GUSTAVO CORONEL
DESARROLLA SOFTWARE

CIENCIA DE DATOS: TALLER DE FUNDAMENTOS DE MACHINE LEARNING CON PYTHON

Módulo 02
ESTRUCTURAS FUNDAMENTALES DE PYTHON: SINTAXIS Y CONTROL DE FLUJO

Dr. Eric Gustavo Coronel Castillo
Docente UNI
gcoronel@uni.edu.pe

ÍNDICE

1 Presentación

Python es uno de los lenguajes más utilizados en ciencia de datos debido a su sintaxis clara y a su ecosistema de bibliotecas. Sin embargo, antes de trabajar con librerías como NumPy o pandas, es indispensable dominar las estructuras fundamentales del lenguaje. Este módulo se enfoca en construir esa base: operaciones y variables, colecciones nativas (listas, tuplas y diccionarios) y el control de flujo mediante condicionales, bucles y funciones. Con estos elementos, el estudiante podrá escribir scripts y notebooks más legibles, modularizar su código y realizar transformaciones básicas de datos de forma confiable. El módulo prioriza ejemplos prácticos orientados a tareas típicas de análisis: filtrado, limpieza elemental, cálculos de indicadores y construcción de estructuras para preparar información antes del modelado.

2 Objetivo

2.1 Objetivo general

- Dominar las estructuras fundamentales de Python (variables, colecciones nativas y control de flujo) para implementar procesos básicos de manipulación y preparación de datos con código claro, reproducible y modular.

2.2 Objetivos específicos

- Reconocer y aplicar operadores, tipos de datos y reglas básicas de nombres para construir expresiones correctas y legibles.
- Construir y manipular listas, tuplas y diccionarios usando indexación, slicing, métodos principales y comprensiones, según el problema.
- Implementar estructuras selectivas (if/elif/else) para validar datos y aplicar reglas de negocio simples (por ejemplo, rangos válidos y etiquetas).
- Implementar estructuras iterativas (for/while) para recorrer colecciones, acumular resultados y generar resúmenes (conteos, totales, promedios).
- Definir y utilizar funciones para reutilizar lógica, reducir duplicidad y organizar el análisis en pasos claros.
- Identificar y evitar errores frecuentes (referencias compartidas, conversiones de tipo, claves inexistentes, límites de índice) mediante prácticas básicas de verificación.

3 OPERACIONES Y VARIABLES

3.1 Conceptos esenciales

3.1.1 Variables: qué son y por qué importan

Una variable es un nombre que referencia un valor en memoria. En Python, el tipo de la variable se infiere automáticamente a partir del valor asignado (tipado dinámico). Esto facilita comenzar rápido, pero exige verificar tipos cuando se combinan datos (por ejemplo, texto con números) para evitar resultados inesperados.

Asignación

Ejemplos de asignación simple y múltiple:

```
x = 10
y = 3
a, b, c = 1, 2, 3  # asignación múltiple
m = n = p = 50     # asignación encadenada; m, n y p toman el mismo valor
x = x + 1          # actualización
```

Buenas prácticas de nombres

Aplique reglas simples para mejorar legibilidad y evitar errores:

- Use nombres descriptivos: total_ventas, promedio_semana.
- Use snake_case (recomendación de PEP 8).
- Evite nombres ambiguos: data, temp, var.
- No use palabras reservadas: if, for, while, etc.

3.1.2 Operaciones: lo mínimo que debe dominar

Operadores aritméticos

Operadores comunes y su uso:

```
x = 17
y = 5
x + y   # suma
x - y   # resta
```

```
x * y   # multiplicación
x / y   # división real (float)
x // y  # división entera (floor)
x % y   # módulo (residuo)
x ** y  # potencia
```

En ciencia de datos, la diferencia entre '/' y '//' es importante: use '/' para cálculos numéricos (promedios, tasas, normalización) y reserve '//' para casos discretos (por ejemplo, agrupar en intervalos enteros).

Orden de operaciones (precedencia)

Python respeta el orden matemático usual. Use paréntesis para dejar la intención explícita y reducir errores al leer el código.

```
2 + 3 * 4        # 14
(2 + 3) * 4      # 20
3 * 3 ** 2 / 9   # 3.0
```

Operadores de comparación y lógicos

Se usan para crear condiciones (filtrar, validar, limpiar):

```
edad = 20
edad >= 18                        # True
(edad >= 18) and (edad < 65)      # True
not (edad < 0)                    # True
```

Advertencia frecuente: no confunda '=' (asignación) con '==' (comparación).

3.1.3 Tipos numéricos y precisión

En ciencia de datos trabajará sobre todo con int (enteros) y float (decimales). Los float representan números en punto flotante; por ello, algunas operaciones pueden mostrar pequeños errores de representación (no es un bug: es una limitación del formato).

```
0.1 + 0.2  # puede mostrar 0.30000000000000004
```

Para comparaciones de decimales, evite '==' si provienen de cálculos. En su lugar, use tolerancias (por ejemplo, con round() o comparaciones por margen).

3.1.4 Conversión de tipos (casting)

El casting es común al leer datos (CSV/Excel) o recibir valores desde formularios/inputs. Aplique conversiones explícitas cuando el tipo sea parte de la lógica.

```
texto = '25'
edad = int(texto)          # 25
precio = float('19.90')    # 19.9
str(edad)                  # '25'
```

3.1.5 2.5 Impresión y formateo para reportar resultados

En notebooks, es buena práctica mostrar resultados con mensajes claros. Los f-strings permiten formateo legible.

```
promedio = 16.0
print(f'Promedio: {promedio:.2f}')  # 2 decimales
```

3.2 Errores típicos y cómo evitarlos

- Mezclar tipos sin convertir (por ejemplo, '10' + 5).
- Dividir enteros esperando un entero (use '//' solo si realmente corresponde).
- Usar nombres poco recordables (x1, x2, dato) en análisis largos.
- No revisar tipos al leer datos: números como texto, fechas como texto, etc.
- Suponer que una comparación de float será exacta.

3.3 Práctica

Ejecute el siguiente bloque en su notebook y observe los resultados:

```
# 1) Variables y operaciones
ingresos = 1250.50
gastos = 980.10
utilidad = ingresos – gastos

# 2) Indicadores simples
margen = utilidad / ingresos

# 3) Reporte
print(f'Utilidad: S/ {utilidad:.2f}')
print(f'Margen: {margen:.2%}')
```

Interpretación esperada:

- **Utilidad**: diferencia entre ingresos y gastos.
- **Margen**: utilidad relativa sobre ingresos (porcentaje).

3.4 Ejercicios propuestos

1. Cree tres variables (a, b, c) y calcule el promedio usando división real.
2. Calcule el residuo de 137 entre 10 y explique su interpretación.
3. Construya una condición que valide si una edad está entre 18 y 65 (inclusive).
4. Convierta los textos '3.1416' y '2025' a float e int respectivamente, y muestre sus tipos con type().
5. Imprima un resultado monetario con dos decimales y un porcentaje con dos decimales.

4 Listas, tuplas y diccionarios

4.1 Introducción

Las estructuras de datos son componentes fundamentales en cualquier lenguaje de programación, y Python no es la excepción. En el contexto de la ciencia de datos, la capacidad de organizar, almacenar y manipular información de manera eficiente es crucial para el procesamiento y análisis de grandes volúmenes de datos. Python ofrece tres estructuras de datos nativas principales para trabajar con colecciones de elementos: listas, tuplas y diccionarios.

Según Van Rossum y Drake (2009), el diseño de estas estructuras de datos en Python busca un equilibrio entre simplicidad, eficiencia y versatilidad. Cada una de estas estructuras tiene características específicas que las hacen más adecuadas para ciertos tipos de problemas y escenarios de uso en ciencia de datos.

En esta separata, exploraremos en profundidad cada una de estas estructuras, analizando sus propiedades, métodos, casos de uso y mejores prácticas para su implementación en proyectos de análisis de datos.

4.2 Listas (Lists)

4.2.1 Definición y Características

Una lista en Python es una estructura de datos ordenada y mutable que puede contener elementos de diferentes tipos. McKinney (2017) señala que las listas son una de las estructuras más versátiles en Python y constituyen la base para muchas operaciones de procesamiento de datos.

Características principales:

- Ordenadas: los elementos mantienen el orden en que fueron insertados
- Mutables: pueden modificarse después de su creación
- Heterogéneas: pueden contener elementos de diferentes tipos de datos
- Dinámicas: pueden crecer o reducirse según sea necesario

4.2.2 Creación de Listas

Existen múltiples formas de crear listas en Python. A continuación, se presentan los métodos más comunes.

Método 1: Sintaxis de corchetes

```
# Lista vacía
lista_vacia = []
# Lista con elementos
numeros = [1, 2, 3, 4, 5]
textos = ['Python', 'Java', 'C++']
mixta = [1, 'dos', 3.0, True]
```

Método 2: Función list()

```
# Convertir otros iterables a lista
lista_desde_tupla = list((1, 2, 3))
lista_desde_cadena = list('Python')
lista_desde_rango = list(range(5))
```

Método 3: List comprehension

```
# Creación mediante comprensión de listas
cuadrados = [x**2 for x in range(10)]
pares = [x for x in range(20) if x % 2 == 0]
```

4.2.3 Operaciones Básicas con Listas

Las listas soportan diversas operaciones que permiten manipular sus elementos de forma eficiente:

Acceso a elementos (indexación)

```
lenguajes = ['Python', 'Java', 'JavaScript', 'C++', 'Ruby']
# Indexación positiva (desde el inicio)
primer_elemento = lenguajes[0]  # 'Python'
# Indexación negativa (desde el final)
ultimo_elemento = lenguajes[-1]  # 'Ruby'
# Slicing (rebanado)
Subconjunto1 = lenguajes[1:4]  # ['Java', 'JavaScript', 'C++']
subconjunto2 = lenguajes[2:]   # ['JavaScript', 'C++', 'Ruby']
```

Modificación de elementos

```
numeros = [1, 2, 3, 4, 5]
# Modificar un elemento individual
numeros[0] = 10  # [10, 2, 3, 4, 5]
# Modificar múltiples elementos mediante slicing
numeros[1:3] = [20, 30, 40]  # [10, 20, 30, 40, 4, 5]
```

Concatenación y repetición

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
# Concatenación
combinada = lista1 + lista2  # [1, 2, 3, 4, 5, 6]
# Repetición
repetida = lista1 * 3  # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

4.2.4 Métodos Principales de Listas

Python proporciona una amplia gama de métodos incorporados para trabajar con listas. La Tabla 1 resume los métodos más utilizados en ciencia de datos.

Tabla 1
Métodos principales de listas en Python

Método	Descripción	Ejemplo
append(x)	Agrega un elemento al final de la lista	lista.append(5)
extend(iterable)	Extiende la lista agregando elementos de un iterable	lista.extend([6,7])
insert(i, x)	Inserta un elemento en la posición indicada	lista.insert(0, 1)
remove(x)	Elimina la primera ocurrencia del valor x	lista.remove(5)
pop([i])	Elimina y devuelve el elemento en la posición i	lista.pop()
sort()	Ordena los elementos de la lista in-place	lista.sort()
reverse()	Invierte el orden de los elementos	lista.reverse()

4.2.5 Aplicaciones en Ciencia de Datos

Las listas son fundamentales en el preprocesamiento de datos. Según Müller y Guido (2016), se utilizan comúnmente para:

- Almacenar secuencias de observaciones o mediciones
- Implementar estructuras de datos más complejas
- Procesar datos antes de convertirlos a arrays de NumPy o DataFrames de Pandas
- Manejar colecciones heterogéneas de datos durante el análisis exploratorio

Ejemplo práctico:

```
# Procesar datos de sensores
temperaturas = [23.5, 24.1, 14.4, 23.8, 25.2, 24.7, 32.4]
# Calcular estadísticas básicas
promedio = sum(temperaturas) / len(temperaturas)
maximo = max(temperaturas)
minimo = min(temperaturas)
# Filtrar valores anómalos
temperaturas_validas = [t for t in temperaturas if 20 <= t <= 30]
```

4.3 Tuplas (Tuples)

4.3.1 Definición y Características

Las tuplas son estructuras de datos ordenadas e inmutables en Python. Ramalho (2015) destaca que la inmutabilidad de las tuplas las convierte en una opción preferida cuando se necesita garantizar que los datos no cambien durante la ejecución del programa.

Características principales:

- Inmutables: una vez creadas, no pueden modificarse
- Ordenadas: mantienen el orden de inserción
- Heterogéneas: pueden contener elementos de diferentes tipos
- Hashables: pueden usarse como claves en diccionarios

4.3.2 Creación de Tuplas

Sintaxis básica:

```
# Tupla vacía
tupla_vacia = ()
# Tupla con elementos
coordenadas = (10.5, 20.3)
info_persona = ('Juan', 25, 'Ingeniero')
# Tupla de un solo elemento (requiere coma)
tupla_unitaria = (42,)
# Sin paréntesis (tuple packing)
datos = 1, 2, 3
```

Función tuple():

```
# Convertir otros iterables a tupla
tupla_desde_lista = tuple([1, 2, 3])
tupla_desde_cadena = tuple('Python')
tupla_desde_rango = tuple(range(5))
```

4.3.3 Operaciones con Tuplas

Acceso y desempaquetado:

```
# Acceso mediante índices
punto = (10, 20, 30)
x = punto[0]  # 10
y = punto[1]  # 20
# Desempaquetado (tuple unpacking)
x, y, z = punto
# Desempaquetado con *
primer, *resto = (1, 2, 3, 4, 5) # primer = 1, resto = [2, 3, 4, 5]
```

Concatenación y repetición:

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
# Concatenación
combinada = tupla1 + tupla2  # (1, 2, 3, 4, 5, 6)
# Repetición
repetida = tupla1 * 2  # (1, 2, 3, 1, 2, 3)
```

Métodos disponibles:

```
datos = (1, 2, 3, 2, 4, 2)
# count(): cuenta ocurrencias
conteo = datos.count(2)  # 3
# index(): encuentra la primera posición
posicion = datos.index(3)  # 2
```

4.3.4 Tuplas Nombradas (Named Tuples)

Python ofrece la clase namedtuple del módulo collections, que proporciona una forma de crear tuplas con campos nombrados, mejorando la legibilidad del código:

```
from collections import namedtuple
# Definir una tupla nombrada
Punto = namedtuple('Punto', ['x', 'y', 'z'])
# Crear instancias
p1 = Punto(10, 20, 30)
p2 = Punto(x=15, y=25, z=35)
# Acceso por nombre o índice
print(p1.x)  # 10
print(p1[0])  # 10
```

4.3.5 Aplicaciones en Ciencia de Datos

Las tuplas son especialmente útiles en ciencia de datos para:

- Representar coordenadas o puntos multidimensionales
- Retornar múltiples valores desde funciones
- Usar como claves en diccionarios cuando se necesita una clave compuesta
- Proteger datos que no deben modificarse durante el análisis

Ejemplo práctico:

```
# Función que retorna estadísticas
def calcular_estadisticas(datos):
     promedio = sum(datos) / len(datos)
     minimo = min(datos)
     maximo = max(datos)
     return (promedio, minimo, maximo)

# Uso
valores = [23, 45, 12, 67, 34]
stats = calcular_estadisticas(valores)
prom, min_val, max_val = stats
```

4.4 Diccionarios (Dictionaries)

4.4.1 Definición y Características

Los diccionarios son estructuras de datos que almacenan pares clave-valor, proporcionando una forma eficiente de mapear información. Beazley y Jones (2013) señalan que los diccionarios son implementados mediante tablas hash, lo que garantiza tiempos de acceso prácticamente constantes O(1) para la mayoría de las operaciones.

Características principales:

- No ordenados (hasta Python 3.6): a partir de Python 3.7, mantienen el orden de inserción
- Mutables: pueden modificarse después de su creación
- Claves únicas: no pueden existir claves duplicadas

- Claves hashables: las claves deben ser objetos inmutables (strings, números, tuplas)

4.4.2 Creación de Diccionarios

Método 1: Sintaxis de llaves

```
# Diccionario vacío
diccionario_vacio = {}

# Diccionario con elementos
estudiante = {
     'nombre': 'Ana García',
     'edad': 22,
     'carrera': 'Ciencia de Datos',
     'promedio': 18.5
}
```

Método 2: Función dict()

```
# Desde pares clave-valor
producto = dict(nombre='Laptop', precio=1200, stock=15)

# Desde lista de tuplas
items = [('a', 1), ('b', 2), ('c', 3)]
diccionario = dict(items)
```

Método 3: Dictionary comprehension

```
# Crear diccionario mediante comprensión
# Resultado: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
cuadrados = {x: x**2 for x in range(6)}
```

4.4.3 Operaciones Básicas con Diccionarios

Acceso a elementos:

```
# diccionario
estudiante = {'nombre': 'Ana', 'edad': 22, 'promedio': 18.5}

# Acceso directo (lanza error si no existe)
nombre = estudiante['nombre']

# Método get() (devuelve None o valor por defecto)
edad = estudiante.get('edad')
telefono = estudiante.get('telefono', 'No registrado')
```

Modificación y adición:

```
# Modificar valor existente
estudiante['edad'] = 23

# Agregar nuevo par clave-valor
estudiante['email'] = 'ana@universidad.edu'

# Actualizar múltiples valores
estudiante.update({'edad': 24, 'semestre': 8})
```

Eliminación de elementos:

```
# Eliminar con del
del estudiante['semestre']

# Eliminar y obtener valor con pop()
promedio = estudiante.pop('promedio')

# Eliminar último elemento insertado (Python 3.7+)
ultimo = estudiante.popitem()
```

4.4.4 Métodos Principales de Diccionarios

La **Tabla *2*** presenta los métodos más utilizados para trabajar con diccionarios.

Tabla 2

Métodos principales de diccionarios en Python

Método	Descripción	Ejemplo
keys()	Retorna una vista de todas las claves	d.keys()
values()	Retorna una vista de todos los valores	d.values()
items()	Retorna una vista de pares (clave, valor)	d.items()
get(key, default)	Obtiene el valor de una clave o un valor por defecto	d.get('x', 0)
setdefault(key, default)	Obtiene valor o establece uno por defecto si no existe	d.setdefault('x', [])
update(otro)	Actualiza el diccionario con pares de otro diccionario	d.update(d2)

4.4.5 Iteración sobre Diccionarios

Formas comunes de iterar:

```
estudiante = {'nombre': 'Ana', 'edad': 22, 'carrera': 'Ciencia de Datos'}

# Iterar sobre claves
for clave in estudiante:
     print(clave)

# Iterar sobre valores
for valor in estudiante.values():
     print(valor)

# Iterar sobre pares clave-valor
for clave, valor in estudiante.items():
     print(f'{clave}: {valor}')
```

4.4.6 Aplicaciones en Ciencia de Datos

Los diccionarios son fundamentales en ciencia de datos para diversas aplicaciones. VanderPlas (2016) destaca su importancia en:

- Representar registros de datos con campos nombrados
- Contar frecuencias de elementos
- Mapear identificadores a valores
- Configurar parámetros de modelos de machine learning
- Almacenar resultados de análisis con metadatos

Ejemplo práctico - Conteo de frecuencias:

```
# Contar frecuencia de palabras en un texto
texto = "python es un lenguaje de programación python es versátil"
palabras = texto.split()
frecuencias = {}
for palabra in palabras:
     frecuencias[palabra] = frecuencias.get(palabra, 0) + 1
print(frecuencias) # {'python': 2, 'es': 2, 'un': 1, 'lenguaje': 1, ...}
```

4.5 Comparación y Criterios de Selección

La selección apropiada de la estructura de datos es crucial para el rendimiento y la claridad del código. La **Tabla *3*** presenta una comparación de las características principales.

Tabla 3
Comparación de estructuras de datos en Python

Característica	**Lista**	**Tupla**	**Diccionario**
Mutabilidad	Mutable	Inmutable	Mutable
Orden	Ordenada	Ordenada	Ordenada (3.7+)
Indexación	Por posición	Por posición	Por clave
Duplicados	Permitidos	Permitidos	Claves únicas
Uso típico	Secuencias modificables	Datos inmutables	Mapeo clave-valor

4.6 Conclusiones

El dominio de las estructuras de datos fundamentales en Python es esencial para el desarrollo efectivo en ciencia de datos. Las listas proporcionan flexibilidad para colecciones mutables, las tuplas garantizan integridad de datos inmutables, y los diccionarios ofrecen acceso eficiente mediante claves. La elección apropiada de la estructura de datos no solo afecta el rendimiento del código, sino también su legibilidad y mantenibilidad.

A medida que avancemos en el curso hacia librerías especializadas como NumPy y Pandas, estas estructuras fundamentales servirán como base conceptual para comprender estructuras más complejas como arrays multidimensionales y DataFrames.

4.7 Ejercicios Propuestos

4.7.1 Ejercicios con Listas

1. Cree una lista con las temperaturas registradas durante una semana (7 valores). Calcule el promedio, identifique la temperatura máxima y mínima, y cuente cuántos días la temperatura superó los 25°C.
2. Dada una lista de números enteros, escriba un programa que genere dos listas nuevas: una con los números pares y otra con los impares. Use list comprehension.
3. Implemente una función que reciba una lista de palabras y retorne una nueva lista con las palabras ordenadas por longitud (de menor a mayor).

4.7.2 Ejercicios con Tuplas

4. Cree una tupla que represente las coordenadas (x, y, z) de un punto en el espacio tridimensional. Escriba una función que calcule la distancia euclidiana entre dos puntos.
5. Implemente una función que retorne los estadísticos básicos (media, mediana, moda) de una lista de números usando una tupla para el retorno múltiple.
6. Utilice namedtuple para crear una estructura que represente información de estudiantes (nombre, edad, carrera, promedio). Cree una lista de 5 estudiantes y encuentre el estudiante con el promedio más alto.

4.7.3 Ejercicios con Diccionarios

7. Cree un diccionario que almacene información de productos (nombre, precio, stock, categoría). Implemente funciones para agregar productos, actualizar stock y buscar productos por categoría.
8. Escriba un programa que cuente la frecuencia de cada carácter en una cadena de texto usando un diccionario. Luego, identifique los 3 caracteres más frecuentes.
9. Dados dos diccionarios que representan las ventas de dos meses diferentes, fusione ambos diccionarios sumando las ventas de productos comunes.

4.7.4 Ejercicios Integradores

10. Cree un sistema de gestión de calificaciones que use diccionarios para almacenar información de estudiantes (nombre como clave) y listas para sus calificaciones. Implemente funciones para agregar estudiantes, registrar calificaciones y calcular el promedio de cada estudiante.
11. Implemente un analizador de texto que reciba un párrafo y retorne un diccionario con estadísticas: número de palabras, número de oraciones, palabra más frecuente, y longitud promedio de palabras.
12. Desarrolle un programa que simule un inventario de biblioteca. Use diccionarios para representar libros (título, autor, año, disponible) y listas para gestionar múltiples copias. Implemente funciones para préstamo, devolución y búsqueda de libros.

5 7. Referencias

Beazley, D., & Jones, B. K. (2013). *Python Cookbook (3rd ed.).* O'Reilly Media. https://www.oreilly.com/library/view/python-cookbook-3rd/9781449357337/

McKinney, W. (2017). *Python for Data Analysis (2nd ed.).* O'Reilly Media. https://www.oreilly.com/library/view/python-for-data/9781491957653/

Müller, A. C., & Guido, S. (2016). *Introduction to Machine Learning with Python.* O'Reilly Media. https://www.oreilly.com/library/view/introduction-to-machine/9781449369880/

Python Software Foundation. (2026). *The Python Language Reference*. https://docs.python.org/3/reference/

Python Software Foundation. (2026). *The Python Tutorial*. https://docs.python.org/3.13/tutorial/

Ramalho, L. (2015). *Fluent Python.* O'Reilly Media. https://www.oreilly.com/library/view/fluent-python/9781491946237/

Van Rossum, G., Warsaw, B., & Coghlan, A. (2001). *PEP 8 – Style guide for Python code*. Python Enhancement Proposals. https://peps.python.org/pep-0008/

VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data.* O'Reilly Media. https://doi.org/https://jakevdp.github.io/PythonDataScienceHandbook/