Tema 1 – Introducción a los lenguajes de programación

1. Conceptos

Programación → Proceso de diseño y construcción de un programa informático. Cuenta con una herramienta que permite escribir y traducir el código al lenguaje máquina (el que comprende el ordenador).

Codificar → Proceso de escribir códigos de un idioma a otro. Los ordenadores no entienden el lenguaje natural, solo el binario.

Algoritmo → Conjunto de instrucciones o reglas definidas y no ambiguas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades.

Variable → Espacio de memoria que contiene una cantidad de información conocida o desconocida, es decir un valor, y que tienen un nombre simbólico asociado a dicho espacio.

Función → proporción o bloque de código que se puede volver a usar y que realiza una tarea determinada.

Objetos → unidad dentro de un programa que tiene un estado (atributo) y un comportamiento (métodos). Se crean instanciando clases.

Clases → representación de una entidad o concepto

2. Tipos de lenguajes de programación (cercanía a la máquina)

Según su nivel:

- **Bajo nivel** → Lenguaje parecido al de la máquina. Depende del ordenador
  Lenguaje máquina: difícil de programar pero muy rápido.
  Lenguaje ensamblador: Derivado del lenguaje máquina. Son abreviaciones, letras y números. Se traduce luego a lenguaje máquina.
- **Alto nivel** → Lenguaje más parecido al nuestro, es independiente de la máquina. (El programador se olvida del hardware)

Según su ejecución:

- **Compilados** → Código → Proceso de compilación → Ordenador.
- **Interpretado** → Código → Proceso de compilación (interprete: traduce el código al lenguaje máquina) → Ordenador.

3. Introducción a Python

Características del lenguaje

1) **Interpretado** → Todo lo hace en tiempo de ejecución.
2) **Sintaxis elegante y tipado dinámico** → Usa identación (tab) para separar los bloques de código.
3) **Tipado dinámico fuerte** → La misma variable puede tomar otros valores en diferentes momentos.
4) **Multiparadigma** → Puedes programar de diferentes modos.
5) **Cantidad abundante de librerías que facilitan la programación**.
6) **Sencillez y velocidad**.
7) Es **multiplataforma**. → Disponible en todos los dispositivos que tengan el intérprete.
8) Es **gratuito**.

9) **Software libre. (FLOSS).**
10) **Propósito general** → Utilizable para cualquier cosa que se nos ocurra.

¿Cómo vemos el Zen de Python? Si ponemos el comando >>> import this.

Otros principios de Python

- El desarrollador no es estúpido, no se hace nada para bloquear al desarrollador, se encuentran todo tipo de soluciones a los problemas.
- Incorpora dosctring para documentar el código y preparar pruebas unitarias.
- Incluye una librería estándar y de terceros que facilitan la programación y permiten que se pueda interactuar con otros lenguajes.
- Duck typing: el fondo es más importante que la forma, que el aspecto funcional es más importante que el técnico.
  ``*Si veo un animal que vuela como un pato y nada como un pato, entonces es un pato*´´

Tema 2 – Entornos de programación en python

1. Definición de IDE.

IDE → Integrated Development Enviroment o Entorno de Desarrollo Integrado.

- Herramienta que facilita la programación a los desarrolladores.
- Cuenta con;
  - editores de código fuerte
  - herramientas de construcción automática y depuradores de código.
  - IntilliSense, un autocompletado inteligente de código.
  - Algunos tienen compilador y/o un intérprete (NetBeans y VisualStudio).
- Buscan maximizar la productividad con gran variedad de herramientas en un mismo entorno.
- Git hub → Sistema de Control de versiones.

2. Entornos en Python

- **Consola Python**: no es un IDE, permite lanzar cualquier instrucción, ver la ayuda, el zen, licencia …etc.
- **Jupyter**
- **Visual Studio code**
- **Pycharm**
- **Spyder**

Tema 3 – Elementos del lenguaje en Python

1. Variables y tipos de datos

Variables: almacenan valores de datos, pueden ser declaradas con cualquier tipo de datos (cogerá el de la asignación que la pongan) y pueden cambiar de tipo después de ser asignadas y usadas.

* Cada variable en Python es un objeto, soporta instrucciones genéricas que nos informan de los objetos.

```
#help (objetc) muestra información de cómo usar objetos.
#dir (objetc) muestra la estructura interna del objeto (sus métodos y atributos)
#ejemplos
x = 1
help (x)
dir (x)
```

Para definir una variable deben cumplir las siguientes normas:

1) Deben comenzar por una letra o un guion bajo.
2) No pueden comenzar por un número.
3) Solo pueden tener caracteres alfanuméricos y guiones bajos.
4) Cuidado con las mayúsculas. **(Case sensitive)**

```
#no son las mismas variables
EDAD = 18
edad = 18
```

con el = se asignan valores

```
#asignación multiple de variables.
x,y,z= 1,2,3
print (x)
print (y)
```

```
#asignación múltiple con el mismo valor a diferentes variables.
x=y=z="Hola"
print ("El valor de x es ``+x")
print (y)
```

Se puede obtener el tipo de dato de una variable (objeto) usando la función **type ()**

```
x = "Hola"
y = 1
type (x)
type (y)
```

Tipos de datos avanzados

Las variables, almacenan **diferentes tipos de datos** que permiten realizar **distintos tipos de acciones y operaciones.**

Categorías de tipos de datos

El tipo de dato de una variable se fija cuando se asigna un valor a esa variable.

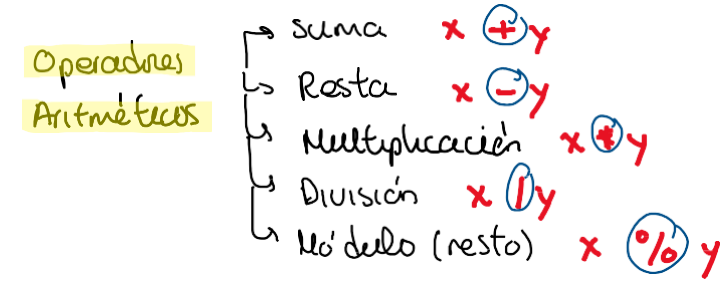
- Texto
- Numerico
- Secuencia
- Mappins
- set
- Boolean
- Binary

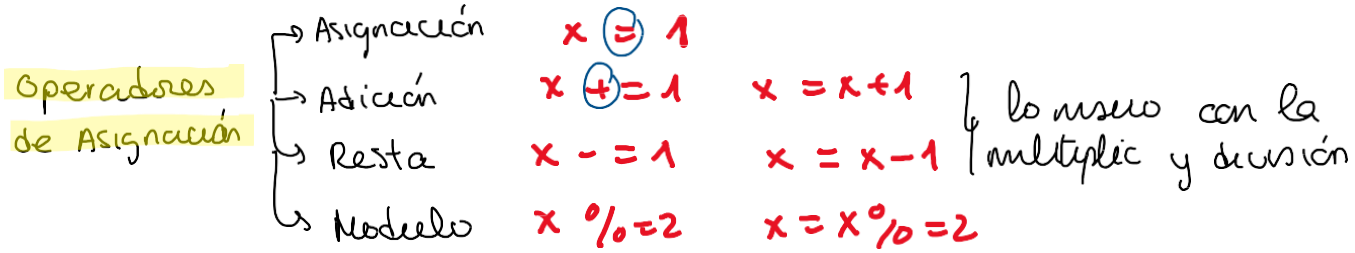
- **Str o strings**: cadenas de caracteres, se pueden indicar con comillas simples o dobles.
  Cuando queremos definir más de una línea, usamos 3 comillas dobles o simples.
  X = “hola mundo”
- **Float o ``números de punto flotante´´:** son números positivos o negativos con decimales.
  x = 0.5
- **Int o integrer:** son números enteros (sin decimales), positivos o negativos de longitud ilimitada.
  x = 7
- **Bool o boolean:** representan valores true o false.
  x = True

- Habrá veces que tendremos que **especificar un tipo de dato en una variable o tendremos que convertir (casting) un tipo de dato en otro.**
- **El casting** en Python se hace usando las funciones de construcción:
  Int() – conversión a entero.
  Float() – conversión a decimal.
  Str() – conversión a cadena de texto.

Tipos de datos – operadores.

Se usan para realizar operaciones con las variables y sus valores.





Operadores de Comparación		
Ejemplo	**Nombre**	**Operador**
x == y	**Igual**	==
x != y	**Distinto**	!=
x > 5	**Mayor que**	>
x < 2	**Menor que**	<
x >= 2	**Mayor o igual que**	>=
x <= 2	**Menor o igual que**	<=

Operadores de Lógicos		
Ejemplo	**Nombre**	**Operador**
x < 5 and x < 10	**Y**	**and**
x < 4 or x < 6	**O**	**or**
not (x == 5 or x == 6)	**Invierte el resultado si es verdadero**	**not**

2. Estructuras de control

Permiten modificar el flujo de ejecución de las instrucciones de un programa.

- De acuerdo con una condición, ejecutar un grupo u otro de sentencias (if).
- Ejecutar un grupo de sentencias hasta que se cumpla una condición (while).
- Ejecutar un grupo de sentencias un número determinado de veces (for).

Condicionales

- **Sentencia if:** permite ejecutar una o varias instrucciones cuando se cumple la condición establecida.

```
#sentencia if
x = 1
y= 2
if x<y:
    print ("Menor")
```

Su ejecución → la condición se evalúa siempre:

- Si el resultado es *True*, se ejecuta el bloque de sentencias.
- Si el resultado es *False*, no se ejecuta el bloque de sentencias.

- **Sentencia else:** establece un bloque de instrucciones que se ejecuta en caso de que no se cumplan las condiciones establecidas anteriormente.

```
x = 1
y= 2
if x<y:
    print ("Menor")
else:
    print ("Mayor")
```

Su ejecución → la condición se evalúa siempre:

- Si el resultado es *True*, se ejecuta el bloque de sentencias del if.
- Si el resultado es *False*, se ejecuta el bloque de sentencias del else.

- **Sentencia elif:** indica otras condiciones adicionales a la primera condición de la sentencia if.
- Necesita una sentencia if previa.
- Puede incluirse varios elif para evaluar distintas condiciones.

```
x = 1
y= 2
if x<y:
    print ("Menor")
elif x > y:
    print ("Mayor")
else:
    print ("Iguales")
```

- **Indentación:** el alcance de los bloques de código en python se establecen con la identación (tabular la línea de código) en vez de usar las llaves. Sirve para que el código sea más legible.

```
x = 3
y =3
if x == y:
    print ("Iguales")
```

Bucles o ciclos

- Secuencia que se repetirá x veces hasta que la condición asignada a ese bucle, deja de cumplirse.
- Bucles más usados: while, for y do – while.
- En python existen dos instrucciones de bucles primitivos; while y for.

- **While:** permite repetir la ejecución de un grupo de instrucciones mientras esa condición sea verdadera.

➔ **Resultado true:** se ejecuta el cuerpo del bucle, se repite el proceso evaluando de nuevo la condición, mientras la condición sea cierta.

➔ **Resultado false:** no se ejecuta el cuerpo del bucle y se ejecuta el resto del programa.

Esas variables que aparecen en la condición son **variables de control**. Tienen que definirse antes de iniciar el bucle while y modificarse dentro del while.

```
i = 1
while i < 6:
    print (i) #si terminasemos aqui el código, solo imprimiría 1. Se genera un bucle infito.
    i +=1 #se imprime por pantalla; 1, 2, 3, 4, 5.
```

```
# Definir el número positivo
num = 5

# Validar que el número ingresado sea positivo
while num <= 0:
    num += 1

# Imprimir los números del 1 al número ingresado
i = 1
while i <= num:
    print(i)
    i += 1
```

El número positivo es 5, pero puedes cambiarlo por cualquier otro número. La validación de que el número sea positivo se realiza en el ciclo while utilizando el operador de comparación <= (menor o igual). Si el número es menor o igual a cero, el ciclo while seguirá sumando 1 hasta que el número sea positivo. Una vez que el número es positivo, el código imprimirá los números del 1 al número ingresado utilizando un ciclo while.

- **For:** repite (iteración) el bloque (cuerpo del bucle) un número predeterminado de veces.

```
for i in range (6):
    print (i)
```

Break → se para el bucle antes de que finalicen los elementos iterables.

Los strings son objetos iterables, que tienen una secuencia de caracteres.

Elemento iterable → *es cualquier objeto que se puede iterar, es decir, que se puede recorrer uno por uno los elementos que lo componen. Por lo general, los elementos iterables en Python son objetos que contienen una secuencia de valores, como listas, tuplas, cadenas de caracteres, conjuntos y diccionarios.*

Para iterar sobre un elemento iterable en Python, se puede utilizar un bucle for. En cada iteración, el bucle toma uno de los elementos del iterable y realiza una acción sobre él.

Continue → se para la iteración del bucle y continuar con el siguiente.

Else → se ejecuta un bloque de código cuando haya finalizado.

- Bucles anidados:

- Es un bucle dentro de otro bucle.
- El “interno” se ejecutará una vez por cada iteración del bucle “externo”.

3. Tipos de datos avanzados

Lista → Conjunto ordenado, alterable de elementos (permite elementos duplicados: números, cadenas, listas).

Tupla → Conjunto ordenado y no alterable de elementos (permite elementos duplicados).

Set → Conjunto desordenado y no indexado de elementos (no permite elementos duplicados).

Diccionario → Conjunto desordenado, alterable e indexado de elementos (no permite elementos duplicados)

Listas

- Se delimitan por los corchetes [] y los elementos separados por las comas.

```
nombre_lista = ["elemento1", "elemento2", "etc"]
print (nombre_lista) #sintaxis
```

1) Pueden contener elementos del mismo tipo o diferente.

```
nombres = ["Pedro", 1, "Ana"]
```

2) También pueden contener otras listas.

```
lista_multiple = [["Thor", 2011],["Doctor Strange", 2016]]
```

3) Pueden tener muchos niveles de anidamiento.

```
lista_multiple = [["Carapapa",["Thoe",2011]],["Superman", [1999]]]
```

4) Es posible acceder directamente a cada elemento de una lista mediante el índice numérico.

```
list_names = ["Ana", "Pedro", "Ramón"]
print (list_names[2])
```

5) Podemos acceder a los elementos a través de la indexación negativa, comenzando desde el final de la lista, -1 se refiere al último elemento, -2 se refiere al segundo último elemento, etc.

```
list_names = ["Ana", "Pedro", "Ramón", “Sara”]
print (list_names[-2])
```

6) Se puede especificar un rango de índices, indicando donde comienza y dónde termina el rango. Cuando se especifica un rango, el valor de retorno será una nueva lista con los elementos especificados.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
print(lst_names[2:3])
```

** El primer índice es un 0. La búsqueda empezará en el primer índice indicado (incluido) y terminará en el último indicado (no incluido).

7) Dejando vacío el primer valor del rango, el rango comenzará en el primer elemento.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
print(lst_names[:3])
```

8) Dejando vacío el valor final del rango, el rango continuará hasta el final de la lista.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
print(lst_names[2:])
```

9) Obtener un rango con índices negativos, de forma que comenzará la búsqueda desde el final de la lista.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
print(lst_names[-4:-1])
```

10) Cambiar el valor de los elementos de la lista, haciendo referencia al índice del elemento.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
lst_names[2] = "José"
print(lst_names)
```

11) Comprobar la existencia de un elemento dentro de la lista:

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
if "Pedro" in lst_names:
  print("Existe!!")
```

12) Len () → permite obtener el nº de elementos que tiene la lista.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
print(len(lst_names))
```

13) Añadir elementos

a) **Append ()** → añadir un elemento al final de la lista.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
lst_names.append("Lucas")
print(lst_names)
```

b) **Insert ()** → añadir un elemento en el lugar indicado de la lista.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
lst_names.insert(2,"Lucas")
print(lst_names)
```

14) Eliminar elementos

a) **Remove ()** → eliminar un elemento especificado.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
lst_names.remove("Ramón")
print(lst_names)
```

b) Pop () → elimina el índice especificado, si no se especifica se elimina el último índice.

```
lst_names = ["Ana", "Pedro", "Ramón", "Sara"]
lst_names.pop(2)
print(lst_names)
```

c) Del → elimina el elemento del índice especificado. Sin el índice se puede eliminar toda la lista.

d) Clear () → vacía la lista.

¿Cómo hacemos una copia de una lista?

- **Lista2 = lista 1** → copia una referencia de la lista1 en la lista2, cada cambio en una lista se replica en la otra.
- **Copy ()** → realiza una copia de una lista en otra sin que una sea referencia de la otra. También podemos usar **list ().**

Otras operaciones.

- **Unir listas → + o extend ()**
- **Ordenar** → sort () orden alfabético y reverse () para un orden inverso.

Tuplas conjuntos ordenados y no alterables de elementos. Se delimitan por paréntesis () y los elementos se separan por comas.

Sintaxis →

```
nombre_tupla = (item1, item2, etc.)
```

```
tp_names = ("Ana", "Pedro", "Alex")
print(tp_names)
```

- Pueden contener elementos del mismo tipo o diferente.
- Se puede acceder a ellas con el índice numérico o mediante la indexación negativa.
- Podemos buscar por rangos, positivos o negativos.
- Son inalterables. Una vez creadas no se pueden cambiar valores ni borrarlos.
- Podemos convertirlas a una lista para cambiar los elementos de la tupla, y convertir la lista en tupla de nuevo.
- Se puede usar for para recorrer sus elementos.
- Si queremos comprobar la existencia de un elemento dentro de una lista, lo podemos hacer directamente.
- Len () para obtener el número de elementos que tiene una tupla.
- Si hay una tupla de un mismo elemento, debemos poner coma tras el elemento, sino la variable no será de tipo tupla, sino string.

- Unir listas → con +.
- Contar los elementos → count ().

Diccionarios

- Los diccionarios están formados por una clave y un valor.
- Las claves son siempre cadenas de texto (str) y los valores pueden ser del mismo tipo o de diferente.

```
dict1 = {
    "Edad":      18,
    "Nombre": "Xia",
    "Apellido1": "Martinez",
    "Apellido2": "Espinosa",
    "Altura": 1.82,
    "Aficiones": ["Guitarra", "Musica", "Correr"]
}
print (dict1)
#acceder a un elemento de un diccionario haciendo referencia al nombre de su
clave.
print (dict1["Aficiones"])

print (dict1.get("Aficiones")) #otra manera de acceder al elemento.

dict1 ["Altura"]= 1.83 #un diccionario es una estructura de datos alterable,
así se cambia un valor mediante su clave.

#podemos usar un diccionario para realizar iteraciones en un bucle for.
#cuando se recorre un diccionario, el valor que se imprime son las claves.

for x in dict1:
    print (x)
 #para poder devolver el valor de las claves, se deben usar los corchetes
for x in dict1:
    print (dict1[x])

for x in dict1.values ():
    print (x) #otra manera para devolver los valores.

#acceso a claves y valores con item ()
for x, y in dict1.items():
    print (x,y)

#¿cómo comprobamos que existe una clave dentro de un diccionario?

if "Nombre" in dict1:
    print ("Existe nombre")

#también podemos usar len () para obtener el número de elementos (pares clave
- valor) que contiene.
print (len(dict1))  #imprimirá por pantalla el número 6.

#para añadir elementos en un diccionario, hay que asignar una nueva clave y
su valor.

dict1 ["Correo"] = "correoprueba@gmail.com"
print (dict1)
```

```
#eliminar elementos en un diccionario:
dict1.pop ("Correo") #elimina el elemento con la clave especificada.

dict1.popitem() #elimina el úlimo elemento insertado.

del dict1 ("Correo") #elimina el elemento con la clave especificada.
del dict1 #también podemos usar del para borrar completamente el diccionario
si no especificamos la clave.

dict1.clear () #vacía el diccionario.

#copia de diccionarios

dict2 = dict1 #copia una referencia del dict1 en el dict2. Cada cambio en un
diccionario se replica en otro.
#copy () #realiza una copia de un diccionario en otro sin que sea una
referencia del anterior. También podemos usar dict().

#Anidar diccionarios

dic2 = {
    "dict2": {
    "nombre": "Alex",
    "apellidos": "Perez Lopez"
    },
    "dict3": {
    "nombre": "Paco",
    "apellidos": "Sierra Lopez"
    }
}
print (dict2)
```

Funciones

- Son bloques de código con un nombre asociado.
- Pueden recibir parámetros de entrada.
- Ejecutan una serie de instrucciones para realizar las acciones deseadas y devuelve un valor y/o realiza una tarea.

```
#definición de una función
def func1 ():
    print ("Hello from a function")

#la definición de funciones se realiza mediante def + nombre de la
función descriptivo.
def NOMBRE (LISTA_DE_PARÁMETROS):
    """DOCSTRING_DE_FUNCIÓN""" #cadena de texto usada para documentar la
función.
    SENTENCIAS #bloque de sentencias o instrucciones que realizarán la
función.
    RETURN [EXPRESION]
#return; sentencia de devolución de la función. No es obligatoria.
#expresión; expresión o variable que devuelve la función.
#los bloques de función deben de estar indentados correctamente, al igual
que los bloques de otrasestructuras de control.
```

El uso de funciones es fundamental. Ventajas:

- **Modularización**: permite segmentar un programa complejo en una serie de partes o módulos más simples.
- **Reutilización**: reutiliza una misma función en distintos programas.

Argumentos: Tantos argumentos como parámetros existan en la función. (Si hay uno de menos = error).

Devolver un valor del tipo que sea, se utiliza la sentencia **return**.

Parámetros de entrada; información que se pasa a las funciones, se especifican después del nombre de la función, entre los paréntesis. Si son varios parámetros hay que separarlos por comas.

Excepciones

- Error en el programa → mensaje de error y se finaliza el programa.
- Error = excepción. → En Python lo manejamos usando **try, except.** Permiten controlar el error y realizar acciones para mantener la ejecución del programa.

```
try:
    print (x)
except:
    print ("Excepción x no existe")
#cuando se establece un bloque de sentencias dentro de un tru, cualquier
excepción que se produzca hará que se ejecute el bloque except.
#sin esos bloques el programa se dentendrá y dará error.
```

- Podemos definir tantos **except** como queramos. **Lo haremos estableciendo el tipo de excepción.**
- Si no se produce ningún error usamos **else.**
- Si queremos ejecutar un código haya o no error lo haremos con **finally.**
- Tendremos que generar una excepción si se produce una condición o situación determinada. Para lanzar la excepción usaremos **raise.**

TEMA 4 - PROGRAMACIÓN ORIENTADA A OBJETOS

.

- Tipo de diseño de software.
- Los programadores definen el tipo de dato de una estructura de datos (objeto) y sus operaciones que se pueden aplicar a la estructura de datos.
- El control lo tienen los programadores.

Un objeto en POO

- ente abstracto que permite separar los diferentes componentes de un programa.
- integran procedimientos, variables y datos relacionados con el objeto.

Objeto; tiene 3 partes →

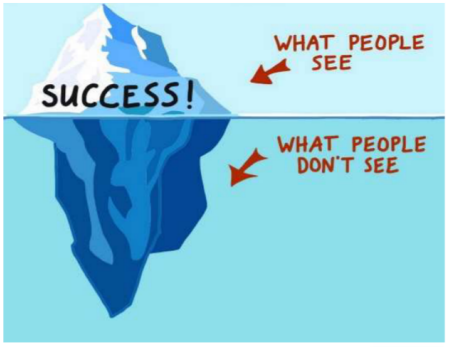
1) Métodos: funciones que permiten interactuar con los objetos.
2) Eventos: aquellas acciones donde el objeto reconoce que se está interactuando con él.
3) Atributos: datos que tiene el objeto.

¿Que es un objeto?
Cualquier representación de lo que queramos dentro de un programa.

Clase → se definen los atributos y sus métodos. Representación de una entidad o concepto.
Instancia → creación de un objeto de una clase determinada

Características principales de la Programación orientada a objetos OJO

- Abstracción → permite ocultar detalles de implementación. Tiene mucho que ver con la encapsulación.
- Encapsulación → deniega el acceso a los atributos y métodos internos de la clase desde el exterior.



- Polimorfismo → Polimorfo;
  - Sobrecarga (overload)
  - Sobreescritura (overriding): (marcada dentro de la herencia): puedes cambiar el método de la clase padre para cambiar su contenido en la clase hija.
- Herencia → organiza y facilita el polimorfismo y el encapsulamiento.
- Modularidad → propiedad que permite subdividir una aplicación en partes más pequeñas. (módulos → independientes).
- Recolección de basura →

TEMA 4 - PROGRAMACIÓN ORIENTADA A OBJETOS

Lenguajes orientados a objetos

- Los lenguajes de programación orientados a objetos, tratan los problemas como conjuntos de objetos que interactúan entre ellos para realizar acciones o tareas.
- Permite que los objetos sean más sencillos de escribir, mantener y favoreciendo la reutilización.
- Los objetos tienen información (atributos), los diferencia de otros pertenecientes a otra clase.
- Por medio de métodos se comunican con los objetos de una misma clase o una diferente produciendo el cambio de estado de los objetos.

Ventajas y desventajas:

Reusabilidad	Documentación extensa
Mantenibilidad	Complejidad
Modificabilidad	Diferentes interpretaciones
Fiabilidad	

- Python permite varios paradigmas de programación.
- POO → Codigo mas efectivo, organizado y reutilizable.
- **Todo es un objeto**

¿Cómo creamos una clase en Python?

```
class nombreclase:   #definimos la clase.
    propiedad = 1

#ahora debemos crearla o instanciarla.
obj = nombreclase()
print (type(object))
    #se crea un objeto de tipo nombreclase con todos los datos y
metodos definidos en la clase.
```

```
#Generar nuestra primera clase!!!!!
class Saludo:
    texto = "Hola"
    def saludar (self, name):
        print (self.texto + " " + name + "!!!!")

p1 = Saludo ()
p1.saludar ("Clase")
```

- __init__ → es el constructor de la clase, es el método al que se llamará cuando se instancie un objeto de esa clase.
  Se usa para inicializar los datos (atributos) de la clase o cualquier otra operación necesaria cuando el objeto es creado.

```
class Saludo:
    def __init__ (self,texto):
        self.texto = texto
    def saludar (self,name):
        print (self.tecto + " " + name + "!!!")
p1 = Saludo ("Hola")
p1.saludar ("Clase")
```

**Se pueden incluir métodos (funciones) para interactuar con el objeto.*
*se crean iguales las funciones pero pertenecen al objeto.**

Para acceder a un atributo o para ejecutar una función de un objeto → self.texto
¿Que es self? → referencia interior al objeto. (no hace falta que sea self, se puede usar cualquier nombre válido que se desee. **SIEMPRE DEBE SER EL PRIMER PARÁMETRO DE CADA FUNCIÓN EXISTENTE EN LA CLASE.**

```
class Saludo:
    def __init__ (mio,texto):
        mio.texto = texto
```

- Modificar una propiedad → cambiamos el valor de una propiedad accediendo mediante el objeto.

```
class Saludo:
    def __init__(self, nombre):
        self.texto = "Hola"
        self.nombre = nombre

    def saludar(self):
        print(self.texto + " " + self.nombre)

p1 = Saludo("clase")
p1.nombre = "Esteban"
p1.saludar()
```

- Borrar una propiedad → **del**

```
class Saludo:
    def __init__(self, nombre):
        self.texto = "Hola"
        self.nombre = nombre

    def saludar(self):
        print(self.texto + " " + self.nombre)

p1 = Saludo("clase")
print(dir(p1))
del p1.texto
print(dir(p1))
```

- Borrar un objeto → borrar el objeto entero

```
class Saludo:
    def __init__(self, nombre):
        self.texto = "Hola"
        self.nombre = nombre

    def saludar(self):
        print(self.texto + " " + self.nombre)

    def despedirse(self):
        print("Adios " + self.nombre)

p1 = Saludo("clase")
p1.saludar()
p1.despedirse()
```

- Llamar a una función → se puede llamar a una función que pertenezca a un objeto.

```
class Saludo:
    def __init__(self, nombre):
        self.texto = "Hola"
        self.nombre = nombre

    def saludar(self):
        print(self.texto + " " + self.nombre)

    def despedirse(self):
        print("Adios " + self.nombre)

p1 = Saludo("clase")
p1.saludar()
p1.despedirse()
```

*Una de las características de POO es la **encapsulación**, consiste en denegar el acceso a datos y/o métodos internos de la clase desde el exterior de la misma. La forma de hacer esto en Python es precediendo a los atributos y métodos de dos guiones bajos (_ _).*

Cualquier intento de acceder desde el exterior del objeto a estos atributos o métodos para visualizarlos, modificarlos o ejecutarlos provocará un error del tipo AttributeError.

TEMA 5 - HERENCIA Y POLIMORFISMO

Herencia → permite generar una estructura de clases que heredan información una de otras.

``Mecanismo por el cual una clase permite heredar las características (atributo y métodos) de otra clase´´

- **Clase padre o base:** clase de la que se hereda.

  class NombreClase:
  propiedad = 1

  Resultado por pantalla: “Xia Martinez”

- **Clase hija o derivada:** clase que se hereda de otra clase. (hereda tanto los métodos como los atributos).

  class NombreClase (ClasePadre): #Definición clase hija.

*Cuando queremos definir una clase, una función o un método de una clase que no haga nada (sin funcionalidad), se debe usar la palabra **pass**, (indica al intérprete de que la función no realiza nada). Todo esto evita un *IndentationError: expected an indented block.*

*Si no se define un constructor de la clase hija, el intérprete llamará al constructor de la clase padre. El constructor de la clase hija estará **sobreescribiendo (overriding)** el constructor de la clase padre.

- **super ()** → accedemos a los atributos y métodos de la clase padre.

```
class Alumno(Persona):

      def __init__ (self, nombre, apellidos):

            super().__init__(nombre, apellidos)
```

- En las clases hijas o derivadas, se pueden implementar tantos nuevos atributos como métodos. Sólo estarán disponibles en esta clase y en sus clases hijas, pero nunca en la clase padre.

- **Herencia múltiple:** en Python, es posible heredar de múltiples clases, la forma de realizar esto es similar a la forma de indicar la herencia de una clase pero se añadirán separados por comas tantas clases padre como se desee (ClasePadre1, ClasePadre2,... ClasePadreN).

```
class NombreClase(ClasePadre1, ClasePadre2):
    # Definición clase hija
```

- **¿Cómo se diferencia el comportamiento de una clase hija de su clase padre?** SOBRESCRIBIENDO MÉTODOS → Se sustituye el comportamiento del método de la clase padre por el de la hija.

- Para determinar si un objeto es una instancia de una clase o no, Python proporciona la función isinstance:

```
isinstance(Object,class_type)
```

- Determinar si una clase es una subclase (clase hija de otra):

```
issubclass(classinfo, classinfo)
```

Polimorfismo → Muchas formas.
Si en una porción de código se invoca un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto.
Distintos objetos pueden tener un método con un mismo nombre (firma), pero que realice diferentes operaciones.

¿Cómo se implementa? → Creando una clase base y teniendo una o varias clases derivadas que implementan métodos con el mismo nombre.
Cualquier otra función o método que manipule estos objetos puede llamar a los mismos métodos independientemente del tipo de objeto que se esté usando, sin tener una verificación del tipo antes.

```python
#Clase base
class Vehiculo:
    #Constructor
    def __init__(self):
        self.peso = 0
        self.largo = 0
        self.ancho = 0

    #Métodos
    def GetPeso(self):
        return self.peso

    def GetLargo(self):
        return self.largo

    def GetAncho(self):
        return self.ancho
```

```python
#Clase derivada
class Coche(Vehiculo):
    #Constructor
    def __init__(self, peso, largo, ancho, marca, modelo):
        super().__init__()
        #Inicialición datos
        self.SetPeso(peso)
        self.SetLargo(largo)
        self.SetAncho(ancho)
        self.marca = marca
        self.modelo = modelo

    #Métodos
    def SetPeso(self, peso):
        self.peso = peso

    def SetLargo(self, largo):
        self.largo = largo

    def SetAncho(self, ancho):
        self.ancho = ancho
```

Dos vías para implementar el polimorfismo.

- Sobrecarga (overload): dos métodos con el mismo nombre, tienen diferentes parámetros de entrada con una funcionalidad diferente.
- LA SOBRECARGA EN PYTHON no es factible si se implementa el mismo código en Python. Se obtiene el siguiente error:

TypeError: unMetodo() missing 1 required positional argument: 's'

```python
#Clase base
class Sobrecarga:
    #Métodos
    def unMetodo(self):
        print("Sobrecarga")

    def unMetodo(self, s):
        print(s)


s = "método sobrecargado"
#Instancia de Sobrecarga
sobre= Sobrecarga()
sobre.unMetodo()
sobre.unMetodo(s)
```

Simular una sobrecarga → Usamos parámetros opcionales y controlando el bloque de código del método la acción que se debe realizar en base.

```
#Clase base
class Sobrecarga:
    #Métodos
    def unMetodo(self, s = None):
        if s is not None:
            print(s)
        else:
            print("Sobrecarga")


s = "método sobrecargado"
#Instancia de Sobrecarga
sobre= Sobrecarga()
sobre.unMetodo()
sobre.unMetodo(s)
```

Sobreescritura (overriding) → dos métodos tienen el mismo nombre, pero pueden tener distinta cabecera y pueden realizar acciones distintas.

```
class Vehiculo(object):
    def __init__(self):
        self.peso = 0

    def getPeso(self):
        return self.peso

class Coche(Vehiculo):
    def getPeso(self):
        return self.peso + 2500

c = Coche()
print(c.getPeso())
```