenedor principal ocupe toda la altura de la pantalla.
- dark: Clase utilizada para activar el modo oscuro.
- border, rounded-lg, shadow-sm: Añaden bordes, esquinas redondeadas y sombras suaves.
- transition, hover:shadow-lg: Crea un efecto de transición en el hover.

294.7 6. Ejecución del Proyecto

Con todo el código en su lugar, puedes ejecutar tu aplicación:

```
npm run dev
```

Visita http://localhost:5173 en tu navegador para ver la lista de productos de la Fake Store API estilizados con Tailwind CSS.

294.7.1 Crear un Build de Producción

Para crear un build optimizado para producción, ejecuta el siguiente comando:

```
npm run build
```

Este comando generará una carpeta dist con los archivos optimizados para producción.

> **Tip**
>
> Para probar el build de producción, puedes usar npx para ejecutar el servidor de pruebas sin necesidad de instalación global.
> Ejecuta el siguiente comando:
>
> ```
> npx serve -s dist
> ```
>
> Luego, abre tu navegador en http://localhost:5000 para ver la aplicación en funcionamiento.

295 Conclusiones

En este tutorial, aprendiste a consumir una API con React y a estilizar la aplicación con Tailwind CSS. También implementaste la funcionalidad de cambio de tema entre oscuro y claro. Ahora puedes aplicar estos conceptos en tus propios proyectos para crear aplicaciones web modernas y atractivas.

296 CRUD en React con API REST

296.1 Requisitos previos:

- Node.js y npm (o yarn, o pnpm) instalados en tu sistema.
- Conocimientos básicos de React y JavaScript.
- Backend (API) funcionando, por ejemplo, usando Django REST API u otra API compatible.

1. Inicializa el proyecto de React

Primero, necesitas crear un proyecto de React. Vamos a usar npm create vite@latest para hacerlo de forma rápida:

```
npm create vite@latest nombre-del-proyecto
```

Sigue las indicaciones y selecciona react como plantilla. Luego navega al directorio del proyecto e instala las dependencias necesarias:

```
cd nombre-del-proyecto
npm install
```

2. Configura Axios para las solicitudes a la API

Axios es una librería que facilita las solicitudes HTTP. Crea un archivo api.js en la carpeta src/utils para configurar Axios con la URL base de tu API:

```
mkdir -p src/utils
touch src/utils/api.js
```

Ahora, edita el archivo api.js:

```
import axios from 'axios';

const api = axios.create({
  baseURL: 'https://tu-api-backend.com/api', // Reemplaza esta URL con la URL de tu API
  headers: {
    'Content-Type': 'application/json',
  },
});

export default api;
```

Este archivo configura Axios con la base de la URL de tu API, facilitando la reutilización de este cliente HTTP en toda tu aplicación.

3. Configura las rutas y el sistema de navegación con react-router-dom

Para manejar las rutas de la aplicación, instala react-router-dom:

```
npm install react-router-dom
```

Modifica tu archivo src/App.jsx para agregar rutas:

```
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import ProductList from './components/ProductList';
import ProductDetail from './components/ProductDetail';
import ProductForm from './components/ProductForm';

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/products" element={<ProductList />} />
        <Route path="/products/new" element={<ProductForm />} />
        <Route path="/products/edit/:id" element={<ProductForm />} />
        <Route path="/products/:id" element={<ProductDetail />} />
      </Routes>
    </Router>
  );
};

const Home = () => (
  <div>
    <h1>CRUD en React con API REST</h1>
    <p>Esta es una aplicación de ejemplo para realizar operaciones CRUD (Crear, Leer, Act
    <p>Dirígete a <Link to="/products">Productos</Link> para comenzar.</p>
  </div>
);

export default App;
```

Aquí estamos usando react-router-dom para definir nuestras rutas. Tenemos rutas para listar los productos, crear uno nuevo, editar un producto existente y ver los detalles de un producto específico. También hemos agregado una ruta para una página de inicio con instrucciones.

4. Crea el componente ProductList (Leer)

Este componente mostrará una lista de productos obtenidos de la API:

```
mkdir -p src/components
touch src/components/ProductList.jsx
```

Ahora, edita ProductList.jsx para obtener y mostrar la lista de productos:

```
import { useState, useEffect } from 'react';
import api from '../utils/api';
import { Link } from 'react-router-dom';

const ProductList = () => {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    // Obtener todos los productos
    api.get('/products')
      .then(response => {
        setProducts(response.data);
      })
      .catch(error => {
        console.error("Error al obtener los productos: ", error);
      });
  }, []);

  return (
    <div>
      <h1>Lista de Productos</h1>
      <ul>
        {products.map(product => (
          <li key={product.id}>
            <Link to={`/products/${product.id}`}>{product.name}</Link>
          </li>
        ))}
      </ul>
      <Link to="/products/new">Crear Nuevo Producto</Link>
    </div>
  );
};

export default ProductList;
```

Utilizamos useEffect para hacer la solicitud GET a la API cuando el componente se monta. Los productos obtenidos se almacenan en el estado products y se muestran en una lista con enlaces a los detalles de cada producto.

5. Crea el componente ProductDetail (Leer uno y eliminar)

Este componente mostrará los detalles de un producto específico y permitirá eliminarlo:

```
touch src/components/ProductDetail.jsx
```

Edita ProductDetail.jsx:

```
import { useState, useEffect } from 'react';
import { useParams, useNavigate, Link } from 'react-router-dom';
import api from '../utils/api';

const ProductDetail = () => {
  const { id } = useParams();
  const [product, setProduct] = useState(null);
  const navigate = useNavigate();

  useEffect(() => {
    // Obtener los detalles de un producto
    api.get(`/products/${id}`)
      .then(response => {
        setProduct(response.data);
      })
      .catch(error => {
        console.error("Error al obtener el producto: ", error);
      });
  }, [id]);

  const handleDelete = () => {
    // Eliminar producto
    api.delete(`/products/${id}`)
      .then(() => {
        navigate('/products');
      })
      .catch(error => {
        console.error("Error al eliminar el producto: ", error);
      });
  };

  if (!product) return <div>Cargando...</div>;

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <button onClick={handleDelete}>Eliminar</button>
      <Link to={`/products/edit/${product.id}`}>Editar</Link>
    </div>
  );
};

export default ProductDetail;
```

Aquí usamos useParams para obtener el ID del producto de la URL. Hacemos una solicitud GET para obtener los detalles del producto y mostramos su nombre y descripción. También tenemos una función handleDelete que elimina el producto y redirige a la lista de productos.

6. Crea el componente ProductForm (Crear y Actualizar)

Este componente servirá tanto para crear un nuevo producto como para editar uno existente:

```
touch src/components/ProductForm.jsx
```

Edita ProductForm.jsx:

```
import { useState, useEffect } from 'react';
import { useNavigate, useParams } from 'react-router-dom';
import api from '../utils/api';

const ProductForm = () => {
  const [name, setName] = useState('');
  const [description, setDescription] = useState('');
  const { id } = useParams();
  const navigate = useNavigate();

  useEffect(() => {
    if (id) {
      // Si hay un `id`, estamos editando un producto
      api.get(`/products/${id}`)
        .then(response => {
          const product = response.data;
          setName(product.name);
          setDescription(product.description);
        })
        .catch(error => {
          console.error("Error al obtener los detalles del producto: ", error);
        });
    }
  }, [id]);

  const handleSubmit = (e) => {
    e.preventDefault();
    const product = { name, description };

    if (id) {
      // Actualizar producto
      api.put(`/products/${id}`, product)
        .then(() => {
          navigate(`/products/${id}`);
```

```
        })
        .catch(error => {
          console.error("Error al actualizar el producto: ", error);
        });
    } else {
      // Crear nuevo producto
      api.post('/products', product)
        .then(() => {
          navigate('/products');
        })
        .catch(error => {
          console.error("Error al crear el producto: ", error);
        });
    }
  };

  return (
    <div>
      <h1>{id ? 'Editar Producto' : 'Crear Producto'}</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Nombre</label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
          />
        </div>
        <div>
          <label>Descripción</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            required
          ></textarea>
        </div>
        <button type="submit">{id ? 'Actualizar' : 'Crear'}</button>
      </form>
    </div>
  );
};

export default ProductForm;
```

Este formulario maneja tanto la creación como la edición de productos. Si se pasa un ID a través de la URL, el formulario se rellenará con los datos del producto y permitirá la edición. Si no hay ID, se considera una creación de nuevo producto.

7. Ejecuta y prueba la aplicación

Finalmente, inicia el servidor de desarrollo y verifica que todo funcione correctamente:

```
npm run dev
```

Abre tu navegador y navega a http://localhost:5173 (o la URL que indique el servidor de desarrollo) para probar tu aplicación CRUD.

297 Conclusiones

En este tutorial, aprendiste a crear una aplicación CRUD en React que interactúa con una API REST. Implementamos las operaciones de Crear, Leer, Actualizar y Eliminar productos utilizando Axios para las solicitudes HTTP y react-router-dom para la navegación. Este es un ejemplo básico que puedes ampliar y personalizar según tus necesidades. ¡Esperamos que te haya sido útil!

> **Tip**
>
> Si deseas profundizar en el tema, te recomendamos explorar otras funcionalidades como paginación, autenticación, validación de formularios, entre otros.

Part VII

Ejercicios

298 Desarrollo del Backend para un E-Commerce con Django Rest Framework

298.1 1. Configuración Inicial del Proyecto

298.1.1 1.1. Crear un Proyecto Django

Abre tu terminal y ejecuta los siguientes comandos para crear un nuevo proyecto Django:

```
python -m venv env
source env/bin/activate
pip install django==4.2
django-admin startproject ecommerce_project .
cd ecommerce_project
```

298.1.2 1.2 Crear una Aplicación Django

Dentro del directorio del proyecto, crea una aplicación para manejar el e-commerce:

```
python manage.py startapp products
```

298.1.3 1.3 Instalar Django Rest Framework

Instala DRF usando pip:

```
pip install djangorestframework
```

298.1.4 1.4 Configurar el Proyecto

Añade 'rest_framework' y tu nueva aplicación 'products' a la lista INSTALLED_APPS en ecommerce_project/settings.py:

```
INSTALLED_APPS = [
    # ... otras apps
    'rest_framework',
    'products',
]
```

298.2 2. Definir el Modelo de Datos

298.2.1 2.1 Crear Modelos en products/models.py

Define los modelos para el e-commerce, como Product, Category, y Order:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    category = models.ForeignKey(Category, related_name='products', on_delete=models.CASC
    stock = models.PositiveIntegerField()

    def __str__(self):
        return self.name

class Order(models.Model):
    product = models.ForeignKey(Product, related_name='orders', on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField()
    total_price = models.DecimalField(max_digits=10, decimal_places=2)
    order_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Order {self.id} - {self.product.name}"
```

298.2.2 2.2 Crear y Aplicar Migraciones

Genera y aplica las migraciones para los modelos:

```
python manage.py makemigrations
python manage.py migrate
```

298.3 3. Crear Serializers

298.3.1 3.1 Definir Serializers en products/serializers.py

Los serializers se encargan de transformar los modelos en formatos JSON y viceversa:

```
from rest_framework import serializers
from .models import Category, Product, Order

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = '__all__'

class ProductSerializer(serializers.ModelSerializer):
    category = CategorySerializer()

    class Meta:
        model = Product
        fields = '__all__'

class OrderSerializer(serializers.ModelSerializer):
    product = ProductSerializer()

    class Meta:
        model = Order
        fields = '__all__'
```

298.4 4. Crear Vistas y Rutas

298.4.1 4.1 Definir Vistas en products/views.py

Utiliza las vistas basadas en clases de DRF para crear y manejar las operaciones CRUD:

```
from rest_framework import generics
from .models import Category, Product, Order
from .serializers import CategorySerializer, ProductSerializer, OrderSerializer

class CategoryListCreate(generics.ListCreateAPIView):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class CategoryDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class ProductListCreate(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

class ProductDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
```

```
    serializer_class = ProductSerializer

class OrderListCreate(generics.ListCreateAPIView):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer

class OrderDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer
```

298.5 4.2 Configurar Rutas en products/urls.py

Define las rutas para acceder a las vistas:

```
from django.urls import path
from . import views

urlpatterns = [
    path('categories/', views.CategoryListCreate.as_view(), name='category-list-create'),
    path('categories/<int:pk>/', views.CategoryDetail.as_view(), name='category-detail'),
    path('products/', views.ProductListCreate.as_view(), name='product-list-create'),
    path('products/<int:pk>/', views.ProductDetail.as_view(), name='product-detail'),
    path('orders/', views.OrderListCreate.as_view(), name='order-list-create'),
    path('orders/<int:pk>/', views.OrderDetail.as_view(), name='order-detail'),
]
```

298.6 4.3 Incluir las URLs en ecommerce_project/urls.py

Añade las URLs de la aplicación al archivo principal de URLs del proyecto:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('products.urls')),
]
```

298.7 5. Probar la API

298.7.1 5.1 Ejecutar el Servidor de Desarrollo

Inicia el servidor de desarrollo de Django:

```
python manage.py runserver
```

298.8 5.2 Probar los Endpoints

Utiliza herramientas como Postman o cURL para probar los endpoints:

- Listar categorías: GET /api/categories/
- Crear categoría: POST /api/categories/
- Obtener categoría específica: GET /api/categories/{id}/
- Actualizar categoría: PUT /api/categories/{id}/
- Eliminar categoría: DELETE /api/categories/{id}/

Y lo mismo para productos y pedidos.

- Listar productos: GET /api/products/
- Crear producto: POST /api/products/
- Obtener producto específico: GET /api/products/{id}/
- Actualizar producto: PUT /api/products/{id}/
- Eliminar producto: DELETE /api/products/{id}/

299 Extra

Agreguemos Swagger a nuestro proyecto para tener una documentación de nuestra API.

299.1 1. Instalar Django Rest Swagger

Instala Django Rest Swagger usando pip:

```
pip install drf-yasg
```

299.2 2. Configurar Django Rest Swagger

Añade 'rest_framework_swagger' a la lista INSTALLED_APPS en ecommerce_project/settings.py:

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="E-commerce API",
        default_version='v1',
        description="API documentation for the E-commerce project",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="contact@ecommerce.local"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('products.urls')),
    path('docs/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-u
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

299.3 3. Configurar las URLs

Añade las URLs de Swagger al archivo principal de URLs del proyecto:

```
'''
from rest_framework_swagger.views import get_swagger_view

schema_view = get_swagger_view(title='E-Commerce API')

urlpatterns = [
    '''
    path('docs/', schema_view),
]
```

> **Tip**
>
> Para evitar un error común es necesario instalr **setuptools** con el siguiente comando:
>
> ```
> pip install setuptools
> ```

Finalmente es necesario agregar el siguiente código al final del archivo settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'drf_yasg',
    'rest_framework',
    'products',
]

'''
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema'
}

CORS_ALLOWED_ORIGINS = [
    "http://localhost:8000",
    "http://127.0.0.1:8000",
    # Añade otros orígenes permitidos aquí
]
```

299.4 4. Probar la Documentación

300 Ejercicios de Git y Github

300.0.1 Ejercicio 1

1. Crear un repositorio en Github
2. Clonar el repositorio en tu computadora
3. Crear un archivo de texto con tu nombre y subirlo al repositorio
4. Hacer un commit con el mensaje "Agrego mi nombre"
5. Hacer un push al repositorio

Respuesta:

```
git clone [url del repositorio]
cd [nombre del repositorio]
echo "Mi nombre es: [Tu nombre]" > nombre.txt
git add nombre.txt
git commit -m "Agrego mi nombre"
git push origin master
```

300.0.2 Ejercicio 2

1. Crear un repositorio en Github
2. Clonar el repositorio en tu computadora
3. Crear un archivo de python que imprima tu nombre
4. Hacer un commit con el mensaje "Agrego archivo de python"
5. Hacer un push al repositorio

Respuesta:

```
git clone [url del repositorio]
cd [nombre del repositorio]
echo "print('Mi nombre es: [Tu nombre]')" > nombre.py
git add nombre.py
git commit -m "Agrego archivo de python"
git push origin master
```

300.0.3 Ejercicio 3

1. Crear un repositorio en Github
2. Clonar el repositorio en tu computadora
3. Crear un archivo de python que imprima un saludo de bienvenida

4. Hacer un commit con el mensaje “Agrego saludo de bienvenida”
5. Hacer un push al repositorio

Respuesta:

```
git clone [url del repositorio]
cd [nombre del repositorio]
echo "print('Hola, bienvenido')" > saludo.py
git add saludo.py
git commit -m "Agrego saludo de bienvenida"
git push origin master
```

300.0.4 Ejercicio 4

1. Crear un repositorio en Github
2. Clonar el repositorio en tu computadora
3. Crear un archivo de python que imprima un saludo de despedida
4. Hacer un commit con el mensaje “Agrego saludo de despedida”
5. Hacer un push al repositorio

Respuesta:

```
git clone [url del repositorio]
cd [nombre del repositorio]
echo "print('Adios, hasta luego')" > despedida.py
git add despedida.py
git commit -m "Agrego saludo de despedida"
git push origin master
```

300.0.5 Ejercicio 5

1. Crear un repositorio en Github
2. Clonar el repositorio en tu computadora
3. Crear un archivo de python que imprima un saludo de bienvenida y un saludo de despedida
4. Hacer un commit con el mensaje “Agrego saludo de bienvenida y despedida”
5. Hacer un push al repositorio

Respuesta:

```
git clone [url del repositorio]
cd [nombre del repositorio]
echo "print('Hola, bienvenido')" > saludo.py
echo "print('Adios, hasta luego')" > despedida.py
git add saludo.py despedida.py
git commit -m "Agrego saludo de bienvenida y despedida"
git push origin master
```

301 Ejercicios Python - Nivel 1

301.1 Ejercicio 1

- Crear un programa que muestre por pantalla la cadena “Hola Mundo!”.

Solución

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

301.2 Ejercicio 2

- Crear un programa que muestre por pantalla tu nombre.

Solución

```
print("Tu nombre")
```

301.3 Ejercicio 3

- Crear un programa que pida al usuario que introduzca su nombre y muestre por pantalla la cadena “Hola”, seguido del nombre y un signo de exclamación.

Solución

```
nombre = input("Introduce tu nombre: ")
print("Hola", nombre, "!")
```

Otra forma de hacerlo:

```
nombre = input("Introduce tu nombre: ")
print(f"Hola {nombre}!")
```

301.4 Ejercicio 4

- Crear un programa que pregunte al usuario por el número de horas trabajadas y el coste por hora. Después debe mostrar por pantalla la paga que le corresponde.

Solución

```
horas = float(input("Introduce tus horas de trabajo: "))
coste = float(input("Introduce lo que cobras por hora: "))
paga = horas * coste
print("Tu paga es de", paga)
```

301.5 Ejercicio 5

- Crear un programa que pida al usuario una cantidad de dolares, una tasa de interés y un número de años. Mostrar por pantalla en cuanto se habrá convertido el capital inicial transcurridos esos años si cada año se aplica la tasa de interés introducida.
- Formula del interés compuesto: Cn = C * (1 + x/100) ^ n

Solución

```
cantidad = float(input("¿Cantidad a invertir? "))
interes = float(input("¿Interés porcentual anual? "))
años = int(input("¿Años?"))
print("Capital final: ", round(cantidad * (interes / 100 + 1) ** años, 2))
```

302 Ejercicios Python - Nivel 2

302.1 Ejercicio 1

- Crear una función que reciba una lista de números y devuelva su media aritmética.
- Ejemplo: media_aritmetica([1, 2, 3, 4, 5]) -> 3.0

Posible solución

```
def media_aritmetica(lista):
    return sum(lista) / len(lista)
```

302.2 Ejercicio 2

- Crear una función que reciba una lista de números y devuelva su mediana.
- Ejemplo: mediana([1, 2, 3, 4, 5]) -> 3.0

Posible solución

```
def mediana(lista):
    lista_ordenada = sorted(lista)
    n = len(lista_ordenada)
    if n % 2 == 0:
        return (lista_ordenada[n // 2 - 1] + lista_ordenada[n // 2]) / 2
    else:
        return lista_ordenada[n // 2]
```

302.3 Ejercicio 3

- Crear una función que reciba una lista de números y devuelva su moda.
- Si hay más de una moda, devolver una lista con todas las modas.
- Si no hay moda, devolver una lista vacía.
- La moda es el número que más veces se repite en una lista.
- Si todos los números se repiten el mismo número de veces, no hay moda.
- Ejemplo: moda([1, 2, 3, 2, 3, 4]) -> [2, 3]

Posible solución

```
def moda(lista):
    frecuencias = {}
    for numero in lista:
        if numero in frecuencias:
            frecuencias[numero] += 1
        else:
            frecuencias[numero] = 1
    max_frecuencia = max(frecuencias.values())
    modas = [numero for numero, frecuencia in frecuencias.items() if frecuencia == max_fr
    return modas if len(modas) > 1 else modas[0] if modas else []
```

302.4 Ejercicio 4

- Crear una función que reciba una lista de números y devuelva su desviación típica.
- La desviación típica es la raíz cuadrada de la varianza.
- La varianza es la media de los cuadrados de las diferencias entre cada número y la media aritmética.
- Ejemplo: desviacion_tipica([1, 2, 3, 4, 5]) -> 1.4142135623730951

Posible solución

```
def desviacion_tipica(lista):
    media = sum(lista) / len(lista)
    varianza = sum((numero - media) ** 2 for numero in lista) / len(lista)
    return varianza ** 0.5
```

302.5 Ejercicio 5

- Crear una función que reciba una lista de números y devuelva su coeficiente de variación.
- El coeficiente de variación es la desviación típica dividida por la media aritmética.
- Ejemplo: coeficiente_variacion([1, 2, 3, 4, 5]) -> 0.4472135954999579

Posible solución

```
def coeficiente_variacion(lista):
    media = sum(lista) / len(lista)
    desviacion_tipica = sum((numero - media) ** 2 for numero in lista) / len(lista) ** 0.
    return desviacion_tipica / media
```

303 Ejercicios Python - Nivel 3

303.1 Ejercicio 1:

- Crear una lista vacía y agregar elementos a la misma hasta que el usuario ingrese "fin".

Posible solución

```
lista = []
while True:
    elemento = input("Ingrese un elemento: ")
    if elemento == "fin":
        break
    lista.append(elemento)
print(lista)
```

303.2 Ejercicio 2:

- Crear una lista con los números del 1 al 10 y mostrar los números pares.

Posible solución

```
lista = list(range(1, 11))
for numero in lista:
    if numero % 2 == 0:
        print(numero)
```

303.3 Ejercicio 3:

- Crear una lista con los números del 1 al 10 y mostrar los números impares.

Posible solución

```
lista = list(range(1, 11))
for numero in lista:
    if numero % 2 != 0:
        print(numero)
```

303.4 Ejercicio 4:

- Crear una lista de nombres de estudiantes y mostrar aquellos cuyos nombres comiencen con la letra "A".

Posible solución

```
nombres = ["Ana", "Juan", "Pedro", "Andrea", "Lucía", "Antonio"]
for nombre in nombres:
    if nombre[0].lower() == "a":
        print(nombre)
```

303.5 Ejercicio 5:

- Crear una lista de números y mostrar aquellos que sean mayores a 100.

Posible solución

```
numeros = [10, 20, 150, 200, 300, 400, 500]
for numero in numeros:
    if numero > 100:
        print(numero)
```

304 Ejercicios Python - Nivel 4

304.1 Ejercicio 1:

- Crear un conjunto vacío y agregar elementos al mismo hasta que el usuario ingrese "fin".

Posible solución

```
conjunto = set()
while True:
    elemento = input("Ingrese un elemento o 'fin' para terminar: ")
    if elemento == "fin":
        break
    conjunto.add(elemento)
print(conjunto)
```

304.2 Ejercicio 2:

- Crear un diccionario vacío y agregar elementos al mismo hasta que el usuario ingrese "fin".
- Cada elemento debe ser una tupla con dos elementos, el primero será la clave y el segundo el valor.
- Si el usuario ingresa una clave que ya existe, se debe mostrar un mensaje de error y no agregar el elemento.

Posible solución

```
diccionario = dict()
while True:
    clave = input("Ingrese una clave o 'fin' para terminar: ")
    if clave == "fin":
        break
    if clave in diccionario:
        print("La clave ya existe")
        continue
    valor = input("Ingrese un valor: ")
    diccionario[clave] = valor
print(diccionario)
```

304.3 Ejercicio 3:

- Crear un diccionario con los nombres de los meses como claves y la cantidad de días que tienen como valor.
- Mostrar los meses que tienen 31 días.
- Mostrar los meses que tienen 30 días.
- Mostrar los meses que tienen 28 días.

Posible solución

```
meses = {
    "enero": 31,
    "febrero": 28,
    "marzo": 31,
    "abril": 30,
    "mayo": 31,
    "junio": 30,
    "julio": 31,
    "agosto": 31,
    "septiembre": 30,
    "octubre": 31,
    "noviembre": 30,
    "diciembre": 31
}

meses_31 = [mes for mes, dias in meses.items() if dias == 31]
meses_30 = [mes for mes, dias in meses.items() if dias == 30]
meses_28 = [mes for mes, dias in meses.items() if dias == 28]

print("Meses con 31 días:", meses_31)
print("Meses con 30 días:", meses_30)
print("Meses con 28 días:", meses_28)
```

304.4 Ejercicio 4:

- Crear un diccionario con los nombres de los meses como claves y la cantidad de paises de sur america mostrando mediante el pais la capital.
- Mostrar la capital de Argentina.
- Mostrar la capital de Brasil.
- Mostrar la capital de Ecuador.

Posible solución

```
paises = {
    "Argentina": "Buenos Aires",
    "Bolivia": "La Paz",
    "Brasil": "Brasilia",
```

```
    "Chile": "Santiago",
    "Colombia": "Bogotá",
    "Ecuador": "Quito",
    "Guyana": "Georgetown",
    "Paraguay": "Asunción",
    "Perú": "Lima",
    "Surinam": "Paramaribo",
    "Uruguay": "Montevideo",
    "Venezuela": "Caracas"
}

print("La capital de Argentina es", paises["Argentina"])
print("La capital de Brasil es", paises["Brasil"])
print("La capital de Ecuador es", paises["Ecuador"])
```

304.5 Ejercicio 5:

- Crear un diccinario con los nombres de los presidentes de Ecuador y la fecha en la que asumieron el cargo.
- Mostrar la fecha en la que asumió el presidente Eloy Alfaro.
- Mostrar la fecha en la que asumió el presidente Garcia Moreno.

Posible solución

```
presidentes = {

    "Gustavo Noboa": "23 de noviembre de 2023",
    "Guillermo Lasso": "24 de mayo de 2021",
    "Lenín Moreno": "24 de mayo de 2017",
    "Rafael Correa": "15 de enero de 2007",
    "Jamil Mahuad": "10 de agosto de 1998",
    "Abdalá Bucaram": "10 de agosto de 1996",
    "Sixto Durán Ballén": "10 de agosto de 1992",
    "Rodrigo Borja": "10 de agosto de 1988",
    "León Febres Cordero": "10 de agosto de 1984",
    "Osvaldo Hurtado": "10 de agosto de 1981",
    "Jaime Roldós": "10 de agosto de 1979",
    "Guillermo Rodríguez": "24 de mayo de 1972",
    "José María Velasco Ibarra": "1 de septiembre de 1968",
    "Otto Arosemena": "16 de febrero de 1966",
    "Carlos Julio Arosemena": "1 de septiembre de 1961",
    "Camilo Ponce Enríquez": "1 de septiembre de 1956",
    "José María Velasco Ibarra": "1 de septiembre de 1952",
    "Galalza Castro": "1 de septiembre de 1947",
    "Carlos Arroyo del Río": "1 de septiembre de 1940",
    "Andrés Córdova": "1 de septiembre de 1938",
```

```
    "Alberto Enríquez Gallo": "1 de septiembre de 1937",
    "Federico Páez": "1 de septiembre de 1935",
    "José María Velasco Ibarra": "1 de septiembre de 1934",
    "Abelardo Montalvo": "1 de septiembre de 1933",
    "Neptalí Bonifaz": "1 de septiembre de 1931",
    "Isidro Ayora": "1 de septiembre de 1926",
    "Gonzalo Córdova": "1 de septiembre de 1924",
    "José Luis Tamayo": "1 de septiembre de 1920",
    "Leónidas Plaza": "1 de septiembre de 1912",
    "Emilio Estrada": "1 de septiembre de 1911",
    "Carlos Freile Zaldumbide": "1 de septiembre de 1907",
    "Eloy Alfaro": "1 de septiembre de 1906",
    "Leónidas Plaza": "1 de septiembre de 1901",
    "Eloy Alfaro": "1 de septiembre de 1897",
    "Antonio Flores Jijón": "1 de septiembre de 1888",
    "José Plácido Caamaño": "1 de septiembre de 1883",
    "Pedro José de Arteta": "1 de septiembre de 1882",
    "Francisco Xavier León": "1 de septiembre de 1878",
    "Antonio Borrero": "1 de septiembre de 1875",
    "Gabriel García Moreno": "1 de septiembre de 1861",
    "Francisco Robles": "1 de septiembre de 1856",
    "Diego Noboa": "1 de septiembre de 1850",
    "José Joaquín de Olmedo": "1 de septiembre de 1845",
    "Juan José Flores": "1 de septiembre de 1830",
}

print("Eloy Alfaro asumió el", presidentes["Eloy Alfaro"])
print("García Moreno asumió el", presidentes["Gabriel García Moreno"])
```