Curso de Python Avanzado

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Bienvenida

Bienvenidos al curso de **Python Avanzado** para Scotiabank, brindado por Belatrix.

En este curso aprenderás a desarrollar una programación avanzada con Python.

El curso está dirigido a profesionales relacionados al área de desarrollo, soporte y análisis de datos.

Se requieren conocimientos previos de Python Intermedio para poder cubrir satisfactoriamente este curso.

Temario

Módulo 1: Programación Funcional en Python

1. Introducción a la programación funcional
2. Funciones de orden superior
3. Lambdas y expresiones generadoras

Módulo 2: Diseño de Patrones Avanzados

1. Patrones de diseño comunes en Python
2. Aplicación práctica de patrones en el desarrollo de software

Módulo 3: Manipulación Eficiente de Datos

1. Uso avanzado de NumPy y Pandas
2. Operaciones vectorizadas y broadcasting

Módulo 4: Concurrencia y Paralelismo en Python

1. Hilos y procesos en Python
2. Multiprocessing y Asyncio

Módulo 5: Optimización de Código

1. Estrategias para mejorar el rendimiento
2. Perfilado de código y herramientas de optimización

Módulo 6: Seguridad en Desarrollo Python

1. Principios básicos de seguridad
2. Mejores prácticas de codificación segura

Módulo 7: Despliegue y Escalabilidad

1. Configuración de entornos de producción
2. Estrategias para escalabilidad horizontal y vertical

Introducción a la Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación basado en el

concepto de "objetos", que pueden contener datos, en forma de campos, a menudo conocidos como atributos; y código, en forma de procedimientos, a menudo conocidos como métodos. Python, siendo un lenguaje de programación versátil, soporta plenamente la programación orientada a objetos, ofreciendo una manera poderosa y flexible de organizar tu código. Aquí te presento una introducción a la programación orientada a objetos en Python:

Conceptos Básicos de POO

- **Clases**: Una clase es un plano o prototipo a partir del cual se crean los objetos. Define un conjunto de atributos y métodos que caracterizan a cualquier objeto de esa clase.
- **Objetos**: Un objeto es una instancia de una clase. Un objeto en Python es una encapsulación de variables y funciones en una sola entidad. Los objetos obtienen sus variables y funciones de las clases.
- **Atributos**: Son las características de la clase. Se utilizan para almacenar información. Los atributos son variables definidas dentro de una clase.
- **Métodos**: Son funciones definidas dentro de una clase y se utilizan para describir los comportamientos de los objetos.

Creando una Clase en Python

Para definir una clase en Python, se utiliza la palabra clave `class` seguida del nombre de la clase y dos puntos. A continuación, se definen los métodos y atributos de la clase:

```
class MiClase:
    # Constructor de la clase
    def __init__(self, atributo1, atributo2):
        self.atributo1 = atributo1
        self.atributo2 = atributo2

    # Método de la clase
    def mi_metodo(self):
        print(f"Atributo 1 es {self.atributo1} y Atributo 2 es {self.atributo2}")
```

Instanciando Objetos

Para crear un objeto (instancia de una clase), simplemente llame a la clase usando el nombre de la clase seguido de paréntesis, pasando los argumentos que el método constructor `__init__` espera:

```
mi_objeto = MiClase("valor1", "valor2")
mi_objeto.mi_metodo()  # Salida: Atributo 1 es valor1 y Atributo 2 es valor2
```

Pilares de la POO en Python

La programación orientada a objetos en Python se basa en cuatro principios fundamentales:

1. **Encapsulación**: Se refiere a la agrupación de datos (atributos) y código (métodos) que opera sobre los datos en una sola unidad o clase, y restringir el acceso a algunos de los componentes de un objeto.
2. **Abstracción**: Permite ocultar la complejidad real de un sistema y mostrar solo lo necesario al exterior. Se logra mediante clases abstractas e interfaces.
3. **Herencia**: Permite a una clase heredar atributos y métodos de otra, facilitando la reutilización de código y la creación de relaciones jerárquicas entre clases.
4. **Polimorfismo**: Significa la capacidad de una entidad (método o objeto) de referirse a instancias de diferentes clases. Python permite el polimorfismo, permitiendo que un mismo método funcione de manera diferente en función del objeto que lo invoca.

Conclusión

La programación orientada a objetos en Python es un paradigma poderoso que facilita la escritura de código organizado, reutilizable y fácil de mantener. Al dominar la POO, podrás diseñar soluciones más eficientes y efectivas para tus problemas de programación.

```
class Producto:
  def __init__(self, nombre, precio):
    #print(f"Voy a construir un objeto a partir de nombre={nombre} precio=${preci
    self.nombre = nombre
    self.precio = precio
    self.iva = precio * 0.16
    self.precio_con_iva = precio + self.iva

  def describir(self):
    print("{:<20} ${:>6.2f} (IVA ${:>6.2f}) | ${:>6.2f}".format(self.nombre, self

producto1 = Producto("Coca-Cola", 30.5) # producto1 -> objeto | Producto -> clase

print(producto1)
print(producto1.nombre, type(producto1.nombre))
print(producto1.precio, type(producto1.precio))
print(producto1.iva, type(producto1.iva))
print(producto1.precio_con_iva, type(producto1.precio_con_iva))

producto1.describir()
```

```
    <__main__.Producto object at 0x797e99c499f0>
    Coca-Cola <class 'str'>
    30.5 <class 'float'>
    4.88 <class 'float'>
    35.38 <class 'float'>
    Coca-Cola            $ 30.50 (IVA $  4.88) | $ 35.38
```

```
import random

for i in range(10):
  producto_i = Producto(f"Producto {i + 1}", random.uniform(1, 100))
  producto_i.describir()
```

```
    Producto 1           $ 62.35 (IVA $  9.98) | $ 72.33
    Producto 2           $ 33.77 (IVA $  5.40) | $ 39.17
    Producto 3           $ 74.87 (IVA $ 11.98) | $ 86.85
    Producto 4           $ 39.92 (IVA $  6.39) | $ 46.30
    Producto 5           $ 34.22 (IVA $  5.48) | $ 39.69
    Producto 6           $ 68.71 (IVA $ 10.99) | $ 79.71
    Producto 7           $ 69.61 (IVA $ 11.14) | $ 80.74
    Producto 8           $ 83.05 (IVA $ 13.29) | $ 96.33
    Producto 9           $ 45.32 (IVA $  7.25) | $ 52.57
    Producto 10          $ 80.41 (IVA $ 12.87) | $ 93.27
```

```
class Carrito:
  def __init__(self):
    self.productos = []
```

```
  def agregarProducto(self, producto):
    self.productos.append(producto)

  def calcularTotal(self):
    from functools import reduce
    return reduce(lambda total, producto: total + producto.precio, self.productos

  def calcularTotalIva(self):
    from functools import reduce
    return reduce(lambda total, producto: total + producto.iva, self.productos, 0

  def calcularTotalConIva(self):
    from functools import reduce
    return reduce(lambda total, producto: total + producto.precio_con_iva, self.p

  def describir(self):
    print("Carrito de compras | Total de productos ({})".format(len(self.producto
    print("-" * 54)
    if len(self.productos) > 0:
      for producto in self.productos:
        producto.describir()
      print("-" * 54)
      total = self.calcularTotal()
      total_iva = self.calcularTotalIva()
      total_con_iva = self.calcularTotalConIva()
      print("Total {} ${:>6.2f} (IVA ${:>6.2f}) | ${:6.2f}".format(" " * 14, tota
    else:
      print("  El carrito está vacío 🥲")
    print("-" * 54)
    print()

carrito1 = Carrito() # self.productos = []

carrito1.describir()

carrito1.agregarProducto(Producto("Coca-Cola 600ml", 17.5))
carrito1.agregarProducto(Producto("Pesi 600ml", 16.5))
carrito1.agregarProducto(Producto("Gansito", 21.5))

carrito1.describir()
```

```
Carrito de compras | Total de productos (0)
------------------------------------------------------
  El carrito está vacío 🥲
------------------------------------------------------

Carrito de compras | Total de productos (3)
------------------------------------------------------
Coca-Cola 600ml      $ 17.50 (IVA $  2.80) | $ 20.30
Pesi 600ml           $ 16.50 (IVA $  2.64) | $ 19.14
Gansito              $ 21.50 (IVA $  3.44) | $ 24.94
------------------------------------------------------
Total                $ 55.50 (IVA $  8.88) | $ 64.38
------------------------------------------------------
```

Módulo 2: Diseño de Patrones Avanzados

1. Patrones de diseño comunes en Python
2. Aplicación práctica de patrones en el desarrollo de software

201. Patrones de diseño comunes en Python

Los patrones de diseño son soluciones generales y reutilizables para los problemas comunes que nos encontramos en el diseño de software. No son plantillas que se puedan traducir directamente en código, sino más bien directrices para abordar ciertos problemas en contextos específicos. En Python, al igual que en otros lenguajes de programación, los patrones de diseño pueden clasificarse en tres tipos principales: creacionales, estructurales y de comportamiento.

Patrones Creacionales

Se enfocan en cómo se crean las instancias de objetos. Simplifican la creación de objetos cuando existen complejidades involucradas.

1. **Singleton**: Asegura que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. En Python, se puede implementar mediante una clase base o decoradores. Es útil manejar recursos compartidos, como una conexión a una base de datos.
2. **Factory Method**: Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Facilita la extensión del código existente y es útil en entornos

colaborativos para definir y expandir tipos de objetos procesados.

3. **Builder**: Separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones. Esto es particularmente útil para crear configuraciones complejas de experimentos o visualizaciones de datos de forma ordenada.

Patrones Estructurales

Tratan sobre cómo se componen las clases y objetos para formar estructuras más grandes.

1. **Adapter (Adaptador)**: Permite que interfaces incompatibles trabajen juntas. Esto es útil cuando se utilizan bibliotecas o APIs que tienen interfaces distintas de las que el código existente espera.
2. **Decorator**: Adjunta responsabilidades adicionales a un objeto de manera dinámica. Los decoradores ofrecen una alternativa flexible a la subclase para extender la funcionalidad, muy útil en Python para añadir funcionalidades en tiempo de ejecución, como logging o manejo de errores en funciones específicas.
3. **Proxy**: Proporciona un sustituto o marcador de posición para otro objeto para controlar el acceso a él. Útil para controlar el acceso a recursos que pueden ser costosos de crear.

Patrones de Comportamiento

Se centran en la comunicación efectiva y la asignación de responsabilidades entre objetos.

1. **Observer**: Define una dependencia de uno-a-muchos entre objetos de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Muy útil para implementar dashboards dinámicos o visualizaciones que se actualizan en respuesta a cambios de datos.
2. **Strategy**: Define una familia de algoritmos, encapsula cada uno de ellos, y los hace intercambiables. La estrategia permite que el algoritmo varíe independientemente de los clientes que lo utilizan. Esto es beneficioso para alternar entre diferentes algoritmos de procesamiento de datos o métodos de análisis sin cambiar el código cliente.
3. **Command**: Encapsula una solicitud como un objeto, permitiendo parametrizar clientes con colas, solicitudes y operaciones. Puede ser utilizado para implementar operaciones deshacer/rehacer o para programar tareas que pueden ser ejecutadas en diferentes momentos.

Estos patrones no solo ayudan a escribir código más limpio y mantenible sino que también

facilitan la colaboración, donde múltiples usuarios pueden estar trabajando en el mismo proyecto. Implementar estos patrones correctamente puede llevar a una mejor organización del código, mayor reutilización y una integración más fácil de nuevas características o bibliotecas.

```
# Ejemplo del Singletón

class DataSource:
  def __init__(self, token):
    if token != "123":
      raise Exception("Esta clase requiere un token válido para ser instancia")
    self.data = []

  def addPoint(self, point):
    self.data.append(point)

  def describe(self):
    print(f"Points ({len(self.data)})")
    for point in self.data:
      print(point)

class DataSourceSingleton:
  shared = DataSource("123")

DataSourceSingleton.shared.addPoint({ "x": 123, "y": 456 })
DataSourceSingleton.shared.addPoint({ "x": 344, "y": 657 })
DataSourceSingleton.shared.addPoint({ "x": 563, "y": 111 })
DataSourceSingleton.shared.addPoint({ "x": 243, "y": 223 })
DataSourceSingleton.shared.addPoint({ "x": 987, "y": 143 })
DataSourceSingleton.shared.addPoint({ "x": 143, "y": 998 })

DataSourceSingleton.shared.describe()
```

```
Points (6)
{'x': 123, 'y': 456}
{'x': 344, 'y': 657}
{'x': 563, 'y': 111}
{'x': 243, 'y': 223}
{'x': 987, 'y': 143}
{'x': 143, 'y': 998}
```

```
ds = DataSource("???")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-50-0659533c04d3> in <cell line: 1>()
----> 1 ds = DataSource("???")

<ipython-input-49-eb3976317f34> in __init__(self, token)
      4   def __init__(self, token):
      5     if token != "123":
----> 6       raise Exception("Esta clase requiere un token válido para ser
instancia")
      7     self.data = []
      8

Exception: Esta clase requiere un token válido para ser instancia
```

```
# Ejemplo de Factory Method (Métodos de Fábrica)

class SaludarEspañol:

  def __init__(self, nombre):
    self.nombre = nombre

  def saludar(self):
    print(f"Hola {self.nombre} 😊")

class SaludarIngles:

  def __init__(self, fecha, veces):
    self.fecha = fecha
    self.veces = veces

  def saludar(self):
    for i in range(self.veces):
      print(f"[{self.fecha}] Hello world 👋")

class SaludarRobot:

  def __init__(self, marca, modelo):
    self.marca = marca
    self.modelo = modelo

  def saludar(self):
    print(f"[{self.marca} / {self.modelo}] Pipu pipu 🤖")

class Saludador:
```

```
  def crearSaludoEspañol(self, nombre):
    self.instance = SaludarEspañol(nombre)
    return self

  def crearSaludoIngles(self, fecha, veces):
    self.instance = SaludarIngles(fecha, veces)
    return self

  def crearSaludoRobot(self, marca, modelo):
    self.instance = SaludarRobot(marca, modelo)
    return self

saludador = Saludador().crearSaludoEspañol("Beto")

saludador.instance.saludar()
```

```
    Hola Beto 😊
```

```
saludador = Saludador().crearSaludoIngles("9th, april 2024", 4)

saludador.instance.saludar()
```

```
    [9th, april 2024] Hello world 👋
    [9th, april 2024] Hello world 👋
    [9th, april 2024] Hello world 👋
    [9th, april 2024] Hello world 👋
```

```
saludador = Saludador().crearSaludoRobot("Tesla", "Tux")

saludador.instance.saludar()
```

```
    [Tesla / Tux] Pipu pipu 🤖
```

```
# Ejemplo de Builder (Constructor por partes)

class TerminalVenta:

  def __init__(self, id):
    self.id = id
    self.iniciada = False

  def setVendedor(self, vendedor):
    if not vendedor.activo:
      raise Exception("El vendedor no está activo")
    self.vendedor = vendedor

  def iniciar(self):
    import random
    self.sesionId = random.randint(10_000, 1_000_000)
    self.iniciada = True

  def cobrar(self):
    if not self.iniciada:
      raise Exception("La terminal no está iniciada")

    print(f"La terminal {self.id} ha iniciado el proceso de cobro con el vendedor

class Vendedor:

  def __init__(self, id, nombre, activo):
    self.id = id
    self.nombre = nombre
    self.activo = activo

# terminal1 = TerminalVenta(123)
# terminal1.cobrar() # ERROR: La terminal no está iniciada
```

```
vendedores = [
    Vendedor(1001, "Juan Pérez", True),
    Vendedor(1002, "John Poe", False),
    Vendedor(1003, "Jorge Jiménez", True),
    Vendedor(1004, "Paty Rampírez", True),
]

def crearTerminalVenta(terminalVentaId, vendedorId):
  # 1. Crea la terminal con el terminalVentaId
  terminal = TerminalVenta(terminalVentaId)

  # 2. Ajusta el vendedor
  # Buscar el vendedor
  from functools import reduce
  vendedor = reduce(lambda encontrado, vendedor: vendedor if vendedor.id == vende

  if vendedor == None:
    raise Exception(f"El vendedor con id {vendedorId} no existe")

  terminal.setVendedor(vendedor)

  # 3. Iniciamos la terminal
  terminal.iniciar()

  return terminal


terminal = crearTerminalVenta(34, 1004)

terminal.cobrar()
```

```
# Ejemplo de Observador

class ObservadorCarrito:

  def __init__(self):
    self.listeners = [] # son funciones que reciben los eventos

  def addListener(self, listener):
    self.listeners.append(listener)

  def onProductosChange(self, producto, productos): # Renotifica a los listeners
    for listener in self.listeners:
      # El evento envia un diccionario con los datos del evento
      listener({ "event": "productoChange", "data": (producto, productos) })

class Carrito(ObservadorCarrito): # --> HERENCIA

  def __init__(self):
    ObservadorCarrito.__init__(self) # LLAMADA AL CONSTRUCTOR SUPERIOR
    self.productos = []

  def agregarProducto(self, producto):
    self.productos.append(producto)
    self.onProductosChange(producto, self.productos) # desencadena el evento ("no

  def quitarProducto(self, producto):
    producto = self.productos.remove(producto)
    self.onProductosChange(producto, self.productos) # desencadena el evento ("no


carrito1 = Carrito()

total = 0

def updateTotal(evento):
  global total
  if evento["event"] == "productoChange":
    producto, productos = evento["data"]
    from functools import reduce
    total = reduce(lambda total, producto: total + producto.precio, productos, 0.
  print(f"El total ha sido actualizado: {total}")

def tieneCocaCola(evento):
  if evento["event"] == "productoChange":
    producto, productos = evento["data"]

    encontrada = False
```

```
    for producto in productos:
      if producto.nombre == "Coca-Cola":
        encontrada = True

    if encontrada:
      print("Hay una Coca-Cola dentro de los productos")
    else:
      print("No hay una Coca-Cola dentro de los productos")

carrito1.addListener(updateTotal)
carrito1.addListener(tieneCocaCola)

cocaCola = Producto("Coca-Cola", 100)

carrito1.agregarProducto(cocaCola)
carrito1.agregarProducto(Producto("Pepsi", 200))
carrito1.agregarProducto(Producto("Galletar Marías", 500))
carrito1.quitarProducto(cocaCola)

print(total)
```

```
El total ha sido actualizado: 100.0
Hay una Coca-Cola dentro de los productos
El total ha sido actualizado: 300.0
Hay una Coca-Cola dentro de los productos
El total ha sido actualizado: 800.0
Hay una Coca-Cola dentro de los productos
El total ha sido actualizado: 700.0
No hay una Coca-Cola dentro de los productos
700.0
```

202. Aplicación práctica de patrones en el desarrollo de software

```
# Sistema de Venta de Frutas

# 1. Definir las clase Fruta y CanastaFrutas

class Fruta:
  def __init__(self, id, nombre, precio):
    self.id = id
    self.nombre = nombre
    self.precio = precio

  def describir(self):
    ...

class CanastaFrutas:

  def __init__(self):
    self.frutas = []

  def agregarFruta(self, id, nombre, precio):
    self.frutas.append(Fruta(id, nombre, precio))

  def quitarFruta(self, id):
    for index, fruta in enumerate(self.frutas):
      if fruta.id == id:
        self.frutas.pop(index)
        return
    raise Exception(f"La fruta con id {id} no existe")

# > Realiza unas pruebas y calcula los totales

# 2. Definir el ObservadorCansataFrutas

# > Observa cuando se agrega una fruta
# > Observa cuando se quita una fruta

# 3. Redefinir la CanastaFrutas y adaptar el ObservadorCansataFrutas
```