

# Análisis de Datos con Python

Escuela de Código  
Pílares Topiltzin/EL Reloj

## Objetivo General del Curso

El estudiantado empleará los conocimientos en lenguaje de programación Python en los procesos de reunión, análisis e interpretación de datos a través de prácticas y ejercicios de programación.



# Temario

## **I. Introducción al lenguaje de programación**

- a. ¿Qué es Python?
- b. Instalación de Python

## **II. Números enteros y reales**

## **III. Operadores aritméticos**

- a. Booleanos
- b. Operadores lógicos
- c. Cadenas

## **IV. Tipos de colección de datos**

- a. Listas
- b. Tuplas
- c. Dicionarios

## **V. Operadores relacionales**

## **VI. Sentencias condicionales**

- a. Bucles
- b. Funciones

## **VII. Programación orientada a objetos**

- a. Clases
  - b. Objetos
  - c. Herencia
  - d. Herencia múltiple
  - e. Encapsulación
  - f. Polimorfismo
-



# Temario

## **VIII. Métodos de los objetos**

- a. Cadenas
- b. Listas
- c. Diccionarios

## **IX. Programación funcional**

- a. Función de orden superior
- b. MAP
- c. FILTER
- d. REDUCE
- e. Funciones lambda

## **X. Comprensión de listas**

## **XI. Generadores**

## **XII. Clases decoradoras**

## **XIII. Excepciones**

## **XIV. Entrada Estándar rawInput**

## **XV. Salida Estándar rawInput**

## **XVI. Módulos**

## **Instalación de un ambiente de desarrollo**

---

# I. Introducción al lenguaje de programación

Objetivo:

Conocer el lenguaje de programación python, sus características y sus aplicaciones.

**a. ¿Qué es Python?**

**b. Instalación de Python**

---

# ¿Qué es Python

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 cuyo nombre está inspirado en el grupo de cómicos ingleses “Monty Python”.

Posee una sintaxis muy limpia que favorece un código legible. Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos.

# Características

## **Lenguaje interpretado o de script**

Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados).

## **Fuertemente tipado**

No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o string) no podremos tratarla como un número (sumar la cadena “9” y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

# Características

## **Multiplataforma**

El intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

## **Orientado a objetos**

La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos. Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.



# Instalación

1.- Comprobar si la computadora ejecuta la versión 32 bits de Windows o la de 64.

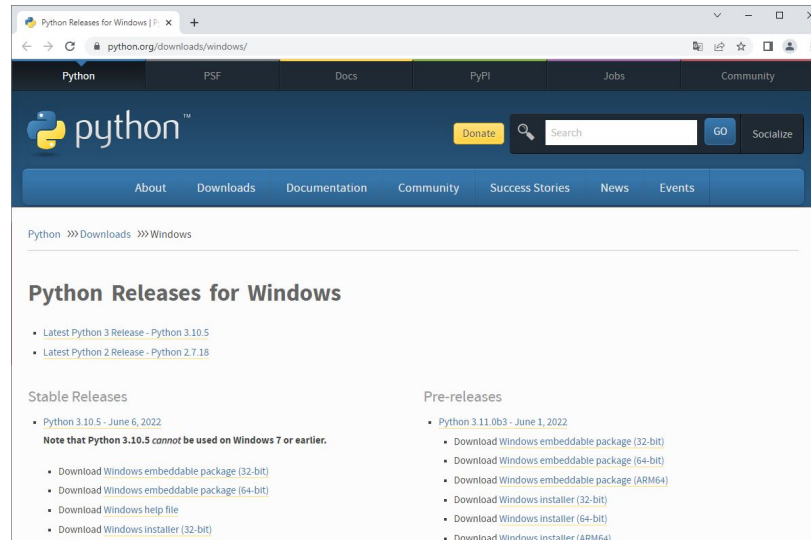
## Acerca de Especificaciones del dispositivo

### HP EliteDesk 705 G4 SFF

Nombre del dispositivo	DESKTOP-0Q8MQDH
Procesador	AMD Ryzen 3 PRO 2200G with Radeon Vega Graphics 3.50 GHz
RAM instalada	8.00 GB (6.93 GB utilizable)
Id. del dispositivo	D9818ABC-B3F7-48E1-8091-5154AFEF1226
Id. del producto	00330-52088-80388-AAOEM
Tipo de sistema	Sistema operativo de 64 bits, procesador x64
Lápiz y entrada táctil	La entrada táctil o manuscrita no está disponible para esta pantalla

# Instalación

2.- Acceder al sitio: <https://www.python.org/downloads/windows/>.



# Instalación

## 3.- Clic en: Latest Python 3 Release - Python X.XX.X

Python Release Python 3.10.5 | P...  
python.org/downloads/release/python-3105/

basically a resistance to decay against strong force and electromagnetism. This means that any particle that contains a strange quark can not decay due to strong force (or electromagnetism), but instead with the much slower weak force. It was believed that this was a 'strange' method of decay, which is why the scientists gave the particles that name.

[Full Changelog](#)

### Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
<a href="#">Gzipped source tarball</a>	Source release		d87193c077541e22f892ff1353fac76c	25628472	<a href="#">SIG</a>
<a href="#">XZ compressed source tarball</a>	Source release		f05727cb3489aa93cd57eb561c16747b	19361320	<a href="#">SIG</a>
<a href="#">macOS 64-bit universal2 installer</a>	macOS	for macOS 10.9 and later	cdc2e4c5a91477ae446689711c53aa72	40430804	<a href="#">SIG</a>
<a href="#">Windows embeddable package (32-bit)</a>	Windows		86be4156e8a5d5c9added8aab2bc83d1	7596969	<a href="#">SIG</a>
<a href="#">Windows embeddable package (64-bit)</a>	Windows		d97e3c0c7a19db2c5019f534bcb0b19	8558134	<a href="#">SIG</a>
<a href="#">Windows help file</a>	Windows		43c924ac87daed65acd85596eed1e33	9319556	<a href="#">SIG</a>
<a href="#">Windows installer (32-bit)</a>	Windows		eb59401a8da40051ec3b429897ae1203	27478768	<a href="#">SIG</a>
<a href="#">Windows installer (64-bit)</a>	Windows	Recommended	9a99ae597902b70b1273e88cc8d41abd	28637720	<a href="#">SIG</a>

About

Applications

Downloads

All releases

Documentation

Docs

Community

Diversity

Success Stories

Arts

News

Python News

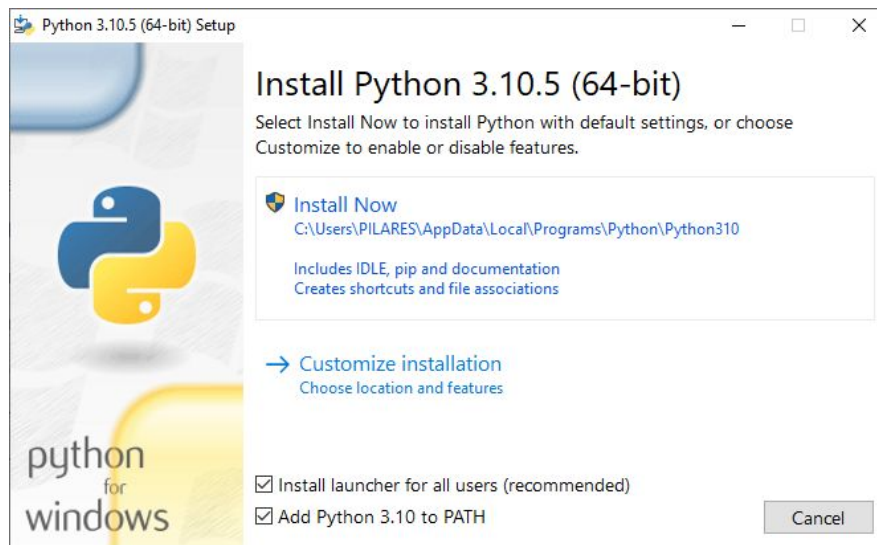
# Instalación

4.- Descargar Windows installer (64-bit) o Windows installer (32-bit) de acuerdo al SO.

Windows installer (32-bit)	Windows		eb59401a8da40051ec3b429897ae1203	27478768	SIG
Windows installer (64-bit)	Windows	Recommended	9a99ae597902b70b1273e88cc8d41abd	28637720	SIG

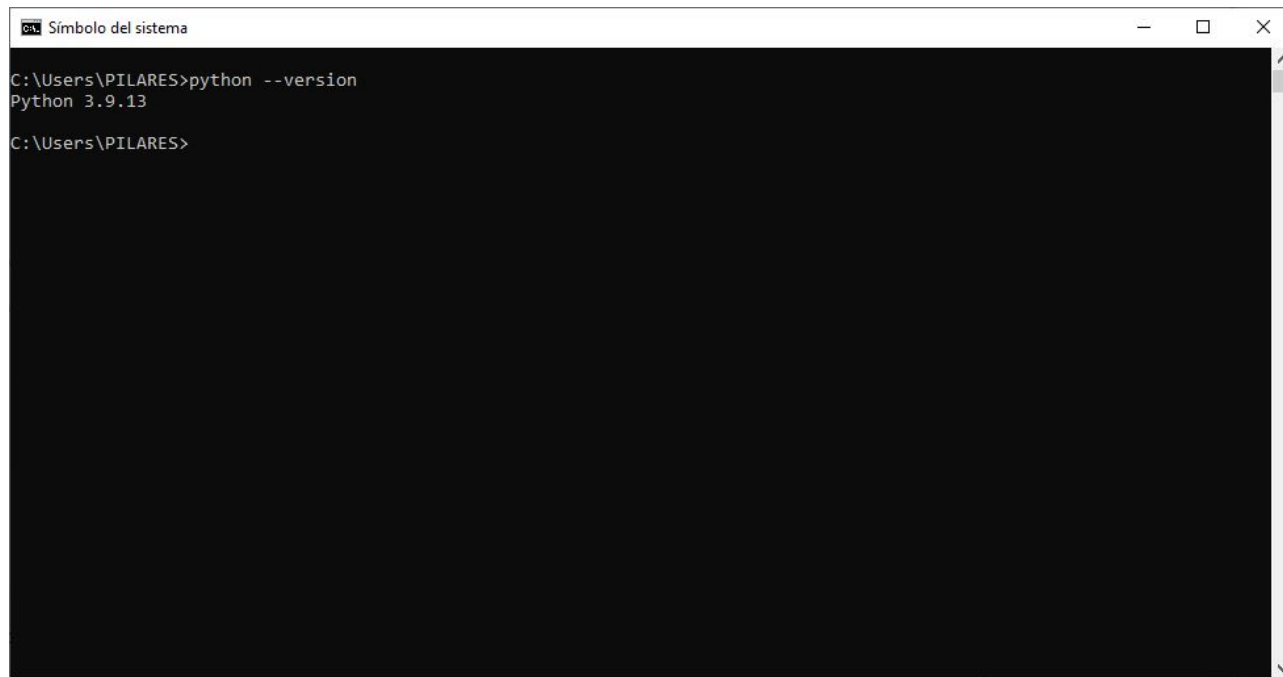
# Instalación

## 5.- Marcar casilla Add Python 3.10 to PATH



# Instalación

## 6.- Verificar instalación



```
Símbolo del sistema

C:\Users\PILARES>python --version
Python 3.9.13

C:\Users\PILARES>
```

## II. Tipos de datos

Objetivo:

Identificar los tipos de datos que existen en python y su manipulación en el lenguaje python.

**a. Enteros**

**b. Reales**

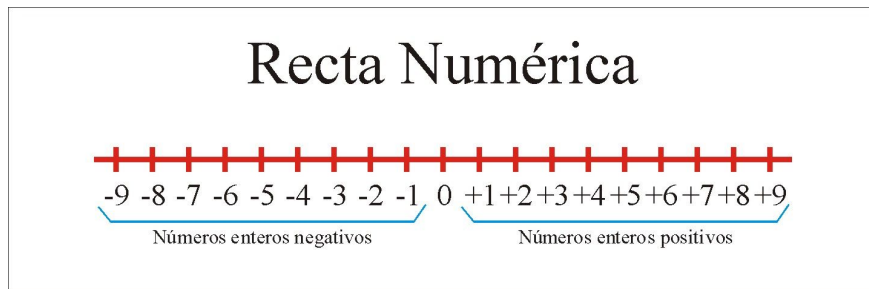
**c. Cadenas**

---

# Enteros

Este tipo de dato se corresponde con números enteros, es decir, sin parte decimal. En python se expresan mediante el tipo int.

Rango (-9.223.372.036.854.775.808  $\leftarrow \rightarrow$  9.223.372.036.854.775.808)





# Enteros

numInt = 5

numInt1 = 9 223 372 036 854 775 807

numLong = 9 223 372 036 854 775 808

# Flotantes

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo float.

Rango ( $2.2250738585072014\text{e-}308 - 1.7976931348623157\text{e+}308$ )

`numfloat= 5.5`

`numfloat1 = 1.7976931348623157e+308`

`numDouble = 1.7976931348623157e+309`

# Cadenas

Las cadenas no son más que texto encerrado entre comillas simples('cadena')o dobles (“cadena”). Dentro de las comillas se pueden añadir caracteres especiales marcandolos con \, como \n, el carácter de nueva línea, o \t, el de tabulación.

saludo = “Hola Mundo”

# Cadenas

También, es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter `\n`, así como las comillas sin tener que escaparlas.

```
triple = """primera linea
```

```
esto se vera en otra linea"""
```

# Concatenación

saludo = “Hola ”

nombre = “Oscar”

completo = saludo + nombre

print (completo)

# III. OPERADORES

Objetivo:

Identificar el uso de los operadores en Python.

- a. Aritméticos**
  - b. Relacionales**
  - c. Booleanos**
  - d. Lógicos**
-

# Operadores Aritméticos

Operador	Descripción	Ejemplo
+	Suma	$r = 3 + 2$ # $r$ es 5
-	Resta	$r = 4 - 7$ # $r$ es -3
-	Negación	$r = -7$ # $r$ es -7
*	Multiplicación	$r = 2 * 6$ # $r$ es 12
**	Exponente	$r = 2 ** 6$ # $r$ es 64
/	División	$r = 3.5 / 2$ # $r$ es 1.75
//	División entera	$r = 3.5 // 2$ # $r$ es 1.0
%	Módulo	$r = 7 \% 2$ # $r$ es 1

Para operaciones más complejas podemos recurrir al módulo math.

# Operadores Aritméticos

La diferencia entre división y división entera no es otra que la que indica su nombre.

En la división el resultado que se devuelve es un número real, mientras que en la división entera el resultado que se devuelve es solo la parte entera.

No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que queremos que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo,  $3 / 2$  y  $3 // 2$  sería el mismo: 1.

Si quisiéramos obtener los decimales necesitaríamos que al menos uno de los operadores fuera un número real, bien indicando los decimales.



# Operadores Relacionales

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	<pre>&gt;&gt;&gt; 5 == 3 False</pre>
!=	¿son distintos a y b?	<pre>&gt;&gt;&gt; 5 != 3 True</pre>
<	¿es a menor que b?	<pre>&gt;&gt;&gt; 5 &lt; 3 False</pre>
>	¿es a mayor que b?	<pre>&gt;&gt;&gt; 5 &gt; 3 True</pre>
<=	¿es a menor o igual que b?	<pre>&gt;&gt;&gt; 5 &lt;= 5 True</pre>
>=	¿es a mayor o igual que b?	<pre>&gt;&gt;&gt; 5 &gt;= 3 True</pre>

# Operadores lógicos

AND

$5 == 5$  and  $4 != 3$

OR

$5 != 5$  or  $4 == 3$

# Operadores Booleanos

Una variable de tipo booleano sólo puede tener dos valores: **True** (cierto) y **False** (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles.

El tipo bool (el tipo de los booleanos) es una subclase del tipo int.

```
estado1 = true
```

```
estado2 = false
```

# Operadores Booleanos

SÍMBOLO	DESCRIPCIÓN	EJEMPLO	BOOLEANO
==	IGUAL QUE	5 == 7	FALSE
!=	DISTINTO QUE	ROJO != VERDE	TRUE
<	MENOR QUE	8 < 12	TRUE
>	MAYOR QUE	12 > 7	TRUE
<=	MENOR O IGUAL QUE	16 <= 17	TRUE
>=	MAYOR O IGUAL QUE	67 >= 72	FALSE

# Programación Python

```
saludo = "Hola Mundo"
```

```
print("Hola Mundo")
```

```
saludo2 = str(input("Escribe un saludo: "))
```

```
print(saludo2)
```

# IV. TIPOS DE COLECCIÓN DE DATOS

Objetivo:

Identificar las colecciones de datos utilizadas en Python.

- a. Listas**
- b. Tuplas**
- c. Diccionesarios**

---

# Colección de datos

Definición:

Una colección representa un grupo de objetos, conocidos como sus elementos. Las colecciones permiten elementos duplicados y los mismos no están ordenados. Los elementos de una colección no son accedidos a través de un índice. Por el contrario su única forma de recuperación es uno a uno, secuencialmente.

Collection en inglés significa colección, conjunto, acopio, agregación.

La característica fundamental de las colecciones es que aceptan elementos duplicados.

# Listas

La lista es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por arrays, o vectores.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos y también listas.

Crear una lista es tan sencillo como indicar entre corchetes, y separados por comas, los valores que queremos incluir en la lista:

```
l = [22, True, "una lista", [1, 2]]
```



# Listas

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes. Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1:

```
l = [11, False]
```

```
mi_var = l[0] # mi_var vale 11
```

# Listas

Si queremos acceder a un elemento de una lista incluida dentro de otra lista tendremos que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:

```
l = [22, True, "una lista", [1, 2]]
```

```
l[3][0] #Muestra el valor 1
```

```
l[3][1] #Muestra el valor 2
```

# Listas

También podemos utilizar este operador para modificar un elemento de la lista si lo colocamos en la parte izquierda de una asignación:

```
l = [22, True]
```

```
l[0] = 99 # Con esto l valdrá [99, True]
```

El uso de los corchetes para acceder y modificar los elementos de una lista es común en muchos lenguajes, pero Python nos depara varias sorpresas muy agradables.

# Listas

Una curiosidad sobre el operador `[]` de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con `[-1]` accederemos al último elemento de la lista, con `[-2]` al penúltimo, con `[-3]`, al antepenúltimo, y así sucesivamente.

# listas

```
>>> x = [1,2,3,4,5,6,7,8,9,10]
```

```
>>> x[:]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> x[0:]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> x[:0]
```

```
[]
```

```
>>> x[9:]
```

```
[10]
```

```
>>> x[:9]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> x[0:9]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> x[0:10]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> x[-1]
```

```
10
```

```
>>> x[-10]
```

```
1
```

```
>>> x[:-10]
```

```
[]
```

```
>>> x[-10:]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Operaciones en Listas

## 1. Operaciones de definición

Estas operaciones nos permiten definir o crear una lista.

### 1.1. []

Crea una lista vacía.

```
y = []
```

## 2. Operaciones mutables

Estas operaciones nos permiten trabajar con listas alterando o modificando su definición previa.

# Operaciones en Listas

## **append**

Añade un único elemento al final de la lista.

```
>>> x = [1,2]
```

```
>>> x.append("Uno")
```

```
>>> print(x)
```

```
[1, 2, 'Uno']
```

# Operaciones en Listas

## **extend**

Añade otra lista al final de una lista.

```
>>> x = [1, 2]
```

```
>>> x.extend([3, 4])
```

```
>>> print(x)
```

```
[1, 2, 3, 4]
```



# Operaciones en Listas

## **insert**

Inserta un nuevo elemento en una posición determinada de la lista, este método recibe un primer argumento que equivale a la posición, y un segundo argumento que corresponde al elemento por agregar.

```
>>> x = [1, 2]
```

```
>>> x.insert(0, "y")
```

```
>>> print(x)
```

```
['y', 1, 2]
```

# Operaciones en Listas

## **del**

Elimina el elemento ubicado en el índice descrito, a su vez este método tiene la capacidad de eliminar una sección de elementos de la lista, mediante el operador “:” que permite definir un punto de [inicio:fin] sin considerar el punto final, esto puntos pueden ser obviados, con lo cual como punto de inicio se tomará la posición 0 y punto de fin la última posición de la lista.

```
>>> x = [1, 2, 3]
```

```
>>> del x[1]
```

```
>>> print(x)
```

```
[1, 3]
```

```
>>> y = [1,2,3,4,5]
```

```
>>> del y[:2]
```

```
>>> print(y)
```

```
[3, 4, 5]
```

# Operaciones en Listas

## **remove**

Remueve la primer coincidencia del elemento especificado.

```
>>> x = [1,2,"h",3,"h"]
```

```
>>> x.remove("h")
```

```
>>> print(x)
```

```
[1, 2, 3, 'h']
```

# Operaciones en Listas

## **reverse**

Invierte el orden de los elementos de la lista, dando como resultado una lista con los elementos del final al inicio y del inicio al final.

```
>>> x = [1,2,3,4,5,6,7,8,9]
>>> x.reverse()
>>> print(x)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

# Operaciones en Listas

## **sort**

Por defecto este método ordena los elementos de la lista de menor a mayor, este comportamiento puede modificarse mediante el parámetro `reverse=True`.

```
>>> x = [4,2,3,1]
>>> x.sort()
>>> print(x)
[1, 2, 3, 4]
>>> y = [14,12,13,11]
>>> y.sort(reverse=True)
>>> print(y)
[14, 13, 12, 11]
```

# Operaciones en Listas

## 3. Operaciones inmutables

Estas operaciones nos permiten trabajar con listas sin alterar o modificar su definición previa.

### **sorted**

Este método nos permite crear una nueva lista ordenada a partir de otra lista no ordenada.

Algo interesante del método `sorted`, es que este no se encuentra limitado a las listas, sino que a su vez es aplicable a las tuplas y los diccionarios, algo que en otro artículo estaremos abordando.

```
>>> lista1 = [2,5,8,1,4,5,8,3,2,5]
```

```
>>> lista2 = sorted(lista1)
```

```
>>> print(lista1)
```

```
[2, 5, 8, 1, 4, 5, 8, 3, 2, 5]
```

```
>>> print(lista2)
```

```
[1, 2, 2, 3, 4, 5, 5, 5, 8, 8]
```

# Operaciones en Listas

+

Esta operación nos permite concatenar o unir dos listas diferentes en nueva lista.

```
>>> x = [1,2,3,4,5]
```

```
>>> y = [6,7,8,9,10]
```

```
>>> print(x+y)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Operaciones en Listas

\*

Esta operación nos permite replicar una lista hasta la cantidad de veces indicada.

```
>>> x = [1,2,3]
```

```
>>> print(x * 3)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```



# Operaciones en Listas

## **min**

Este método retorna el elemento más pequeño dentro de una lista.

## **max**

Al contrario del metodo min, este método retorna el elemento más grande dentro de una lista.

```
>>> x = [40,100,3,9,4]
```

```
>>> print(min(x))
```

```
3
```

```
>>> print(max(x))
```

```
100
```

```
>>> print(min(x),max(x))
```

```
3 100
```

# Operaciones en Listas

## **index**

Retorna la posición en la lista del elemento especificado.

```
>>> x = [10,30,20]  
>>> print(x.index(30))  
1
```

# Operaciones en Listas

## **count**

Retorna la cantidad de veces que el elemento especificado se encuentra en la lista.

```
>>> x = [10,30,20,30,30,20,10]  
>>> print(x.count(30))  
3
```

# Operaciones en Listas

## **sum**

Este método realiza la suma de los elementos de la lista, esto siempre que los mismos puedan ser sumados. Sum es un método muy utilizado con listas de tipo numéricas.

```
>>> x = [2.5,3,3.5]  
>>> print(sum(x))  
9.0
```

# Operaciones en Listas

## **in**

El método nos permite determinar si un elemento específico se encuentra en una lista, este retorna únicamente dos posibles valores True cuando el elemento se encuentre en la lista, y False cuando este no lo esté. Este método es ampliamente utilizado para evitar excepciones en métodos tales como: index y remove, ya que en caso que el elemento buscado no se encuentre en la lista, dará como resultado una excepción.

```
>>> x = ["h",2,"a",6,9]
```

```
>>> print("a" in x)
```

```
True
```

# Operaciones en Listas

## **len**

La función len devuelve la longitud (el número de elementos) de un objeto.

```
>>> x = ["h",2,"a",6,9]
```

```
>>> print(len(x))
```

```
5
```

# Duplas

Todo lo que hemos explicado sobre las listas se aplica también a las tuplas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

```
t = (1, 2, True, "python")
```

En realidad el constructor de la tupla es la coma, no el paréntesis, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, con claridad.

```
>>> t = 1, 2, 3
```

```
>>> type(t)
```

```
type "tuple"
```

# Duplas

Para referirnos a elementos de una tupla, como en una lista, se usa el operador `[]`:

```
mi_var = t[0] # mi_var es 1  
mi_var = t[0:2] # mi_var es (1, 2)
```

Podemos utilizar el operador `[]` debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados secuencias. Permitirme un pequeño inciso para indicar que las cadenas de texto también son secuencias, por lo que no extrañará que podamos hacer cosas como estas:

```
c = "hola mundo"  
c[0] # h  
c[5:] # mundo  
c[:3] # hauo
```



# Diferencias Listas/Duplas

La diferencia es que las listas presentan una serie de funciones adicionales que permiten un amplio manejo de los valores que contienen. Basándonos en esta definición, puede decirse que las listas son dinámicas, mientras que las tuplas son estáticas.

# Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor. Por ejemplo, veamos un diccionario de películas y directores:

`d = {"Love Actually ": "Richard Curtis",`

`"Kill Bill": "Tarantino",`

`"Amélie": "Jean-Pierre Jeunet"}"`

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, ... pero no listas o diccionarios, dado que son mutables.

# Diccionarios

Esto es así porque los diccionarios se implementan como **tablas hash**, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

```
d["Love Actually"] # devuelve "Richard Curtis"
```

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

```
d["Kill Bill"] = "Quentin Tarantino"
```

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los diccionarios no son secuencias, si no mappings (mapeados, asociaciones).

# V. OPERADORES RELACIONALES

Objetivo:

Utilizar operadores relacionales  
en expresiones lógicas en  
Python

**a. Operadores**

---

# Tabla

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	<pre>&gt;&gt;&gt; 5 == 3 False</pre>
!=	¿son distintos a y b?	<pre>&gt;&gt;&gt; 5 != 3 True</pre>
<	¿es a menor que b?	<pre>&gt;&gt;&gt; 5 &lt; 3 False</pre>
>	¿es a mayor que b?	<pre>&gt;&gt;&gt; 5 &gt; 3 True</pre>
<=	¿es a menor o igual que b?	<pre>&gt;&gt;&gt; 5 &lt;= 5 True</pre>
>=	¿es a mayor o igual que b?	<pre>&gt;&gt;&gt; 5 &gt;= 3 True</pre>

# VI. Estructuras de control en Python

Objetivo:

Conocer y utilizar las estructuras condicionales e iterativas con Python.

- a. if/else
- b. while
- c. do while
- d. for

---

# Estructuras de Control

Las estructuras de control se utilizan para controlar el flujo de un programa (o bloque de instrucciones), son métodos que permiten especificar el orden en el cual se ejecutarán las instrucciones en un algoritmo. Si no existieran las estructuras de control, los programas se ejecutarían linealmente desde el principio hasta el fin, sin la posibilidad de tomar decisiones.

Por lo general, en la mayoría de lenguajes de programación encontraremos dos tipos de estructuras de control. Encontraremos un tipo que permite la ejecución condicional de bloques del programa y que son conocidas como estructuras condicionales. Por otro lado, encontraremos las estructuras iterativas que permiten la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición.

Aquí es donde cobran su importancia el tipo booleano y los operadores lógicos y relacionales.

# if/else

Una sentencia if en Python esencialmente indica:

"Si la expresión evaluada, resulta ser verdadera(True), **entonces ejecuta una vez el código** en la expresión. Si sucede el caso contrario y la expresión es falsa, **entonces no ejecutes el código que sigue.**"

Una sentencia if consiste en:

- La palabra reservada if , da inicio al condicional if .
- La siguiente parte es la condición. Esta puede evaluar si la declaración es verdadera o falsa. En Python estas son definidas por las palabras reservadas (True or False).
- Paréntesis (()) Los paréntesis son opcionales, no obstante, ayudan a mejorar la legibilidad del código cuando más de una condición está presente.
- Dos puntos : cuya función es separar la condición de la declaración de ejecución siguiente.
- Una nueva línea.
- Un nivel de indentación de cuatro espacios, que es una convención en Python. El nivel de indentación es asociado con la estructura de la declaración que sigue.
- Finalmente, la estructura de la sentencia. Este es el código que será ejecutado, únicamente si la sentencia a ser evaluada es verdadera. Es posible tener múltiples líneas en la estructura de código que pueden ser ejecutadas; en este caso es necesario tener cautela en cuanto a que todas las líneas tengan el mismo nivel de indentación.



# if/else

Estructura básica

```
if (True):  
    print("La expresión es Verdadera")
```

Una sentencia if ejecuta el código, solo en caso de cumplirse la condición especulada.  
¿Qué sucede si queremos ejecutar el código alternativo en caso de no cumplirse las condiciones? En este caso es necesario usar else.

```
if condicion:  
    ejecutar codigo si la condicion es True  
else:  
    ejecutar codigo si la condicion es False
```

**elif**

## **¿Qué si necesitamos más de dos opciones?**

En vez de decir: "Si la primera condición es verdadera, realiza esto, si no, realiza esto otro", ahora le indicamos al programa, "Si esto no es verdadero, intenta esto otro, y si todas las condiciones fallan en ser verdaderas, entonces haz esto.

```
if condicion1:  
    ejecutar codigo  
elif condicion2:  
    ejecutar codigo  
else:  
    ejecutar codigo
```

# Ciclos

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los ciclos nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

# while

El ciclo **while** permite ejecutar un bloque de instrucciones mientras se cumpla la condición dada. Primero comprueba que en efecto se cumple la condición dada y entonces, ejecuta el segmento de código correspondiente hasta que la condición no se cumpla.

Estructura básica:

```
numero = 0
while numero <= 10:
    print (numero)
    numero += 1
```

```
numero = 0
while numero <= 10:
    print (random.randint(1,10))
    numero += 1
```

# while

Estructura básica:

```
import random
```

```
numero = 0
```

```
while numero <= 10:
```

```
    print (random.randint(1,10))
```

```
    numero += 1
```

## do while

La sentencia do while es una estructura de repetición o iterativa que se utiliza en muchos lenguajes de programación.

Cuando utilizamos este bucle nos aseguramos que las instrucciones se ejecuten por lo menos una vez.

Estructura básica:

# for

Los ciclos `for` permiten ejecutar una o varias instrucciones de forma iterativa, una vez por cada elemento en la colección.

Las colecciones pueden ser de varios tipos, el `for` puede recibir una colección predefinida o directamente de la salida de una función.

Estructura básica:

```
for contador in range(1,10):  
    print (contador)
```

# Iteraciones sobre una lista (for)

Los ciclos for se utilizan en Python para recorrer secuencias, por lo que vamos a utilizar un tipo de secuencia lista.

Leamos la cabecera del bucle como si de lenguaje natural se tratara: “para cada elemento en secuencia”. Y esto es exactamente lo que hace el bucle: para cada elemento que tengamos en la secuencia, ejecuta estas líneas de código. Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia y almacenarlo en una variable de nombre elemento. Por esta razón en la primera iteración del bucle elemento valdrá “uno”, en la segunda “dos”, y en la tercera “tres”.

Estructura básica:

```
numeros = ["uno", "dos", "tres"]
```

```
for indice in numeros:  
    print (indice)
```



# Ejercicios

- 1.- Poblar una lista con 100 números (reales) aleatorios
- 2.- Poblar una lista con 10 números aleatorios y obtener la suma total
- 3.- Poblar una lista con datos introducidos en el teclado
- 4.- Poblar una lista de tuplas pares, que contengan Nombre y edad de personas.

# Funciones

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos.

En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java. Además de ayudarnos a programar y depurar dividiendo el programa en partes las funciones también permiten reutilizar código. En Python las funciones se declaran de la siguiente forma:

# Funciones

## Estructura

básica:

1. La palabra clave def
2. Un nombre de función
3. Paréntesis '()', y dentro de los paréntesis los parámetros de entrada, aunque los parámetros de entrada sean opcionales. (Argumentos)
4. Dos puntos ':'
5. Algún bloque de código para ejecutar
6. Una sentencia de retorno (opcional)

```
def mi_funcion(param1, param2):  
    print param1  
    print param2  
    return
```

# Funciones

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de docstring (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):  
    """Esta función imprime los dos valores pasados como parámetros"""  
    print param1  
    print param2  
    return
```

# **VII. Programación orientada a objetos**

Objetivo:

Introducción a POO con  
lenguaje de programación  
Python.

- a. Clases**
  - b. Objetos**
  - c. herencia**
  - d. herencia múltiple**
  - e. encapsulación**
-

# POO

## ¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

# P00

Con el paradigma de **Programación Orientado a Objetos** lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

# P00

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores DRY (Don't Repeat Yourself), para evitar duplicar el código y crear de esta manera programas eficientes. Además, evita el acceso no deseado a los datos o la exposición de código propietario mediante la encapsulación y la abstracción.



# POO

¿Cómo se crean los programas orientados a objetos? En resumen, consiste en hacer clases y crear objetos a partir de estas clases. Las clases forman el modelo a partir del que se estructuran los datos y los comportamientos.

El primer y más importante concepto de la POO es la **distinción entre clase y objeto**.

# Objetos y Clases

Una **clase** es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse 'animal' y tener una serie de **atributos**, como 'nombre' o 'edad' (que normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

# Clases y Objetos

Las **clases** proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo *tipo* de objeto, permitiendo crear nuevas *instancias* de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Un **objeto** es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.

# Sintaxis

Para crear una clase vamos a emplear la palabra reservada `class` seguido de un nombre escrito en minúscula, a excepción de la primera letra de cada palabra, que se escribe en mayúscula, y sin guiones bajos. (Clase vacía)

```
class Alumno:  
    pass
```

# Sintaxis

Sabemos que las clases pueden contener funciones, a las que llamamos *métodos*. Para ello vamos a usar la misma nomenclatura que aprendimos en el apartado anterior, con la diferencia que esta vez todo nuestro código estará indentado cuatro espacios, para indicar que queremos ubicarlo dentro de la clase.

```
class Alumno:

    def saludar(self):
        """Imprime un saludo en pantalla."""
        print("¡Hola, mundo!")
```

Todas las funciones definidas dentro de una clase deberán tener, al menos, un argumento, que por convención se le llama `self` y es una referencia a la *instancia* de la clase.

# Sintaxis

Creamos una instancia de nuestra clase.

```
alumno = Alumno()
```

```
alumno.saludar()
```

# Sintaxis

Una clase también puede contener variables, a las que se conoce con el nombre de *atributos*. Para crear atributos definimos un método especial llamado `__init__()`, que es invocado por Python automáticamente siempre que se crea una instancia de la clase (conocido también como *constructor* o *inicializador*).

```
class Alumno:
```

```
    def __init__(self):  
        self.nombre = "Pablo"
```

```
    def saludar(self):  
        """Imprime un saludo en pantalla."""  
        print(f"¡Hola, {self.nombre}!")
```

```
alumno = Alumno()
```

# Sintaxis

Para el código anterior es conveniente permitir que, al definir una instancia, se pase como argumento el nombre del alumno y éste se almacene en el atributo `self.nombre`.

```
class Alumno:

    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        """Imprime un saludo en pantalla."""
        print(f"¡Hola, {self.nombre}!")

alumno = Alumno("Pablo")
alumno.saludar()
```



# Herencia

La herencia define **relaciones jerárquicas entre clases**, de forma que atributos y métodos comunes puedan ser reutilizados. Las clases principales extienden atributos y comportamientos a las clases secundarias. A través de la definición en una clase de los atributos y comportamientos básicos, se pueden crear clases secundarias, ampliando así la funcionalidad de la clase principal y agregando atributos y comportamientos adicionales.

# Herencia

La herencia es una herramienta fundamental para la orientación a objetos. Permite definir jerarquías de clases que comparten diversos métodos y atributos. Por ejemplo, consideremos la siguiente clases:

```
class Rectangulo:
    """
    Define un rectángulo según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def area(self):
        return self.b * self.h

rectangulo = Rectangulo(20, 10)
print("Área del rectángulo: ", rectangulo.area())
```

```
class Triangulo:
    """
    Define un triángulo según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def area(self):
        return (self.b * self.h) / 2
```

# Herencia

Los códigos son muy similares, a excepción del método `area()`. Pero dado que el método `__init__()` es el mismo, podemos abstraerlo en una *clase padre* de la cual *hereden* tanto Rectángulo como Triángulo.

```
class Poligono:
    """
    Define un polígono según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h
```

```
class Rectangulo(Poligono):

    def area(self):
        return self.b * self.h
```

```
class Triangulo(Poligono):

    def area(self):
        return (self.b * self.h) / 2
```

```
rectangulo = Rectangulo( 20, 10)
triangulo = Triangulo( 20, 12)
```

```
print("Área del rectángulo: " , rectangulo.area())
print("Área del triángulo:" , triangulo.area())
```

# Herencia

Una misma clase puede heredar de varias clases (*herencia múltiple*); en ese caso, se especifican los nombres separados por comas.

```
class ClaseA(ClaseB, ClaseC, ClaseD):  
    pass
```

# Encapsulación

La encapsulación contiene **toda la información importante de un objeto dentro del mismo** y solo expone la información seleccionada al mundo exterior.

Esta propiedad permite asegurar que la información de un objeto esté oculta para el mundo exterior, agrupando en una Clase las características o atributos que cuentan con un acceso privado, y los comportamientos o métodos que presentan un acceso público.

La encapsulación de cada objeto es responsable de su propia información y de su propio estado. La única forma en la que este se puede modificar es mediante los propios métodos del objeto. Por lo tanto, los atributos internos de un objeto deberían ser **inaccesibles desde fuera**, pudiéndose modificar sólo llamando a las funciones correspondientes. Con esto conseguimos mantener el estado a salvo de usos indebidos o que puedan resultar inesperados.

Usamos de ejemplo un coche para explicar la encapsulación. El coche comparte información pública a través de las luces de freno o intermitentes para indicar los giros (interfaz pública). Por el contrario, tenemos la interfaz interna, que sería el mecanismo propulsor del coche, que está oculto bajo el capó. Cuando se conduce un automóvil es necesario indicar a otros conductores tus movimientos, pero no exponer datos privados sobre el tipo de carburante o la temperatura del motor, ya que son muchos datos, lo que confundiría al resto de conductores.

# Abstracción

La abstracción es cuando **el usuario interactúa solo con los atributos y métodos seleccionados de un objeto**, utilizando herramientas simplificadas de alto nivel para acceder a un objeto complejo.

En la programación orientada a objetos, los programas suelen ser muy grandes y los objetos se comunican mucho entre sí. El concepto de abstracción **facilita el mantenimiento de un código de gran tamaño**, donde a lo largo del tiempo pueden surgir diferentes cambios.

Así, la abstracción se basa en usar **cosas simples para representar la complejidad**. Los objetos y las clases representan código subyacente, ocultando los detalles complejos al usuario. Por consiguiente, supone una extensión de la encapsulación. Siguiendo con el ejemplo del coche, no es necesario que conozcas todos los detalles de cómo funciona el motor para poder conducirlo.

# Poliformismo

El polimorfismo consiste en **diseñar objetos para compartir comportamientos**, lo que nos permite procesar objetos de diferentes maneras. Es la capacidad de presentar la misma interfaz para diferentes formas subyacentes o tipos de datos.

Al utilizar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute diferentes comportamientos de dos formas: anulación de método y sobrecarga de método.

# Getters / Setters

Los “Getters” y “Setters” se utilizan en POO para garantizar el principio de la encapsulación de datos.

Claramente el **getter** se emplea para obtener los datos y el **setter** para cambiar el valor de los datos. Son decoradores y se identifican por tener un @.

Por lo general, estos se usan en Python:

- Si queremos añadir una lógica de validación para obtener y establecer un valor.
- Para evitar el acceso directo a un atributo de clase (un usuario externo no puede acceder directamente a las variables privadas ni modificarlas).



# “Getters / Setters” Syntax

```
class Humano:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
        #self.valEdad()

    @property
    def listaNombre(self):
        return self.nombre

    @listaNombre.setter
    def listaNombre(self, nuevoNombre):
        self.nombre = nuevoNombre

    @property
    def listaEdad(self):
        return self.edad

    @listaEdad.setter
    def listaEdad(self, nuevaEdad):
        self.edad = nuevaEdad
        #self.valEdad()
```

# VIII. Métodos de los Objetos

Objetivo:

Implementar las funciones que integra Python para el proceso de información

**a. Cadenas**

**b. Listas**

**c. Diccionarios**

---

# Cadenas (String)

`cadena.count(sub[, start[, end]])`

Devuelve el número de veces que se encuentra sub en la cadena. Los parámetros opcionales start y end definen una subcadena en la que buscar.

```
>>> mensaje = "Hola Mundo, Escuela deCodigo"
>>> mensaje.count("o")
4
>>> mensaje.count("o", 0, 9)
1
>>> mensaje.count("o", 0, 10)
2
```

# Cadenas (String)

`cadena.find(sub[, start[, end]])`

Devuelve la posición en la que se encontró por primera vez sub en la cadena o -1 si no se encontró.

```
>>> mensaje = "Hola Mundo, Escuela deCodigo"
>>> mensaje.find("a")
3
>>> mensaje.find("a", 0, 2)
-1
```

# Cadenas (String)

`cadena.join(sequence)`

Devuelve una cadena resultante de concatenar las cadenas de la secuencia que se encuentran separadas por la cadena sobre la que se llama el método.

```
>>> list1 = ['Hola', 'Escuela', 'de', 'Codigo']
>>> s = "-"
>>> s = s.join(list1)
>>> s
'Hola-Escuela-de-Codigo'
```

# Cadenas (String)

## cadena.partition(sep)

Busca el separador sep en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.

```
>>> mensaje = "Hola Escuela de codigo"
>>> mensaje.partition('de')
('Hola Escuela ', 'de', ' codigo')
>>> mensaje.partition(' de ')
('Hola Escuela', ' de ', 'codigo')
>>> mensaje.partition('Escuela')
('Hola ', 'Escuela', ' de codigo')
>>> mensaje.partition('No')
('Hola Escuela de codigo', '', '')
```

# Cadenas (String)

`cadena.replace(old, new[, count])`

Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena `old` por la cadena `new`. Si se especifica el parámetro `count`, este indica el número máximo de ocurrencias a reemplazar.

```
>>> mensaje = "Hola Escuela deCodigo"
>>> mensaje.replace("Escuela", "Pilares")
'Hola Pilares deCodigo'
>>> mensaje = "Hola Escuela deCodigo,Hola Escuela deCodigo,Hola Escuela deCodigo"
>>> mensaje.replace("Escuela", "PILARES")
'Hola PILARES deCodigo,Hola PILARES deCodigo,Hola PILARES deCodigo'
>>> mensaje = "Hola Escuela deCodigo,Hola Escuela deCodigo,Hola Escuela deCodigo"
>>> mensaje.replace("Escuela", "PILARES", 1)
'Hola PILARES deCodigo,Hola Escuela deCodigo,Hola Escuela deCodigo'
```

# Cadenas (String)

`cadena.split([sep [,maxsplit]])`

Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividir las por el delimitador `sep`. En el caso de que no se especifique `sep`, se usan espacios. Si se especifica `maxsplit`, este indica el número máximo de particiones a realizar.

```
>>> mensaje = "Hola Escuela deCodigo"
>>> mensaje.split()
['Hola', 'Escuela', 'de', 'Codigo']
```



# Listas

## **L.append(object)**

Añade un objeto al final de la lista.

## **L.count(value)**

Devuelve el número de veces que se encontró value en la lista.

## **L.extend(iterable)**

Añade los elementos del iterable a la lista.

## **L.index(value[, start[, stop]])**

Devuelve la posición en la que se encontró la primera ocurrencia de value. Si se especifican, start y stop definen las posiciones de inicio y fin de una sublista en la que buscar.

## **L.insert(index, object)**

Inserta el objeto object en la posición index.

## **L.pop([index])**

Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

# Listas

## **L.remove(value)**

Eliminar la primera ocurrencia de value en la lista.

## **L.reverse()**

Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

## **L.sort(cmp=None, key=None, reverse=False)**

Ordena la lista. Si se especifica cmp, este debe ser una función que tome como parámetro dos valores x e y de la lista y devuelva -1 si x es menor que y, 0 si son iguales y 1 si x es mayor que y. El parámetro reverse es un booleano que indica si se debe ordenar la lista de forma inversa, lo que sería equivalente a llamar primero a L.sort() y después a L.reverse().

# Diccionarios

## **D.get(k[, d])**

Busca el valor de la clave k en el diccionario. Es equivalente a utilizar D[k] pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis D[k], de no existir la clave se lanzaría una excepción.

## **D.has\_key(k)**

Comprueba si el diccionario tiene la clave k. Es equivalente a la sintaxis k in D.

## **D.items()**

Devuelve una lista de tuplas con pares clave-valor. D.keys() Devuelve una lista de las claves del diccionario.

## **D.pop(k[, d])**

Borra la clave k del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve d si se especificó el parámetro o bien se lanza una excepción. D.values() Devuelve una lista de los valores del diccionario.

# **IX. Programación Funcional**

Objetivo:

Implementar las funciones que integra Python para el proceso de información

- a. función de orden superior**
  - b. MAP**
  - c. FILTER**
  - d. REDUCE**
  - e. funciones lambda**
-

# Programación Funcional

La programación funcional es un paradigma en el que la programación se basa casi en su totalidad en funciones, entendiendo el concepto de función según su definición matemática, y no como los simples subprogramas de los lenguajes imperativos que hemos visto hasta ahora.

En los lenguajes funcionales puros un programa consiste exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida. Python, sin ser un lenguaje puramente funcional incluye varias características tomadas de los lenguajes funcionales como son las funciones de orden superior o las funciones lambda (funciones anónimas).

# Función de orden superior

Una función se denomina **Función de orden superior** si esta contiene otras funciones como parámetros de entrada o si devuelve una función como salida, es decir, las funciones que operan con otra función se conocen como **Funciones de orden superior en Python**.

Ejemplo 1 :

```
def funcion_superior(fun):  
    fun()  
  
def saludo():  
    print("Hola Mundo")  
  
f = funcion_superior(saludo)
```

# Función de orden superior

Ejemplo 2 :

Crear funciones que calculen la raíz y el cuadrado de un número, de acuerdo a los parámetros que indique el usuario.

```
def operacion(x, fun):  
    return fun(x)  
  
def raiz(x):  
    return x**0.5  
  
def cuadrado(x):  
    return x**2  
  
res1 = operacion(5, cuadrado)  
res2 = operacion(5, raiz)  
res3 = operacion(res2, cuadrado)  
print(res1)  
print(res2)  
print(res3)  
  
25  
2.23606797749979  
5.000000000000001
```

# Función de orden superior

Ejemplo 3 :

Crear una función que realice un saludo, en el idioma indicado por el usuario.

```
def saludar(idioma):  
    def saludar_es():  
        print("Hola")  
  
    def saludar_fr():  
        print("Salut")  
  
    def saludar_en():  
        print("Hello")  
  
    idioma_lista = {"es": saludar_es, "fr": saludar_fr, "en": saludar_en}  
    return idioma_lista[idioma]  
  
f = saludar("en")  
f()
```



# Funciones de Orden Superior en Python

Dentro de Python, tenemos tres funciones principales que se encajan en esta categoría, las cuales son *filter()*, *map()* y *reduce()*.

# filter()

La función **filter()** devuelve un iterador donde los elementos se filtran a través de una función para probar si el elemento es aceptado o no.

Es decir, a partir de una lista o iterador y una función condicional, es capaz de devolver una nueva colección con los elementos filtrados que cumplan la condición.

Sintaxis:

```
filter(function, iterable)
```

# filter()

Ejemplo:

A partir de una lista de números enteros, obtener únicamente los números pares empleando para ello la función de orden superior de python filter():

```
def es_par(numero):  
    if numero%2==0:  
        return numero  
    else:  
        pass  
  
lista = [1,2,3,4,5,6,7,8,9,10]  
  
lista_par = list(filter(es_par, lista))  
print(lista_par)
```

# map()

La función **map()** ejecuta una función específica para cada elemento en un iterable. El objeto se envía a la función como parámetro.

Esta función trabaja de una forma muy similar a `filter()`, con la diferencia que en lugar de aplicar una condición a un elemento de una lista o secuencia, aplica una función sobre todos los elementos y como resultado se devuelve un iterable de tipo `map`:

Sintaxis:

```
map(function, iterables)
```

# map()

Ejemplo:

A partir de una lista de números enteros, obtener el cubo de cada uno de los elementos.

```
def cubo(numero):  
    numero=numero**3  
    return numero  
  
lista = [1,2,3,4,5,6,7,8,9,10]  
  
lista_cubo = list(map(cubo,lista))  
print(lista_cubo)
```

# reduce()

La función **reduce()** acepta una función y una secuencia y devuelve un único valor calculado de la siguiente manera:

1. Inicialmente, se llama a la función con los dos primeros elementos de la secuencia y se devuelve el resultado.
2. A continuación, se vuelve a llamar a la función con el resultado obtenido en el paso 1 y el siguiente valor de la secuencia. Este proceso se repite hasta que hay elementos en la secuencia.

Sintaxis:

```
from functools import reduce  
reduce(function, iterables)
```

# reduce()

Ejemplo:

A partir de una lista de números enteros, obtener la suma total de todos los elementos.

```
from functools import reduce

def suma(a,b):
    return a + b

lista = [1,2,3,4,5,6,7,8,9,10]

lista_suma = reduce(suma,lista)
print(lista_suma)
```

# Funciones Lambda

En Python, una función Lambda se refiere a una pequeña función anónima. Las llamamos “funciones anónimas” porque técnicamente carecen de nombre. Al contrario que una función normal, no la definimos con la palabra clave estándar `def` que utilizamos en Python. En su lugar, las funciones Lambda se definen como una línea que ejecuta una sola expresión. Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión.



# Funciones Lambda Características

- Las lambdas de Python son funciones pequeñas y anónimas, sujetas a una sintaxis más restrictiva pero más concisa que las funciones normales de Python.
- Las **Lambda Functions** es una declaración especial de una función en Python que no tiene un identificador (Nombre).
- Las funciones Lambda pueden contener el número de argumentos que se necesiten pero solo puede ser declarada en una línea de código.

Sintaxis:

**lambda** argumentos: expresión

# Funciones Lambda

Ejemplo:

Convertir en función anónima la siguiente función de suma:

```
def suma(a,b) :  
    return a+b
```

Solución:

```
suma = lambda a,b : a + b  
  
resultado = suma(a,b)
```

# Funciones Lambda

Ejercicio:

Crear una expresión anónima con Python usando la función Lambda para evaluar una ecuación cuadrática. La idea de la función es que permita colocar los coeficientes de la ecuación a, b y c y que permita colocar el punto donde se desea evaluar la expresión  $f(x)$ :

Solución:

```
poly = lambda x, a, b, c: a * x**2 + b * x + c
```

```
#llamar la función anónima
```

```
poly(5, 1, 2, 1)
```

# Funciones Lambda

Ejercicios propuestos:

Crear dos funciones anónimas que:

1. Reciba el valor en Fahrenheit y retorne el valor en Celsius
2. Reciba el valor en Celsius y Retorne el valor en Fahrenheit
3. Implementar función Lambda en funciones `filter()`, `map()`, `reduce()` de los ejemplos vistos.

# X. Comprensión de listas

Objetivo:

Análisis de estructuras de datos en Python.

## 1.- Ejemplos

---

# Comprensión de listas

La comprensión de listas, del inglés **list comprehensions**, es una funcionalidad que nos permite crear listas avanzadas en una misma línea de código. Para mejorar en este ámbito se revisarán los siguientes ejemplos.

Ejemplo 1: Crear una lista con las letras de una palabra.

```
# Método tradicional
lista = []
for letra in 'casa':
    lista.append(letra)
print(lista)

# Con comprensión de listas
lista = [letra for letra in 'casa']
print(lista)

['c', 'a', 's', 'a']
['c', 'a', 's', 'a']
```

Como vemos, gracias a la comprensión de listas podemos indicar directamente cada elemento que va a formar la lista, en este caso la letra, a la vez que definimos el for:

# La lista está formada por cada letra que recorremos en el for

```
lista = [letra for letra in 'casa']
```

# Comprensión de listas

Ejemplo 2: Crear una lista con los cuadrados de los primeros 10 números.

```
# Método tradicional
lista = []
for numero in range(0,11):
    lista.append(numero**2)
print(lista)
```

```
# Con comprensión de listas
lista = [numero**2 for numero in range(0,11)]
print(lista)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

De este ejemplo podemos aprender que es posible modificar al vuelo los elementos que van a formar la lista.

# Comprensión de listas

Ejemplo 3: Crear una lista con todos los múltiplos de 2 entre 0 y 10.

```
# Método tradicional
lista = []
for numero in range(0,11):
    if numero % 2 == 0:
        lista.append(numero)
print(lista)

# Con comprensión de listas
lista = [numero for numero in range(0,11) if numero%2==0]
print(lista)

[0, 2, 4, 6, 8, 10]
[0, 2, 4, 6, 8, 10]
```

En este caso podemos observar que incluso podemos marcar una condición justo al final para añadir o no el elemento en la lista.



# Comprensión de listas

Ejemplo 4: Crear una lista de pares a partir de otra lista creada con los cuadrados de los primeros 10 números.

```
# Método tradicional
lista = []
for numero in range(0,11):
    lista.append(numero**2)
```

```
pares = []
for numero in lista:
    if numero % 2 == 0:
        pares.append(numero)
```

```
print(pares)
```

```
# Con comprensión de listas
lista = [numero for numero in
         [numero**2 for numero in range(0,11)]
         if numero%2==0]
print(lista)
```

Crear listas a partir de listas anidadas nos permite llevar la comprensión de listas al siguiente nivel y además no hay un límite.

# XI. Generadores

Objetivo:  
Análisis de estructuras de datos  
en Python.

## 1.- Ejemplos

---

# Generadores

Un generador es una función que produce una secuencia de resultados en lugar de un único valor. Es decir, cada vez que llamemos a la función nos darán un nuevo resultado. Para construir generadores sólo tenemos que usar la orden `yield`. Esta orden devolverá un valor igual que hace `return` pero, además, pasará la ejecución de la función hasta la próxima vez que le pidamos un valor.

**`range()`** es una especie de función generadora. Por regla general las funciones devuelven un valor con **`return`**, pero la peculiaridad de los generadores es que van *cediendo* valores sobre la marcha, en tiempo de ejecución.

La función generadora **`range(0,11)`**, empieza cediendo el **0**, luego se procesa el `for` comprobando si es par y lo añade a la lista, en la siguiente iteración se cede el **1**, se procesa el `for` se comprueba si es par, en la siguiente se cede el **2**, etc.

Con esto se logra ocupar el mínimo de espacio en la memoria y podemos generar listas de millones de elementos sin necesidad de almacenarlos previamente.

# Generadores

Ejemplo: Crear una función generadora de pares:

Como vemos, en lugar de utilizar el **return**, la función generadora utiliza el **yield**, que significa ceder. Tomando un número busca todos los pares desde 0 hasta el número+1 sirviéndonos de un `range()`.

```
def pares(n):  
    for numero in range(n+1):  
        if numero % 2 == 0:  
            yield numero
```

```
pares(10)
```

Sin embargo, fijaros que al imprimir el resultado, éste nos devuelve un objeto de tipo generador.

De la misma forma que recorremos un **range()** podemos utilizar el bucle `for` para recorrer todos los elementos que devuelve el generador:

```
[numero for numero in pares(10) ]
```

# **XII. Funciones Decoradoras**

## **1.- Ejemplos**

---

# Decoradores

Los decoradores son funciones que **modifican el comportamiento de otras funciones**, ayudan a reducir nuestro código y hacen que sea más sencillo de interpretar.

Los decoradores son un patrón de diseño de software que alteran dinámicamente y agregan funcionalidades adicionales a los métodos, funciones o clases de Python sin tener que usar subclases o cambiar el código fuente.

Los decoradores toman una función como argumento y luego retornan también una función. Si el decorador no retorna la función, esta se ejecutará y lanzará un `Error NoneType`.

# Decoradores

Sintaxis

Ejemplo 1:

```
def decorador(fun):  
    print("Esto es un decorador")  
    return fun
```

```
@decorador  
def hola():  
    print("Hola Mundo")
```

```
hola()
```

La notación del decorador es mediante el símbolo @ seguido del nombre de la función que cumple el papel de decorador.

# Decoradores

## Ejemplo 2

```
def decorador(fun):  
    def funcion():  
        print("Esto es un decorador 1")  
        f = fun()  
        print("Esto es un decorador 2")  
        return f  
    return funcion
```

```
@decorador  
def hola():  
    print("Hola Mundo")
```

```
hola()
```



# **XIII. Excepciones**

**1.- try, except, finally**

---

# Excepciones

Los errores de ejecución son llamados comúnmente excepciones y por eso de ahora en más utilizaremos ese nombre. Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continúa propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa.

# Excepciones

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias `try`, `except` y `finally`.

# try-except

En este caso, se levantó la excepción `ZeroDivisionError` cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque `try-except`.

Dado que dentro de un mismo bloque `try` pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques `except`, cada uno para capturar un tipo distinto de excepción.

```
def div(a,b):  
    try:  
        return a / b  
    except:  
        print("No se puede dividir entre 0")
```

```
def div2(a,b):  
    try:  
        return a / b  
    except ZeroDivisionError:  
        print("No se puede dividir entre 0")
```

## finally

Este bloque se suele usar si queremos ejecutar algún tipo de **acción de limpieza**. Si por ejemplo estamos escribiendo datos en un fichero pero ocurre una excepción, tal vez queramos borrar el contenido que hemos escrito con anterioridad, para no dejar datos inconsistentes en el fichero.

# finally

En el siguiente código vemos un ejemplo. Haya o no haya excepción el código que haya dentro de `finally` será ejecutado.

```
try:
    # Forzamos excepción
    x = 2/0
except:
    # Se entra ya que ha habido una excepción
    print("Entra en except, ha ocurrido una excepción")
finally:
    # También entra porque finally es ejecutado siempre
    print("Entra en finally, se ejecuta el bloque finally")

#Entra en except, ha ocurrido una excepción
#Entra en finally, se ejecuta el bloque finally
```

## **XIV. Entrada Estándar Rawinput**

## **XV. Salida Estándar Rawinput**

**1.- rawinput**

---

# Rawinput

Entrada de datos por teclado en Python

Python proporciona dos funciones integradas para leer una línea de texto de la entrada estándar por teclado.

Estas funciones son: `raw_input()` , `input()`

La función `raw_input`

El `raw_input` ([indicacion]) función lee una línea de la entrada estándar y lo devuelve como una cadena (quitando el salto de línea final).

Sintaxis : `str= raw_input('interaccion ')`



# Rawinput

Cabe señalar que la función `raw_input()` recibe y procesa de forma literal lo ingresado por teclado, por ejemplo si ingresamos `1+2+4` el intérprete lo maneja como texto y no lo procesa:

```
>>> str = raw_input("Ingrese algo por teclado: ")
Ingrese algo por teclado: 1+2+4
>>> print str
1+2+4
>>>
```

La función `input([indicacion])` es equivalente a `raw_input`, excepto que devuelve el resultado evaluado en su caso.

```
>>> str = input("Ingrese algo por teclado: ")
Ingrese algo por teclado: 1+2+4
>>> print str
7
>>>
```

# **XVI. Módulos**

- 1.Módulos
- 2.Bibliotecas



# Módulos

Para facilitar el mantenimiento y la lectura los programas demasiado largos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica de nuestro código.

Los ficheros son su contrapartida física: cada archivo Python almacenado en disco equivale a un módulo. Vamos a crear nuestro primer módulo entonces creando un pequeño archivo `modulo.py` con el siguiente contenido:

```
#modulo.py

def saludo():
    print ("Hola Escuela de Código")

def suma(a,b):
    return a+b
```

# Módulos

Si quisiéramos utilizar la funcionalidad definida en este módulo en nuestro programa tendríamos que importarlo. Para importar un módulo se utiliza la palabra clave `import` seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión. Como ejemplo, creemos un archivo `programa.py` en el mismo directorio en el que guardamos el archivo del módulo (esto es importante, porque si no se encuentra en el mismo directorio Python no podrá encontrarlo), con el siguiente contenido:

```
import modulo  
  
modulo.saludo()  
  
modulo.suma(5,3)
```

# Módulos

```
#modulo.py

def saludo():
    print ("Hola Escuela de Código")

def suma(a,b):
    print(a+b)

#Ejemplo

import modulo

modulo.saludo()

modulo.suma(5,3)

Hola Escuela de Código
8
```

# Módulos

El `import` no solo hace que tengamos disponible todo lo definido dentro del módulo, sino que también ejecuta el código del módulo. Por esta razón nuestro programa, además de imprimir el texto “una función” al llamar a `mi_funcion`, también imprimiría el texto “un módulo”, debido al `print` del módulo importado. No se imprimiría, no obstante, el texto “una clase”, ya que lo que se hizo en el módulo fue tan solo definir de la clase, no instanciarla. La cláusula `import` también permite importar varios módulos en la misma línea.

# Módulos y Bibliotecas

1. Collections
2. CSV – file Handling
3. Random – generation
4. Tkinter – GUI applications
5. Requests – HTTP requests
6. BeautifulSoup4 – web scraping
7. Numpy
8. Pandas
9. Matplotlib