Módulo 1: Fundamentos de Python

Introducción a la Programación en Python para Aprendizaje por Refuerzo

Dr. Darío Ezequiel Díaz

SAE

Marzo 2025

Contenidos de la clase

Historia de Python

- Creado por Guido van Rossum a finales de los 80s, lanzado en 1991
- Nombrado por el grupo cómico Monty Python, no por la serpiente
- Historia de versiones:
  - Python 1.0 (1994): Características funcionales como lambda, map, filter
  - Python 2.0 (2000): Recolector de basura, soporte Unicode
  - Python 3.0 (2008): Cambios importantes no compatibles con versiones anteriores
  - Python 3.11+ (Actual): Mejoras significativas en velocidad y diagnóstico de errores
- Filosofía: .El Zen de Python"(PEP 20) - Legibilidad y simplicidad

El Zen de Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
...
```

Características principales

- Lenguaje interpretado de alto nivel
- Tipado dinámico y fuerte
- Sintaxis clara y legible que enfatiza la indentación
- Multiparadigma: soporta programación orientada a objetos, imperativa y funcional
- Extensa biblioteca estándar ("baterías incluidas")
- Alta portabilidad: funciona en múltiples plataformas
- Extensible: permite integrar código C/C++ cuando se necesita mayor rendimiento
- Comunidad activa y en crecimiento constante

Aplicaciones de Python

Python destaca en numerosos campos:

- **Ciencia de datos** y análisis estadístico (NumPy, Pandas, SciPy)
- **Aprendizaje automático** (TensorFlow, PyTorch, scikit-learn)
- **Aprendizaje por refuerzo** (OpenAI Gym, Stable Baselines)
- Desarrollo web (Django, Flask)
- Automatización y scripting
- Aplicaciones de escritorio (PyQt, Tkinter)
- Procesamiento de imágenes y visión computacional (OpenCV)
- Computación científica y visualización (Matplotlib)
- Bioinformática, astronomía, física cuántica

El intérprete de Python

- Diferentes formas de ejecutar código Python:
  - Intérprete interactivo (REPL): útil para experimentar
  - Ejecución de scripts (.py): para programas completos
  - Notebooks (Jupyter): ideal para datos y educación
  - Entornos como Google Colab: sin instalación local
- Para nuestro curso utilizaremos Google Colaboratory
- Permite acceso gratuito a recursos computacionales
- Facilita compartir y colaborar en código

Tu primer programa Python

```python
# Este es un comentario en Python
print(" Hola , mundo!")  # Imprime un mensaje

# Variables y asignación básica
nombre = "Estudiante"
edad = 25
pi_aproximado = 3.14159

# Mostrando información con formato
print(f"Hola {nombre}, tienes {edad} años")

# Operación sencilla
año_nacimiento = 2025 - edad
print(f"Naciste aproximadamente en {año_nacimiento}")
```

Sintaxis: Características clave

- **Indentación significativa**:
  - Python usa espacios (habitualmente 4) para delimitar bloques
  - No se utilizan llaves {} ni palabras clave end
  - La indentación incorrecta produce errores
- **Comentarios**:
  - Línea: comienzan con # (hasta el final de la línea)
  - Multilínea: entre triples comillas (”’comentario”’ o ¸comentario”)
- **Case sensitive**: distingue entre mayúsculas y minúsculas
- **Nombres de variables**: letras, números y guiones bajos (no pueden empezar con número)

Entrada y salida básica

```
1  # Salida básica
2  print("Esto es un texto")
3  print("Múltiples", "argumentos", "separados", "por", "comas"
       )
4  print("Puedes controlar el separador", end=" -> ")
5  print("y el final de línea")
6
7  # Entrada de usuario
8  nombre = input(" C ómo te llamas? ")
9  print(f"Hola {nombre}, bienvenido al curso de Python")
10
11 # Conversión de tipos en la entrada
12 edad = int(input(" Cu ál es tu edad? "))
13 print(f"En 10 años tendrás {edad + 10} años")
```

Tipos numéricos

- **Enteros** (int): números sin parte decimal
  - Sin límite práctico de tamaño en Python 3
- Ejemplos: 42, -7, 0, $1_0 00_0 00$(*separadoresparalegibilidad*)
- **Punto flotante** (float): números con parte decimal
  - Representación aproximada (cuidado con comparaciones)
  - Ejemplos: 3.14159, -0.001, 1.0, 2e10 (notación científica)
- **Complejos** (complex): números con parte real e imaginaria
  - Formato: real + imagj
  - Ejemplo: 3+4j, 2.5-1j

Operaciones con números

```python
# Operaciones básicas
a = 10
b = 3

print(a + b)     # Suma: 13
print(a - b)     # Resta: 7
print(a * b)     # Multiplicación: 30
print(a / b)     # División: 3.3333... (siempre devuelve
    float)
print(a // b)    # División entera: 3
print(a % b)     # Módulo (resto): 1
print(a ** b)    # Potencia: 1000

# Funciones matemáticas (importando el módulo math)
import math
print(math.sqrt(16))    # Raíz cuadrada: 4.0
print(math.sin(math.pi/2))  # Seno: 1.0
print(math.log(100, 10))  # Logaritmo base 10: 2.0
```

Cadenas de texto (strings)

- En Python, los strings se pueden definir con diferentes tipos de comillas:
  - `'Comillas simples'`
  - `Çomillas dobles"`
  - `'''Texto con múltiples líneas'''`

```
1 # Diferentes formas de definir strings
2 s1 = 'Comillas simples'
3 s2 = "Comillas dobles"
4 s3 = '''Texto con
5 múltiples líneas'''
```

Operaciones con strings

```python
nombre = "Python"
print(len(nombre))        # Longitud: 6
print(nombre[0])          # Primer carácter: P
print(nombre[-1])         # Último carácter: n
print(nombre[0:2])        # Subcadena (slice): Py
print(nombre + " 3.10") # Concatenación: Python 3.10
print(nombre * 3)         # Repetición: PythonPythonPython
```

Métodos útiles para strings

```
s = "aprendizaje por refuerzo"
print(s.upper())          # Mayúsculas: APRENDIZAJE POR REFUERZO
print(s.title())          # Capitalización: Aprendizaje Por Refuerzo
print(s.replace("por", "de")) # Reemplaza: aprendizaje de refuerzo
print("python" in s)      # Verificar contención: False
```

Formato de strings

```python
nombre = "Ana"
edad = 30
pi = 3.14159

# Método format()
print("Hola, {} tienes {} años".format(nombre, edad))

# Especificando posición de argumentos
print("{1} tiene {0} años".format(edad, nombre))

# Con nombres
print("{persona} tiene {años} años".format(
    persona=nombre, años=edad))

# f-strings (Python 3.6+) - Más moderno y recomendado
print(f"Hola, {nombre} tienes {edad} años")
print(f"El valor de pi es {pi:.2f}")  # Con formato: 3.14
print(f"{nombre.lower()} en mayúsculas: {nombre.upper()}")
```

Booleanos en Python

- En Python, los valores booleanos son:
  - `True` (Verdadero)
  - `False` (Falso)
- Son el resultado de expresiones lógicas o comparaciones.

```
1 # Valores booleanos
2 verdadero = True
3 falso = False
```

Operadores de comparación

```
a = 5
b = 10

print(a == b)     # Igualdad: False
print(a != b)     # Desigualdad: True
print(a < b)      # Menor que: True
print(a <= b)     # Menor o igual: True
print(a > b)      # Mayor que: False
print(a >= b)     # Mayor o igual: False
```

Operadores lógicos

```
# Operadores lógicos
print(True and False)  # AND lógico: False
print(True or False)   # OR lógico: True
print(not True)        # NOT lógico: False

# Combinando operadores
c = 15
print(a < b and b < c)  # True
print(a < b < c)        # Equivalente y más pytónico
```

Conversión entre tipos en Python

- Python permite convertir entre diferentes tipos de datos con estas funciones:
  - `int()`: convierte a entero (número sin decimales)
  - `float()`: convierte a punto flotante (número con decimales)
  - `str()`: convierte a cadena de texto
  - `bool()`: convierte a booleano (True/False)

Ejemplos de conversión entre tipos

```
# Conversión básica entre tipos
print(int(5.7))        # 5 (trunca la parte decimal)
print(float(5))        # 5.0
print(str(5.7))        # "5.7"
print(bool(5))         # True
```

Conversión desde strings

```
1 print(int("42"))        # 42
2 print(float("3.14"))    # 3.14
```

Reglas importantes

- Valores que convierten a `False`:
  - `bool(0)`, `bool('')`, `bool([])`, `bool(None)`
- Cualquier otro valor no vacío convierte a `True`
- `int('42')` funciona, pero `int('42.0')` genera error
- Solución para textos con decimales:

```
1 valor = "42.8"
2 # print(int(valor))  # ERROR!
3 entero = int(float(valor))  # Correcto: 42
```

Estructura condicional: if-else

- Permite ejecutar diferentes bloques de código dependiendo de una condición.
- Sintaxis básica:

```
edad = 18

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Estructura condicional: if-elif-else

- Se usa cuando hay múltiples condiciones a evaluar.

```
puntuacion = 85

if puntuacion >= 90:
    calificacion = "A"
elif puntuacion >= 80:
    calificacion = "B"
elif puntuacion >= 70:
    calificacion = "C"
elif puntuacion >= 60:
    calificacion = "D"
else:
    calificacion = "F"

print(f"Tu calificación es: {calificacion}")
```

Expresión condicional en una línea

- En Python, se puede escribir una condición en una sola línea usando el operador ternario.
- Sintaxis: `valor_si_verdadero if condición else valor_si_falso`

```
edad = 20
mensaje = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(mensaje)  # Mayor de edad
```

Múltiples condiciones en if-elif-else

- Se pueden encadenar varias condiciones para evaluar diferentes rangos de valores.

```
temperatura = 25

if temperatura < 0:
    estado = "Congelado"
elif 0 <= temperatura < 15:
    estado = "Frío"
elif 15 <= temperatura < 25:
    estado = "Templado"
else:
    estado = "Caluroso"

print(estado)  # Templado
```

Anidamiento de condiciones

- Es posible colocar una estructura condicional dentro de otra.
- Esto se usa, por ejemplo, en sistemas de autenticación.

```
usuario = "admin"
contraseña = "12345"

if usuario == "admin":
    if contraseña == "12345":
        print("Acceso concedido al panel de administración")
    else:
        print("Contraseña incorrecta")
else:
    print("Usuario no reconocido")
```

Bucle for

```python
# Iteración sobre una secuencia de números
for i in range(5):  # 0, 1, 2, 3, 4
    print(i, end=" ")  # 0 1 2 3 4

# Especificando inicio, fin (exclusivo) y paso
for i in range(2, 10, 2):  # 2, 4, 6, 8
    print(i, end=" ")  # 2 4 6 8

# Iteración sobre una lista
colores = ["rojo", "verde", "azul"]
for color in colores:
    print(f"Color: {color}")

# Iteración con índice usando enumerate
for i, color in enumerate(colores):
    print(f"Índice {i}: {color}")

# Iteración sobre un string
for letra in "Python":
    print(letra, end="-")  # P-y-t-h-o-n-
```

Bucle while

```python
# Bucle while básico
contador = 0
while contador < 5:
    print(contador, end=" ")  # 0 1 2 3 4
    contador += 1

# Bucle con break (salir del bucle)
num = 0
while True:  # Bucle infinito
    if num >= 5:
        break  # Sale del bucle cuando num >= 5
    print(num, end=" ")  # 0 1 2 3 4
    num += 1

# Bucle con continue (saltar a la siguiente iteración)
for i in range(10):
    if i % 2 == 0:  # Si es par
        continue  # Salta a la siguiente iteración
    print(i, end=" ")  # 1 3 5 7 9
```

Bucles anidados y control de flujo

```python
# Bucles anidados: tabla de multiplicar
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j}")
    print("-----")

# Uso de else con bucles (se ejecuta si el bucle termina normalmente)
for i in range(5):
    print(i, end=" ")
else:
    print("\nBucle completado sin interrupciones")

# Cuando else no se ejecuta (por break)
for i in range(5):
    if i == 3:
        break
    print(i, end=" ")
else:
    print("\nEsto no se imprimirá porque hubo un break")
```

Listas en Python: Creación y acceso

- Una lista es una colección ordenada de elementos.
- Puede contener distintos tipos de datos.

```python
# Creación de listas
numeros = [1, 2, 3, 4, 5]
mixta = [1, "dos", 3.0, True, [5, 6]]
vacia = []

# Acceso a elementos
print(numeros[0])  # Primer elemento: 1
print(numeros[-1]) # Último elemento: 5
```

Listas en Python: Slicing y manipulación

- Se puede acceder a partes de la lista con `slice`.

```
print(numeros[1:3])   # Slice: [2, 3]
print(numeros[:3])    # Desde inicio hasta índice 3 (excl): [1, 2, 3]
print(numeros[2:])    # Desde índice 2 hasta el final: [3, 4, 5]
print(numeros[::2])   # Cada 2 elementos: [1, 3, 5]
print(numeros[::-1]) # Orden inverso: [5, 4, 3, 2, 1]
```

Listas en Python: Modificación

- Las listas son mutables, lo que significa que sus elementos pueden cambiarse.

```
# Modificación de listas
numeros[0] = 10          # Modificar un elemento
numeros.append(6)        # Añadir al final: [10, 2, 3, 4, 5, 6]
numeros.insert(1, 15)    # Insertar en posición: [10, 15, 2, 3, 4, 5, 6]
numeros.extend([7, 8]) # Extender con otra lista: [10, 15, 2, 3, 4, 5, 6, 7, 8]
```

Listas en Python: Eliminación de elementos

- Se pueden eliminar elementos por índice o por valor.

```
numeros = [10, 15, 2, 3, 4, 5, 6, 7, 8]

# Eliminación de elementos
eliminado = numeros.pop()          # Elimina y devuelve el ú
    ltimo: 8
print(numeros)                     # [10, 15, 2, 3, 4, 5, 6, 7]

eliminado = numeros.pop(1)         # Elimina y devuelve índice
    1: 15
numeros.remove(5)                  # Elimina primera ocurrencia
     de 5
print(numeros)                     # [10, 2, 3, 4, 6, 7]
```

Listas en Python: Otras operaciones útiles

```
# Otras operaciones útiles con listas
print(len(numeros))                 # Longitud: 6
print(2 in numeros)                 # Verificar existencia: True
print(numeros.index(4))             # Índice de primera
    ocurrencia: 3

numeros.sort()                      # Ordena la lista in-place:
    [2, 3, 4, 6, 7, 10]
numeros.reverse()                   # Invierte in-place: [10, 7,
     6, 4, 3, 2]

copia = numeros.copy()              # Crea una copia de la lista
```

Diccionarios en Python: Creación y acceso

- Un diccionario es una colección de pares `clave: valor`.
- Permite acceso rápido a los valores a través de las claves.

```
# Creación de diccionarios
persona = {
    "nombre": "Ana",
    "edad": 28,
    "profesion": "científica de datos"
}

# Diccionario vacío
vacio = {}  # o vacio = dict()

# Acceso a valores
print(persona["nombre"])  # Ana
print(persona.get("altura", "dato no disponible"))  # Valor por defecto
```

Diccionarios en Python: Modificación

- Se pueden agregar, modificar y eliminar elementos de un diccionario.

```
# Modificación de diccionarios
persona["edad"] = 29                    # Modifica un valor existente
persona["ciudad"] = "Buenos Aires" # Añade un nuevo par clave-valor

# Eliminación de elementos
edad = persona.pop("edad")              # Elimina y devuelve el valor
print(persona)                          # {'nombre': 'Ana', 'profesion': '...', 'ciudad': '...'}
```

Diccionarios en Python: Operaciones comunes

```python
# Operaciones comunes
print(len(persona))          # Número de pares clave-valor: 3
print("nombre" in persona) # Verifica si la clave existe:
    True

# Obtener claves, valores y pares clave-valor
print(persona.keys())    # dict_keys(['nombre', 'profesion',
    'ciudad'])
print(persona.values()) # dict_values(['Ana', 'científica
    ...', 'Buenos Aires'])
print(persona.items())   # dict_items([('nombre', 'Ana'),
    ...])
```

Tuplas en Python: Creación

- Una tupla es una colección ordenada e **inmutable** de elementos.
- Se crean usando paréntesis () o sin ellos.

```
# Creación de tuplas
coordenadas = (10, 20)
singleton = (5,)          # Tupla de un elemento (coma
    necesaria)
vacia = ()                # Tupla vacía
tupla_mixta = (1, "dos", 3.0, True)
```

Tuplas en Python: Empaquetado y desempaquetado

- Se pueden asignar múltiples valores a una tupla (empaquetado).
- Se pueden extraer valores individuales (desempaquetado).

```
# Empaquetado
punto = 5, 10           # Creación implícita de tupla: (5, 10)

# Desempaquetado
x, y = punto            # x=5, y=10

# Tuplas como valores de retorno
def obtener_coordenadas():
    return (30, 40)     # Devuelve una tupla

lat, long = obtener_coordenadas()  # lat=30, long=40
```

Tuplas en Python: Características y operaciones

```python
# Acceso a elementos
print(coordenadas[0])  # 10
print(coordenadas[1:]) # (20,)

# Inmutabilidad
# coordenadas[0] = 15  # ERROR: las tuplas no se pueden
    modificar

# Operaciones comunes
print(len(coordenadas))   # Longitud: 2
print(20 in coordenadas)  # Verificar existencia: True
```

Conjuntos en Python: Creación y características

- Un conjunto (`set`) es una colección **desordenada** de elementos **únicos**.
- Se crean con llaves `{}` o con `set()`.

```
# Creación de conjuntos
colores = {"rojo", "verde", "azul"}
vacio = set()  # Set vacío (no se puede usar {})
numeros = set([1, 2, 3, 2, 1])  # Desde lista con duplicados

print(numeros)  # {1, 2, 3} (elimina duplicados)
```

Conjuntos en Python: Operaciones básicas

- Se pueden modificar dinámicamente con `add()` y `remove()`.

```
numeros.add(4)           # Añade un elemento: {1, 2, 3, 4}
numeros.remove(2)        # Elimina un elemento: {1, 3, 4}
# numeros.remove(5)      # ERROR: KeyError si no existe
numeros.discard(5)       # No da error si no existe

print(len(numeros))      # Longitud: 3
print(1 in numeros)      # Verificar existencia: True
```

Conjuntos en Python: Operaciones entre conjuntos

```
1 a = {1, 2, 3}
2 b = {3, 4, 5}
3
4 print(a | b)    # Unión: {1, 2, 3, 4, 5}
5 print(a & b)    # Intersección: {3}
6 print(a - b)    # Diferencia: {1, 2}
7 print(a ^ b)    # Diferencia simétrica: {1, 2, 4, 5}
8 print(a.issubset(b))  #  Es  subconjunto?: False
```

Conversiones entre estructuras de datos

```python
# Lista a partir de otras estructuras
numeros = [1, 2, 3, 2, 1]
print(list("Python"))           # ['P', 'y', 't', 'h', 'o', 'n']
print(list({1: "uno", 2: "dos"}))  # [1, 2] (solo las claves)

# Tupla a partir de otras estructuras
print(tuple(numeros))           # (1, 2, 3, 2, 1)
print(tuple("Python"))          # ('P', 'y', 't', 'h', 'o', 'n')

# Conjunto a partir de otras estructuras
print(set(numeros))             # {1, 2, 3} (elimina duplicados)
print(set("Mississippi"))       # {'M', 'i', 's', 'p'}

# Diccionario a partir de pares
pares = [("a", 1), ("b", 2)]
print(dict(pares))              # {'a': 1, 'b': 2}
```

Comprensión de listas en Python

- Permite crear listas de manera concisa y eficiente.
- Comparación entre método tradicional y comprensión de listas.

```python
# Lista tradicional vs comprensión
numeros = [1, 2, 3, 4, 5]

# Tradicional
cuadrados1 = []
for n in numeros:
    cuadrados1.append(n**2)

# Usando comprensión de listas
cuadrados2 = [n**2 for n in numeros]  # [1, 4, 9, 16, 25]
```

Comprensión de listas con condiciones

- Se pueden filtrar elementos usando condiciones en la comprensión.

```
# Lista con condición (solo números pares)
pares = [n for n in numeros if n % 2 == 0]   # [2, 4]
```

Comprensiones de listas más avanzadas

- Se pueden usar comprensiones anidadas para manipular estructuras más complejas.

```
# Aplanando una matriz
matriz = [[1, 2], [3, 4], [5, 6]]
aplanada = [num for fila in matriz for num in fila]  # [1, 2, 3, 4, 5, 6]

# Generando una matriz identidad 3x3
matriz_id = [[1 if i == j else 0 for j in range(3)] for i in range(3)]
# [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Comprensión de diccionarios y conjuntos

```python
# Comprensión de diccionarios
numeros = [1, 2, 3, 4, 5]
cuadrados_dict = {n: n**2 for n in numeros}
# {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Filtrado con comprensión de diccionarios
pares_dict = {n: n**2 for n in numeros if n % 2 == 0}
# {2: 4, 4: 16}

# Comprensión de conjuntos
cuadrados_set = {n**2 for n in numeros}
# {1, 4, 9, 16, 25}

# Invertir clave-valor en un diccionario
inventario = {"manzanas": 10, "naranjas": 5, "bananas": 10}
conteo_inv = {valor: [k for k in inventario if inventario[k]
    == valor]
            for valor in set(inventario.values())}
# {10: ['manzanas', 'bananas'], 5: ['naranjas']}
```

Funciones lambda en Python

- Una función **lambda** es una función anónima de una sola línea.
- Se usa cuando una función pequeña es necesaria temporalmente.

```
# Funciones lambda (anónimas)
sumar = lambda x, y: x + y
print(sumar(5, 3))  # 8
```

Uso de map() con funciones lambda

- La función `map()` aplica una función a cada elemento de un iterable.

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

# Map con lambda
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)  # [1, 4, 9, 16, 25]

# Map con múltiples iterables
n1 = [1, 2, 3]
n2 = [4, 5, 6]
sumas = list(map(lambda x, y: x + y, n1, n2))
print(sumas)  # [5, 7, 9]
```

Uso de filter() y combinación con map()

- `filter()` filtra elementos de un iterable según una condición.
- Se puede combinar con `map()` para transformaciones más avanzadas.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filter con lambda (filtrar pares)
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # [2, 4, 6, 8, 10]

# Combinando map y filter
cuadrados_pares = list(map(lambda x: x**2,
                           filter(lambda x: x % 2 == 0,
    numeros)))
print(cuadrados_pares)  # [4, 16, 36, 64, 100]
```

Ordenamiento básico con sort()

- La función `sort()` ordena una lista en el lugar (modifica la original).

```
# Ordenamiento básico
numeros = [3, 1, 4, 1, 5, 9, 2, 6]
numeros.sort()  # Ordena la lista in-place
print(numeros)  # [1, 1, 2, 3, 4, 5, 6, 9]
```

Ordenamiento avanzado con sorted()

- La función `sorted()` ordena una lista sin modificar la original.
- Se pueden aplicar claves de ordenamiento con el argumento `key`.

```
# Ordenamiento con key
palabras = ["python", "aprendizaje", "por", "refuerzo"]
palabras.sort(key=len)  # Ordena por longitud
print(palabras)  # ['por', 'python', 'refuerzo', 'aprendizaje']

# Ordenamiento con sorted (no modifica el original)
original = [3, 1, 4, 1, 5]
ordenado = sorted(original, reverse=True)  # Orden descendente
print(original)   # [3, 1, 4, 1, 5] (sin cambios)
print(ordenado)   # [5, 4, 3, 1, 1]
```

Ordenamiento de estructuras complejas

- Se pueden ordenar listas de diccionarios con `sorted()` y una función lambda.

```
# Ordenando objetos complejos
estudiantes = [
    {"nombre": "Ana", "nota": 85},
    {"nombre": "Carlos", "nota": 92},
    {"nombre": "Berta", "nota": 78}
]
# Ordenar por nota (descendente)
estudiantes_ordenados = sorted(
    estudiantes,
    key=lambda est: est["nota"],
    reverse=True
)

for e in estudiantes_ordenados:
    print(f"{e['nombre']}: {e['nota']}")
```

Iteración avanzada: Uso de zip()

- La función `zip()` permite combinar múltiples iterables en tuplas.

```python
nombres = ["Ana", "Ben", "Carlos"]
edades = [28, 32, 25]
ciudades = ["Madrid", "Lima", "Buenos Aires"]

# Iteración combinando listas
for nombre, edad, ciudad in zip(nombres, edades, ciudades):
    print(f"{nombre} tiene {edad} años y vive en {ciudad}")

# Convertir a lista de tuplas
datos = list(zip(nombres, edades, ciudades))
print(datos)
# [('Ana', 28, 'Madrid'), ('Ben', 32, 'Lima'), ...]

# Desempaquetado con zip
nombres2, edades2, ciudades2 = zip(*datos)
print(nombres2)  # ('Ana', 'Ben', 'Carlos')
```

Iteración avanzada: Uso de enumerate()

- La función `enumerate()` permite obtener el índice junto con el valor.

```
nombres = ["Ana", "Ben", "Carlos"]

# Iteración con índice
for i, nombre in enumerate(nombres, start=1):
    print(f"Persona {i}: {nombre}")
```

Iteración avanzada: Iterando sobre diccionarios

- La función `.items()` permite recorrer claves y valores en un diccionario.

```
1 persona = {"nombre": "Ana", "edad": 28, "ciudad": "Buenos
      Aires"}
2
3 # Iteración sobre un diccionario
4 for clave, valor in persona.items():
5     print(f"{clave}: {valor}")
```

¿Qué es PEP 8?

- PEP = Python Enhancement Proposal (Propuesta de Mejora de Python)
- PEP 8 es la guía de estilo oficial para código Python
- Creada por Guido van Rossum y otros colaboradores
- Objetivos principales:
  - Mejorar la legibilidad del código
  - Hacerlo consistente entre diferentes proyectos
  - Establecer convenciones comunes para la comunidad
- No todas las reglas son estrictas, pero seguirlas facilita:
  - Colaboración con otros desarrolladores
  - Mantenimiento a largo plazo del código
  - Adaptación a código existente

Indentación en Python

- Python utiliza **indentación** para definir bloques de código.
- Se recomienda usar **4 espacios por nivel de indentación**.

```
# Correcto (4 espacios por nivel de indentación)
def funcion_ejemplo():
    if True:
        x = 5
    return x
```

Uso de espacios en blanco en Python

- Se recomienda dejar espacios en blanco en asignaciones y estructuras de datos.

```
# Correcto
x = 5
y = x + 10
lista = [1, 2, 3, 4]
diccionario = {"a": 1, "b": 2}

# Incorrecto
x=5
y=x+10
lista=[1,2,3,4]
diccionario={"a":1,"b":2}
```

Espacios alrededor de operadores

- Se recomienda dejar espacios alrededor de operadores binarios para mejorar la legibilidad.

```
# Correcto
i = i + 1
x = x * 2 - 1
c = (a + b) * (a - b)

# Excepciones para mejorar la legibilidad
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Nomenclatura y convenciones en Python

- Python sigue convenciones de nombres según el tipo de elemento:

```python
# Variables y funciones: snake_case
nombre_estudiante = "Ana"
def calcular_promedio(valores):
    return sum(valores) / len(valores)

# Clases: CamelCase
class EstudianteGraduado:
    pass

# Constantes: MAYÚSCULAS_CON_GUIONES
VELOCIDAD_LUZ = 299792458
DIAS_SEMANA = 7

# Variables privadas (convención, no realmente privadas)
_contador_interno = 0
__valor_muy_privado = 42  # Name mangling
```

Longitud de línea y organización del código

- Se recomienda un máximo de **79 caracteres por línea** (99 en algunos casos).
- Métodos para dividir líneas largas:

```
# Opción 1: Paréntesis implícitos
resultado = (valor1 + valor2 + valor3
             + valor4 + valor5)

# Opción 2: Operador de continuación de línea
total = valor1 + valor2 + valor3 + \
        valor4 + valor5

# Opción 3: Con los parámetros de una función
def funcion_con_muchos_parametros(
        param1, param2, param3,
        param4, param5):
    print(param1)
```

Buenas prácticas en imports

- Agrupar **importaciones estándar**, de terceros y locales.
- Evitar `import *` (importar todo).

```
# Imports en líneas separadas y agrupados
import os
import sys
from collections import defaultdict, namedtuple

# Evitar imports con * (importa todo)
# Mal: from module import *
```

Comentarios y docstrings en Python

- Los **comentarios** explican el código y mejoran la legibilidad.
- Los **docstrings** describen módulos, funciones o clases.

```
# Este es un comentario de una línea

"""Este es un docstring multilínea.
Describe el propósito del módulo, función o clase."""

def calcular_area_circulo(radio):
    """Calcula el área de un círculo.

    Args:
        radio: El radio del círculo (número positivo)

    Returns:
        El área del círculo

    Raises:
        ValueError: Si el radio es negativo
    """
    if radio < 0:
        raise ValueError("El radio no puede ser negativo")
```

Herramientas para aplicar PEP 8

- **Verificadores de estilo**:
  - **pylint**: Análisis estático de código.
  - **flake8**: Verificador de PEP 8 y complejidad ciclomática.
  - **pycodestyle**: Verificador específico para PEP 8.
  - **black**: Formateador automático de código.
- **En Google Colab**:
  - Instalar verificadores: `!pip install pycodestyle`
  - Verificar un archivo: `!pycodestyle mi_archivo.py`
- **Integración en editores y IDEs**:
  - VS Code, PyCharm, Jupyter, etc. tienen extensiones.
  - Configurables para verificación automática.
  - Algunos permiten auto-formateo al guardar.

Conclusiones y mejores prácticas generales

- **Principios importantes del Zen de Python**:
  - La legibilidad cuenta.
  - Simple es mejor que complejo.
  - Explícito es mejor que implícito.
  - Plano es mejor que anidado.
- **Recomendaciones generales**:
  - Seguir PEP 8 desde el principio para crear buenos hábitos.
  - Mantener funciones pequeñas y con propósito único.
  - Nombrar variables y funciones descriptivamente.
  - Comentar "por qué", no "qué"(el código ya dice qué hace).
  - Refactorizar código repetitivo en funciones.
  - Usar tipos de datos apropiados para cada tarea.
- *."El código se lee muchas más veces de las que se escribe."*

Ejercicio 1: Calculadora básica

- **Instrucciones**:
  - Crear una calculadora que pida dos números y una operación.
  - Operaciones admitidas: suma, resta, multiplicación y división.

```
def calculadora():
    print("Calculadora básica")
    n1, n2 = float(input("Número 1: ")), float(input("Número 2: "))
    op = input("Operación (+, -, *, /): ")

    if op == "+": resultado = n1 + n2
    elif op == "-": resultado = n1 - n2
    elif op == "*": resultado = n1 * n2
    elif op == "/":
        if n2 == 0: return "Error: División por cero"
        resultado = n1 / n2
    else: return "Operación no válida"

    return f"Resultado: {resultado}"
```

Ejercicio 2: Lista de compras

- **Instrucciones**:
  - Gestionar una lista de compras con un diccionario.
  - Permitir añadir, eliminar, ver productos y calcular el total.

```
def lista_compras():
    compras = {}
    while True:
        opcion = input("\n1. Añadir  2. Eliminar  3. Ver  4. Total  5. Salir: ")
        if opcion == "1":
            compras[input("Producto: ")] = float(input("Precio: "))
        elif opcion == "2":
            compras.pop(input("Producto: "), print("No encontrado"))
        elif opcion == "3":
            print("\n".join(f"{p}: ${v:.2f}" for p, v in compras.items()))
        elif opcion == "4":
            print(f"Total: ${sum(compras.values()):.2f}")
        elif opcion == "5":
            break
```

Recursos y próximos pasos

- **Documentación oficial**:
  - Python Docs - Documentación completa.
  - Tutorial oficial.
  - PEP 8 - Guía de estilo.
- **Recursos interactivos**:
  - Google Colab - Prácticas en la nube.
  - Replit - Código en línea y colaboración.
  - Codecademy, DataCamp - Cursos interactivos.
- **Próximo módulo**:
  - Programación Funcional y Orientada a Objetos en Python.
  - Creación de funciones y clases.
  - Conceptos de herencia y polimorfismo.

Tarea para la próxima clase

1. Crear un programa que:
   - Solicite datos para una lista de estudiantes (nombre, edad, calificación).
   - Almacene los datos en estructuras apropiadas.
   - Calcule estadísticas básicas (promedio, máximo, mínimo).
   - Muestre resultados con formato adecuado.
2. Practicar con ejercicios adicionales compartidos en Google Colab.
3. Leer sobre funciones en Python (próximo tema).

¡Gracias por su atención!
¿Preguntas?