TECNOLOGÍAS DE INFORMACIÓN Y CIBERSEGURIDAD

Inacap

PROGRAMACIÓN ORIENTADA A OBJETO (POO)

PROGRAMACIÓN ESTRUCTURADA

1. La programación estructurada es un enfoque de programación que organiza el código en bloques lógicos y secuenciales, utilizando estructuras como bucles y condicionales, para facilitar el diseño, comprensión y mantenimiento del software.

2. Se enfoca en evitar el uso de saltos incondicionales, como "goto", para mejorar la legibilidad y prevenir problemas conocidos como el "spaghetti code".

PROGRAMACIÓN ESTRUCTURADA: EJEMPLO

```python
def calcular_area_triangulo(base, altura):
    return (base * altura) / 2

def main():
    print("Calculadora de Área de Triángulo")
    base = float(input("Ingrese la longitud de la base del triángulo: "))
    altura = float(input("Ingrese la altura del triángulo: "))

    if base <= 0 or altura <= 0:
        print("Error: Los valores deben ser positivos.")
    else:
        area = calcular_area_triangulo(base, altura)
        print(f"El área del triángulo es: {area:.2f} unidades cuadradas.")

main()
```

¿QUÉ ES EL CÓDIGO ESPAGUETI?

1. El "spaghetti code" (código espagueti) es una expresión que se utiliza para describir un código de programación desorganizado, confuso y difícil de entender.

2. Se caracteriza por una excesiva cantidad de saltos incondicionales (como "goto" en lenguajes antiguos), múltiples anidaciones de bucles y condicionales, así como una estructura poco clara.

3. Este tipo de código es complicado de mantener y puede generar errores difíciles de identificar y corregir.

4. La programación estructurada busca evitar la creación de "spaghetti code" mediante la organización lógica y secuencial del código.

EJEMPLOS DE CÓDIGO ESPAGUETI EN C Y PYTHON

```c
#include <stdio.h>

int main() {
    int num, fact, i;
    printf("Ingrese un número para calcular su factorial: ");
    scanf("%d", &num);

    fact = 1;
    i = 1;

    inicio:
    if (i <= num) goto continuar;
    printf("El factorial de %d es %d\n", num, fact);
    goto fin;

    continuar:
    fact = fact * i;
    i = i + 1;
    goto inicio;

    fin:
    return 0;
}
```

```python
def calcular_factorial(num):
    fact = 1
    for i in range(1, num + 1):
        fact = fact * i
    return fact

def main():
    numero = int(input("Ingrese un número para calcular su factorial: "))

    if numero < 0:
        print("El factorial no está definido para números negativos.")
    else:
        resultado = calcular_factorial(numero)
        print(f"El factorial de {numero} es {resultado}.")

    print("Gracias por usar este programa.")

main()
```

PROGRAMACIÓN ORIENTADA A OBJETOS

1 La programación orientada a objetos es un paradigma de programación que organiza el código en objetos, que combinan datos y funciones relacionadas en una sola entidad.

2 Se enfoca en la reutilización de código, la encapsulación y el diseño de jerarquías de clases para facilitar el desarrollo, mantenimiento y escalabilidad del software.

DIFERENCIAS ENTRE PROGRAMACIÓN ESTRUCTURADA Y PROGRAMACIÓN ORIENTADA A OBJETOS

Característica	Programación Estructurada	Programación Orientada a Objetos
Unidades de organización	Funciones o procedimientos	Objetos y clases
Enfoque	Proceso lineal paso a paso	Interacción entre objetos
Reutilización de código	Menos propensa a la reutilización	Fomenta la reutilización del código
Encapsulación	No se enfoca en la encapsulación	Utiliza la encapsulación de datos y métodos
Herencia	No tiene concepto de herencia	Permite heredar propiedades y comportamientos
Polimorfismo	No se utiliza el polimorfismo	Permite que objetos de distintas clases respondan a una misma interfaz
Flexibilidad	Menos flexible	Mayor flexibilidad en el diseño del software
Complejidad	Manejo adecuado de la complejidad	Puede manejar mejor la complejidad a través de jerarquías de clases

PROGRAMACIÓN ESTRUCTURADA VS PROGRAMACIÓN ORIENTADA A OBJETOS

	Programación Estructurada	Programación Orientada a Objetos
Ventajas	- Simple y fácil de aprender.	- Reutilización de código a través de herencia y polimorfismo.
	- Menor complejidad en proyectos pequeños.	- Mayor organización y modularidad del código.
	- Más adecuada para proyectos pequeños y sencillos.	- Facilita el mantenimiento y la escalabilidad del software.
	- Menor sobrecarga de recursos.	- Permite un diseño más cercano al mundo real.
Desventajas	- Dificultad para manejar proyectos grandes y complejos.	- Mayor complejidad de aprendizaje y diseño inicial.
	- No permite una reutilización de código tan eficiente.	- Mayor consumo de recursos en comparación con estructurado.
	- Menos flexible y difícil de extender.	- Puede llevar a una sobreingeniería en proyectos pequeños.
	- Difícil de mantener en aplicaciones a largo plazo.	- Posible aumento de la complejidad en jerarquías de clases.

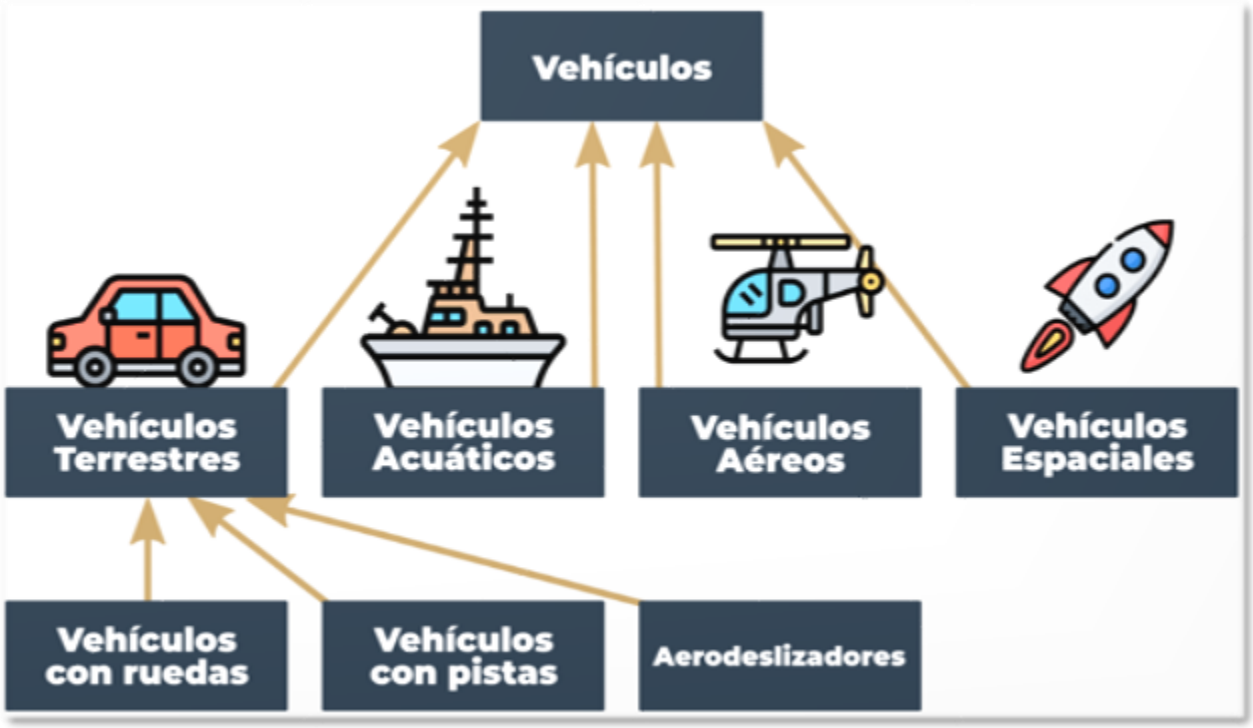
PROGRAMACIÓN ORIENTADA A OBJETOS: COMPONENTES

Componente	Descripción
Clase	Es la plantilla o el "molde" que define la estructura y el comportamiento de un objeto.
Objeto	Es una instancia concreta de una clase, con sus propios datos (atributos) y comportamientos (métodos).
Atributo	Son variables que contienen datos o estados específicos de un objeto.
Método	Son funciones asociadas a una clase que representan el comportamiento o las acciones que puede realizar un objeto.
Encapsulación	Es el principio que oculta los detalles internos de un objeto y expone solo las operaciones necesarias para interactuar con él.
Herencia	Es un mecanismo que permite que una clase adquiera las características (atributos y métodos) de otra clase, estableciendo una jerarquía de clases.
Polimorfismo	Es la capacidad de un objeto para tomar diferentes formas o comportarse de manera diferente según el contexto.
Abstracción	Es el proceso de identificar las características esenciales de un objeto y definir una clase para representarlo de manera simplificada.
Instanciación	Es el proceso de crear un objeto específico a partir de una clase.

PROGRAMACIÓN ORIENTADA A OBJETOS: COMPONENTES

PROGRAMACIÓN ORIENTADA A OBJETOS: JERARQUÍAS

La clase Vehículos es muy amplia. Tenemos que definir clases especializadas. Las clases especializadas son las subclases. La clase Vehículos será una superclase para todas ellas.



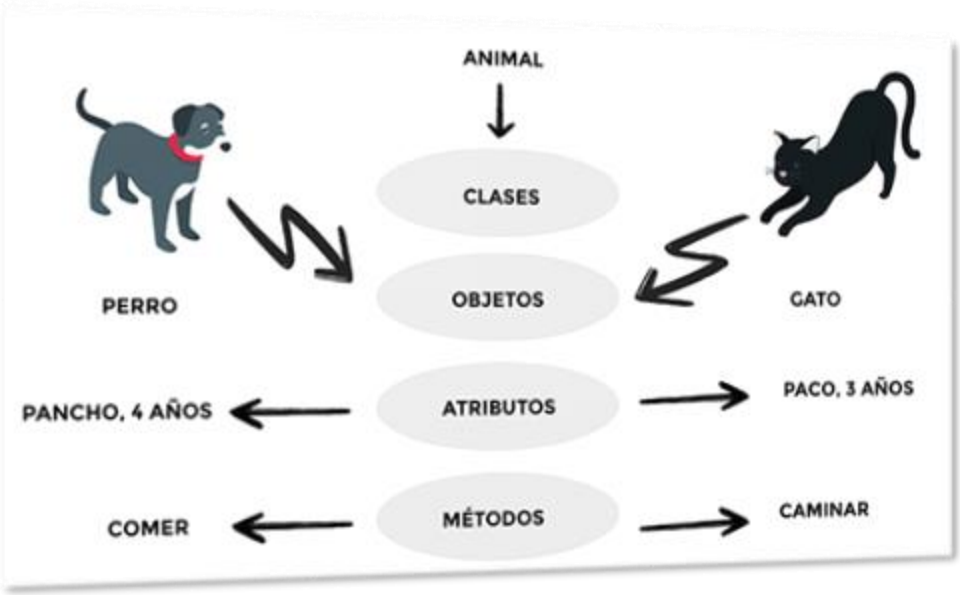
Nota: la jerarquía crece de arriba hacia abajo, como raíces de árboles, no ramas. La clase más general y más amplia siempre está en la parte superior (la superclase) mientras que sus descendientes se encuentran abajo (las subclases).

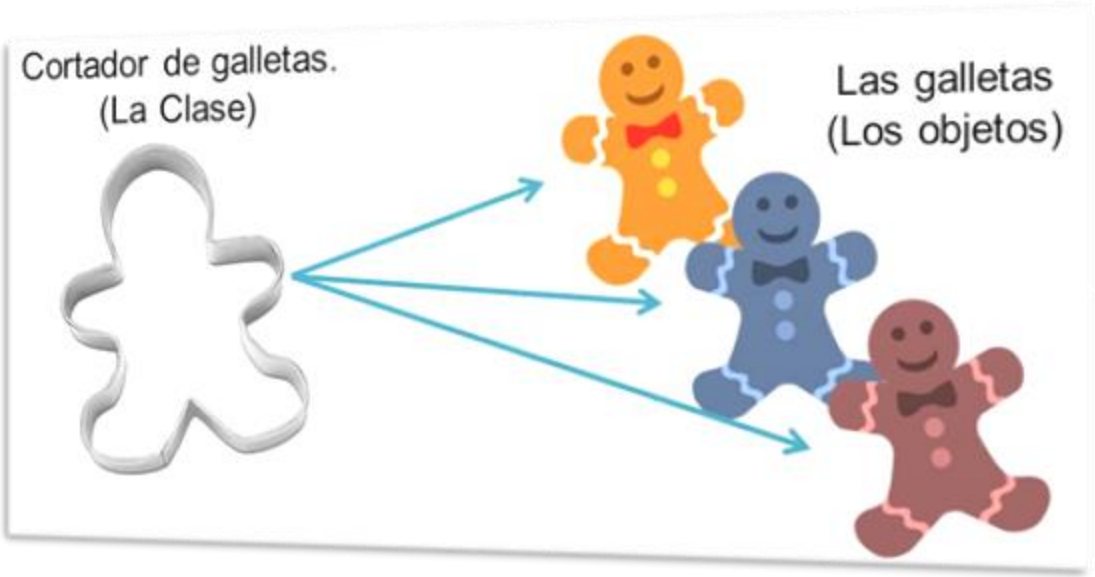
¿QUÉ ES UN OBJETO?

1. Una clase es un conjunto de objetos. Un objeto es un ser perteneciente a una clase.
2. Un objeto es una encarnación de una clase con atributos y métodos.
3. Las clases forman jerarquías, donde un objeto puede pertenecer a múltiples niveles (superclases y subclases)
4. Las subclases son más especializadas que sus superclases, que son más generales.

Por ejemplo: cualquier automóvil personal es un objeto que pertenece a la clase Vehículos Terrestres. También significa que el mismo automóvil pertenece a todas las superclases de su clase local; por lo tanto, también es miembro de la clase Vehículos.

¿QUÉ ES UN OBJETO?





CREANDO UNA CLASE

```
#Definición de la clase
class Galleta:
    pass

#Instanciación (creando objetos)
una_galleta = Galleta()
otra_galleta = Galleta()
```

FUNCIÓN TYPE

```
#Función type() -> Sirve para determinar la clase de un objeto
print(type(una_galleta))
print(type(10))
print(type(3.14))
print(type("Hola"))
print(type([]))
print(type({}))
print(type(()))
print(type(True))
print(type(print))
```

ATRIBUTOS Y MÉTODOS DE CLASE

1. **Atributos:** hacen referencia a las **variables** internas de la clase.

2. **Métodos:** hacen referencia a las **funciones** internas de la clase

```
#Definición de atributos en la clase
class Galleta:
    cobertura = False

#Creamos la instancia y revisamos el valor inicial del atributo
g = Galleta()
print(g.cobertura)

#Cambiamos el valor del atributo para la instancia creada y verificamos
el nuevo valor
g.cobertura = True
print(g.cobertura)
```

ATRIBUTOS Y MÉTODOS DE CLASE

Definición de Atributos dinámicos en los objetos

```
#Definimos atributos dinámicos en el objeto y verificamos
g.sabor = "Salado"
g.color = "Marrón"
print(f"El color de la galleta es {g.color} y su sabor es {g.sabor}")
```

MÉTODO INIT()

- **Se llama automáticamente al crear una instancia de una clase.**

```
#Definimos el método init() dentro de la clase
class Galleta:
    cobertura = False
    def __init__(self):
        print("Se acaba de crear una galleta.")

#Verificamos que al crear una instancia de la clase aparece el
mensaje
g = Galleta()
```

SELF

Self sirve para hacer referencia a los métodos y atributos base de una clase dentro de sus propios métodos.

```
class Galleta:
    cobertura = False
    def __init__(self):
        print("Se acaba de crear una galleta.")

    def chocolatear(self):
        self.cobertura = "Chocolate"

    def tiene_chocolate(self):
        if(self.cobertura):
            print("Soy una galleta cubierta de chocolate :-)")
        else:
            print("Soy una galleta sin cubierta de chocolate :-(")

#Creamos la instancia y probamos los métodos
g = Galleta()
g.tiene_chocolate()
g.chocolatear()
g.tiene_chocolate()
```

PARÁMETROS EN EL INIT (ARGUMENTOS AL INSTANCIAR)

```
class Galleta:
    cobertura = False
    #Agregamos argumentos en el método Init
    def __init__(self, sabor, forma):
        self.sabor = sabor
        self.forma = forma
        print(f"Se acaba de crear una galleta {sabor} y {forma}.")

    def chocolatear(self):
        self.cobertura = "Chocolate"

    def tiene_chocolate(self):
        if(self.cobertura):
            print("Soy una galleta cubierta de chocolate :-)")
        else:
            print("Soy una galleta sin cubierta de chocolate :-(")

#Creamos la instancia y enviamos los argumentos
g = Galleta("salada", "cuadrada")
```

PARÁMETROS POR DEFECTO EN EL INIT

```
class Galleta:
    cobertura = False
    #Agregamos parámetros con valores por defectp en el método Init
    def __init__(self, sabor = None, forma = None):
        self.sabor = sabor
        self.forma = forma
        if sabor is not None and forma is not None:
            print(f"Se acaba de crear una galleta {sabor} y {forma}.")
        else:
            print(f"Se acaba de crear una galleta incípida y deforme")

    def chocolatear(self):
        self.cobertura = "Chocolate"

    def tiene_chocolate(self):
        if(self.cobertura):
            print("Soy una galleta cubierta de chocolate :-)")
        else:
            print("Soy una galleta sin cubierta de chocolate :-(")

#Creamos la instancia sin enviar argumentos
g = Galleta()
#Creamos una instancia enviando argumentos
g2 = Galleta("salada", "cuadrada")
```

CONSTRUCTOR Y DESTRUCTOR

```
class Pelicula:
    #Constructor de clase (al crear la instancia)
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")

    #Destructor de clase (al borrar la instancia)
    def __del__(self):
        print(f"Se borrará la película {self.titulo}")

p = Pelicula("El Padrino", 175, 1972)
print(p.titulo)
#Al reinstanciar la misma variable, se crea de nuevo y se borra la
anterior
p = Pelicula("Volver al Futuro", 125, 1982)
print(p.titulo)
```

FUNCIÓN STR

```
#Para devolver una cadena por defecto al convertir un objeto a
una cadena con str(objeto)

#Revisamos cómo se ve el objeto al imprimirlo
print(p)
#Verificamos el tipo del objeto antes de convertirlo con str
print(type(p))
#Probamos la función str
print(str(p))
#Probamos el tipo del objeto convertido con str
print(type(str(p)))
```

FUNCIÓN STR

```
class Pelicula:
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")

    #Redefinimos el método str
    def __str__(self):
        return f"{self.titulo} lanzada en {self.lanzamiento} con una
duración de {self.duracion} minutos"

p = Pelicula("El Padrino", 175, 1972)

#Probamos la nueva función str
print(str(p))
```

FUNCIÓN LEN

```
#Para devolver un número que simula la longitud del objeto
len(objeto)
print(len(p))
```

FUNCIÓN LEN

```
    #Redefinimos el método len
    def __len__(self):
        return self.duracion

p = Pelicula("El Padrino", 175, 1972)

#Probamos la nueva función len
print(len(p))
```

SOBRECARGA

```
class Pelicula:
    #Constructor de la clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")

    def __init__(self, titulo = None, duracion = None, lanzamiento = None):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento

    #Destructor de la clase
    def __del__(self):
        print(f"Se borrará la película {self.titulo}")

    #Redefinimos el método len
    def __len__(self):
        return self.duracion

p = Pelicula("El Padrino", 175, 1972)
pp = Pelicula()

#Probamos lo que pasa al usar len para cada instancia
print(len(p))
print(len(pp))
```

VARIABLES DE INSTANCIA

```
class Pelicula:

    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")

    def set_director(self, director):
        self.director = director

#Creamos 3 instancias distintas
p = Pelicula("El Padrino", 175, 1972)
pp = Pelicula("Volver al Futuro", 125, 1982)
ppp = Pelicula("Barbie", 120, 2023)

#Añadimos y creamos variables para algunas de ellas
pp.set_director("Robert Zemeckis")
ppp.categoria = "Fantasía"

#Verificamos las diferentes variables de cada instancia
print(p.__dict__)
print(pp.__dict__)
print(ppp.__dict__)
```

FUNCIÓN HASATTR

```
class Pelicula:

    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")

    def set_director(self, director):
        self.director = director

p = Pelicula("El Padrino", 175, 1972)

#Verificamos si el objeto tiene cierto atributo usando hasattr
if hasattr(p, "categoria"):
    print(p.categoria)
if hasattr(p, "titulo"):
    print(p.titulo)
```

RELACIÓN DE OBJETOS: CREANDO UN CATÁLOGO DE PELÍCULAS

```python
class Pelicula:
    #Constructor de la clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print(f"Se ha creado la película {self.titulo}")
    def __str__(self):
        return f"{self.titulo}  {self.lanzamiento}"

class Catalogo:
    peliculas = [] #Almacenará el listado de películas

    def __init__(self, peliculas = []):
        self.peliculas = peliculas
    def agregar(self, p): #p será un objeto Pelicula
        self.peliculas.append(p)
    def mostrar(self):
        for p in self.peliculas:
            print(p) #Este print toma por defecto str(p)

p = Pelicula("El Padrino", 175, 1972)
#Creamos la instancia del catálogo y añadimos películas
c = Catalogo([p])
c.agregar(Pelicula("El Padrino 2", 202, 1974))
#Revisamos las películas guardadas en el catálogo
c.mostrar()
```

ENCAPSULACIÓN: CONCEPTO

```
#Consiste para denegar el acceso a los atributos y métodos de la clase desde el exterior
#En Python no existe como tal, pero se puede simular...

class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera"

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera")

e = Ejemplo()
e.__atributo_privado
```

ENCAPSULACIÓN: ACCEDIENDO

```
#Internamente la clase puede acceder a sus atributos y métodos encapsulados
#El truco consiste en crear sus equivalente "públicos"

class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera"

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()

e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

HERENCIA

```
#Creando una jerarquía de productos con clases
#Superclase Producto
class Producto:
    def __init__(self, identificador, nombre):
        self.identificador = identificador
        self.nombre = nombre
    def __str__(self):
        return f"""
IDENTIFICADOR:  {self.identificador}
NOMBRE:         {self.nombre}"""

#Subclase Adorno
class Adorno(Producto):
    pass
a = Adorno(1, "Jarra Ming", "Jarra de la dinastía Ming")
print(a)

#Subclase Alimento
class Alimento(Producto):
    productor = ""
    distribuidor = ""

    def __str__(self):
        return f"""
IDENTIFICADOR:  {self.identificador}
NOMBRE:         {self.nombre}
PRODUCTOR:      {self.productor}
DISTRIBUIDOR:   {self.distribuidor}"""

al = Alimento(2, "Pan Bimbo")
al.productor = "Bimbo"
al.distribuidor = "Bimbo SA"
print(al)
```

FUNCIÓN ISINSTANCE

```
#Creamos una lista con los productos creados
productos = [a, al]

#Recorremos la lista
for p in productos:
    #Verificamos a qué instancia pertenece el valor actual
    if isinstance(p, Adorno):
        print(f"Producto de la Subclase Adorno:{p}\n")
    elif isinstance(p, Alimento):
        print(f"Producto de la Subclase Alimento:{p}\n")
```

```
#Podemos recorrer los objetos y verificar si uno es una
instancia de cierta clase
for obj in [a, al]:
    for cls in [Producto, Adorno, Alimento]:
        print(isinstance(obj, cls), end = "\t")
    print()
```

FUNCIONES QUE RECIBEN OBJETOS DE CLASES

```
#Los objetos se envían por referencia a las funciones
#Cualquier cambio realizado dentro afectará al propio objeto
def agregar_precio(producto, precio):
    producto.precio = precio
    print(f"""Los nuevos datos para su producto son: {producto}
PRECIO:         {producto.precio}""")

#Llamamos a la función antes creada y le enviamos los valores
solicitados
agregar_precio(al, 1500)
```

COPIA DE OBJETOS

```
#Una copia de un objeto también hace referencia al objeto copiado (como un acceso directo)
copia_al = al
copia_al.nombre = "Pan Integral"
print(copia_al)
```

```
#Para crear una copia al 100% nueva debemos utilizar la función copy del módulo copy
import copy

copia_al = copy.copy(al)
copia_al.nombre = "Pan Integral"
print(al)
print(copia_al)
```

POLIMORFISMO

```
#Se refiere a una propiedad de la herencia por la que objetos de distintas subclases pueden responder a una misma acción

#La siguiente función es capaz de cambiar el nombre al objeto entregado

#La acción de manipular el nombre funcionará siempre que el objeto tenga dicho atributo

#En caso contrario ocasionaría un error

def cambiar_nombre(producto, nombre):
    producto.nombre = nombre
    print(f"El nombre del producto cambió: {producto}")

cambiar_nombre(al, "Pancito")

#La polimorfía es implícita en Python en todos los objetos, ya que todos son hijos de una superclase compun llamada Object
```

FUNCIÓN ISSUBCLASS

```
#Podemos recorrer las clases y verificar si una es subclase de
la otra
for cls1 in [Producto, Adorno, Alimento]:
    for cls2 in [Producto, Adorno, Alimento]:
        print(issubclass(cls1, cls2), end = "\t")
    print()
```

FUNCIÓN SUPER

```
#La función super() en Python se utiliza para llamar métodos o el constructor de la clase padre desde una clase hija.
#Facilita la herencia y reutilización de código al permitir que la clase hija acceda y extienda el comportamiento de la clase padre sin necesidad de referenciar su nombre directamente.

class Animal:
    def __init__(self, especie):
        self.especie = especie
    def hacer_sonido(self):
        print("Haciendo sonido genérico")

class Perro(Animal):
    def __init__(self, especie, raza):
        super().__init__(especie)  # Llamada al constructor de la clase padre
        self.raza = raza
    def hacer_sonido(self):
        print("Guau guau")

mi_perro = Perro(especie="Canino", raza="Golden Retriever")
print(mi_perro.especie)  # Salida: Canino
mi_perro.hacer_sonido()  # Salida: Guau guau

#En este ejemplo, tenemos una clase base Animal que tiene un constructor __init__ y un método hacer_sonido. Luego, definimos una clase Perro que hereda de Animal. Utilizamos super().__init__(especie) en el constructor de Perro para llamar al constructor de la clase padre Animal y asegurarnos de que se inicialice correctamente. Luego, en el método hacer_sonido de Perro, sobrescribimos el comportamiento del método de la clase padre.
```

HERENCIA MÚLTIPLE

```
#Es la capacidad de una clase hija para heredar atributos y métodos de varias clases padres.
#Una clase puede heredar de múltiples clases, lo que permite la reutilización de código y la construcción de jerarquías de clases complejas.
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre
    def hacer_sonido(self):
        pass

class Volador:
    def volar(self):
        print("Volando alto")

class Pajaro(Animal, Volador):
    def hacer_sonido(self):
        print("Pío pío")

pajaro1 = Pajaro(nombre="Loro")
print(pajaro1.nombre)  # Salida: Loro
pajaro1.hacer_sonido()  # Salida: Pío pío
pajaro1.volar()  # Salida: Volando alto

#Tenemos dos clases padres, Animal y Volador, y una clase hija Pajaro que hereda de ambas.
#La clase Animal tiene un atributo nombre y un método hacer_sonido, mientras que la clase Volador tiene un método volar.
#La clase Pajaro hereda tanto el atributo y método de Animal, como el método de Volador.
#Esto permite que un objeto de la clase Pajaro tenga acceso tanto a las características de un animal como a las habilidades de volar.
```

HERENCIA MÚLTIPLE: MRO

```
#El MRO (Method Resolution Order) es el orden en el que Python busca y resuelve los métodos a ser ejecutados en una jerarquía de clases que involucra herencia múltiple.

class A:
    def saludar(self):
        print("Hola desde A")
class B(A):
    def saludar(self):
        print("Hola desde B")
class C(A):
    def saludar(self):
        print("Hola desde C")
class D(B, C):
    pass

objeto_d = D()
objeto_d.saludar()

#En este ejemplo, tenemos cuatro clases: A, B, C y D.
#Las clases B y C heredan de la clase A, y la clase D hereda de ambas, B y C.
#Cuando creamos un objeto de la clase D (objeto_d) y llamamos al método saludar(), Python seguirá el MRO para determinar qué método se ejecutará.
#El MRO se basa en un algoritmo llamado C3 Linearization, que en este caso específico daría el siguiente orden: D -> B -> C -> A. Por lo tanto, al llamar al método saludar() en objeto_d, se imprimirá: "Hola desde B". Esto es porque Python busca el método saludar() primero en la clase más a la izquierda en la lista del MRO que tiene dicho método, y luego continúa hacia la derecha si no lo encuentra.
#Es fundamental para resolver el orden de ejecución de los métodos en clases que utilizan herencia múltiple y asegura que los métodos sean llamados de manera coherente y sin conflictos.
```

EXCEPCIONES CON CLASES

```
#Las excepciones con clases en Python son errores que ocurren durante la ejecución del código y que pueden ser capturados y tratados mediante clases personalizadas.
class MiErrorPersonalizado(Exception):
    def __init__(self, mensaje):
        super().__init__(mensaje)
def dividir(a, b):
    if b == 0:
        raise MiErrorPersonalizado("No se puede dividir por cero.")
    return a / b

try:
    resultado = dividir(10, 0)
    print("El resultado es:", resultado)
except MiErrorPersonalizado as error:
    print("Ha ocurrido un error:", error)

#En este ejemplo, definimos una clase personalizada de excepción llamada MiErrorPersonalizado, que hereda de la clase base Exception. Luego, definimos una función dividir(a, b) que realiza una división y, si el divisor b es cero, levanta una excepción de tipo MiErrorPersonalizado con un mensaje personalizado.
#En el bloque try, llamamos a la función dividir con argumentos 10 y 0. Al intentar dividir por cero, se levanta la excepción personalizada. En el bloque except, capturamos la excepción, y el programa muestra el mensaje de error definido en la clase MiErrorPersonalizado.
#Las excepciones con clases permiten crear errores personalizados y controlar situaciones específicas en el código, mejorando la capacidad de manejo de errores y proporcionando información más significativa sobre las excepciones que pueden ocurrir.
```