

UNIVERSIDAD TECNOLÓGICA DE PANAMÁ  
FACULTAD DE INGENIERÍA DE SISTEMAS  
COMPUTACIONALES  
DEPARTAMENTO DE PROGRAMACIÓN DE  
COMPUTADORAS


MATERIA:  
  
PROGRAMACIÓN II

---

FACILITADOR:  
ING. JOSÉ JAVIER CHIRÚ F.

I SEMESTRE  
2024

# PROGRAMACIÓN ORIENTADA A OBJETOS EN PYTHON



# Programación orientada a objetos en python

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos" y en la interacción entre ellos para modelar sistemas y resolver problemas. En la POO, los objetos son unidades de software que combinan datos (atributos) y comportamientos (métodos) relacionados, lo que permite representar entidades del mundo real de manera más efectiva en el software. Este enfoque favorece la modularidad, la reutilización del código, la organización estructurada de los programas y la abstracción de los detalles internos de implementación, lo que resulta en sistemas más flexibles, mantenibles y escalables.

---



# Clases

Una clase en programación orientada a objetos es una plantilla que define la estructura y el comportamiento de los objetos de un determinado tipo. Esto incluye la especificación de los atributos (datos) que tendrán los objetos, así como los métodos que operarán sobre esos datos y permitirán la interacción entre los objetos.

Los atributos representan las características o propiedades que describen el estado de un objeto, mientras que los métodos representan las acciones o comportamientos que pueden realizar esos objetos. La combinación de atributos y métodos en una clase define cómo se comportarán los objetos creados a partir de esa clase.

---

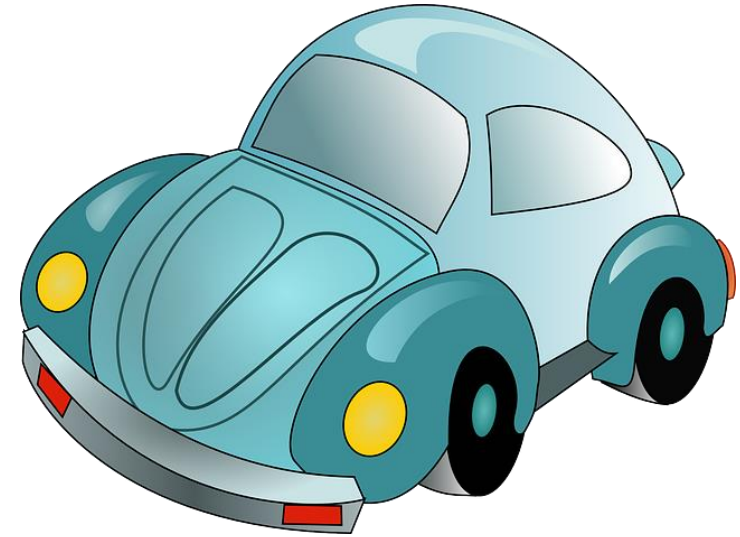
# Objeto

---

Los objetos son/representan cosas

Los objetos pueden ser simples o complejos

Los objetos pueden ser reales o imaginarios



# Abstracción

La abstracción se trata de simplificar la interacción con objetos complejos al exponer solo los detalles necesarios para realizar ciertas tareas, mientras se ocultan los detalles internos complejos.

La abstracción es como usar un control remoto para la televisión. No necesitas saber cómo funciona la televisión por dentro; simplemente presionas los botones para cambiar de canal o ajustar el volumen. De la misma manera, en programación, la abstracción te permite usar objetos y funciones sin tener que entender todos los detalles de cómo están hechos.

La abstracción se basa en usar cosas simples para representar la complejidad.

**Énfasis en el ¿qué hace? Mas que en el ¿Cómo lo hace?**





# Polimorfismo

El polimorfismo es un concepto fundamental en la programación orientada a objetos que se refiere a la capacidad de objetos de diferentes clases de responder de manera específica a una misma llamada de método o acción. En otras palabras, permite que un mismo nombre de método pueda comportarse de manera diferente dependiendo del objeto que lo esté ejecutando. Esto se logra mediante la herencia y la sobrescritura de métodos en las clases hijas, lo que permite que un objeto pueda exhibir comportamientos distintos según su tipo concreto en tiempo de ejecución. El polimorfismo facilita la flexibilidad y la reutilización del código en los sistemas orientados a objetos.

---



```
def my_function(param1, param2, ...):  
    pass
```

Funciones definidas por el usuario





# Declaración de Funciones

Los métodos de una clase son las funciones que definen el comportamiento de los objetos de esa clase.

Se definen como las funciones con la palabra reservada **def**. La única diferencia es que su primer parámetro es especial y se denomina **self**. Este parámetro hace siempre referencia al objeto desde donde se llama el método, de manera que para acceder a los atributos o métodos de una clase en su propia definición se puede utilizar la sintaxis **self.atributo** o **self.método**.

---

# Declaración de Funciones

```
class Saludo:
    mensaje = "Bienvenido "           # Definición de un atributo
    def saludar(self, nombre):        # Definición de un método
        print(self.mensaje + nombre)
        return
```

---



# Funciones recursivas

Una función recursiva es una función que se llama a sí misma dentro de su propio cuerpo de código. Esta llamada se realiza con el objetivo de resolver un problema de manera iterativa, dividiéndolo en subproblemas más simples hasta alcanzar un caso base que detiene la recursión. Esto permite abordar problemas complejos de manera más clara y estructurada, ya que se resuelven de manera repetida utilizando la misma lógica, pero con datos diferentes en cada llamada recursiva.

funcionan de forma similar a las iteraciones, pero debe encargarse de planificar el momento en que dejan de llamarse a sí mismas o tendrá una función recursiva infinita.

---

# Funciones recursivas

```
def cuenta_regresiva(numero):  
    numero -= 1  
    if numero > 0:  
        print (numero)  
        cuenta_regresiva(numero)  
    else:  
        print ("Booooooooooom!")  
        print ("Fin de la función", numero)
```

```
4  
3  
2  
1  
Booooooooooom!  
Fin de la función 0
```



# Funciones de orden superior

Una función de orden superior en Python es aquella que toma una o más funciones como argumentos y/o devuelve una función como resultado. En otras palabras, estas funciones pueden recibir otras funciones como parámetros o retornar funciones como valores.

---

# Funciones de orden superior

```
def operar(v1,v2,fn):  
    return fn(v1,v2)
```

```
def sumar(x1,x2):  
    return x1+x2
```

```
def restar(x1,x2):  
    return x1-x2
```

```
resu1=operar(10,3,sumar)  
print(resu1)
```

```
resu2=operar(10,3,restar)  
print(resu2)
```

```
13  
7
```



# Funciones integradas

El interprete Python tiene un número de funciones integradas (built-in) dentro del módulo `__builtins__`, las cuales están siempre disponibles. Estas funciones están listadas en orden alfabéticos a continuación:

---

# Funciones integradas

Funciones Built-in				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	





# Funciones sin parámetros

```
def diHola():  
    print("Hello!")
```





# Funciones lambda

Una función lambda en Python es una forma de crear funciones pequeñas y anónimas de una sola línea sin necesidad de definirlas formalmente con la palabra clave `def`. Estas funciones son útiles cuando necesitas una función rápida para realizar una operación simple.

Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión.

---

# Funciones lambda

#Aquí tenemos una función creada para sumar.

```
def suma(x,y):  
    return(x + y)
```

#Aquí tenemos una función Lambda que también suma.

```
lambda x,y : x + y
```

---



# Método o atributo privado

En Python no existe una distinción explícita entre métodos/atributos públicos y privados como en otros lenguajes de programación como Java o C++. En lugar de eso, Python utiliza una convención de nomenclatura para indicar el nivel de acceso deseado.

No obstante, como convención se prefija un doble subrayado (\_\_) en los atributos y métodos. Esto debe ser interpretado por el programador como privado.

---



# Diccionario

Un Diccionario es una estructura de datos y un tipo de dato en Python con características especiales que nos permite almacenar cualquier tipo de valor como enteros, cadenas, listas e incluso otras funciones. Estos diccionarios nos permiten además identificar cada elemento por una clave (Key).

---



# Diccionario

Para definir un diccionario, se encierra el listado de valores entre llaves. Las parejas de clave y valor se separan con comas, y la clave y el valor se separan con dos puntos.

```
diccionario = {'nombre' : 'Carlos', 'edad' : 22, 'cursos': ['Python','Django','JavaScript'] }
```

---

# Diccionario

```
lista = [6103, 7540] #Lista
materias = {}# Diccionario vacio
dia="lunes"
materias[dia] = lista
materias["martes"] = 6201
print(materias)
```

```
{'lunes': [6103, 7540], 'martes': 6201}
```

---

# Diccionario

Podemos acceder al elemento de un Diccionario mediante la clave de este elemento, como veremos a continuación

```
lista = [6103, 7540] #Lista
materias = {} #Diccionario vacio
dia = "lunes"
materias[dia] = lista
materias["martes"] = 6201
print(materias[dia]) #Acceso al diccionario
```

```
[6103, 7540]
```



# Diccionario

```
lista = [6103, 7540] #Lista
materias = {}# Diccionario vacio
dia="lunes"
materias[dia] = lista
materias["martes"] = 6201
for dias in materias:
    print (dias, ":", materias[dias])
```

```
lunes : [6103, 7540]
martes : [6103, 7540]
```

---

# Diccionario

```
lista = [6103, 7540] #Lista
materias = {}# Diccionario vacio
dia="lunes"
materias[dia] = lista
materias["martes"] = 6201
for dias, codigos in materias.items():
    print (dias, ":", codigos)
```

```
lunes : [6103, 7540]
martes : 6201
```

---

# Métodos de los Diccionarios

`dict ()`

Recibe como parámetro una representación de un diccionario y si es factible, devuelve un diccionario de datos.

```
dic = dict(nombre='nestor', apellido='Plasencia', edad=22)
```

```
dic → {'nombre' : 'nestor', 'apellido' : 'Plasencia', 'edad' : 22}
```

---

# Métodos de los Diccionarios

zip()

Recibe como parámetro dos elementos iterables, ya sea una cadena, una lista o una tupla. Ambos parámetros deben tener el mismo número de elementos. Se devolverá un diccionario relacionando el elemento i-esimo de cada uno de los iterables.

```
dic = dict(zip('abcd', [1, 2, 3, 4]))
```

```
dic → {'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4}
```



# Métodos de los Diccionarios

## **items()**

Devuelve una lista de tuplas, cada tupla se compone de dos elementos: el primero será la clave y el segundo, su valor.

## **keys()**

Retorna una lista de elementos, los cuales serán las claves de nuestro diccionario.

## **values()**

Retorna una lista de elementos, que serán los valores de nuestro diccionario.

---



# Métodos de los Diccionarios

## **clear()**

Elimina todos los ítems del diccionario dejándolo vacío.

## **copy()**

Retorna una copia del diccionario original

## **get()**

Recibe como parámetro una clave, devuelve el valor de la clave. Si no lo encuentra, devuelve un objeto none.

## **pop()**

Recibe como parámetro una clave, elimina esta y devuelve su valor. Si no lo encuentra, devuelve error.

---

# Métodos de los Diccionarios

## update()

Recibe como parámetro otro diccionario. Si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
dic 1 = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}  
dic 2 = {'c' : 6, 'b' : 5, 'e' : 9 , 'f' : 10}  
dic1.update(dic 2)  
  
dic 1 → {'a' : 1, 'b' : 5, 'c' : 6 , 'd' : 4 , 'e' : 9 , 'f' : 10}
```

# Manejo de archivos

En Python, hay diferentes formas de leer un archivo de texto.

```
open => Abrir  
read => Leer  
readLine => Leer linea  
readLines => Leer lineas  
close => cerrar  
with => Con
```





# Manejo de archivos

En Python, para abrir un archivo usaremos la función `open`, que recibe el nombre del archivo a abrir.

```
open("archivo.txt")
```

Esta función intentará abrir el archivo con el nombre indicado. Si tiene éxito, devolverá una variable que nos permitir

---

Modo	Descripción
'r'	Abra un archivo para leerlo. (predeterminado)
'w'	Abra un archivo para escribir. Crea un nuevo archivo si no existe o trunca el archivo si existe.
'x'	Abra un archivo para su creación exclusiva. Si el archivo ya existe, se produce un error en la operación.
'a'	Abierto para anexar al final del archivo sin truncarlo. Crea un nuevo archivo si no existe.
't'	Abrir en modo texto. (predeterminado)
'b'	Abrir en modo binario.
'+'	Abrir un archivo para actualizarlo (lectura y escritura)

# Manejo de archivos

---

# Manejo de archivos

La operación más sencilla a realizar sobre un archivo es leer su contenido.

```
línea=archivo.readline()
while línea != '':
    # procesar línea
    línea=archivo.readline()
```

```
for línea in archivo:
    # procesar línea
```

```
with open("text.txt","w") as file:
    file.write("I am learning Python!\n")
    file.write("I am really enjoying it!\n")
    file.write("And I want to add more lines to say how much I like it")
```