Curso de Python Avanzado

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

+ Código + Texto

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

Repaso General de Python

```
# Variables de tipos primitivos

nombre = "Daniela" # str
apellidos = 'González Martínez' # str
edad = 23 # int
peso = 54.6 # float
frecuencia = 43.5 + 2.3j # complex
casado = True # bool
trabajo = False # bool

print(f"NOMBRE: {nombre} {apellidos}")
print(f"EDAD: {edad} años | PESO: {peso} kg")
print(f"CASADO: {casado} | TRABAJO: {trabajo}")
```

```
NOMBRE: Daniela González Martínez
EDAD: 23 años | PESO: 54.6 kg
CASADO: True | TRABAJO: False
```

```
print("NOMBRE: {} {}".format(nombre, apellidos))
print("EDAD: {:4} años | PESO: {:>10} kg".format(edad, peso))
bool_to_text = lambda p: "SI" if p else "NO"
print("CASADO: {:>7} | TRABAJO: {:>10}".format(bool_to_text(casado), bool_to_text(tr
```

```
NOMBRE: Daniela González Martínez
EDAD:   23 años | PESO:       54.6 kg
CASADO:      SI | TRABAJO:         NO
```

```
# Colecciones: listas, tuplas, diccionarios

# Listas -> Conjunto secuencial, indexado de valores dinámicos

frutas = ["pera", "manzana", "piña"]

print(frutas)
print(frutas[1]) # 0 - pera, 1 - manzana, 2 - piña

frutas.append("plátano")

print(frutas)

frutas.insert(0, "melón")

print(frutas)

frutas.pop(2) # 2 - manzana

print(frutas)

print(frutas[2:4]) # ["piña", "plátano"]

print(frutas[0::2]) # ["melón", "piña"]

# Tuplas -> Conjunto secuencial, indexado de valores fijos

nombre = "Daniel"
salario = 10_500

datos_daniel = (nombre, salario) # <2-tupla> (<valor 1>, <valor 2>)

print(datos_daniel)

n, s = datos_daniel # Desacomplar la <2-tupla> en 2-variables

print(n) # "Daniel"
print(s) # "salario"

t = (8, "Daniel", 23, True, False) # Es complejo determinar a qué se refiere cada po

id, nombre, edad, casado, trabajo = t # Es complejo recordar cada variable acoplada

# Diccionarios -> Conjunto no secuencial, indexado de valores dinámicos

# Equivale a un "FRAME" de información
d = {
    "id": 8,
    "nombre": "Daniel",
    "edad": 23,
    "casado": True,
```

```
    "trabajo": False
}

print(d)

print(d["nombre"]) # "Daniel"
print(d["edad"]) # 23
print(d["trabajo"]) # False
```

```
['pera', 'manzana', 'piña']
manzana
['pera', 'manzana', 'piña', 'plátano']
['melón', 'pera', 'manzana', 'piña', 'plátano']
['melón', 'pera', 'piña', 'plátano']
['piña', 'plátano']
['melón', 'piña']
('Daniel', 10500)
Daniel
10500
{'id': 8, 'nombre': 'Daniel', 'edad': 23, 'casado': True, 'trabajo': False}
Daniel
23
False
```

Módulo 1: Programación Funcional en Python

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

101. Introducción a la programación funcional

La programación funcional es un paradigma de programación que trata a la computación como la evaluación de funciones matemáticas y evita cambiar el estado y los datos mutables. Es distinta de la programación imperativa, que enfoca la computación en términos de instrucciones que cambian el estado del programa. La programación funcional ofrece varios beneficios, como la facilidad de pruebas y depuración, debido a que las funciones puras (un concepto clave en este paradigma) siempre producen el mismo resultado para los mismos argumentos y no tienen efectos secundarios (como modificar variables globales o el estado del sistema).

Aquí hay algunos conceptos clave en la programación funcional:

1. **Funciones Puras**

- Son funciones donde el resultado solo depende de los argumentos proporcionados y no producen efectos secundarios (como modificar variables externas). Esto facilita entender y predecir el comportamiento del código.

2. **Inmutabilidad**

- Los datos nunca cambian después de su creación. Si quieres modificar un objeto, debes crear uno nuevo con los cambios deseados. Esto previene efectos secundarios no deseados y hace que el código sea más seguro y fácil de razonar.

3. **Funciones de Orden Superior**

- Son funciones que toman otras funciones como argumentos o las devuelven como resultado. Esto permite técnicas como la composición de funciones, donde se combinan varias funciones pequeñas para crear una nueva función.

4. **Expresiones Lambda**

- Son una forma de crear funciones anónimas en el momento, lo que es útil para operaciones cortas que no requieren la definición de una función nombrada completa.

5. **Evaluación Perezosa**

- Significa que la evaluación de una expresión se retrasa hasta que su resultado sea realmente necesario. Esto puede mejorar el rendimiento al evitar cálculos innecesarios y permite estructuras de datos infinitas, como listas infinitas.

6. **Recursividad**

- Dado que la inmutabilidad prohíbe las variables que cambian, la recursividad se utiliza a menudo en lugar de ciclos tradicionales como bucles for o while. La recursividad implica que una función se llame a sí misma dentro de su definición.

7. **Patrones Comunes**

- Mapa, filtro y reducción son patrones de diseño comunes en la programación funcional que operan en listas o secuencias de datos. Permiten la transformación, selección y agregación de datos de manera expresiva y concisa.

Ejemplo Básico en Python

Python no es un lenguaje de programación puramente funcional, pero admite muchos conceptos de programación funcional. Aquí hay un ejemplo simple que muestra el uso de funciones puras y la inmutabilidad:

```
# Función pura
def sumar(a, b):
    return a + b

# Uso de funciones de orden superior y expresiones lambda
numeros = [1, 2, 3, 4, 5]
numeros_cuadrados = list(map(lambda x: x**2, numeros))

# La lista original permanece inalterada, demostrando inmutabilidad
print(numeros)  # [1, 2, 3, 4, 5]
print(numeros_cuadrados)  # [1, 4, 9, 16, 25]
```

Este ejemplo ilustra cómo la programación funcional puede ser utilizada en Python para escribir código conciso y mantenible.

```
# Ejemplos 101

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

suma(5, 6)

    11


suma(1000, 2000)

    3000


suma("Hola", "mundo")

    'Holamundo'


def concatenar(*valores): # Todos los parámetros los acumula en una lista
  buffer = ""
  for valor in valores:
    buffer += valor
  return buffer

concatenar("Hola", " ", "mundo", " ", "😊")
```

```
'Hola mundo 😊 '
```

```
def calcular_tamaño(textos): # Definición de la función
  # textos = ["manzana", "pera", ...]

  # Necesitamos una lista nueva originalmente vacía
  # para guardar los tamaños de cada uno de los textos
  tamaños = []

  # Si textos es una lista de valores de texto,
  # entonces, podemos iterar cada uno de esos valores
  # y cada uno representará un texto
  for texto in textos:
    # Inspeccionamos cada texto y su tamaño en cada iteración
    #print(texto, len(texto))
    # Necesitamos almacenar el tamaño del texto a una lista (agregación)
    tamaños.append(len(texto))

  #print(tamaños)

  return tamaños

frutas = ["manzana", "pera", "mango", "limón", "guanabana", "kiwi"]

calcular_tamaño(frutas) # Llamada a la función
```

```
[7, 4, 5, 5, 9, 4]
```

```
frutas = ["manzana", "pera", "mango", "limón", "guanabana", "kiwi"]
tamaños = calcular_tamaño(frutas)

print(frutas)
print(tamaños)
```

```
['manzana', 'pera', 'mango', 'limón', 'guanabana', 'kiwi']
[7, 4, 5, 5, 9, 4]
```

```
list(zip(frutas, tamaños)) # relación 1-1 entre frutas y tamaños
```

```
[('manzana', 7),
 ('pera', 4),
 ('mango', 5),
 ('limón', 5),
 ('guanabana', 9),
 ('kiwi', 4)]
```

```
dict(zip(frutas, tamaños))
```

```
{'manzana': 7, 'pera': 4, 'mango': 5, 'limón': 5, 'guanabana': 9, 'kiwi': 4}
```

```
# Adaptador no genérico
def calcular_tamaño_con_info(textos): # Definición de la función
  tamaños = []

  for texto in textos:
    tamaños.append({
        "texto": texto,
        "tamaño": len(texto),
        "inicial": texto[0],
        "terminal": texto[-1],
    })

  return tamaños

frutas = ["manzana", "pera", "mango", "limón", "guanabana", "kiwi"]

calcular_tamaño_con_info(frutas) # Llamada a la función
```

```
[{'texto': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'},
 {'texto': 'pera', 'tamaño': 4, 'inicial': 'p', 'terminal': 'a'},
 {'texto': 'mango', 'tamaño': 5, 'inicial': 'm', 'terminal': 'o'},
 {'texto': 'limón', 'tamaño': 5, 'inicial': 'l', 'terminal': 'n'},
 {'texto': 'guanabana', 'tamaño': 9, 'inicial': 'g', 'terminal': 'a'},
 {'texto': 'kiwi', 'tamaño': 4, 'inicial': 'k', 'terminal': 'i'}]
```

102. Funciones de orden superior

Las funciones de orden superior son un concepto fundamental en la programación funcional, que permite una programación más expresiva y flexible. En Python, así como en otros lenguajes que soportan paradigmas de programación funcional, estas funciones son herramientas poderosas que facilitan la escritura de código conciso, legible y eficiente.

Definición

Una función de orden superior es una función que cumple al menos uno de los siguientes criterios:

- Toma una o más funciones como argumentos.
- Devuelve otra función como su resultado.

Este concepto permite abstracciones funcionales y operaciones que pueden reducir significativamente la redundancia en el código y mejorar su modularidad.

Ejemplos de Funciones de Orden Superior en Python

Python ofrece varias funciones incorporadas que son ejemplos de funciones de orden superior. Algunas de las más utilizadas incluyen `map()`, `filter()`,y `reduce()`.

`map(function, iterable, ...)`

- **Propósito**: Aplica una función dada a cada elemento de un iterable (lista, tupla, etc.) y devuelve un iterador que produce los resultados.
- **Ejemplo**:

```
def cuadrado(numero):
    return numero ** 2

numeros = [1, 2, 3, 4, 5]
resultados = map(cuadrado, numeros)
print(list(resultados))  # [1, 4, 9, 16, 25]
```

`filter(function, iterable)`

- **Propósito**: Construye un iterador a partir de aquellos elementos de un iterable para los cuales la función devuelve verdadero.
- **Ejemplo**:

```
def es_par(numero):
    return numero % 2 == 0

numeros = [1, 2, 3, 4, 5, 6]
resultado = filter(es_par, numeros)
print(list(resultado))  # [2, 4, 6]
```

`reduce(function, iterable[, initializer])`

- **Propósito**: Aplica una función de dos argumentos acumulativamente a los elementos de un iterable, de izquierda a derecha, para reducir el iterable a un solo valor. `reduce()` no está disponible globalmente en Python 3, se debe importar desde `functools`.
- **Ejemplo**:

```
from functools import reduce

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

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

```
resultado = reduce(sumar, numeros)
print(resultado)  # 15
```

Ventajas de las Funciones de Orden Superior

- **Composición**: Facilitan la composición de funciones, permitiendo crear nuevas operaciones a partir de las existentes de manera clara y concisa.
- **Reutilización del Código**: Promueven la reutilización del código al permitir que las funciones sean utilizadas como argumentos.
- **Abstracción y Expresividad**: Permiten abstracciones más altas y código más expresivo, especialmente en operaciones que involucran colecciones de datos.

Las funciones de orden superior son un pilar de la programación funcional, pero su utilidad no se limita a este paradigma. Son herramientas versátiles que pueden mejorar significativamente la claridad, concisión y expresividad del código en muchos contextos de programación. ¿Te gustaría explorar más ejemplos o conceptos relacionados con la programación funcional?

```
# Ejemplos 102

# Función de adaptación (adaptador o función compuesto)
def T(x): # Define cómo transformar un único elemento
  return x ** 2

X = [5, 7, 9, 20] # x pertenece a X

Y = list(map(T, X)) # Y = T(x) para cada x en X

print(X)
print(Y)

     [5, 7, 9, 20]
     [25, 49, 81, 400]


# Adaptador genérico
def text_len(text):
  return len(text)

frutas = ["mango", "fresa", "papaya"]

frutas_tamaño = list(map(text_len, frutas))

print(frutas)
print(frutas_tamaño)

     ['mango', 'fresa', 'papaya']
     [5, 5, 6]
```

```
def F(x):
  return x >= 28 and x <= 48

X = [23, 26, 56, 43, 29, 55]

Xp = list(filter(F, X))

print(X)
print(Xp)

     [23, 26, 56, 43, 29, 55]
     [43, 29]


def R(previo, siguiente):
  return previo + siguiente

X = [67.8, 43.9, 72.5, 23.4] # 0 | 0 + 67.8, anterior + nuevo

from functools import reduce

X_suma = reduce(R, X, 0)

print(X)
print(X_suma)

     [67.8, 43.9, 72.5, 23.4]
     207.6
```

103. Lambdas y expresiones generadoras

Las expresiones lambda y las expresiones generadoras son características poderosas de Python que permiten escribir código más conciso y eficiente, especialmente dentro del paradigma de la programación funcional.

Lambdas

Las lambdas en Python son pequeñas funciones anónimas definidas mediante la palabra clave `lambda`. La sintaxis general de una lambda es:

```
lambda argumentos: expresión
```

Las lambdas pueden tener cualquier número de argumentos, pero solo una expresión. La expresión es evaluada y devuelta cuando se llama a la lambda. Estas funciones son útiles cuando se necesita una función por un corto período de tiempo, y no se desea definirla con el estándar `def`.

Ejemplo de Uso de Lambda

```
# Definir una función lambda para calcular el cuadrado de un número
cuadrado = lambda x: x ** 2

print(cuadrado(5))  # Salida: 25

# Usar lambda directamente como argumento de una función de orden superior
numeros = [1, 2, 3, 4]
numeros_cuadrados = list(map(lambda x: x**2, numeros))
print(numeros_cuadrados)  # Salida: [1, 4, 9, 16]
```

Expresiones Generadoras

Las expresiones generadoras proporcionan una manera compacta de generar secuencias de datos sin necesidad de construir una lista en memoria, lo que puede ser muy útil para secuencias grandes. Son similares a las comprensiones de listas pero usan paréntesis en lugar de corchetes, y el resultado es un generador del cual se pueden extraer elementos uno por uno.

Ejemplo de Uso de Expresión Generadora

```
# Crear una expresión generadora para calcular los cuadrados de números del 1 al 5
cuadrados = (x**2 for x in range(1, 6))

# Acceder a los elementos del generador
for cuadrado in cuadrados:
    print(cuadrado)
# Salida: 1, 4, 9, 16, 25
```

Las expresiones generadoras son especialmente útiles para operaciones que implican la transformación de datos o el filtrado de grandes conjuntos de datos, ya que no requieren que todos los elementos estén en memoria a la vez.

Combinando Lambdas y Expresiones Generadoras

Lambdas y expresiones generadoras pueden ser combinadas para crear operaciones poderosas y eficientes en una sola línea de código. Por ejemplo:

```
# Filtrar y calcular el cuadrado de números pares del 1 al 10
numeros_cuadrados = (x**2 for x in range(1, 11) if x % 2 == 0)

for num in numeros_cuadrados:
```

```
    print(num)
 # Salida: 4, 16, 36, 64, 100
```

Este enfoque permite construir pipelines de procesamiento de datos de manera eficiente y concisa, aprovechando lo mejor de ambos mundos: la flexibilidad de las lambdas y la eficiencia de las expresiones generadoras. ¿Hay algo más sobre estos temas que te gustaría explorar o algún otro concepto de Python avanzado?

```
# Ejemplos 103

frutas = ["manzana", "mango", "kiwi", "limón", "pera"]

frutas_tamaños = list(map(lambda fruta: len(fruta), frutas))

print(frutas)
print(frutas_tamaños)
```

```
    ['manzana', 'mango', 'kiwi', 'limón', 'pera']
    [7, 5, 4, 5, 4]
```

```
frutas = ["manzana", "mango", "kiwi", "limón", "pera"]

frutas_tamaños = list(map(lambda fruta: {
    "nombre": fruta,
    "tamaño": len(fruta),
    "inicial": fruta[0],
    "terminal": fruta[-1],
}, frutas))

print(frutas)
print(frutas_tamaños)
```

```
    ['manzana', 'mango', 'kiwi', 'limón', 'pera']
    [{'nombre': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'}, {'nombre':
```

```
print(frutas_tamaños)
```

```
    [{'nombre': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'}, {'nombre':
```

```
frutas_inicial_m = list(filter(lambda d: d["inicial"] == "m", frutas_tamaños))

frutas_inicial_m
```

```
    [{'nombre': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'},
     {'nombre': 'mango', 'tamaño': 5, 'inicial': 'm', 'terminal': 'o'}]
```

```
frutas_terminal_a = list(filter(lambda d: d["terminal"] == "a", frutas_tamaños))

frutas_terminal_a
```

```
    [{'nombre': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'},
     {'nombre': 'pera', 'tamaño': 4, 'inicial': 'p', 'terminal': 'a'}]
```

```
print(frutas_tamaños)
```

```
    [{'nombre': 'manzana', 'tamaño': 7, 'inicial': 'm', 'terminal': 'a'}, {'nombre':
```

```
from functools import reduce

def reduce_iniciales(iniciales, frutaInfo):
  iniciales.append(frutaInfo["inicial"] * frutaInfo["tamaño"])
  return iniciales

reduce(reduce_iniciales, frutas_tamaños, [])
```

```
    ['mmmmmmm', 'mmmmm', 'kkkk', 'llllll', 'pppp']
```

```
frutas = ["mango", "plátano", "piña", "fresa"]

# (<result> <generador>)
# (<result> for <elemento> in <secuencia>)
# Secuencia en frío (no computada hasta su consumo)
generador_frutas = (fruta for fruta in frutas)

for fruta in generador_frutas:
  print(fruta)
```

```
    mango
    plátano
    piña
    fresa
```

```
# Esta función genera un cliente y lo retiene hasta que se lo piden
# Cuando piden el siguiente valor en un iterador, entonces genera uno nuevo
def generador_clientes(n):
  import random

  nombres = ["Ana", "Beto", "Carlos", "Daniela", "Edwin", "Fabiola"]
  apellidos = ["Juárez", "González", "Martínez", "Bustamante", "Hernández", "Ávila"]

  for i in range(n):
    cliente = {
      "nombre": random.choice(nombres),
      "apellido_paterno": random.choice(apellidos),
      "apellido_materno": random.choice(apellidos),
      "edad": random.randint(18, 99),
    }

    yield cliente

for cliente in generador_clientes(5):
  print(cliente)
```

```
{'nombre': 'Daniela', 'apellido_paterno': 'Hernández', 'apellido_materno': 'Mart
{'nombre': 'Beto', 'apellido_paterno': 'Hernández', 'apellido_materno': 'Martíne
{'nombre': 'Edwin', 'apellido_paterno': 'Juárez', 'apellido_materno': 'Ávila', '
{'nombre': 'Edwin', 'apellido_paterno': 'Juárez', 'apellido_materno': 'González'
{'nombre': 'Ana', 'apellido_paterno': 'Bustamante', 'apellido_materno': 'Martíne
```

Ejercicios

```
from functools import reduce


# [1, 2, 3] -> "# ## ###"
# [4, 5, 2] -> "#### ##### ##"

def R(anterior, siguiente):
  print(f"Anterior: {anterior} | Siguiente: {siguiente}")
  nuevo = str(anterior + " " + "#" * siguiente).strip()
  print(f"Nuevo: {nuevo}")
  return nuevo

reduce(R, [3, 5, 2], "🐍")
```

```
    Anterior: 🐍  | Siguiente: 3
    Nuevo: 🐍 ###
    Anterior: 🐍 ### | Siguiente: 5
    Nuevo: 🐍 ### #####
    Anterior: 🐍 ### ##### | Siguiente: 2
    Nuevo: 🐍 ### ##### ##
    '🐍 ### ##### ##'
```

```
# [23, 56, 43, 22, 19] -> 56
# [98, 77, 34, 101, 22, 18] -> 101

def S(mayor, actual):
  if mayor == None:
    return actual
  if mayor > actual:
    return mayor
  else:
    return actual

reduce(S, [98, 77, 34, 101, 22, 18], None)
```

```
    101
```

```
mayor = None

for x in [98, 77, 34, 101, 22, 18]:
  if mayor == None or mayor < x:
    mayor = x

mayor
```

```
    101
```

```
# reduce(R<w, x>, X, w_0)
reduce(lambda mayor, x: x if mayor == None or mayor < x else mayor, [98, 77, 34, 101
```

```
    101
```

```
# map(T<x>, X)
list(map(lambda x: x - 50, [98, 77, 34, 101, 22, 18]))
```

```
    [48, 27, -16, 51, -28, -32]
```

```
# filter(F<x>, X)
list(filter(lambda x: x > 50, [98, 77, 34, 101, 22, 18]))
```

```
    [98, 77, 101]
```

```
import pandas as pd

df = pd.read_csv("https://archive.ics.uci.edu/static/public/53/data.csv")

df
```

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns