Primitivas de Programación en Python

Variables, Expresiones y Estructuras de Control

Dr. Pierre Delice
Departamento de Innovación Tecnológica
Departamento de Ciencias de Datos
Universidad de Guadalajara
pierre.delice@academicos.udg.mx

9 de febrero de 2026

Contenido

Capítulo 1: Propósito y alcance

- Este bloque prepara el terreno para aprender a programar con Python:
  - Conceptos base: representación de información y arquitectura (modelo mental).
  - Qué es programar: algoritmos, lenguajes y pensamiento computacional.
  - Entorno de trabajo: Python, Jupyter, VS Code y herramientas.
  - Flujo de desarrollo y control de versiones con Git/GitHub.
- **Meta del curso (alto nivel):** conectar *conceptos* (modelo mental) con *práctica* (código reproducible).

Python: por qué es un lenguaje de alto nivel relevante

- Sintaxis legible y ecosistema amplio.
- Multiparadigma (imperativo, OO, funcional en práctica).
- Ideal para: prototipado, ciencia de datos, automatización, web, scripting.

Enfoque didáctico

Aprender Python aquí significa dominar: *vocabulario + estructura + razonamiento + depuración + reproducibilidad*.

Preparación: entorno Python, Jupyter y VS Code

- Recomendación práctica:
  - **Jupyter Notebook** para exploración interactiva y bitácora.
  - **VS Code** como IDE para proyectos (archivos, módulos, control de versiones).
- **Entornos virtuales**: aislar dependencias por proyecto.

Checklist

Python instalado → crear env → instalar paquetes → elegir intérprete en VS Code.

Entornos virtuales: conda y venv

Contexto: cada proyecto debe aislar dependencias para evitar conflictos entre paquetes.

Conda (recomendado si ya usas Anaconda/Miniconda)

Instalación: descarga Miniconda/Anaconda e instala.
`conda init bash` (una vez) → `source /.bashrc`
`conda create -n mi_env python=3.11`
`conda activate mi_env`
`python -m pip install paquete`

Python (venv, sin conda)

```
python -m venv .venv
source .venv/bin/activate
python -m pip install paquete
```

Windows (PowerShell / CMD)

Conda: `conda init powershell` → `conda activate mi_env`.
venv (PowerShell): `.venv\Scripts\Activate.ps1`
venv (CMD): `.venv\Scripts\activate.bat`

Tip VS Code

Selecciona el entorno en **Python: Select Interpreter** para usar el env activo.

Bitácora y reproducibilidad (notebooks)

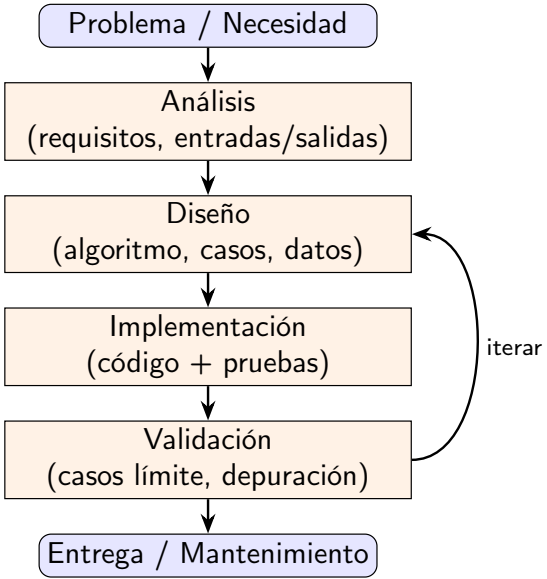
Contexto: un notebook es el cuaderno de laboratorio del curso: evidencia, pruebas y aprendizaje.

- Crea un notebook por capítulo/sección.
- Guarda:
  - Ejemplos ejecutados
  - Errores/observaciones
  - Variantes del código y pruebas
- Esto funciona como **evidencia** + **material de repaso**.

Resolución de problemas (flujo de desarrollo)

Por qué entender esta lámina

En programación real no basta con "que corra": este flujo te ayuda a descomponer el problema, diseñar antes de codificar, validar con casos límite y **iterar** cuando algo falla. Es el mapa para trabajar con orden, ahorrar tiempo y justificar tus decisiones.



Control de versiones: por qué Git es obligatorio

Contexto: Git funciona como "seguro" del proyecto y facilita el trabajo en equipo.

- En proyectos reales cambian archivos **todo el tiempo**.
- Sin control de versiones:
  - se pierden versiones buenas,
  - se rompe algo y no sabes qué cambió,
  - colaboración se vuelve caótica.
- **Git** registra historia y permite volver atrás.
- **GitHub** hospeda repositorios remotos (colaboración + respaldo).

Git mínimo viable (flujo local → remoto)

Contexto: flujo mínimo para publicar tu primer repositorio en GitHub.

```
# 1) Inicializar repo local
git init

# 2) Ver cambios
git status

# 3) Preparar (staging) y confirmar (commit)
git add .
git commit -m "primer avance"

# 4) Enlazar remoto y subir
git remote add origin https://github.com/usuario/repositorio.git
git push -u origin main
```

Git desde cero: crear carpeta local

Contexto: comienza desde una carpeta vacía y crea tu primer commit.

```
# 0) Crear carpeta y entrar
mkdir mi-proyecto
cd mi-proyecto

# 1) Inicializar repo
git init

# 2) Agregar primer archivo
echo "# Mi proyecto" > README.md
git add README.md
git commit -m "inicio"
```

Flujo diario (local ↔ remoto)

Contexto: rutina típica para sincronizar cambios y entregar avances.

```
# Traer cambios del remoto
git pull

# Trabajar... luego revisar y confirmar
git status
git add .
git commit -m "mensaje claro"

# Subir cambios
git push
```

Control de versiones: conceptos clave

Contexto: vocabulario esencial para leer documentación y colaborar en equipos.

- **Commit**: snapshot con mensaje; unidad de avance.
- **Staging**: seleccionar qué cambios entran al commit.
- **Branch**: líneas paralelas de trabajo.
- **Merge**: integrar cambios entre ramas.
- **Remote**: copia en servidor (GitHub/GitLab/Bitbucket).
- **Historial**: `git log` para auditar y volver atrás.

Referencias para estudiar Git

Contexto: usa estas fuentes para practicar y profundizar con ejemplos reales.

- **Pro Git** (Chacon & Straub): libro clásico y gratuito.
- **Git Docs**: manual oficial y comandos por tema.
- **GitHub Docs**: flujo con remotos, issues y PRs.
- **Atlassian Git Tutorials**: guías visuales y ejemplos.
- **Learn Git Branching**: práctica interactiva de ramas.

Representación y sistemas numéricos: qué cubrimos

- Bits y bytes: cómo se almacena la información.
- Bases numéricas (2, 8, 10, 16) y conversiones entre ellas.
- Enteros con y sin signo (rango, desbordamiento).
- Reales en punto flotante (aproximación, precisión).
- Codificaciones comunes: texto (ASCII/Unicode) y colores (RGB/hex).

Idea clave

La representación determina qué valores existen y cómo se interpretan.

¿Por qué enseñar sistemas numéricos en Python?

Idea central

Enseñar binario, hexadecimal y octal es clave para comprender cómo las computadoras almacenan, procesan y manipulan datos. Esto ayuda a optimizar memoria, depurar errores, trabajar con protocolos de red y criptografía, y entender la lógica de los tipos numéricos en Python.

- **Fundamentos de computación**: entender el sistema binario (0 y 1) y la conversión automática a decimal.
- **Representación de datos**: uso de notación `0b`, `0o`, `0x` en hardware, colores y direcciones.
- **Conversión y manipulación**: funciones integradas para cambiar de base y analizar datos.
- **Manejo de precisión**: reconocer límites de punto flotante y cuándo usar `decimal`.

Sistemas numéricos y representación (bases comunes)

Contexto: distintas bases se usan según el problema (hardware, colores, direcciones).

Base	Nombre	Dígitos	Uso típico
2	Binario	0–1	Representación interna (bits)
8	Octal	0–7	Sistemas legacy / compactación
10	Decimal	0–9	Interacción humana
16	Hexadecimal	0–9, A–F	Direcciones, colores, dumps

Mini-demo en Python

```
bin(78), hex(31), int("1F", 16)
```

Ejemplos: conversiones rápidas en Python

Contexto: estas funciones permiten verificar conversiones manuales en segundos.

```
# De decimal a binario/octal/hex
n = 78
print(bin(n))    # '0b1001110'
print(oct(n))    # '0o116'
print(hex(n))    # '0x4e'

# De string en base b a decimal
print(int("1F", 16))   # 31
print(int("27", 8))    # 23
```

Ejercicio rápido

Convierte $(121)_{10}$, $(3E)_{16}$, $(33)_8$ a binario y verifica con `int(...)`.

Capítulo 2: Mapa de contenidos

Este capítulo cubre el **vocabulario** y la **gramática** básica de Python:

- Identificadores, palabras reservadas y reglas de nombrado.
- Tipos de datos: simples y compuestos.
- Variables/constantes.
- Operadores y expresiones.
- Estructura de programas, comentarios y estilo.
- Sentencias simples y compuestas; bloques e indentación.

Referencias a Notebooks

Introducción: `01.intro-to-python/00_intro/00_content.ipynb`
Elementos básicos: `01.intro-to-python/01_elements/00_content.ipynb`
Variables: `00-Python Object.../01-Variable Assignment.ipynb`

Vocabulario: identificadores y palabras reservadas

Contexto: nombrar bien reduce errores y mejora la comunicación del código.

- **Identificador**: nombre que damos a variables, funciones, clases, etc.
- Reglas típicas:
  - Inicia con letra o _, luego letras/dígitos/_
  - **Case-sensitive** (`A1` $\neq$ `a1`)
  - No puede ser palabra reservada

```
1 # validos
2 mi_variable = 10
3 _total = 0
4 NombreCompleto = "Ana Ruiz"
5
6 # invalidos (ejemplos)
7 # 3da = 5        # no puede iniciar con digito
8 # for = 7        # palabra reservada
```

Convenciones de nombrado (comunidad Python)

Contexto: no es obligatorio, pero facilita lectura y trabajo en equipo.

- **snake_case** para variables y funciones: `total_ventas`, `calcular_area()`
- **CamelCase** para clases: `MiClase`
- Constantes (convención): **MAYÚSCULAS**: `PI = 3.14159`
- Subrayados:
  - `_nombre`: "uso interno"
  - `nombre_`: evitar colisión con palabras reservadas
  - `__nombre`: *name mangling* (clases)

Alcance (scope) y resolución de nombres

- **Scope** determina dónde "vive" un nombre:
  - local (función), global (módulo), enclosing (función dentro de función), built-in
- Idea clave: un nombre se resuelve buscando en niveles (de adentro hacia afuera).

Ejercicio

Escribe un ejemplo donde una variable local "oculte" a una global (shadowing). Explica el resultado.

Estructura de programas en Python

Contexto: pasamos de scripts simples a proyectos organizados y mantenibles.

- Un programa puede ser:
  - una secuencia de sentencias (script),
  - un módulo con funciones/clases,
  - un paquete con múltiples módulos.
- Práctica recomendada (curso alto nivel):
  - separar lógica (funciones) de ejecución (`if __name__ == "__main__":`)
  - entradas/salidas controladas

Documentación: comentarios vs docstrings

Contexto: documentar el propósito evita ambigüedades al reutilizar código.

```
1 def area_circulo(r):
2     """Devuelve el area de un circulo de radio r."""
3     # Nota: r debe ser >= 0
4     import math
5     return math.pi * r**2
```

- **Docstring**: describe propósito/uso de función/clase/módulo.
- **Comentario**: explica decisiones locales (el "por qué").

Estilo (PEP 8 como referencia mínima)

- Indentación: **4 espacios** (evita tabs mezcladas).
- Longitud de línea: guía típica **79 caracteres**.
- Espaciado consistente alrededor de operadores.
- Nombres descriptivos y consistentes.

Ejercicio

Toma un fragmento de tu código y "normalízalo" a PEP 8 (nombres, espacios, líneas).

Sentencias simples (catálogo)

Contexto: son las instrucciones más básicas que Python ejecuta línea por línea. Ejemplos de sentencias simples frecuentes:

- **Expresión** (p.ej. llamada a función): `print("hola")`
- **Asignación**: `x = 3`, `x += 1`
- **Entrada/Salida**: `input(...)` / `print(...)`
- Control: `assert`, `pass`, `del`, `return`, `raise`
- Bucles: `break`, `continue`
- Módulos: `import`
- Alcance: `global`, `nonlocal`

Sentencias compuestas (bloques)

Sentencias compuestas crean **bloques de código**:

- Selección: `if`, `if-else`, `if-elif-else`
- Iteración: `while`, `for`
- Abstracción: `def`, `class`
- Excepciones: `try-except`
- Contextos: `with`

Conexión con el material

Las secciones de condicionales y bucles aparecen después como práctica guiada.

Bloques, indentación y espaciado (reglas prácticas)

- Un bloque se define por **dos puntos** : + **indentación**.
- Reglas mínimas:
  - Indenta consistentemente (4 espacios recomendado).
  - No mezcles tabs y espacios.
  - Mantén alineación clara en bloques anidados.

Ejercicio

Indenta a propósito mal un bloque `if/for` y explica el error que arroja Python.

Tipos de datos simples (visión rápida)

Contexto: estos tipos son la base de casi todo cálculo.

Tipo	Ejemplo	Notas
`int`	`42, -7`	enteros arbitrariamente grandes
`float`	`3.14`	punto flotante
`bool`	`True/False`	lógico
`complex`	`3+4j`	parte real + imaginaria

Ejercicio

Crea variables de cada tipo y verifica con `type()`.

Tipos compuestos (estructuras base)

Contexto: permiten modelar colecciones y relaciones reales.

Tipo	Ejemplo	Acceso / propiedad
`str`	`"hola"`	indexable, inmutable
`list`	`[1,2,3]`	indexable, mutable
`tuple`	`(1,2)`	indexable, inmutable
`set`	`{1,2}`	no indexable, sin orden
`dict`	`{"k": 1}`	clave $\rightarrow$ valor

Ejercicio

Da un ejemplo real para cada estructura (catálogo, registro, colección única, secuencia ordenada, etc.).

Variables y constantes

Contexto: nombrar valores hace el código legible y mantenible.

- Variable: nombre $\rightarrow$ referencia a un objeto/valor.
- Constantes: Python no impone constantes reales; se usan **convenciones**: $\texttt{NOMBRE}_{C}ONSTANTE = ...$
- `Constantes integradas comunes: True, False, None.`

Ejercicio

Define una constante $\texttt{TASA}_{I}VAysalaenunclculo(sinmagicalnumbers_{J})$.

Operadores: mapa conceptual

Contexto: combinan valores para producir nuevos resultados.

- Aritméticos: `+ - * / // % **`
- Comparación: `== != < > <= >=`
- Lógicos: `and or not`
- Bitwise: `& | ^ ~ « »`
- Asignación: `=, +=, -=, *=, ...`
- Identidad: `is`, `is not`
- Secuencia: concatenación (+), repetición (*)
- Pertenencia: `in`, `not in`

Funciones integradas útiles (built-ins)

Contexto: Python trae herramientas listas para tareas comunes.

```
# Conversion y tipos
int("123"), float("3.14"), str(42), bool(0)
type(3.14)

# Numericas
abs(-7), round(78.3), pow(2, 3), divmod(13, 5)

# Secuencias
len("hola"), list("abc"), range(5)

# Representacion
bin(9), oct(9), hex(255)
```

Ejercicio

Evalúa: `sum([1,3,5,7,9])/13` y explica el tipo del resultado.

Expresiones: composición correcta

Contexto: una buena expresión evita errores de lógica y mejora la claridad.

- Una expresión combina: **valores** + **variables** + **operadores** + **llamadas a funciones**.
- Reglas clave:
  - Precedencia y asociatividad (usa paréntesis si hay duda).
  - Tipos compatibles (p.ej. `."a"1` falla).
  - Evita "código críptico": claridad > "ingenio".

Ejercicio

Evalúa mentalmente: `23 + 16 / 2` y luego verifica en Python. ¿Por qué no da 19?

¿Qué son las primitivas de programación?

- Las **primitivas** son los elementos básicos y fundamentales de un lenguaje de programación.
- Constituyen los **bloques de construcción** con los que se escriben todos los programas.
- En Python, las primitivas incluyen:
  - Variables y tipos de datos
  - Expresiones aritméticas y lógicas
  - Operadores de asignación
  - Estructuras de control (condicionales e iterativas)
  - Funciones y recursión

Objetivo: Dominar estas primitivas para construir soluciones computacionales efectivas.

Variables: Contenedores de datos

Contexto: en Python el tipo lo define el valor, no una declaración previa.

- Una **variable** es un nombre simbólico asociado a un valor almacenado en memoria.
- En Python, las variables se crean mediante **asignación**.
- No requieren declaración de tipo (tipado dinámico).

```
# Asignacion de variables
nombre = "Ana"           # str (cadena)
edad = 25                # int (entero)
altura = 1.68            # float (punto flotante)
es_estudiante = True     # bool (booleano)
```

Reglas para nombres de variables:

- Deben comenzar con letra o guion bajo (_)
- Solo letras, números y guiones bajos
- Case-sensitive: edad $\neq$ Edad

Tipos de datos básicos en Python

Contexto: reconocer tipos ayuda a predecir comportamientos del programa.

Tipos numéricos:

```python
# Enteros (int)
x = 42
y = -17

# Punto flotante (float)
pi = 3.14159
temperatura = -5.2

# Complejos (complex)
z = 3 + 4j
```

Otros tipos básicos:

```python
# Cadenas (str)
mensaje = "Hola mundo"
inicial = 'A'

# Booleanos (bool)
activo = True
terminado = False

# None (ausencia de valor)
resultado = None
```

Función `type()`**:** determina el tipo de una variable

```
>>> type(42)
<class 'int'>
```

Notebooks de referencia

Conversión entre tipos (type casting)

Contexto: necesario para entradas del usuario y formatos de datos.

- Python permite convertir explícitamente entre tipos de datos.
- Funciones: `int()`, `float()`, `str()`, `bool()`

```
# Conversion de tipos
x = "123"              # String
y = int(x)             # Convierte a entero: 123
z = float(x)           # Convierte a flotante: 123.0

a = 3.14
b = int(a)             # Trunca a entero: 3

c = str(42)            # Convierte numero a string: "42"

# Entrada del usuario (siempre es string)
edad_str = input("Ingresa tu edad: ")
edad = int(edad_str)  # Convierte a entero
```

Expresiones aritméticas

Contexto: se aplican en cálculos y fórmulas cotidianas.

- Una **expresión** combina valores y operadores para producir un resultado.
- Python soporta operadores aritméticos estándar.

Operadores básicos:

```
a = 10
b = 3

suma = a + b          # 13
resta = a - b         # 7
producto = a * b      # 30
division = a / b      # 3.333...
```

Operadores especiales:

```
# Division entera
cociente = a // b    # 3

# Modulo (resto)
resto = a % b        # 1

# Potencia
potencia = a ** b    # 1000
```

Precedencia: sigue reglas matemáticas (PEMDAS)
Paréntesis → Exponentes → Multiplicación/División → Suma/Resta

Expresiones lógicas y operadores de comparación

Contexto: permiten tomar decisiones usando condiciones. **Operadores de comparación:** producen valores booleanos (`True` o `False`)

```
x = 10
y = 20

x == y       # False  (igual a)
x != y       # True   (diferente de)
x < y        # True   (menor que)
x > y        # False  (mayor que)
x <= y       # True   (menor o igual que)
x >= y       # False  (mayor o igual que)
```

Operadores lógicos: combinan expresiones booleanas

```
edad = 25
tiene_licencia = True

# and: ambas condiciones deben ser verdaderas
puede_conducir = (edad >= 18) and tiene_licencia  # True

```

Asignación y operadores de asignación compuesta

Contexto: actualiza estado con expresiones breves y legibles. **Asignación simple:**

```
x = 10          # Asigna el valor 10 a x
```

Asignaciones múltiples:

```
a, b, c = 1, 2, 3      # Asigna valores simultaneamente
x = y = z = 0          # Asigna el mismo valor
```

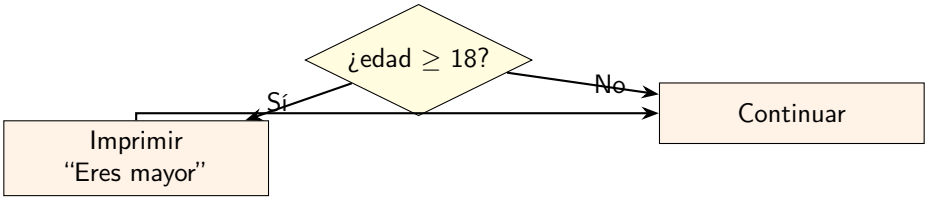
Operadores de asignación compuesta:

```
contador = 10
contador += 5      # contador = contador + 5   (15)
contador -= 3      # contador = contador - 3   (12)
contador *= 2      # contador = contador * 2   (24)
contador /= 4      # contador = contador / 4   (6.0)
contador //= 2     # Division entera           (3.0)
contador %= 2      # Modulo                    (1.0)
contador **= 3     # Potencia                  (1.0)
```

Estructura `if`: ejecución condicional

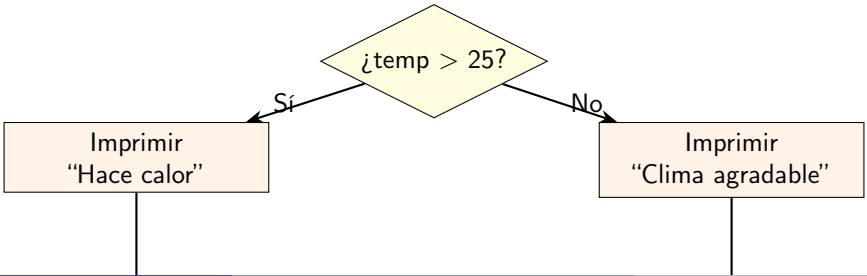
- La estructura `if` permite ejecutar código solo si se cumple una condición.
- **Indentación** define los bloques de código.

```python
edad = 18

if edad >= 18:
    print("Eres mayor de edad")
    print("Puedes votar")

print("Este mensaje siempre se muestra")
```



Estructura `if-else`: alternativa doble

```
1 temperatura = 22
2
3 if temperatura > 25:
4     print("Hace calor")
5     print("Usa ropa ligera")
6 else:
7     print("El clima es agradable")
8     print("Disfruta el dia")
```



Estructura if-elif-else: alternativas múltiples

```
1  calificacion = 85
2
3  if calificacion >= 90:
4      print("Excelente")
5      nivel = "A"
6  elif calificacion >= 80:
7      print("Muy bien")
8      nivel = "B"
9  elif calificacion >= 70:
10     print("Bien")
11     nivel = "C"
12 elif calificacion >= 60:
13     print("Suficiente")
14     nivel = "D"
15 else:
16     print("Insuficiente")
17     nivel = "F"
18
```

Bucle `while`: iteración condicional

- El bucle `while` repite un bloque mientras una condición sea verdadera.
- Se evalúa la condición **antes** de cada iteración.

```
contador = 1

while contador <= 5:
    print(f"Contador: {contador}")
    contador += 1

print("Bucle terminado")
```

Salida:

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
```

Bucle `for`: iteración sobre secuencias

- El bucle `for` itera sobre elementos de una secuencia.
- Más común cuando se conoce el número de iteraciones.

Usando `range()`:

```
1 for i in range(5):
2     print(f"Iteracion: {i}")
```

Iterando sobre una lista:

```
1 frutas = ["manzana", "banana", "naranja"]
2
3 for fruta in frutas:
4     print(f"Me gusta la {fruta}")
```

Control de bucles: break y continue

break: sale inmediatamente del bucle

```
numeros = [3, 7, 2, 9, 5]
objetivo = 9

for num in numeros:
    if num == objetivo:
        print(f"Encontrado: {num}")
        break
    print(f"Revisando: {num}")
```

continue: salta a la siguiente iteración

```
# Imprimir solo numeros impares
for i in range(10):
    if i % 2 == 0:   # Si es par
        continue     # Saltar
    print(i)         # Solo impares
```

Funciones: modularización del código

Contexto: encapsulan pasos para reutilizar y probar más fácil.

- Una **función** es un bloque de código reutilizable.
- Permiten **descomponer** problemas complejos.
- Facilitan la **reutilización** y el **mantenimiento**.

Sintaxis básica:

```
def nombre_funcion(parametros):
    """Documentacion (docstring)"""
    # Instrucciones
    return resultado
```

Ejemplo:

```
def cuadrado(x):
    """Calcula el cuadrado de un numero"""
    return x ** 2

resultado = cuadrado(5)
print(resultado)  # 25
```

Recursión: funciones que se llaman a sí mismas

Contexto: útil en problemas repetitivos, pero exige un caso base claro.

- La **recursión** es cuando una función se invoca a sí misma.
- Útil para problemas divisibles en subproblemas similares.
- Requiere un **caso base** para evitar recursión infinita.

Ejemplo: Factorial

```python
def factorial(n):
    """Calcula n! recursivamente"""
    if n == 0 or n == 1:
        return 1                # Caso base
    else:
        return n * factorial(n - 1)  # Caso recursivo

print(factorial(5))  # 120
```

Estructura General del Curso

Objetivo: Presentar un curso de Python con hilo conductor coherente, eliminando duplicaciones y organizando por complejidad creciente.

Módulos principales:

1. Introducción y Setup
2. Elementos Básicos (variables, tipos)
3. Números y Operaciones
4. Strings y Formato
5. Operadores y Expresiones
6. Estructuras de Control (condicionales e iteración)
7. Estructuras de Datos (listas, tuplas, sets, diccionarios)
8. Funciones
9. Funciones Avanzadas y Programación Funcional
10. Orientación a Objetos (Clases)

Módulo 1: Introducción y Setup

Objetivos: Configurar entorno, entender notebooks, primeros pasos en Python.

Notebooks de referencia:

- `01.intro-to-python/00_intro/00_content.ipynb` — Introducción general
- `01.intro-to-python/00_intro/01_exercises_markdown.ipynb` — Ejercicios de Markdown
- `01.intro-to-python/00_intro/02_review.ipynb` — Revisión

Contenidos clave:

- Instalación de Python y Jupyter
- Uso de notebooks
- Primera ejecución de código
- Control de versiones con Git

Módulo 2: Elementos Básicos

Objetivos: Variables, asignación, tipos básicos, primeras expresiones.

Notebooks de referencia:

- `00-Python Object.../01-Variable Assignment.ipynb` — Asignación de variables
- `01.intro-to-python/01_elements/00_content.ipynb` — Elementos básicos
- `01.intro-to-python/01_elements/01_exercises_print.ipynb` — Ejercicios de print
- `01.intro-to-python/01_elements/03_content.ipynb` — Contenido adicional
- `01.intro-to-python/01_elements/04_exercises_calculator.ipynb` — Calculadora básica
- `01.intro-to-python/01_elements/05_summary.ipynb` — Resumen

Contenidos clave:

- Identificadores y convenciones
- Variables y asignación
- Tipos básicos: int, float, str, bool
- Función `type()`, conversiones

Módulo 3: Números y Operaciones

Objetivos: Operaciones aritméticas, sistemas numéricos, conversiones.

Notebooks de referencia (consolidados):

- `00-Python Object.../01-Numbers.ipynb` — Números básicos
- `01.intro-to-python/05_numbers/00_content.ipynb` — Sistemas numéricos
- `01.intro-to-python/05_numbers/01_content.ipynb` — Operaciones
- `01.intro-to-python/05_numbers/02_content.ipynb` — Precisión y tipos
- `01.intro-to-python/05_numbers/03_appendix.ipynb` — Apéndice
- `01.intro-to-python/05_numbers/04_summary.ipynb` — Resumen

Contenidos clave:

- Operadores aritméticos: +, -, *, /, //, %, **
- Bases numéricas: binario, octal, hexadecimal
- Funciones: `abs()`, `round()`, `pow()`, `divmod()`
- Punto flotante y precisión

Módulo 4: Strings y Formato

Objetivos: Manipulación de cadenas, formato, métodos de strings.

Notebooks de referencia (consolidados):

- `00-Python Object.../02-Strings.ipynb` — Strings básicos
- `00-Python Object.../03-Print Formatting with Strings.ipynb` — Formato
- `01.intro-to-python/06_text/00_content.ipynb` — Texto y strings
- `01.intro-to-python/06_text/01_exercises_palindromes.ipynb` — Ejercicio palíndromos
- `01.intro-to-python/06_text/02_content.ipynb` — Métodos de strings
- `01.intro-to-python/06_text/03_summary.ipynb` — Resumen

Contenidos clave:

- Indexación y slicing
- Métodos: `upper()`, `lower()`, `split()`, `join()`, etc.
- Formato con f-strings, `format()`, `%`
- Inmutabilidad de strings

Módulo 5: Operadores y Expresiones

Objetivos: Operadores de comparación, lógicos, precedencia, expresiones complejas.

Notebooks de referencia:

- `01-Python Comparison Operators/01-Comparison Operators.ipynb`
- `01-Python Comparison Operators/02-Chained Comparison Operators.ipynb`

Contenidos clave:

- Operadores de comparación: ==, !=, <, >, <=, >=
- Operadores lógicos: `and`, `or`, `not`
- Comparaciones encadenadas: `a < b < c`
- Operadores de identidad: `is`, `is not`
- Operadores de pertenencia: `in`, `not in`
- Precedencia y asociatividad

Módulo 6A: Estructuras de Control — Condicionales

Objetivos: Toma de decisiones con if/elif/else.

Notebooks de referencia (consolidados):

- `02-Python Statements/01-Introduction to Python Statements.ipynb`
- `02-Python Statements/02-if, elif, and else Statements.ipynb`
- `01.intro-to-python/03_conditionals/00_content.ipynb`
- `01.intro-to-python/03_conditionals/01_exercises_discounts.ipynb`
- `01.intro-to-python/03_conditionals/02_exercises_fizz-buzz.ipynb`
- `01.intro-to-python/03_conditionals/03_summary.ipynb`

Contenidos clave:

- Estructura `if`
- Estructura `if-else`
- Estructura `if-elif-else`
- Indentación y bloques
- Ejercicios: FizzBuzz, descuentos

Módulo 6B: Estructuras de Control — Iteración

Objetivos: Bucles for y while, control de flujo con break/continue.

Notebooks de referencia (consolidados):

- `02-Python Statements/03-for Loops.ipynb`
- `02-Python Statements/04-while Loops.ipynb`
- `02-Python Statements/05-Useful-Operators.ipynb`
- `01.intro-to-python/04_iteration/00_content.ipynb` — Iteración básica
- `01.intro-to-python/04_iteration/01_exercises_hanoi-towers.ipynb` — Torres de Hanoi
- `01.intro-to-python/04_iteration/02_content.ipynb` — While
- `01.intro-to-python/04_iteration/03_content.ipynb` — For avanzado
- `01.intro-to-python/04_iteration/04_exercises_dice.ipynb` — Simulación dados

Contenidos clave:

- Bucle `for` con `range()`
- Bucle `while`

Módulo 7: Estructuras de Datos — Listas

Objetivos: Crear, manipular y recorrer listas.

Notebooks de referencia (consolidados):

- `00-Python Object.../04-Lists.ipynb` — Listas básicas
- `02-Python Statements/06-List Comprehensions.ipynb` — Comprensiones
- `01.intro-to-python/07_sequences/00_content.ipynb` — Secuencias intro
- `01.intro-to-python/07_sequences/01_content.ipynb` — Listas detalladas
- `01.intro-to-python/07_sequences/02_exercises_lists.ipynb` — Ejercicios
- `01.intro-to-python/07_sequences/03_content.ipynb` — Tuplas y unpacking
- `01.intro-to-python/07_sequences/04_exercises_un-packing.ipynb`

Contenidos clave:

- Creación e indexación
- Métodos: `append()`, `extend()`, `insert()`, `remove()`, `pop()`
- Slicing y mutabilidad
- List comprehensions

Módulo 8: Estructuras de Datos — Tuplas, Sets y Booleans

Objetivos: Entender estructuras inmutables (tuplas) y conjuntos (sets).

Notebooks de referencia:

- `00-Python Object.../06-Tuples.ipynb`
- `00-Python Object.../07-Sets and Booleans.ipynb`
- `01.intro-to-python/07_sequences/03_content.ipynb` — Tuplas
- `01.intro-to-python/07_sequences/05_appendix.ipynb`
- `01.intro-to-python/07_sequences/06_summary.ipynb`

Contenidos clave:

- Tuplas: inmutabilidad, indexación
- Unpacking de tuplas
- Sets: operaciones de conjunto (unión, intersección, diferencia)
- Valores booleanos y operaciones lógicas

Módulo 9: Estructuras de Datos — Diccionarios

Objetivos: Mapeos clave-valor, acceso y manipulación.

Notebooks de referencia (consolidados):

- `00-Python Object.../05-Dictionaries.ipynb`
- `01.intro-to-python/09_mappings/00_content.ipynb`
- `01.intro-to-python/09_mappings/01_exercises_nested-data.ipynb`
- `01.intro-to-python/09_mappings/02_content.ipynb`
- `01.intro-to-python/09_mappings/03_exercises_fibonacci.ipynb` — Fibonacci con memo
- `01.intro-to-python/09_mappings/04_content.ipynb`
- `01.intro-to-python/09_mappings/06_summary.ipynb`

Contenidos clave:

- Creación y acceso: `dict[key]`
- Métodos: `keys()`, `values()`, `items()`, `get()`, `update()`
- Diccionarios anidados
- Aplicaciones: memoización, conteo

Módulo 10: Archivos

Objetivos: Lectura y escritura de archivos de texto.

Notebooks de referencia:

- `00-Python Object.../08-Files.ipynb`

Contenidos clave:

- Apertura de archivos: `open()`, modos (r, w, a)
- Lectura: `read()`, `readline()`, `readlines()`
- Escritura: `write()`, `writelines()`
- Context managers: `with open(...) as f:`
- Manejo de rutas y errores

Módulo 11: Funciones

Objetivos: Definir funciones, parámetros, retorno, documentación.

Notebooks de referencia (consolidados):

- `01.intro-to-python/02_functions/00_content.ipynb`
- `01.intro-to-python/02_functions/01_exercises_sphere-volume.ipynb`
- `01.intro-to-python/02_functions/02_content.ipynb`
- `01.intro-to-python/02_functions/03_summary.ipynb`
- `03-Methods and Functions/01-Methods.ipynb`
- `03-Methods and Functions/02-Functions.ipynb`
- `03-Methods and Functions/03-Function Practice Exercises.ipynb`
- `03-Methods and Functions/04-Function Practice Exercises - Solutions.ipynb`

Contenidos clave:

- Definición: `def nombre(parametros):`
- Parámetros y argumentos
- Return: valores de retorno

Módulo 12: Funciones Avanzadas

Objetivos: Lambda, map, filter, reduce, scope, args/kwargs.

Notebooks de referencia:

- `03-Methods and Functions/05-Lambda-Expressions-Map-and-Filter.ipynb`
- `03-Methods and Functions/06-Nested Statements and Scope.ipynb`
- `03-Methods and Functions/07-args and kwargs.ipynb`
- `01.intro-to-python/08_mfr/00_content.ipynb` — Map, Filter, Reduce
- `01.intro-to-python/08_mfr/01_content.ipynb`
- `01.intro-to-python/08_mfr/02_exercises_outliers.ipynb`
- `01.intro-to-python/08_mfr/04_content.ipynb`
- `01.intro-to-python/08_mfr/05_summary.ipynb`

Contenidos clave:

- Funciones lambda: `lambda x: x**2`
- Map, filter, reduce
- Scope y resolución de nombres (LEGB)

Módulo 13: Orientación a Objetos (Clases)

Objetivos: Introducción a OOP, definición de clases, atributos y métodos.

Notebooks de referencia:

- `01.intro-to-python/11_classes/00_content.ipynb`
- `01.intro-to-python/11_classes/01_exercises_tsp.ipynb` — TSP
- `01.intro-to-python/11_classes/02_content.ipynb`
- `01.intro-to-python/11_classes/03_content.ipynb`
- `01.intro-to-python/11_classes/04_content.ipynb`
- `01.intro-to-python/11_classes/05_summary.ipynb`

Contenidos clave:

- Definición de clases: `class MiClase:`
- Constructor: `__init__()`
- Atributos de instancia y de clase
- Métodos de instancia
- Herencia básica

Evaluaciones y Proyectos

Notebooks de evaluación disponibles:

Estructuras de Datos:

- `00-Python Object.../09-Objects and Data Structures Assessment Test.ipynb`
- `00-Python Object.../10-...-Solution.ipynb`

Statements:

- `02-Python Statements/07-Statements Assessment Test.ipynb`
- `02-Python Statements/08-Statements Assessment Test - Solutions.ipynb`
- `02-Python Statements/09-Guessing Game Challenge.ipynb`
- `02-Python Statements/10-Guessing Game Challenge - Solution.ipynb`

Funciones:

- `03-Methods and Functions/08-Functions and Methods Homework.ipynb`
- `03-Methods and Functions/09-Functions and Methods Homework - Solutions.ipynb`

Ejercicios (entorno + notebooks)

Contexto: aseguran que tu entorno funciona y que documentas el proceso.

1. Crea carpeta del curso y un notebook `comp218start.ipynb` en VS Code.
2. Úsalo como calculadora: prueba expresiones y observa tipos.
3. Ejecuta:
   - concatenación de nombre completo (`first_name`, `last_name`)
   - potencias de 2 con `for i in range(11): p = 2**i; print(bin(p))`

Ejercicios (algoritmos)

1. Escribe pseudocódigo: pasos para hacer una pizza.
2. Escribe pseudocódigo: pasos para cocinar arroz.
3. Dibuja un diagrama de flujo: rutina de la mañana (de levantarte a salida).

Criterio de calidad

Entradas/salidas claras, pasos finitos, casos alternativos (decisiones).

Ejercicios (representación y operaciones binarias)

1. Convierte a binario: $(78)_{10}$, $(1F)_{16}$, $(27)_{8}$, $(121)_{10}$, $(3E)_{16}$, $(33)_{8}$.
2. Completa operaciones binarias: `10111 + 1101`, `1101010 - 101101`, `10101 + 1110`, etc.
3. (Refuerzo) Verifica conversiones con `bin()`, `int(..., base)` en Python.

Ejercicios (identificadores y sentencias)

1. Señala identificadores inválidos y explica por qué: `This`, `3da`, `My_name`, `for`, `i9`, `vote`, `$s`, `_sum_`, `cLearance`, `t5#`
2. Escribe **una sola sentencia** para:
   - leer un entero del usuario en `k`
   - imprimir una etiqueta postal multilínea con tu nombre y dirección
   - imprimir el área de un círculo de radio 13
   - asignar el cubo de 23 a `x`
   - imprimir los cuadrados de 12, 25 y 56 en una línea

Ejercicios (expresiones y trazado mental)

1. Evalúa expresiones: `23 + 16 / 2`, `round(78.3)`, `pow(2,3)+5`, `sum([1,3,5,7,9])/13`, `bin(9)`, `divmod(13,5)`, `int(38.6)//7`, `(3.5+6.7j)+(5.3+12.9j)`, `'Well'*3+'!'`
2. Ejecuta mentalmente y predice salida:
   - `x=5; y=6; print(x+y)`
   - `m=5; k=3; print(f"{m}**{k} = {m**k}")`
   - `m,k=35,12; m//=k; print(m)`
   - `m,k=35,12; m%=k; print(m)`

Ejercicios (programas completos)

1. Lee un `float s` y calcula el área de un cuadrado de lado `s`.
2. Lee dos números y calcula el producto.
3. Estacionamiento: \$2.50 por hora. Lee horas y calcula total.
4. Lee tres números y calcula la media aritmética.
5. Cubo: dado el lado, calcula el área total de superficie.

Entrega sugerida

Repositorio GitHub + README (cómo ejecutar) + pruebas rápidas (3 casos por programa).

Mini-proyecto: registro de gastos

Objetivo: construir un programa de consola que ayude a registrar gastos e ingresos.

- Menú con opciones (`while`): agregar, enlistar, resumen, salida.
- Cada movimiento: `tipo` (gasto/ingreso), `categoria`, `monto`.
- Usa **lista de diccionarios** para almacenar movimientos.
- Implementa funciones: `agregar()`, `listar()`, `resumen()`.
- Valida entradas: números positivos y tipos válidos.

Salida mínima esperada

Totales por tipo y balance final del día/semana.

Ejercicios (variables y casting)

1. Lee un entero `n` desde teclado e imprime:
   - `n` en binario, octal y hexadecimal
   - el tipo de `n` y el tipo de `str(n)`
2. Dado `x = "3.14"`:
   - convierte a `float` y luego a `int`
   - explica por qué el resultado entero cambia (truncamiento)

Ejercicios (condicionales y bucles)

1. Escribe un programa que lea `edad` y muestre:
   - `"Mayor de edad"` si `edad >= 18`, en otro caso `"Menor de edad"`.
2. Escribe un programa que imprima los múltiplos de 3 entre 1 y 50 (usa `for` + `continue`).
3. Escribe un programa que encuentre el primer número >=1 cuyo cuadrado sea mayor a 200 (usa `while` + `break`).

Ejercicios (funciones y recursión)

1. Implementa `es_par(n)` que devuelva `True` si `n` es par.
2. Implementa `max_de_tres(a,b,c)` sin usar `max()`.
3. (Recursión) Implementa `fibonacci(n)` con caso base y explica su complejidad.

Resumen

Primitivas de Programación en Python:

- Variables y tipos de datos (int, float, str, bool)
- Expresiones aritméticas y lógicas
- Operadores de asignación
- Estructuras condicionales (if-elif-else)
- Estructuras iterativas (while, for)
- Descomposición funcional
- Recursión

"Code is read more often than it is written" — Guido van Rossum

¿Preguntas?

pierre.delice@academicos.udg.mx

Universidad de Guadalajara