PROGRAMACIÓN ORIENTADA A OBJETOS Y ESTRUCTURAS DE DATOS EN PYTHON

Luis Miguel Barillas Del Cid -7690-20-12345
Iván Emmanuel Alvarez Villatoro carnet 7690-25-6916

Universidad Mariano Gálvez

RESUMEN
Este documento presenta una investigación ampliada sobre los fundamentos de la Programación Orientada a Objetos (OOP) y las estructuras de datos básicas en Python, con un enfoque práctico para su aplicación en programas de línea de comandos. Se estudian con detalle las clases, objetos, métodos, atributos, encapsulamiento y herencia; además de listas, diccionarios, tuplas y conjuntos, incluyendo sus operaciones clave (agregar, eliminar, buscar). Se explican funciones, modularidad y el ámbito de variables, incorporando ejemplos de código comentados y buenas prácticas (docstrings, validación, propiedades y composición). El contenido está orientado a soportar la segunda parte del proyecto (desarrollo) ofreciendo bases sólidas para diseñar menús, validar entradas y organizar módulos. Como aportes, se integran notas sobre errores comunes, recomendaciones de estilo, y una síntesis de complejidad computacional para operaciones frecuentes. La investigación concluye que combinar OOP con estructuras nativas de Python promueve software mantenible y escalable, y facilita la implementación de estructuras jerárquicas como árboles con raíz sin recurrir a librerías externas.
Palabras claves: orientación a objetos, clases y objetos, encapsulamiento, herencia, listas y diccionarios, funciones y ámbito, Python

DESARROLLO DEL TEMA

1. Conceptos básicos de Programación Orientada a Objetos (OOP)

1.1 Clases y objetos: qué son, cómo crear e interactuar

Una clase define la estructura (atributos) y el comportamiento (métodos) de un conjunto de objetos. Un objeto es una instancia concreta de esa clase. En Python, la interacción se realiza mediante el parámetro self y el operador punto (.).

```
class Persona:
    def __init__(self, nombre: str, edad: int):
        self.nombre = nombre          # atributo de instancia
        self.edad = edad

    def saludar(self) -> str:
        return f"Hola, soy {self.nombre} y tengo {self.edad} años."

p1 = Persona("Luis", 18)
print(p1.saludar())              # Interacción: invocar método
p1.edad += 1                     # Interacción: modificar atributo
```

1.2 Métodos: funciones dentro de las clases

Los métodos operan sobre los atributos del objeto. Existen métodos de instancia (más comunes), de clase (decorador @classmethod) y estáticos (@staticmethod).

```
class Calculadora:
    factor_global = 1.0  # atributo de clase

    def __init__(self, a: float, b: float):
        self.a = a
        self.b = b

    def sumar(self) -> float:          # método de instancia
        return (self.a + self.b) * self.factor_global

    @classmethod
    def set_factor(cls, valor: float): # método de clase
        cls.factor_global = valor

    @staticmethod
    def es_numero(x) -> bool:          # método estático
        return isinstance(x, (int, float))

Calculadora.set_factor(1.2)
c = Calculadora(10, 5)
print(c.sumar())                        # 18.0
```

1.3 Atributos: de instancia y de clase

Los atributos de instancia pertenecen a cada objeto; los de clase son compartidos por todas las instancias. Es buena práctica documentarlos en __init__ y usar nombres claros.

```
class Auto:
    ruedas = 4                          # atributo de clase compartido
    def __init__(self, marca: str, modelo: str):
        self.marca = marca              # atributos de instancia
        self.modelo = modelo
```

1.4 Encapsulamiento: mantener datos privados con getters/setters

En Python se usa el convenio de nombre con doble guion bajo (__saldo) para denotar atributos privados. Se recomiendan propiedades (@property) para exponer acceso controlado.

```
class CuentaBancaria:
    def __init__(self, saldo_inicial: float = 0.0):
        self.__saldo = max(0.0, saldo_inicial)

    @property
    def saldo(self) -> float:
        return self.__saldo

    @saldo.setter
    def saldo(self, cantidad: float):
        if cantidad < 0:
            raise ValueError("El saldo no puede ser negativo.")
        self.__saldo = cantidad

cuenta = CuentaBancaria(100)
cuenta.saldo += 50
print(cuenta.saldo)
```

1.5 Herencia: reutilización y especialización

La herencia permite crear subclases que amplían o redefinen comportamientos de una clase base. Evita duplicación e incrementa la cohesión. También existe la composición como alternativa.

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor

class NodoEtiquetado(Nodo):        # hereda de Nodo
    def __init__(self, valor, etiqueta):
        super().__init__(valor)    # reutiliza inicialización de la base
        self.etiqueta = etiqueta

n = NodoEtiquetado(10, "raiz")
print(n.valor, n.etiqueta)
```

2. Estructuras de datos básicas

2.1 Listas: crear, modificar, acceder; añadir, eliminar, buscar

Listas son contenedores ordenados y mutables. Operaciones comunes: append, insert, pop, remove, in, index, slicing. Búsqueda lineal O(n).

```
numeros = [3, 1, 4]
numeros.append(1)          # [3, 1, 4, 1]
numeros.insert(1, 9)       # [3, 9, 1, 4, 1]
existe = 4 in numeros      # True
pos = numeros.index(9)     # 1
el = numeros.pop()         # elimina último
```

2.2 Diccionarios: pares clave-valor

Permiten acceso promedio O(1) por clave. Métodos: get, keys, values, items, pop, update. Útiles para mapear identificadores a datos.

```
persona = {"nombre": "Luis", "edad": 18}
persona["ciudad"] = "Guatemala"
edad = persona.get("edad", 0)
for clave, valor in persona.items():
    print(clave, valor)
```

2.3 Tuplas y conjuntos: diferencias y usos

Tuplas son inmutables (seguras para datos que no deben cambiar). Conjuntos (set) no admiten duplicados y permiten operaciones matemáticas eficientes.

```
punto = (10, 20)                    # tupla
colores = {"rojo", "azul", "azul"}  # set -> {"rojo", "azul"}
otros = {"azul", "verde"}
print(colores | otros)              # unión
print(colores & otros)              # intersección
print(colores - otros)              # diferencia
```

3. Funciones y modularidad

3.1 Escribir funciones para organizar el código

Usar funciones con parámetros y valores de retorno mejora la reutilización. Documente con docstrings y valide entradas.

```
def altura_arbol(nodo) -> int:
    """Calcula recursivamente la altura de un árbol con raíz.
    Retorna -1 para árbol vacío por convenio clásico.
    """
    if nodo is None:
        return -1
    return 1 + max((altura_arbol(h) for h in nodo.hijos), default=-1)
```

3.2 Funciones: parámetros y valor de retorno

```
def buscar(lista, objetivo):
    for i, x in enumerate(lista):
        if x == objetivo:
            return i
    return -1
```

3.3 Ámbito de variables: local y global

Las variables definidas dentro de una función son locales. Las globales se definen fuera. Use global solo cuando sea imprescindible.

```
x = 10     # global
def f():
    x = 5   # local
    return x
print(f(), x)  # 5 10
```

3.4 Modularidad práctica: módulos y paquetes

Divida el proyecto en archivos (módulos) y use importaciones claras. Por ejemplo, un módulo nodo.py, otro utilidades.py y un main.py con el menú CLI.

Buenas prácticas y errores comunes

• Usar propiedades (@property) para validar atributos.
• Evitar variables globales; preferir pasar datos como parámetros.
• Nombrar con lower_snake_case (funciones) y UpperCamelCase (clases).
• Manejar excepciones (try/except) al leer entradas del usuario.
• Escribir docstrings y comentarios breves, no redundantes.

OBSERVACIONES Y COMENTARIOS

La OOP facilita mapear estructuras jerárquicas (como árboles con raíz) y separar responsabilidades en clases y funciones. Las estructuras nativas de Python permiten gestionar colecciones con pocas líneas y complejidad conocida. En un entorno sin librerías externas, las propiedades, la composición y el manejo cuidadoso de entradas del usuario son claves para robustez y claridad.

CONCLUSIONES

1) La combinación de OOP y estructuras nativas promueve software mantenible y extensible.
2) El encapsulamiento y las propiedades elevan la calidad y seguridad del código.
3) Listas, diccionarios, tuplas y sets cubren la mayoría de necesidades de almacenamiento base.
4) Funciones bien definidas y módulos separados mejoran pruebas y colaboración del equipo.
5) Estos fundamentos son suficientes para implementar árboles con raíz sin librerías externas.

BIBLIOGRAFÍA

Lutz, M. (2023). Learning Python (6th ed.). O'Reilly Media.
Ramalho, L. (2022). Fluent Python (2nd ed.). O'Reilly Media.
Zelle, J. (2022). Python Programming: An Introduction to Computer Science. Franklin, Beedle & Associates.
Python Software Foundation. (2025). The Python Language Reference (Version 3.12). https://docs.python.org/3/reference/
Guttag, J. (2023). Introduction to Computation and Programming Using Python. MIT Press.