# APUNTES
# PYTHON

**José Juan Urrutia Milán**

# Reseñas

- Curso python desde 0: https://www.youtube.com/watch?v=G2FCfQj-9ig&list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS
- Descargar python: https://www.python.org/downloads/
- Descargar Sublime Text 4: https://www.sublimetext.com/download
- Documentación de python: http://pyspanishdoc.sourceforge.net/
- Manual de interfaces gráficas: https://docs.python.org/3/library/tk.html
- Manual de expresiones regulares: https://docs.python.org/3/howto/regex.html

# Siglas/Vocabulario

- **API:** Interfaz de Programación para Aplicaciones (Vienen explicadas todas los paquetes, clases, interfaces, métodos y constantes).
- **IDE:** Entorno de Desarrollo Integrado (App para programar).
- **IDLE:** Entorno de Desarrollo Python por defecto.
- **POO:** Programación Orientada a Objetos.
- **OS:** Operating System
- **Layout:** Es la forma en la que se disponen los diferentes elementos en un marco o lámina (no es una sigla).
- **Cte/Ctes:** Abreviatura de Constante/Constantes.
- **BBDD:** Bases de Datos.
- **SGDB:** Sistemas Gestores de Bases de Datos.
- **GUI:** Interfaces gráficas
- **Convención nominal:** En programación, una convención nominal es un formato que se adopta por todos los programadores por ninguna razón en concreto. Por ejemplo: Las constantes de clase se indican en mayúsculas.

# Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

## Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a:  Hace referencia a cualquier caracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var: Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

# Índice

- split()
- strip()
- replace(old, new, max)

## Título IV: Operadores

- Aritméticos
- Comparación
- Lógicos
- Asignación
- Especiales

## Título V: Funciones o métodos

- Estructura
- Tuplas como parámetros
- Diccionarios como parámetros

## Título VI: Listas/Colecciones (arrays/arreglos)

- Sintaxis
- Llamadas
- Métodos de listas
    - append(var)
    - insert(i, var)
    - extend(var[])
    - index(var)
    - Variable en lista
    - remove(var)
    - pop()
- Convertir lista en tupla

## Título VII: Tuplas

- Sintaxis
- Métodos de tuplas

## Título XII: Bucles

### for

Tipo range(x)

### while

### Instrucciones especiales (con while y for)

break

continue

pass

else

## Título XIII: Método print(var)

Modificar final print(var, end=x)

Saltos de línea en el mismo print

Concatenar string y variables numéricas

## Título XIV: Generadores

Sintaxis

Objetos iterables

yield from

## Título XV: Excepciones

Control de excepciones: try except

finally

as

### Lanzar excepciones (manualmente)

## Título XVI: Módulos

Uso de módulos

## Título XVII: Paquetes

Uso de paquetes

config(bd=x)

config(relief=x)

config(cursor=x)

pack()

pack(side=x)

pack(anchor=x)

pack(expand=x)

pack(fill=x)

## Título III: Widgets

### Empaquetado

pack()

place(x=n, y=m)

grid(row=n, colum=m)

opciones

## Título IV: Widget Label

## Título V: Widget Entry

insert(n, txt)

delete(start, end)

get()

Detectar cada vez que se introduce una letra

## Título VI: Widget Text

Scrollbar

## Título VII: Widget Button

Acciones de botones

## Título VIII: Widget Radiobutton

Sincronizar Radiobutton

Obtener estado del botón

# Introducción

Durante este curso, aprenderemos una parte básica y moderadamente avanzada del lenguaje Python.
A lo largo de este curso, usaremos el IDE Sublime Text (El cual podemos descargar desde el enlace en el apartado **Reseñas**)

Configuración Sublime Text (vídeo 3, min 10:00)
Uso Sublime Text (vídeo 5, min 6:45)
Mostrar consola en Sublime Text (vídeo 10, min 16:20)
Mejor consola en Sublime Text (vídeo 12, min 2:32)
Explorador de archivos en Sublime Text (vídeo 36, min 3:25)

Lenguaje de muy alto nivel, gramática sencilla, clara y muy legible.
No es necesario poner ; al final de una sentencia.
Distinción fuerte entre tipos de variables, como Java.
Python es 100% orientado a objetos: Sobrecarga de constructores, herencia múltiple, encapsulación, interfaces, polimorfismo.
Open Source.
Librería estándar muy amplia.
Interpretado (no compilado).
Funciones: Aplicaciones de escritorio, aplicaciones de servidor, web…
Multiplataforma.

# Capítulo I: Programación básica

## Título I: Introducción

- Si queremos ejecutar varias sentencias en la misma línea, insertar ; entre una sentencia y otra.
- Se puede hacer un salto de línea en una sentencia y seguir en la siguiente como si fuera la misma línea usando "\".
- Si queremos hacer referencia a un texto, lo colocaremos entre dos **“”** o **‘’** , similar a Java.
- Para hacer referencia a una variable, escribiremos su nombre.
- No podemos concatenar variables str o texto con variables int o número. Para ello, deberemos convertirlo todo al mismo tipo de variable, recurriendo a los castings (ver título IX).
- Python es case sensitive, por lo que distingue entre mayúsculas y minúsculas.

### Hola mundo

**print(“Hola Mundo”)**

### print(a)

Imprime **a** en consola.

### Comentarios

Los comentarios es especifican con una almohadilla, que comenta toda la línea:

**#Esto es un comentario**

Si queremos comentar varias líneas, colocamos tres dobles comillas antes y después del párrafo a comentar:

**“””Esto**
**son**
**varios**
**comentarios”””**

### Importar clases

**import <u>nombre</u>**
**import math** *(por ejemplo)*
(ver títulos XVI y XVII)

# Título II: Tipos de datos y variables

### Tipos

- Numéricos: enteros (int), coma flotante (float) y complejos.
- Textos (str).
- Booleanos (que puede ser True (1) o False (0)).

<u>El signo de decimal en python es un punto</u> (5.2 SI | 5,2 NO).

### Declaración

Una variable no puede empezar por un número, aunque puede llevarlo.
El tipo de variable no lo establece el contenedor, sino el contenido, por lo que no hace falta indicar el tipo de dato que se almacena.
- Si queremos declarar una variable de tipo float tendremos que especificar el signo decimal. Si el número en concreto no tiene parte decimal, esta será 0:
**decimal = 5.0**
- Si queremos indicar que una variable será de tipo string, haremos lo siguiente:
**texto = StringVar()**
- Si queremos indicar que una variable será de tipo int, haremos lo siguiente:
**texto = IntVar()**

### Borrado

Podemos borrar variables con la instrucción:
**del <u>variable</u>**

**global**

Si creamos una variable fuera de todo, esta tiene ámbito global.
Si creamos una variable dentro de una función o bucle, esta tiene ámbito local.
Podemos crear una variable global dentro de una función si le ponemos delante la palabra reservada **global:**

```
def prueba():
    global variable = 5
print(variable)
```

**type(a)**

Devuelve el tipo de variable que es la variable especificada (**a**).

```
nombre = 5
type(nombre)
```

(Salida: **<class 'int'>**)

```
for i in [1, 2, 3]:
    print(i)
```

*(Salida: 1 2 3) (en diferentes líneas)*

**Math.sqrt(x)**

Devuelve (float) la raíz cuadrada de un número (**x**) (int o float)
*Importante importar la clase math.

**round(x)**

Devuelve (int) el redondeo de un float.

# Título III: string

Normalmente los textos se definen entre **''**. (**nombre = 'Jose'**)
También pueden ser definidos por "", aunque no se recomienda.

Aunque también se puede definir entre **“”””””””** si este incorpora saltos de línea:
**nombre = “””” Jose**
**Antonio**
**Guzmán””””**

Una variable string puede ser interpretada como “una lista”; si especificamos un índice, nos devuelve el caracter del string que se encuentra en esa posición:
**nombre = “Juan”**
**print(nombre)** *(Salida: Juan)*
**print(nombre[0])** *(Salida: J)*
**print(nombre[1])** *(Salida: u)*
**print(nombre[2])** *(Salida: a)*
**print(nombre[3])** *(Salida: n)*

### len(str)

Devuelve (int) el número de caracteres que tiene el texto (**str**) indicado.

### lower()

Pasa la variable string a la que se le aplica a minúsculas.
**nombre = “Juan”**
**nombre.lower()** *(nombre = juan)*

### upper()

Pasa la variable string a la que se le aplica a mayúsculas.
**nombre = “Juan”**
**nombre.upper()** *(nombre = JUAN)*

### capitalize()

Establece la primera letra del string en mayúsculas.

**count(a)**

Cuenta cuantas veces aparece un string (**a**) dentro de otro string.

**find(a)**

Devuelve (int) en qué índice aparece un string dentro de otro.

**rfind(a)**

Devuelve (int) en qué índice aparece un string dentro de otro (empezando por el final).

**isdigit()**

Devuelve un boolean que nos indica si el string es un dígito o no.

**isalum()**

Devuelve True si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter. Si no, False.

**isalpha()**

Devuelve True si hay sólo letras y espacios. False si no.

**split()**

Devuelve una lista separando las palabras según los espacios.

**strip()**

Borra espacios sobrantes al principio y al final.

**replace(old, new, *max*)**

Reemplaza todas las veces que aparezca el string **old** por **new**.
**max** no es opcional, pero indica cuantas veces se quiere que **old** sea reemplazado por **new**.

*Existen más métodos de manipulación de strings. (Ver Capítulo VI).

# Título IV: Operadores

## Aritméticos

+ Suma.
- Resta.
* Multiplicación.
/ División.
% Resto.
** Exponente. 5^3 = 5**3 (En mates / En python)
// División entera. (Devuelve un entero, fumándose el resto).

## Comparación

== Igual que.
!= Distinto que.
< Menor que.
> Mayor que.
<= Menor o igual que.
>= Mayor o igual que.

## Lógicos

AND
OR
NOT

## Asignación

= Se establece igual a.
+= Incrementa en.
-= Decrementa en.
*= Se multiplica por.
/= Se divide entre.
%= Se le hace el resto por.
**= Se eleva a.
//= Se hace la división entera por.

### Especiales

IS

IS NOT

IN — Comprueba si algo (un elemento) está dentro de algo (una lista o tupla).

NOT IN — Comprueba si algo (un elemento) no está dentro de algo (una lista o tupla).

# Título V: Funciones o métodos

Las funciones en python pueden o no devolver valores.
Las funciones en python pueden o no recibir valores.
Las funciones (como tal) en python no pertenecen a ninguna clase.
(Consultar funciones anónimas o funciones lambda (Título XVIII)).

### Estructura

**def nombre (parametros** *(opcional)***):**
    **código**
    **return código** *(opcional)*

En los parámetros, no hace falta especificar el tipo de parámetro que recibe, simplemente su nombre. Los parámetros se separan entre ellos por comas:

```
def suma(num1, num2):
    print(num1 + num2)
```

Especificar después de **return** lo que queremos devolver, si queremos devolver algo.

```
def suma(num1, num2):
    resul = num1 + num2
```

**return resul**

El final de la función lo indicamos con los indentados, por lo que es importante que el código que esté dentro de nuestra función tenga al menos un tabulado.

Para llamar a una función, establecemos su nombre y entre unos paréntesis, los parámetros que esta solicita. Si no necesita parámetros, indicar sólo paréntesis.
**nombre(parametros)**
**nombre()**

## Tuplas como parámetros

(Consultar Título VII)
Podemos especificar a una función que va a recibir un número indefinido de elementos que se van a almacenar dentro de una tupla, colocando un asterisco delante de la variable que será la tupla:
**def funcion(*tupla):**
*Convención nominal:
Por convención nominal, esta tupla suele ser llamada **args** (arguments):
**def funcion(*args):**

## Diccionarios como parámetros

(Consultar Título VIII)
También podemos especificar a una función que va a recibir un número indefinido de parámetros que se van a almacenar dentro de un diccionario, colocando dos asteriscos delante de la variable que será el diccionario:
**def funcion(**dic):**
*Convención nominal:
Por convención nominal, este diccionario suele ser llamado **kwargs** (key-word arguments)**:**

**def funcion(**kwargs)**
(Consultar Capítulo VII, Título II).

# Título VI: Listas/Colecciones (arrays/arreglos)

Las listas en python pueden guardar diferentes tipos de datos a la vez.
Se pueden expandir dinámicamente añadiendo nuevos elementos (como un **ArrayList** de Java).
Las listas empiezan a contar desde 0.
Se pueden crear listas vacías (**lista = []**)

## Sintaxis

**nombre = [dato1, dato2, dato3, ...]**
**meses = [“enero”, “febrero”, “marzo”, “abril”, “mayo”, “junio”,/ “julio”, “agosto”, “septiembre”, “octubre”, “noviembre”,/ “diciembre”]**

## Llamadas

Las listas a la hora de realizar llamadas, se especifica el nombre de la lista y entre corchetes, que valores se desean obtener. Indicar : para imprimir la lista entera.

**nombres = [“Juan”, “Lola”, “Antonia”]**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Lola’, ‘Antonia’])*
**print(nombres[1])** *(Salida: Lola)*

*Podemos indicar como índice un número negativo.
En este caso, se empezará a contar desde la derecha y desde el -1:
**print(nombres[-1])** *(Salida: Antonia)*
**print(nombres[-3])** *(Salida: Juan)*

- Se puede acceder a porciones de listas:
**print(nombres[0:2])** *(Salida: [‘Juan’, ‘Lola’])*

En este caso, el 0 se incluye dentro de la salida y el 2 se excluye.

- Si el primer índice es 0, se puede obviar:

**print(nombres[:2])** *(Salida: [‘Juan’, ‘Lola’])*

- Si queremos extraer hasta el último, también se puede obviar:

**print(nombres[1:])** *(Salida: [Lola’, ‘Antonia’])*

## Métodos de listas

### append(var)

Para agregar más elementos al final de una lista ya creada, usaremos el método **append(var)** sobre nuestra lista:

**nombres = [“Juan”, “Lola”, “Antonia”]**
**nombres.append(“Paca”)**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Lola’, ‘Antonia’, ‘Paca’])*

### insert(i, var)

Para agregar más elementos en medio de una lista ya creada, usaremos el método **insert(i, var)** sobre nuestra lista:

**nombres = [“Juan”, “Lola”, “Antonia”]**
**nombres.insert(1, “Paca”)**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Paca’, ‘Lola’, ‘Antonia’])*

### extend(var[])

Para agregar otra lista al final de una lista ya creada, usaremos el método **extend(var[])** sobre nuestra lista:

**nombres = [“Juan”, “Lola”, “Antonia”]**
**nombres.extend([“Paca”, “Pepe”])**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Lola’, ‘Antonia’, ‘Paca’, ‘Pepe’])*

### index(var)

Devuelve la posición de la lista en la que está el elemento especificado.

**nombres = [“Juan”, “Lola”, “Antonia”]**
**print(nombres.index(“Juan”))** *(Salida: 0)*

**print(nombres.index(“Antonia”))** *(Salida: 2)*

*Si hay dos elementos repetidos, siempre devolverá el primer elemento:
**nombres = [“Juan”, “Lola”, “Antonia”, “Juan”]**
**print(nombres.index([“Juan”]))** *(Salida: 0)*

## Variable en lista

Para comprobar si un elemento se encuentra en una lista, especificaremos el elemento, la palabra reservada **in** y la lista. Devuelve **true** o **false**:
**nombres = [“Juan”, “Lola”, “Antonia”]**
**print(“Juan” in nombres)** *(Salida: True)*
**print(“Paca” in nombres)** *(Salida: False)*

## remove(var)

Elimina el elemento especificado de la lista.
**nombres = [“Juan”, “Lola”, “Antonia”]**
**nombres.remove(“Lola”)**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Antonia’])*

## pop()

Elimina el último elemento de la lista.
**nombres = [“Juan”, “Lola”, “Antonia”]**
**nombres.pop()**
**print(nombres[:])** *(Salida: [‘Juan’, ‘Lola’])*

*Se pueden concatenar dos listas:
**nombres1 = [“Juan”, “Lola”]**
**nombres2 = [“Antonia”, “Marta”]**
**nombres3 = nombres1 + nombres2**
**print(nombres3[:])** *(Salida: [‘Juan’, ‘Lola’, ‘Antonia’, ‘Marta’])*

*Se pueden multiplicar las listas:
**nombres1 = [“Juan”, “Lola”] * 2**
**print(nombres1[:])** *(Salida: [‘Juan’, ‘Lola’, ‘Juan’, ‘Lola’])*

**Convertir lista en tupla**

**lista = [“Juan”, “Lola”]**
**tupla= tuple(lista)**

# Título VII: Tuplas

Las tuplas son listas inmutables; listas en las que no podemos ni añadir, eliminar ni mover elementos.
Permiten extraer porciones pero el resultado de esta extracción es una tupla nueva.
Permiten búsquedas y comprobaciones (index(var), IN).

Son más rápidas que las listas, ocupan menos espacio, formatean Strings y pueden utilizarse como claves en diccionarios a diferencias de las listas.

A la hora de almacenar elementos para no modificarlos, sólo extraerlos, es más eficiente usar una tupla que una lista.

**Sintaxis**

**nombre = (dato1, dato2, dato3, ...)** (los paréntesis son opcionales)
**nombres = (“Juan”, “Lola”, “Antonia”)**
**print(nombres[1])** *(Salida: Lola)*

Las tuplas empiezan a contar desde 0.
Las tuplas se llaman igual que las listas.
*Es posible crear una tupla unitaria, pero esta debe llevar una coma después del elemento
**tupla_unitaria = (“Juan”,)**

## Métodos de tuplas

### index(var)

Devuelve la posición de la tupla en la que está el elemento especificado **var** .
**nombres = “Juan”, “Lola”, “Antonia”**
**print(nombres.index(“Juan”))** *(Salida: 0)*
**print(nombres.index(“Antonia”))** *(Salida: 2)*

*Si hay dos elementos repetidos, siempre devolverá el primer elemento:
**nombres = [“Juan”, “Lola”, “Antonia”, “Juan”]**
**print(nombres.index(“Juan”))** *(Salida: 0)*

### Variable en tupla

Para comprobar si un elemento se encuentra en una tupla, especificaremos el elemento, la palabra reservada **in** y la tupla. Devuelve **true** o **false**:
**nombres = “Juan”, “Lola”, “Antonia”**
**print(“Juan” in nombres)** *(Salida: True)*
**print(“Paca” in nombres)** *(Salida: False)*

### count(var)

Imprime cuantas veces se encuentra un elemento (**var**) dentro de una tupla.
**nombres = (“Juan”, “Lola”, “Juan”)**
**print(nombres.count(“Juan”))** *(Salida: 2)*

### len(tupla)

Imprime cuántos elementos tiene una tupla (**tupla**).
**nombres = (“Juan”, “Lola”, “Juan”)**
**print(len(nombres))** *(Salida: 3)*

### Convertir tupla en lista

**tupla = (“Juan”, “Lola”)**
**lista = list(tupla)**

### Desempaquetado de tupla

Se pueden separar los distintos elementos de una tupla en variables independientes:
**tupla = (“Juan”, 31, 1, 2016)**
**nombre, dia, mes, ano = tupla**

# Título VIII: Diccionarios

Son estructuras de datos que permiten almacenar diferentes valores de diferentes tipos (int, str, float...) e incluso listas y otros diccionarios. Se crea una asociación **clave:valor** para cada elemento almacenado. Los elementos almacenados no están ordenados, ya que no existe un orden.

Las claves y los valores pueden ser de cualquier tipo de dato, permitiéndonos mezclarlos a nuestro gusto.

### Sintaxis

**nombre = {clave1:valor1, clave2:valor2, ...}**
**capitales = {“Alemania”:”Berlín”, “España”:”Madrid”}**

### Acceder a valores

Para acceder a valores, en vez de indicar el índice como en las tuplas o en las listas, deberemos indicar la clave:
**print(capitales[“Alemania”])** *(Salida: Berlín)*

Para acceder al diccionario entero, no indicar ni corchetes ni claves:
**print(capitales)**
*(Salida: {‘Alemania’: ’Berlín’, ‘España’: ’Madrid’})*

## Agregar más elementos / modificar valores de claves

**capitales[“Italia”]=”Lisboa”**
**capitales[“Italia”]=”Roma”**
**capitales[“España”]=”Granada”**

## Eliminar elementos

**del capitales[“Italia”]**
Haciendo esto, se borra tanto la clave (**“Italia”**) como el valor (**“Roma”**).

## Usar listas como claves y valores

**pais = [“Alemania”, “España”]**
**capital = [“Berlín”, “Madrid”]**
**diccionario1 = {pais[0]:”Berlín”, pais[1]:”Madrid”}**
**diccionario2 = {“Alemania”:capital[0], “Eespaña”:capital[1]}**
**diccionario3 = {pais[0]:capital[0], pais[1]:capital[1]}**
Todo lo anterior es correcto.

**diccionario4 = {“Dia”:23, “Ano”:[2000, 2001, 2002, 2003]}**
Se puede asignar una lista entera como valor de una clave.
Se puede asignar un diccionario entero como valor de una clave.

## keys()

Devuelve todas las keys que tiene un diccionario.
**print(capitales.keys())** *(Salida: dict_keys([‘Alemania’, ‘España’]))*

## values()

Devuelve todos los valores que tiene un diccionario.
**print(capitales.values())** *(Salida: dict_values([‘Berlín’, ‘Madrid’]))*

## len(diccionario)

Devuelve (int) el número de parejas de **clave:valor** que tiene un diccionario.
**print(len(capitales))** *(Salida: 2)*

# Título IX: Introducción de datos por teclado

Para guardar en una variable un valor introducido por teclado, deberemos usar el método **input()** , el cual detiene la ejecución de ese hilo hasta que el usuario introduzca el dato.
Simplemente igualamos la variable que queremos que almacene el dato a este método:strin
**dato = input()**

Podemos pasar datos al método **input()** de forma que nos imprima un texto delante del dato a insertar:
**dato = input("Introduzca un dato: ")**

*Todo lo introducido a través del método **input()** será texto, por lo que si queremos recibir un número, deberemos realizar un casting a int (ver título IX):
**dato = int(input())**

# Título X: Castings

Para realizar un casting, se nos presentan diferentes funciones:

**int(var)**

Para convertir cualquier tipo de variable a int, siempre y cuando sea posible:
**letra = "5"**
**numero = int(letra)**

**str(var)**

Para convertir cualquier tipo de variable a string, siempre y cuando sea posible:
**numero = 5**
**letra = str(numero)**

## float(var)

Para convertir cualquier tipo de variable a float, siempre y cuando sea posible:
**letra = “5”**
**numero = float(letra)**

## Concatenación

Python no permite concatenar strings con valores numéricos por ejemplo, en un print, por lo que podemos recurrir a los castings para pasar el valor numérico a texto o usar la siguiente notación:
(Donde numero es una variable de tipo int igual a 3):
**print(f”El valor de la variable es {numero}”)**

*(Salida: El valor de la variable es 3)*
Donde indicamos la f para indicar que es una función de tipo f y la variable entre llaves, dentro del texto.

- Existe otra forma de concatenar texto con números:
**print(“El valor de la variable es ”, 4, “ centímetros”)**
(Separando texto y números por comas)

- Otra forma más:
**nombre = "Juan"**
**edad = 25**
**lugar = "Madrid"**
**print("{}: tengo {} años y vivo en {}.".format(nombre, edad, lugar))** *#Salida: Juan: tengo 25 años y vivo en Madrid.*

# Título XI: Condicionales

## if

**if <u>condición</u>:**
    **<u>código</u>**
**elif <u>condición</u>:**  *#opcional*
    **<u>código</u>**
**else:**  *#opcional*
    **<u>código</u>**

Donde **<u>condición</u>** es una estructura igual a True o False o que devuelve uno de estos dos mediante funciones o operadores lógicos. **<u>condición</u>** también puede ser una variable booleana:

```
condicion = True
if condicion:
    código
```

### Concatenar operaciones lógicas

Esto es posible en python (numero es una variable int):

```
if 0<numero<100:
    código
```

### AND y OR

Estos operadores nos permite concatenar diferentes condiciones, indicando que todas se tienen que cumplir (AND) o que con tan sólo una es True (OR).
En comparación con Java, && = AND y || = OR .

### IN

Podemos determinar una condición que sea que un valor esté dentro de una tupla. De esta forma, si este valor está dentro de la tupla devolverá True y si no, False:

**num = 5**
**if num in (1, 2, 3, 4, 6):**
    **código**

En este caso, la condición **num in (1, 2, 3, 4, 6)** es False. Si num fuera igual a 1, 2, 3, 4, o 6, esta sería True.

### switch

Python no dispone de una estructura **switch** como tal, aunque ofrece algunas alternativas como las vistas en el apartado del **if .**

# Título XII: Bucles

## for

**for variable in elemento_a_recorrer:**
    **código**

No hace falta declarar el valor de **variable**, ya que esta tomará los valores de **elemento_a_recorrer**.
**elemento_a_recorrer** puede ser una lista, una tupla, una cadena de texto…
El bucle se ejecutará en función del número de elementos que haya en la lista o tupla, haciendo que i en cada caso valga el valor de la lista correspondiente (el bucle for en python se puede asemejar a los bucles for : each en otros lenguajes de programación, como en Java):

**for i in [1, 2, 3]:**
    **print(i)**
*(Salida: 1 2 3) (en diferentes líneas cada número)*

**for i in [“perro”, “casa”, “coche”]:**
    **print(i)**
*(Salida: perro casa coche) (en diferentes líneas cada palabra)*

Si por el contrario, se indica un string como elemento a recorrer, se ejecutará en función de las letras de la cadena de texto. Haciendo que i valga las letras cada vez que se ejecute el bucle:

**for i in “Hola”:**
    **print(i)**
*(Salida: H o l a) (en diferentes líneas cada letra)*

*Al igual que en las funciones y los condiciones, el cuerpo del bucle se indica con una indentación.

### Tipo range(x)

Este tipo de variable nos devuelve algo parecido a una lista, la cual contiene una lista de datos desde el 0 hasta **x**-1 :
**range(5) = [0, 1, 2, 3, 4]**

Por lo tanto:
**for i in range(3):**
    **print(i)**

*(Salida: 0, 1, 2) (números en diferentes líneas)*

Range también permite indicar el primer valor de la sucesión:
**range(5, 10) = [5, 6, 7, 8, 9]**
(El primero se toma en cuenta, el segundo no).

Range también permite indicar de cuanto en cuanto van a aumentar los valores de la sucesión:
**range(1, 2, 8) = [1, 3, 5, 7]**

## while

**while <u>condición</u>:**

**código**

El código que esté dentro del bucle se repetirá todas las veces posibles hasta que la **condición** no se cumpla.

## Instrucciones especiales (con while y for)

### break

Si la ejecución del programa se topa con esta palabra, saldrá del bucle (e ignorará la posible instrucción **else**):

```
while 5==5:
    break
```

Este bucle sólo se ejecutará 1 vez aunque la condición pueda causar un bucle infinito.

### continue

Salta a la siguiente interacción de bucle, ignorando el código que haya detrás de esta palabra en esa vuelta (o interacción).

### pass

Devuelve **null** en cuanto se lee.
Se suele utilizar para declarar bucles, funciones y clases sin código, simplemente para declararlas temporalmente.

### else

Se ejecuta cuando el bucle termina su ejecución de forma normal (sin ayuda de **break**).
Si este termina con un **break**, el **else** no se ejecutará.

## Título XIII: Método print(var)

El método print(var) se usa para imprimir información en consola.

Los datos que se especifiquen dentro de los paréntesis se imprimirán en consola, haciendo un salto de línea cada vez que el método se ejecuta. De esta forma:

**print(“Hola”)**
**print(“Hola”)**

*(Salida:*
*Hola*
*Hola)*

## Modificar final print(var, end=x)

Si queremos modificar lo último que hace el método print() después de imprimir el dato deseado (**var**), lo especificaremos en el apartado (**x**). De esta forma, si queremos que entre print y print haya un espacio. Haremos lo siguiente:
**print(“Hola”, end = “ ”)**
**print(“Hola”, end = “ ”)**
*(Salida: Hola Hola )*

## Saltos de línea en el mismo print

Para añadir un salto de línea en el mismo print, indicaremos el texto **“\n”** :
**print(“Hola\nMi nombre es Juan\nTengo 6 años”)**
*#Salida:*
Hola
Mi nombre es Juan
Tengo 6 años

## Concatenar string y variables numéricas

Consultar Título IX, **Conctatenación**

# Título XIV: Generadores

Son estructuras que extraen valores de una función y los almacenan en objetos iterables (que se pueden recorrer por un iterador).
Estos valores se almacenan de uno en uno.
Cada vez que un generador almacena un valor, este permanece en estado pausado hasta que se solicita el siguiente. Este estado es conocido como "suspensión de estado".

Los generadores son como funciones que devuelven objetos iterables.

Son más eficientes que las funciones, ya que si estas tienen que devolver más de una variable, reservarán más espacio en memoria.
Son muy útiles con listas de valores infinitos.
A veces, es muy útil que un generador nos devuelva valores de uno en uno.

**Sintaxis**

**def nombre(parámetros):** *(**parámetros** opcionales)*
    **código**
    **yield variable** *(yield es similar al return salvo que devuelve un objeto iterable)*

**yield** no hace que la función se detenga, si no que carga al objeto iterable.

```
def numeros(limite):
    num = 1
    while num<limite:
        yield num
        num += 1

pares = numeros(10)
```

**for i in pares:**
    **print(i, end=" ")**  *(Salida: 1 2 3 4 5 6 7 8 9)*

### Objetos iterables

Podemos usar el método **next(obj_iterable)** para imprimir los valores de un objeto iterable. De esta forma, si pares es la variable anterior:
**print(next(pares))** *(Salida: 1)*
**print(next(pares))** *(Salida: 2)*
**print(next(pares))** *(Salida: 3)*

Entre llamada y llamada, estos objetos entran en estado de suspensión, guardando el último valor que fue llamado.

### yield from

Esta instrucción nos permite simplificar código en el caso de que tengamos bucles anidados dentro de un generador:

| | | |
|---|---|---|
| **for elem in grupo:**<br>  **for subElem in elem:**<br>  **yield subElem** | = | **for elem in grupo:**<br>  **yield from elem** |

## Título XV: Excepciones

Las excepciones son errores en tiempo de ejecución.
Es recomendable, o en algunos casos obligatorio, el buen uso de las excepciones, ya que si hay una línea que puede causar errores, podemos ignorarlo y hacer que el programa no se caiga.

### Control de excepciones: try except

Podemos controlar las excepciones con estructuras que intentan reproducir una línea de código pero en el caso de que esta lance un error determinado, se ejecutará otra línea diferente:
**try:**

```
        division = num1/num2
except ZeroDivisionError:
        print("No se puede dividir entre 0")
```

En este caso, se intentará reproducir el código que hay dentro del **try** pero si este genera el **ZeroDivisionError** (error de dividir entre 0), se ejecutará el código que hay dentro del **except** .
El tipo de error nos lo especifica la consola cuando este se produce, haciendo que el programa se caiga.
A esta estructura se le conoce como **try except** y es similar a la de otros lenguajes, como por ejemplo el **try catch** de Java.
Si el **try** se ejecuta, el **except** no lo hará y viceversa.

*Si hay más de una línea que generan varios errores, se pueden meter dentro del mismo **try except**, ya que cuando se detecte el error, se ignora el código del **try** y se ejecuta el del **except** .

```
try:
        division = num1/num2
        division2 = num3/num4
except ZeroDivisionError:
        print("No se puede dividir entre 0")
```

*Se pueden utilizar tantos **except** para capturar diferentes excepciones como queramos.

```
try:
        numero = int(input())
        division = 5/numero
except ZeroDivisionError: #cuando divides entre 0
        print("No se puede dividir entre 0")
except ValueError: #cuando intentas pasar a int algo que no se
                   #puede pasar a int
        print("Datos no válidos")
```

**Si no se indica un error en el **except**, este se ejecutará cuando dentro del **try** se ejecute una excepción independientemente de qué error se haya cometido:

```
try:
    numero = int(input())
    division = 5/numero
except:
    print("Ha ocurrido un error")
```

### finally

El código que haya dentro del **finally** se ejecuta independientemente de que no haya ningún error o de que haya errores:

```
try:
    numero = int(input())
    division = 5/numero
except:
    print("Ha ocurrido un error")
finally:
    print("Programa finalizado") #Esto siempre se ejecuta
```

*Se puede usar una estructura **try except** sin el **except**, siempre y cuando que esta lleve un finally.
Si no capturamos el error con **catch** el programa se caerá (salvo las líneas que estén dentro de un **finally**)

### as

Nos permite obtener el texto que imprime la excepción en consola para hacer otra cosa con él (este se puede personalizar, ver **Lanzar excepciones (manualmente)**), como mostrárselo al usuario:

```
try:
    division = num1/num2
except ZeroDivisionError as DivisionError:
```

**print(DivisionError)** *(Salida: division by zero)*

En este caso se llama DivisionError, pero podemos asignarle el nombre que queramos, como si de una variable se tratara.

## Lanzar excepciones (manualmente)

Se pueden lanzar excepciones con la palabra reservada **raise** , acompañada del nombre de la excepción que queremos lanzar. (Opcional) podemos especificar entre dos paréntesis el texto que imprimirá en consola nuestra excepción (ver **as**):
**raise ZeroDivisionError("No se puede dividir entre 0")**

Estas excepciones, que se lanzan manualmente, se pueden capturar como si fuesen excepciones "naturales".

# Título XVI: Módulos

Los módulos son archivos .py que sirven para organizar y reutilizar código. Es recomendable a la hora de hacer un programa, dividirlo en diferentes módulos. Podemos extraer clases, funciones y variables de los módulos.

### Uso de módulos

Hemos creado un módulo (llamado: funcion_suma.py) con varias funciones:

```
def sumar(num1, num2):
    print(num1+num2)

def restar(num1, num2):
    print(num1-num2)
```

Y queremos acceder a él desde otro módulo que se encuentra en la misma carpeta:

**import funcion_suma**
**funcion_suma.sumar(5, 3)**

*Si sólo queremos importar una función (como en este caso) es más simple importar sólo esa función y no tendremos que usar el nombre del módulo:
**from funcion_suma import sumar**
**sumar(5, 3)**

*Si queremos importar todas las funciones:
**from funcion_suma import ***
**sumar(5, 3)**
**restar(5, 3)**

# Título XVII: Paquetes

Los paquetes son directorios donde se almacenan módulos relacionados entre sí.
Sirven para organizar y reutilizar módulos.
Para crear un paquete, debemos crear una carpeta que contenga un archivo llamado **__init__.py** (este debe estar vacío) y los demás módulos que queremos que estén dentro del paquete.
Para acceder al paquete, debemos crear el script .py en el mismo directorio que el paquete o usar paquetes destribuilbes.

### Uso de paquetes

**from paquete.modulo_a_usar import** *(funciones, clases, var, * ...)*

### Subpaquetes

Se pueden crear subpaquetes creando un directorio dentro del directorio del paquete, siempre y cuando que todas estas tengan un archivo **__init__.py** (vacío).
**from paquete.subpaquete.modulo import …** *(sustituir … )*

## Paquetes distribuibles

### Creación

Para crear un paquete distribuible (este puede contener subpaquetes), debemos crear un directorio que contenga:

- El paquete a distribuir.
- Un archivo **setup.py** : que contenga el siguiente código:

```
from setuptools import setup
setup(
    name="nombrepaquete",
    version="1.0", #debe ser un float (el que quieras)
    description="descripción del paquete",
    author = "autor",
    author_email="autor@email.com",        #opcional
    url="paginaWeb",                       #opcional
    packages=["paquete", "paquete.subpaquete", ...] #(nombres
                                            #de directorio)
)
```

A continuación, nos dirigimos al directorio que contiene el archivo **setup.py** con cmd y establecemos el comando:
**python setup.py sdist**
y pulsamos ENTER.

Al hacer esto, se nos crean dos carpetas:
dist y nombrepaquete.egg-info
Dentro de dist tenemos el archivo comprimido .tar.gz que podemos compartir.

### Instalar paquete

Para instalar un paquete, primero debes descargarlo y guardarlo en una carpeta.

Posteriormente, nos dirigimos a esa carpeta desde cmd y ejecutamos el siguiente comando:
**pip3 install nombrepaquete-version.tar.gz**

Una vez instalado este paquete, podremos acceder a él desde cualquier archivo .py localizado en cualquier parte de nuestro ordenador (siempre y cuando lo importemos (**uso de paquetes**)).

### Desinstalar paquetes

Abrimos cmd (da igual la ruta en la que estemos) y ejecutamos:
**pip3 uninstall nombrepaquete**
y aceptamos con 'y'.

## Título XVIII: Funciones lambda

Las funciones lambda son funciones anónimas que se usan en python para abreviar, resumir una función normal en una función lambda.
Las funciones lambda son capaces de realizar más cosas que las funciones normales.
Las funciones sencillas sí que son susceptibles de ser convertidas a funciones lambda; por otra parte, las complejas no (ni tampoco las que tengan varios condicionales o algún bucle).
Las funciones lambda también son llamadas expresiones lambda, funciones on the go, funciones on demand o funciones online.

Para comprender las funciones lambda, pondremos el ejemplo de una función normal que posteriormente transformaremos en lambda.

```
def area_triangulo(base, altura):
        return (base*altura)/2
triangulo = area_triangulo(5, 7)

area_triangulo = lambda base, altura: (base*altura)/2
```

**triangulo = area_triangulo(5, 7)**

*(Ver título IX)
El ejemplo del título IX se puede simplificar bastante con una función lambda:

```
lista = [15, 8, 6, 21, 23, 27, 10]
pares = list(filter(lambda num: num%2==0, lista))
#pares = [8, 6, 10]
```

# Título IXX: Función filter

La función filter nos permite comprobar que los elementos de una secuencia cumplen una condición, devolviendo un iterador con aquellos elementos que sí la cumplen.
Podríamos decir que esta función es una especie de filtro.
La función filter recibe dos parámetros: la función que comprobará la condición y la lista de elementos a evaluar.
La función filter nos devuelve un objeto, aunque este lo podemos convertir en una lista si usamos la función **list()** y le pasamos como parámetro este objeto.

Un ejemplo de para qué sirve la función filter puede ser el extraer todos los valores que sean pares de una lista a otra:

```
def par(num):
    if num % 2 == 0:
        return True
lista = [15, 8, 6, 21, 23, 27, 10]
pares = list(filter(par, lista))
#pares = [8, 6, 10]
```

*La función indicada sólo puede recibir un parámetro, que será cada uno de los elementos de la lista iterable.

# Título XX: Función map

La función map aplica una función a cada elemento de una lista iterable (como listas o tuplas), devolviendo una lista con los resultados.
La función Map recibe dos parámetros, uno con la función y otro con la lista iterable.
La función Map es similar a la filter, aunque esta no discrimina a ningún objeto, sino que calcula el resultado de todos los objetos.

*La función indicada sólo puede recibir un parámetro, que será cada uno de los elementos de la lista iterable.

Un ejemplo es: tenemos una lista con elementos numéricos y queremos dividir todos los números entre 10:

```
def entre10(num):
    return num/10
lista = [100, 50, 36, 574, 5]
resul = map(entre10, lista)
#resul = [10, 5, 3.6, 57.4, 0.5]
```

# Capítulo II: POO

Teoría: vídeos 24 - 25

## Título I: Creación de una clase

**class nombre():**
**____código**

Dentro del código, especificaremos los campos de clase y los métodos.

## Campos de clase

Los campos de clase se especifican igual que las variables.

```
class Coche():
    largo = 150
    ancho = 50
```

Esto no es recomendable, ya que los campos de clase se declaran dentro del método constructor.

## Método constructor

Este es capaz de darle estados iniciales a campos de clase:

```
def __init__(self):
    self.largo = 150
    self.ancho = 50
```

Puede recibir parámetros:

```
def __init__(self, longitud, ancho):
    self.largo = longitud
    self.ancho = ancho
```

### self

**self** hace referencia a la propia clase. (Igual que **this** en Java).

Siempre que nombremos a un campo de clase dentro de la clse, nos veremos obligados a usar **self**.

# Métodos

Los métodos son funciones que pertenecen a clases.
Estos se declaran igual que las funciones.
Los métodos pueden recibir o no parámetros (*).
*Todos los métodos obligatoriamente tienen que recibir (mínimo) como primer parámetro la palabra reservada **self** .
Los métodos pueden devolver o no parámetros.

```
def nombre(self):
    pass        #Instrucción que ponemos cuando sólo hemos
                #declarado la función y no la hemos creado como tal
                #todavía (consultar capítulo I: título XI: pass)
```

Si hacemos referencia a campos de clase dentro de un método, estos deberán ir con la ayuda de la palabra reservada **self**:

```
def setLargo(self):
    self.largo = 100
```

Si nuestro método sólo recibe un **self**, no hará falta indicar nada dentro de los paréntesis:

```
miCoche = Coche()
miCoche.setLargo()
```

Métodos que reciben parámetros:

```
def setLargo(self, longitud)
    self.largo = longitud

miCoche = Coche()
miCoche.setLargo(150)
```

### Creación de objetos / Instanciar clases

Para instanciar una clase, nombraremos el nombre que le queremos dar al objeto y lo igualamos a la clase (como si de una variable se tratara) (no se usa el operador **new**, como en otros lenguajes).

**nombre = nombre_clase()**

**miCoche = Coche()**

*Para acceder a campos de clase y métodos usamos la nomenclatura del punto como en otros lenguajes como Java.

### Encapsulación

Para encapsular campos de clase o métodos (hacerlos accesibles sólo dentro de la clase), deberemos hacer que su nombre empiece por __ (dos barras bajas):

```
class Prueba():
    def __init__(self):
        self.__largo = 10
        self.__ancho = 5
    def setLargo(self, n)
        self.__largo = n
    def __setAncho(self, n)
        self.__ancho = n
```

El campo largo no será accesible fuera de la clase:

```
obj = Prueba()
obj.largo           #Esto no es posible
obj.setLargo(5)     #Esto si es posible
obj.setAncho(2)     #Esto no es posible
obj.__setAncho(2)   #Esto tampoco es posible
```

# Título II: Herencia

Para que una clase herede de otra, debemos indicar la clase padre dentro de los paréntesis de la clase (en python, una clase puede heredar de más de una clase):

```
class Vehiculo():
    def __init__(self):
        self.largo = 5
    def getLargo(self)
        return self.largo

class Coche(Vehiculo):
    pass
```

La clase hijo puede acceder al método constructor, campos de clase y métodos de la clase padre:

**miCoche = Coche()**
**print(miCoche.getLargo())** *(Salida: 5)*

## Sobrescribir métodos

Si creamos un método en la clase hijo que se llama igual que un método de la clase padre, cuando llamemos a este método en un objeto de la clase hijo, se ejecutará el método de la clase hijo:

```
class Padre():
    def saludar(self):
        print("Hola")

class Hijo(Padre):
    def saludar(self):
        print("Adios")

obj = Hijo()
obj.saludar() (Salida: Adios)
```

### Herencia múltiple

Python nos permite heredar más de una clase, indicando las clases deseadas en los paréntesis de la clase separadas por comas.
Cuando hay herencia múltiple, se da preferencia a la primera clase que se indique; esto es, si ambas tienen métodos con el mismo nombre (incluido el método constructor), esta clase sólo heredará el método de la primera clase que se indique.
El objeto hijo hereda todos los métodos de sus respectivos padres.

### super()

Esta función llama a un método de la clase padre:

```
class Persona():
    def __init__(self, edad):
        self.edad = edad
class Empleado(Persona):
    def __init__(self, edad, sueldo):
        super().__init__(edad)
        self.sueldo = sueldo
```

De esta forma, podemos crear un objeto de la clase Empleado que tenga un valor de edad:

```
Jose = Empleado(45, 1200)
```

El método **super()** también se puede usar a la hora de sobrescribir métodos.

### isinstance(objeto, clase)

Devuelve True si el objeto **objeto** pertenece o hereda de la clase **clase**, False si no.

## Título III: Polimorfismo

El siguiente ejemplo es posible:

```
class Coche():
    def desc(self):
        print("Soy un coche")
class Camion():
    def desc(self):
        print("Soy un camión")
class Moto():
    def desc(self):
        print("Soy una moto")

def descripcion(vehiculo):
    vehiculo.desc()

Micarro = Coche()
descripcion(Micarro) #Salida: Soy un coche
Micarro = Camion()
descripcion(Micarro) #Salida: Soy un camión
Micarro = Moto()
descripcion(Micarro) #Salida: Soy una moto
```

## Título IV: Método __str__(self)

En python, la mayoría de clases tienen un método **__str__(self)** que lo que hace es devolvernos la información de la clase convertida en un string (similar al **toString()** de Java).
Si imprimimos un objeto, lo que realmente pasará es que se imprimirá el método **__str__** de ese objeto.

## Título V: Objetos anónimos

Se pueden crear objetos anónimos y usar sus métodos de la siguiente forma (aunque si es así, este objeto sólo lo podremos usar una vez):

```
class Persona():
    def __init__(self, nombre):
```

```
            self.nombre = nombre
        def mostrar(self):
            print(self.nombre)

objeto = Persona("Antonio")
objeto.nombre()

//

Persona("Antonio").nombre()
```

# Capítulo III: Archivos externos

Para manejar archivos externos debemos seguir siempre 4 pasos:

1. Creación
2. Apertura.
3. Manipulación.
4. Cierre.

## Título I: Escritura en archivos .txt

Usaremos el módulo io, que nos proporciona la creación de streams de datos. (**from io import open**)

1. Para crear/abrir un archivo externo, crearemos una variable que igualamos al método **open(archivo, modo_de_apertura),** donde especificaremos el nombre del archivo que queremos crear/abrir (si no existe se crea y se abre, si existe sólo se abre) y el modo en que queremos acceder a él (lectura, escritura (**w**), append ...).
Hecho todo esto, se creará el archivo con el nombre y el tipo de archivo indicado en el mismo directorio que nuestro programa .py .
**archivo = open(“archivo.txt”, “w”)**
*Podemos abrir un archivo en lectura y escritura especificando **“r+”** .

2. Para escribir en él, debemos usar el método **write(string)** sobre él, en el que le indicaremos la información a escribir.
**archivo.write(“Hola a todos.”)**
(Se sustituirá toda la información del archivo por este string).
2’. Podemos escribir una lista en él usando el método **writelines(lista)** sobre él:
**archivo.writelines([“Hola a todos \n”, “Esto es un curso”])**

3. Cerramos nuestro archivo con el método **close()** .
**archivo.close()**

# Título II: Agregar información a archivos .txt

1. Para abrir un archivo externo, crearemos una variable que igualamos al método **open(archivo, modo_de_apertura),** donde especificaremos el nombre del archivo que queremos abrir y el modo en que queremos acceder a él (lectura, escritura, append (a) ...). Hecho todo esto, tendremos el archivo a nuestra disposición en modo de agregar información.
**archivo = open(“archivo.txt”, “a”)**

2. Para escribir en él, debemos usar el método **write(string)** sobre él, en el que le indicaremos la información a escribir.
**archivo.write(“Hola a todos.”)**

3. Cerramos nuestro archivo con el método **close()** .
**archivo.close()**

# Título III: Leer archivos .txt

Usaremos el módulo io, que nos proporciona la creación de streams de datos. (**from io import open**)

1. Para abrir un archivo externo, crearemos una variable que igualamos al método **open(archivo, modo_de_apertura),** donde especificaremos el nombre del archivo que queremos abrir y el modo en que queremos acceder a él (lectura (**r**), escritura, append ...). Hecho todo esto, tendremos el archivo a nuestra disposición en modo de lectura.
**archivo = open(“archivo.txt”, “r”)**
*Podemos abrir un archivo en lectura y escritura especificando **“r+”** .

2. Para extraer la información del archivo, igualamos una variable string al método **read()** sobre el objeto que maneja nuestro archivo:

**texto = archivo.read()**
*Si no especificamos nada, el método **read()** leerá el archivo entero, pero si especificamos un parámetro en **read(n)**, se leerán el número de caracteres especificados (**n**).
2'. Para extraer la información también podemos usar el método **readlines()** en vez del método **read()**. La única diferencia es que este método nos devuelve una lista donde cada elemento es una línea en el archivo físico.
**lineas = archivo.readlines()**

3. Cerramos nuestro archivo con el método **close()** .
**archivo.close()**

# Título IV: Manejo del cursor

El cursor se sitúa al final de donde acaba nuestro método **read()**, por lo que si ya hemos leído por completo este archivo y lo volvemos a leer otra vez, esta segunda lectura no devuelve nada, ya que el cursor está al final de todo el texto.

```
archivo = open("archivo.txt", "r")
print(archivo.read())
print(archivo.read())
archivo.close()
```

*Esto sólo imprime el archivo una vez*

Para poder manejar la posición en la que se encuentra el cursor, debemos usar el método **seek(n)** para indicar el hueco en el que se tiene que situar. Siendo cada hueco un carácter y empezando a contar desde el 0. De esta manera, antes del segundo **read()** tendríamos que incluir un **seek(0):**

```
archivo = open("archivo.txt", "r")
print(archivo.read())
archivo.seek(0)
```

**print(archivo.read())**
**archivo.close()**
*Esto imprime el archivo dos veces.*

# Título V: Serialización

La serialización consiste en guardar en un fichero en código binario una colección, tupla, diccionario o un objeto.
Esto se hace con la finalidad de recuperarlos tal y como estaban.

## Guardado de datos

**import pickle**

1. Creamos el elemento que queremos guardar (en este caso una lista):
**lista = [“Juan”, “Alberto”, “María”]**

2. Creamos el fichero en el que vamos a guardar los datos (se crea en el mismo directorio que el archivo .py de nuestro programa) y lo abrimos en modo escritura binaria (**wb**) (write binary):
**archivo = open(“archivo”, “wb”)**
(agregar información binaria: **“ab+”**)

3. Escribimos en el fichero el elemento con el método **pickle.dump(elemento, fichero)**:
**pickle.dump(lista, archivo)**

4. Cerramos el fichero con **close()**:
**archivo.close()**

*Si queremos serializar varios objetos, es interesante crear una lista con esos objetos y serializar la lista, para mayor eficiencia.

## Lectura de datos

**import pickle**

1. Abrimos el fichero con el modo de lectura binaria (**rb**) (read binary):
**archivo = open(“archivo”, “rb”)**

2. Guardamos el contenido del archivo en una variable con la ayuda del método **pickle.load(fichero)**
**lista = pickle.load(archivo)**

3. Cerramos el fichero con **close()**:
**archivo.close()**

# Capítulo IV: Interfaces gráficas

(Consultar en Reseñas: Manual de interfaces gráficas).
Librería **Tkinter:** **from tkinter import ***
Aunque existen otras como: **WxPython**, **PyQT**, **PyGTK** ...

**Estructura**

La estructura de una interfaz gráfica en python consta de:

- **Raíz (tk)**: O ventana (una **Raíz** es un **widget**).
- **Frame**: Contenedor donde se establecen los **widgets** (un **Frame** es un **widget**). Este actuará de lámina.
- **widgets**: Componentes.

## Título I: Creación de Raíz o ventana

Importamos la librería tkinter, creamos nuestro objeto que será nuestra raíz con el constructor default de la clase **Tk** y le ejecutamos el método **mainloop()** (crea una especie de bucle infinito para que la ventana sea siempre visible).

```
from tkinter import *
raiz = Tk()
raiz.mainloop()        #Esta instrucción debe estar siempre al final de
                       #TODA la interfaz gráfica.
```

Con esto, creamos una ventana vacía a tamaño default con un icono default.

*Si hacemos doble click en el archivo .py desde windows, se nos ejecutará una ventana cmd con la consola y la interfaz gráfica.
Si no queremos que se nos muestre el cmd con la consola, debemos cambiar la extensión de .py a .pyw .

# Métodos

### title(titulo)

Sobre nuestra raíz para indicar un título (**titulo**) a nuestra ventana.

### geometry(tamano)

Establece el tamaño de la ventana (tamano = “650x250” ...):
**raiz.geometry(‘650x250’)**

*También se le puede indicar el espacio de la pantalla en el que aparacerá de la siguiente forma:
(tamano=’650x250+300+400’)
**raiz.geometry(‘650x250+300+400’)**
De esta forma, aparecerá una ventana de 650x250 a 300px de la izqda y a 400px del margen superior.

### resizable(width, height)

Establece mediante True y False si la ventana se podrá redimensionar a lo ancho y a lo largo.

### iconbitmap(foto)

Establece el icono de la ventana (mover foto a la misma carpeta que el programa .py para no tener que indicar ruta).
*Los iconos deben estar en formato .ico
**raiz.iconbitmap(“foto.ico”)**

### destroy()

Para cerrar la ventana (terminar / cerrar la aplicación).

# config()

Todas las propiedades se pueden usar a la vez dentro del mismo método **config()** , siempre y cuando estén separados por comas.

**config(width=x, height=x)**

(sustituir x por un número en formato de string)
Para darle un tamaño al frame.
**marco.config(width=”600”, height=”300”)**

**config(bg=x)**

(sustituir x por un string que hace referencia a un color en inglés o a un código hexadecimal)
Para darle un color de fondo al frame.
**marco.config(bg=”red”)**

**config(bd=x)**

(sustituir x por un número en formato int)
Establece el tamaño del borde del frame.

**config(relief=x)**

(sustituir x por: “groove”, “sunken”, “flat”, “raised”, “ridge”)
Para darle un marco especial al frame.

**config(cursor=x)**

(sustituir x por: “hand2”, “pirate”, ...)
Cambia la apariencia del cursor al introducirlo dentro del frame.

**Cambiar función del botón cerrar**

Se puede cambiar la funcionalidad del botón cerrar del marco de la app:
solo tenemos que indicar el método **protocol()** sobre nuestro objeto raíz al que le pasaremos como parámetro un texto (**“WM_DELETE_WINDOW”**) y una función.
Simplemente debemos crear una función y meter dentro el código que se ejecute cuando el usuario pulse en cerrar.

**raíz.protocol(“WM_DELETE_WINDOW”, hadler)**

```
def handler():
    print("Cerrando...")
    raíz.destroy()
```

# Título II: Creación de Frame

Antes del método **mainloop()**, creamos nuestro frame con el constructor default de la clase **Frame**. Y lo empaquetamos (agregamos) a nuestra ventana con el método **pack()** (ver métodos: pack).

```
marco = Frame()
marco.pack()
```

También podemos usar el constructor en el que especificamos contenedor y opciones del método **config()**:

```
marco = Frame(raiz, width="600", height="300")
marco.pack()
```

Es obligatorio darle tamaño a un frame para hacerlo visible. Para ello, nos aseguraremos de que nuestra ventana no tiene ejecutado el método **geometry()** (ya que la ventana se adapta al tamaño del frame). Y usamos el método **config()** para darle un tamaño:

```
marco.config(width="600", height="300")
```

Un frame, por defecto, tiene un tamaño fijo. Si creamos una raíz con un frame con distintos colores y redimensionamos la ventana, podemos ver cómo el frame permanece inmóvil mientras que la ventana se ensancha y encoge.

# config()

Todas las propiedades se pueden usar a la vez dentro del mismo método **config()** , siempre y cuando estén separados por comas.

**config(width=x, height=x)**

(sustituir x por un número en formato de string)
Para darle un tamaño al frame.
**marco.config(width=”600”, height=”300”)**

**config(bg=x)**

(sustituir x por un string que hace referencia a un color en inglés o a un código hexadecimal)
Para darle un color de fondo al frame.
**marco.config(bg=”red”)**

**config(bd=x)**

(sustituir x por un número en formato int)
Establece el tamaño del borde del frame.

**config(relief=x)**

(sustituir x por: “groove”, “sunken”, ...)
Para darle un marco especial al frame.

**config(cursor=x)**

(sustituir x por: “hand2”, “pirate”, ...)
Cambia la apariencia del cursor al introducirlo dentro del frame.

# pack()

Existen diferentes alternativas al método **pack()** que cambian el comportamiento del frame.
Todas las propiedades se deben usar a la vez dentro del mismo método **pack()** , siempre y cuando estén separados por comas.

**pack(side=x)**

(sustituir x por: “right”, “left”, “top”, “bottom”)

Hace que el frame se adhiera a una parte de la ventana.

**pack(anchor=x)**

(sustituir x por: “n”, “s”, “e”, “w” (north, south, east, west))
Hace que el frame se adhiera a una parte de la ventana (combinar con side para que este se adhiera a una esquina).

**pack(expand=x)**

(sustituir x por: “True”, “False”)
Da permisos a la propiedad **fill** para redimensionar el frame o no.

**pack(fill=x)**

(sustituir x por: “y”, “x”, “both”, “none”)
Establece que el frame se redimensiona en el eje y, x, en ambos o en ninguno. Algunos necesitan permisos de **expand** .

# Título III: Widgets

Los widgets son componentes gráficos que agregamos a un container (generalmente un frame).

Existen de diferentes clases (ver resto del capítulo), pero todos tienen cosas en común.

Para crear un widget, debemos crear un objeto perteneciente a una clase. Cada tipo de widget tiene una clase asociada. Este objeto generalmente lo crearemos con un constructor al que le tenemos que pasar por norma general su container y una serie de opciones (ver **opciones**).

Una vez creado nuestro objeto que hace referencia a un widget, hay que empaquetarlo, o agregarlo a su componente. Existen diferentes métodos de empaquetado (ver **Empaquetado**).

# Empaquetado

Existen 3 opciones a la hora de elegir cómo colocar los componentes: (Los tres métodos se usan sobre el objeto a empaquetar)

### pack()

Ajusta el tamaño del objeto contenedor al tamaño del objeto hijo (consultar título II: **pack()**).

### place(x=n, y=m)

Establece la distancia en píxeles (int) entre el componente y los bordes izquierdo (**n**) y superior (**m**).

### grid(row=n, column=m)

Divide el componente padre en una tabla y establece la posición (int) de la casilla en la que queramos almacenar cada componente.
Existen otros parámetros del método **grid()**:

**-sticky = x**

(x es un punto cardinal: “n”, “s”, “e”, “w”, “ne”, “se”, “sw”, “nw”)
(“nsew” para hacer que ocupe la casilla entera)
Establece la posición de la casilla donde aparece el componente.

**-padx = x**

(x es un número (int))
Establece la distancia en píxeles entre el componente y los límites de la casilla en el eje x (por la izquierda y por la derecha).

**-pady = x**

(x es un número (int))
Establece la distancia en píxeles entre el componente y los límites de la casilla en el eje y (por arriba y por abajo).

### opciones

Todas las opciones se implementan de la siguiente forma:

**nombreOpcion = “valor”**
Todas ellas son a su vez opciones del método **config()** (ver capítulo I: **config()** y capítulo II: **config()**)

**-text:** Texto que se muestra en el widget .
-**anchor:** Controla la posición del texto (center por defecto).
**-bg:** Color de fondo.
**-bitmap:** Mapa de bits que se mostrará como gráfico.
**-bd:** Grosor del borde (2px por defecto).
**-font:** Tipo de fuente. (*)
**-fg:** Color de la fuente.
**-width:** Ancho del widget en caracteres (no px).
**-height:** Altura del widget en caracteres (no px).
**-image:** Muestra la imagen en el widget en lugar de texto. (**)
**-justify:** Justificación del texto del widget (“center”, “right”, “left”).
**-show:** (Entry) Indica qué carácter se verá en vez del texto. (***)
-**textvariable:** Mantiene relacionada una variable string con el texto del widget si esta variable cambia, el texto del widget también.
**-cursor:** Cambia el cursor cuando este entra en el widget. (título II)
**-bd:** Especifica el tamaño del borde del widget. (título II, **config()**)
**-relief:** Le da un borde al widget. (título II, **config()**)
**-columnspan:** Indica que este componente ocupará varias columnas a la vez (como un entry largo) (especificar con un int).
**-command:** Establece el nombre de una función, como oyente.
**...**

***font=”nombre de la fuente tamaño_fuente tipo de la fuente”**
**font=”Times New Roman 18 bold”**
**font=”Arial 50 italic”**

**Esta debe ser o .png o .gif y debe estar en el mismo directorio que nuestro programa .py o .pyw . Se implementa así:
**imagen = PhotoImage(file=”foto.png”)**

**etiqueta = Label(frame, image=imagen)**
**etiqueta.place(x=100, y=200)**

*** Si ponemos: **show=”*”** , por cada carácter que se introduzca en un **Entry**, este representará un *.

# Título IV: Widget Label

Widget de etiqueta: nos permite mostrar texto o imágenes (como el JLabel de Java).

Creamos un objeto de la clase **Label** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

**objetoLabel = Label(contenedor, opcion1, opcion2, opcion3...)**
**objetoLabel.pack()**
(Número de opciones a declarar indefinido).

# Título V: Widget Entry

Widget de input de texto. Recuadro donde podemos introducir una línea de texto (similar al JTextField de Java).

Creamos un objeto de la clase **Entry** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

**objetoEntry = Entry(contenedor, opcion1, opcion2, opcion3...)**
**objetoEntry.pack()**
(Número de opciones a declarar indefinido).

*Para crear Entry para contraseñas, consultar título III: **opciones: show**

**insert(n, txt)**

Para insertar en la posición **n** del entry el texto **txt**. (no borra)

**delete(start, end)**

Borra la cadena de texto que se encuentra desde la posición **start** hasta la posición **end** (ambos int).

**get()**

Devuelve un string con el texto del entry.

**Detectar cada vez que se introduce una letra**

Para esto, debemos crear una variable que se asocie con el texto introducido en el entry. Posteriormente, ejecutamos el método **trace_add()** sobre esta variable, a la que le pasamos un texto (**"write"**) y la función que se ejecuta cada vez que alguien escribe o borra algo. Esta función debe recibir ilimitados parámetros (***args**)

```
variable = StringVar()
entry = Entry(root, textvariable=variable)
entry.pack()
variable.trace_add("write", actualizar)
def actualizar(*args):
    print("Alguien está escribiendo")
```

# Título VI: Widget Text

Widget de input de texto. Recuadro donde podemos introducir infinitas líneas de texto (similar al JTextArea de Java).

Creamos un objeto de la clase **Text** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

```
objetoText = Text(contenedor, opcion1, opcion2, opcion3...)
objetoText .pack()
```

(Número de opciones a declarar indefinido).

*Para borrar el Text, consultar título V, **delete()**
**Para introducir texto, consultar título V, **insert()**

### Scrollbar

1. Para añadir un **Scrollbar** a nuestro **Text**, creamos un objeto perteneciente a **Scrollbar** al que le pasamos como parámetros de constructor el container del **Text** y la opción **command** que igualamos a nuestro objeto **Text** .yview (o .xview para uno horizontal).
2. Una vez creado, debemos empaquetarlo sobre nuestro container (ver título III: **Empaquetado** (recomendable usar el **grid(row=n, column=m, sticky="nsew")**) (indicar misma fila que el **Text** pero una columna hacia la derecha. (o misma columna pero una fila más abajo para uno horizontal)).
3. Una vez hecho todo esto, debemos añadir la opción de **config()** **yscrollcommand=nombreScrollBar.set** (o xscrollcommand para uno horizontal):

```
area = Text(frame)
area.grid(row=0, column=0)

barra = Scrollbar(frame, command=area.yview)
barra.grid(row=0, column=1, sticky="nsew")

area.config(yscrollcommand=area.set)
```

## Título VII: Widget Button

Widget de un botón (similar al JButton de Java).

Creamos un objeto de la clase **Button** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

**objetoButton = Button(contenedor, opcion1, opcion2, opcion3...)**
**objetoButton .pack()**
(Número de opciones a declarar indefinido).

### Acciones de botones

Para que al pulsar nuestro botón, este desencadene una función, añadiremos una nueva función del método **config()** llamada **command**, donde especificaremos el nombre de la función que desencadena (sin paréntesis):

```
def funcion():
    print("Hola")

boton = Button(frame, text="Saludar", command=funcion)
boton.pack()
```

*Se les puede pasar parámetros a estas funciones siempre y cuando especifiquemos que se trata de una función lambda:

```
def funcion(msg):
    print(msg)

boton = Button(frame, text="Saludar", command= lambda: funcion('Hola'))
boton.pack()
```

Lo que estamos haciendo así es crear una función lambda que devuelve y hace lo mismo que la función que estamos indicando.

## Título VIII: Widget Radiobutton

Widget de botón de radio (similar a los JRadioButton de Java).

Creamos un objeto de la clase **Radiobutton** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

**objetoRb = Radiobutton(contenedor, opcion1, opcion2, opcion3...)**
**objetoRb .pack()**
(Número de opciones a declarar indefinido).

## Sincronizar Radiobutton

Podemos sincronizar diversos Radiobutton para que sólo uno esté seleccionado a la vez.
Para hacer esto, debemos indicarle dos valores a dos características del **config()**:
**-variable:** debemos declarar una variable de tipo int y asignarle la misma variable a ambas.
**-value:** debemos establecer un valor (a modo de ID) a cada botón.

```
var = IntVar()
RadioButton(frame, variable=var, value=1).pack()
RadioButton(frame, variable=var, value=2).pack()
```

## Obtener estado del botón

Al igual que los botones, podemos incluir una función que actuará de oyente del botón (ver título VII: **Acciones de botones**).
Dentro de esta función, podemos usar el método **get()** sobre la variable que especificamos en el campo **variable** . Este nos devolverá el **value** del botón seleccionado:

```
def oye():
    print(var.get())
#Si el 1º está seleccionado, imprime 1, si es el 2º, 2

var = IntVar()
```

**RadioButton(frame, variable=var, value=1, command=oye).pack()**
**RadioButton(frame, variable=var, value=2, command=oye).pack()**

# Título IX: Widget Checkbutton

Widget de botón de check (similar al JCheckButton de Java).

Creamos un objeto de la clase **Checkbutton** y usamos su constructor. Una vez creado el objeto, lo empaquetamos (ver título III: **Empaquetado**).

**objetoCb = Checkbutton(contenedor, opcion1, opcion2, opcion3...)**
**objetoCb .pack()**
(Número de opciones a declarar indefinido).

## Eventos de checkbutton

Para agregar eventos a los checkbutton y obtener su estado, necesitamos añadir 4 funciones:
**-command:** Donde indicamos la función oyente.
**-variable:** Donde indicamos la variable de tipo int que almacenará el estado de nuestro botón de check.
**-onvalue:** Valor que se almacenará dentro de la variable especificada en **variable** cuando el botón esté seleccionado.
**-offvalue:** Valor que se almacenará dentro de la variable especificada en **variable** cuando el botón no esté seleccionado.

```
var = IntVar()
def oyente():
    print(var)
Checkbutton(frame, variable=var, onvalue=1, offvalue=0,
command=oyente).pack()
```

De esta forma, cuando el botón esté seleccionado, la función imprimirá 1 y cuando este no lo esté, 0.
*Si tenemos diferentes botones de check, necesitaremos asignar diferentes variables a cada uno.

# Título X: Widget Menu

Widget de menú (similar a la clase JMenu y derivados de Java).

### Barra del menú

Creamos un objeto de la clase **Menu** y usamos su constructor especificando como container la raíz. Una vez creado el objeto, lo empaquetamos de forma especial:

**barraMenu = Menu (contenedor)**
**raiz.config(menu=barraMenu)**

### Opciones del menú

Creamos otro objeto de la clase **Menu** y usamos su constructor, especificando como container la barra de menú e igualando la propiedad **tearoff** a 0 para que no agregue ningún subelemento por defecto. Una vez creado el objeto, lo agregamos a la barra de menú:

**archivo = Menu(barraMenu, tearoff=0)**
**barraMenu.add_cascade(label=”Archivo”, menu=archivo)**
**ayuda= Menu(barraMenu, tearoff=0)**
**barraMenu.add_cascade(label=”Ayuda”, menu=ayuda)**
**…**

### Subelementos (desplegables) de opciones

Ejecutamos el método **add_command(label=x)**
Sobre nuestro objeto de opciones del menú (sustituyendo la x por el nombre que le queramos dar al subelemento).

Para agregar separadores (barras horizontales), llamar al método **add_separator()**

```
archivo.add_command(label=”Nuevo...”)
archivo.add_command(label=”Guardar”)
archivo.add_separator()
archivo.add_command(label=”Salir”)
```

*Agregar funciones los subelementos con la propiedad **command** haciendo referencia a una función:

```
def funcion():
    print(“Hola”)

archivo.add_command(label=”Nuevo...”, command=funcion)
```

# Título XI: Ventanas emergentes

**from tkinter import messagebox**

Ventanas modales para informar, avisar o permitir realizar tareas al usuario (similar a la clase JOptionPane de Java).
Todas las ventanas existentes:

```
messagebox.showinfo(title, message)
messagebox.showwarning(title, message)
messagebox.showerror(title, message)
messagebox.askquestion(title, message)
messagebox.askokcancel(title, message)
messagebox.askretrycancel(title, message)
messagebox.askyesno(title, message)
messagebox.askyesnocancel(title, message)
```

**Ventana informativa**

Ventana con un título, un icono informativo (círculo azul con “i”), un texto y un botón de aceptar. Al aparecer, reproduce el sonido de windows cuando algo es imposible.

**messagebox.showinfo(title, text)**
Donde **title** es el título de la ventana emergente y **text** es el texto que nos muestra.

## Ventana de advertencia

Ventana con un título, un icono de advertencia (triángulo amarillo con “!”), un texto y un botón de aceptar. Al aparecer, reproduce el sonido de windows cuando algo es imposible.

**messagebox.showwarning(title, text)**
Donde **title** es el título de la ventana emergente y **text** es el texto que nos muestra.

## Ventana Sí/No

Ventana con un título, un icono de pregunta (círculo azul con “?”), un texto y dos botones: Sí y No. Al aparecer no reproduce ningún sonido.

**messagebox.askquestion(title, text)**
Donde **title** es el título de la ventana emergente y **text** es el texto (la pregunta) que nos muestra.

*Devuelve un string según el botón pulsado (Sí = “yes”, No = “no”).

## Ventana Aceptar/Cancelar

Ventana con un título, un icono de pregunta (círculo azul con “?”), un texto y dos botones: Aceptar y Cancelar. Al aparecer no reproduce ningún sonido.

**messagebox.askokcancel(title, text)**

Donde **title** es el título de la ventana emergente y **text** es el texto que nos muestra.

*Devuelve un boolean según el botón pulsado.
(Aceptar = True, Cancelar = False).

### Ventana Reintentar/Cancelar

Ventana con un título, un icono de pregunta (triángulo amarillo con "!"), un texto y dos botones: Reintentar y Cancelar. Al aparecer, reproduce el sonido de windows cuando algo es imposible.

**messagebox.askretrycancel(title, text)**
Donde **title** es el título de la ventana emergente y **text** es el texto que nos muestra.

*Devuelve un boolean según el botón pulsado.
(Reintentar = True, Cancelar = False).

## Título XII: Ventanas emergentes de archivos

**from tkinter import filedialog**

Estas ventanas nos permiten trabajar con archivos, permitiéndonos elegir rutas para guardar archivos o para abrir archivos.
Todas las ventanas existentes:

filedialog.asksaveasfilename()
filedialog.asksaveasfile()
filedialog.askopenfilename()
filedialog.askopenfile()
filedialog.askdirectory()
filedialog.askopenfilenames()
filedialog.askopenfiles()

## Selector de archivos

Abre una ventana que nos permite seleccionar un archivo entre nuestros directorios.

**filedialog.askopenfilename(title=x, initialdir=i, filetypes=f)**
*#initialdir y filetypes opcionales*

(Sustituir x por (string) el título de la ventana)
(Sustituir i por (string) la ruta que se abre al ejecutar la ventana)
(Sustituir f por una tupla que contiene una subtupla, con el nombre que le damos al archivo y la extensión (ver **))

*Devuelve la ruta del archivo.
****filetypes = ((“Ficheros de texto”, “*.txt”), (“Ficheros de python”, “*.py”), ...)*** (mínimo 2) (Todos los ficheros = “*.*”).
Esto nos permite que sólo veamos los directorios y los archivos con esa extensión.

# Capítulo V: BBDD

Python puede acceder a multitud de SGDB, pero para todos estos es necesario saber **SQL**.
En este capítulo, veremos cómo acceder a BBDD usando SQLite.

## Título I: Pasos a la hora de conectarse a una BBDD

1. Abrir-Crear la conexión con la BBDD.
2. Crear un puntero (nos permite hacer querys y manejar los resultados de esta).
3. Ejecutar un query (consulta) SQL
4. Manejar los resultados de la query.
    4.1. Create, Read, Update, Delete (CRUD).
5. Cerrar el puntero
6. Cerrar la conexión con la BBDD

## Título II: SQLite

Este SGDB es de código abierto y se guarda como un único fichero en host. Ocupa muy poco espacio en memoria, es muy eficiente y rápido, es multiplataforma, no necesita configuración.
No admite cláusulas anidadas, ni existen usuarios.
La BBDD se almacena como un fichero byte (sin extensión) en el mismo directorio que donde se encuentra nuestro programa .py
Dado que es un archivo byte no se puede visualizar nada, aunque es posible descargar un visor SQLite de Google para poder ver gráficamente la BBDD.

## Título III: Conexión a una BBDD SQLite

Para empezar, debemos importar la librería correspondiente para trabajar con SQLite: **import sqlite3**

1. Creamos un objeto que igualamos a la instrucción **sqlite3.connect(str nombreBBDD)** , donde le indicamos el nombre de la BBDD. Este objeto hará referencia a nuestra BBDD.
Si nuestra BBDD no existe, la creará; si esta existe, se conectará a ella. (es interesante meter dentro de un try catch la instrucción de crear la tabla, para saber si la BBDD existía (se ejecutará el catch) o no (el try))

2. Creamos nuestro objeto cursor igualándolo al objeto creado anteriormente y al método **cursor()**: **objetoBBDD.cursor()**

3. Para ejecutar el query, escribimos el objeto cursor acompañado de **.execute(str q)** y dentro de él ejecutamos la sentencia SQL.
*IMPORTANTE* La secuencia SQL debe ir entre comillas dobles y las comillas de la sentencia deben ser simples.
*IMPORTANTE* Si ejecutamos un query que cambia la estructura o el contenido de una tabla, debemos ejecutar el método **commit()** sobre nuestro objeto de BBDD, para confirmar la ejecución del query.

4. Para manejar los resultados de una query, primero ejecutamos la query **select** de la forma que hemos visto en el punto 3 y a continuación, creamos una variable donde se almacenará una lista de tuplas. Para ello, igualamos esta variable a nuestro cursor, ejecutando el método **fetchall(): nombre = cursor.fetchall()**

5. Cerramos el cursor ejecutando el método **close()** sobre nuestro objeto cursor.

6. Cerramos la conexión ejecutando el método **close()** sobre el objeto que creamos para crear la BBDD.

```
import sqlite3
conexion = sqlite3.connect("BBDD")
```

```
cursor = conexion.cursor()
cursor.execute(sentenciaSQL)
resultado = cursor.fetchall()
cursor.close()
conexion.close()
```

## Insertar varios registros a la vez

Si queremos insertar varios registros, es recomendable crear una lista de varias tuplas en la que cada tupla contenga la información de un registro y posteriormente, ejecutamos el método **executemany** en el que le indicamos la instrucción SQL dedicada a insertar nuevos registros, pero en **values()** introduciremos tantos interrogantes separados por comas como campos vayamos a rellenar. terminamos la instrucción SQL, indicamos una coma y posteriormente la lista que contiene nuestros registros:

```
cursor.execute("create table productos (nombre varchat(20),
precio integer )")
lista = [
        ("Camiseta", 10),
        ("Chaqueta", 20),
        ("Pantalones", 15),
]
cursor.executemany("insert into productos values(?,?)", lista)
```

*Si uno de los campos de nuestra BBDD es una key autoincrementable, no podemos especificar un ?. Especificar un null en su lugar:

```
cursor.execute("create table productos (id integer primary key
autoincrement , nombre varchat(20), precio integer )")
lista = [
        ("Camiseta", 10),
        ("Chaqueta", 20),
```

```
        ("Pantalones", 15),
]
cursor.executemany("insert into productos values(null,?,?)", lista)
```

# Capítulo VI: Expresiones Regulares

Las expresiones regulares o regex son una secuencia de caracteres que forman un patrón de búsqueda.
Sirven para el trabajo y procesamiento de texto, por ejemplo:
Para buscar un texto que se ajusta a un formato determinado, para buscar si existe o no una cadena de caracteres dentro de un texto, contar el número de coincidencias dentro de un texto…
(Consultar reseñas, manual de expresiones regulares).

## Título I: Métodos

Para trabajar con cualquier expresión regular, tendremos que importar el siguiente módulo:
**import re**

### re.search(palabra, texto)

Recibe dos parámetros: el texto a localizar y el texto donde lo queremos localizar. Si devuelve un objeto, el texto está y si no, devuelve **None**.

```
texto = “Hola, estoy aprendiendo python”
if search(aprendiendo, texto) is not None:
        print(“Aprendiendo está en el texto”)
elif:
        print(“Aprendiendo no está en el texto”)
```

### start()

Se aplica sobre el objeto que devuelve **re.search()** , devuelve (int) el carácter donde empieza la palabra a buscar en el re.**search()** .

### end()

Se aplica sobre el objeto que devuelve **re.search()** , devuelve (int) el carácter donde termina la palabra a buscar en el re.**search()** .

**span()**

Se aplica sobre el objeto que devuelve **re.search()** , devuelve una tupla de dos ints, el primero con el resultado de **start()** y el segundo con el resultado de **end()** .

**re.findall(palabra, texto)**

Recibe dos parámetros: primero el texto a buscar y segundo el texto donde queremos que se busque el primer parámetro.
Devuelve una lista con el primer parámetro repetido tantas veces como se ha encontrado en el texto, por lo que si ejecutamos el método **len()** y le pasamos como parámetro el resultado de este método, podemos ver el número de veces que aparece el primer texto en el segundo texto.

**re.match(palabra, texto)**

Devuelve True si la cadena **palabra** está al principio de la cadena **texto**, False si no.

**re.match(palabra, texto, re.IGNORECASE)**

Desactiva el case sensitive.

# Título II: Metacaracteres

Estos caracteres nos permiten afinar o cambiar cosas en las búsquedas con los métodos vistos anteriormente.

**Encontrar todas las cadenas de texto que comienzan por la misma cadena de texto**

Para localizar todas las cadenas de texto que comienzan por la misma palabra, podemos usar el método **re.findall**, especificando como

primer parámetro la palabra por la que queremos que empiece la frase después del carácter '^':

**nombres = ['Juan Antonio', 'Juan Alberto', 'Juan José', 'Antonio']**

**lista = re.findall('^Juan', nombres)**

*#lista = ['Juan Antonio', 'Juan Alberto', 'Juan José']*

*Este metacaracter también puede encontrar cadenas de texto unidas:

**protocolo = re.findall('^https', 'https://laksjdfjasldkf')**

## Encontrar todas las cadenas de texto que terminan por la misma cadena de texto

Para localizar todas las cadenas de texto que terminan por la misma palabra, podemos usar el método **re.findall**, especificando como primer parámetro la palabra por la que queremos que empiece la frase seguido del carácter '$':

**nombres = ['Juan López', 'José López', 'Juanjo López', 'Antonio']**

**lista = re.findall('López$', nombres)**

*#lista = ['Juan López', 'José López', 'Juanjo López']*

*Este metacaracter también puede encontrar cadenas de texto unidas:

**es = re.findall('.es$', 'https://laksjdfjasldkf.es')**

## Encontrar un caracter / caracteres en una cadena de texto

Podemos usar el método **re.findall** para localizar un carácter en específico dentro de una cadena de texto. Si queremos localizar la ñ en un texto:

**buscar = re.findall('[ñ]', 'España')**

En caso de que la ñ esté en España, buscar será igual a España.

*SI queremos buscar varios caracteres:

**buscar = re.findall('[rst]', 'canal fiesta radio')**

En caso de que la r, la s y la t (no importa el orden) estén en 'canal fiesta radio', buscar será igual a canal fiesta radio.

## Encontrar en variante masculina o femenina

Si queremos buscar una palabra en un texto con la instrucción **re.findall** que sea susceptible de tener género (niño/niña), podemos utilizar los metacaracteres para sólo hacer una búsqueda:
**buscar = re.findall('niñ[oa]s', lista)**
En caso de que en la lista **lista** aparezca la palabra niños o niñas, buscar será igual a esta palabra.

*Este metacaracter no tiene por qué usarse con flexiones de género, aunque este es de los casos más comunes para este metacaracter, junto con las tildes: **re.findall('cami[oó]n', lista)**

## Búsqueda por rango de caracteres

Podemos hacer que **re.findall** nos devuelva todos los elementos que contienen en su interior un carácter entre uno y otro (ordenados alfabéticamente):
**buscar = re.findall('[a-d]', 'abecedario')**
En este caso, **buscar** será iguala a **abecedario**, ya que esta contiene o la a, o la b, o la c, o la d en su interior.

*Es case sensitive por lo que si queremos un rango de minúsculas, especificaremos como límite dos letras minúsculas: [a-c]
por el contrario, si queremos un rango de mayúsculas, especificaremos como límite dos mayúsculas: [A-C] .

**El rango también puede ser numérico: [0-5] .

***El rango también puede ser negado, devolviendonos los que no cumplen esa condición: [^0-5] o [^a-c] .

****Podemos especificar diferentes rangos con relación disyuntiva, mientras que uno de estos se cumpla, se mostrará la palabra: [0-5a-c] .

## Búsqueda ignorando una letra

Podemos hacer que la búsqueda ignore algunos caracteres si sustituimos ese carácter por un punto:

**nombre = 'Sara Martínez'**

**re.match('.ara', nombre)**

Si nombre comienza por un carácter cualquiera seguido de 'ara', esta instrucción devolverá **True**.

## Buscar si una cadena comienza por un número

Podemos hacer que **re.match()** busque si una cadena comienza por un número o no indicando **\d** en lugar del número

**cadena = '1234'**

**cadena2 = 'a1234'**

**re.match('\d', cadena)** *#True*

**re.match('\d', cadena2)** *#False*

# Capítulo VII: Funciones Decoradoras

Las funciones decoradores son unas funciones que “decoran” a otras funciones, añadiendoles más funcionalidades.
La estructura de un decorador depende de 3 funciones (A, B y C), donde A recibe como parámetro a B para devolver C.

```
def funcionA(funcionB):
    def funcionC():
        código
    return funcionC
```

Un decorador devuelve una función.
Los decoradores nos permiten añadir la misma funcionalidad a diferentes funciones.

### Utilidades de funciones decoradoras

Algunas utilidades de las funciones decoradoras pueden ser las de, a la hora de trabajar con BBDD, decorar con una función decoradora que se encargue de cerrar la base, todas las funciones que accedan a esta.

## Título I: Construir una función decoradora sencilla

Para comenzar con los decoradores, crearemos un decorador cuya función es nula pero nos servirá para familiarizarnos con la sintaxis.
El ejemplo consiste en que tenemos dos funciones que imprimen ciertos valores y queremos hacer que estas dos funciones impriman dos líneas de ‘_’ , una antes y otra después:
1. Primero, declaramos la función decoradora y todas las funciones que queramos decorar.
2. Posteriormente, colocamos en la línea inmediatamente anterior a la función que queremos decorar una ‘@’ y el nombre de la función decoradora.

```
def funcion_decoradora(funcion_parametro):
    def funcion_interna():
        print('_____________')
        funcion_parametro()
        print('_____________')
    return funcion_interna

@funcion_decoradora
def f1():
    print(1)

@funcion_decoradora
def f2():
    print(2)
```

## Título II: Funciones decoradoras que reciben parámetros

Si las funciones que queremos decorar reciben parámetros, tenemos que hacer que nuestra función decoradora también reciba parámetros. Tomando un ejemplo similar al título I, lo que tenemos que hacer es indicarle a las funciones normales los parámetros que reciben e indicar a la función C y al uso de la B que van a recibir un número indefinido de parámetros:
También podemos indicar que la decoradora puede recibir un número indeterminado de elementos y además un número indeterminado de elementos clave, valor (o sólo de uno).
(Consultar Capítulo I, Título V, **tuplas como parámetros**)
(Consultar Capítulo I, Título V, **diccionarios como parámetros**)

```
def funcion_A(funcion_B):
    def funcion_C(*args, **kwargs):
```

```
            print('_________')
            funcion_B(*args, **kwargs)
            print('_________')
        return funcionC

@funcion_A
def suma(num1, num2, num3):
        print(num+num2+num3)

@funcion_A
def resta(num1, num2):
        print(num1-num2)

@funcion_A
def potencia(base, exponente):
        print(pow(base, exponente))

suma(10, 5, 7)
resta(9, 6)
potencia(base=5, exponente=6) #Aquí estamos usando el **kwargs
```

# Capítulo VIII: Documentación

La documentación es el incluir comentarios en clases, métodos, módulos… nos permite trabajar en equipo o en aplicaciones complejas.

## Título I: Ver la documentación

Podemos ver la documentación de un código (imprimiendo en consola o en una GUI).
Esto lo hacemos haciendo referencia al bloque donde está contenida seguido de **.__doc__** :

```
def funcion():
    """Esta función imprime hola"""
    print('Hola')
print(funcion.__doc__)
```

*Salida: Esta función imprime hola*.

*También podemos hacer algo similar a esto indicando el método **help()** al que le pasamos como parámetro el nombre de la función que no entendemos:

```
help(funcion)
```

Las clases también se pueden usar en este método, el cual nos devuelve información sobre toda la clase.

**Si estas funciones son métodos de una clase, tenemos que indicar el nombre de la clase, . y el nombre del método:

```
help(Clase.metodo)
```

## Título II: Pruebas: doctest.testmod()

Podemos comprobar si ciertas funciones/métodos… que acabamos de programar funcionan bien con la ayuda del módulo **doctest**

Para comprobar si un método funciona, debemos primero crear la documentación y después de la documentación pero dentro del comentario, debemos escribir tres '>' seguidos de la línea de código que queremos probar. En el siguiente renglón, especificamos el resultado que esta función nos tendría que dar.

Posteriormente, importamos el módulo **doctest** y ejecutamos el método **doctest.testmod()** , el cual sólo ejecutará sentencias como la que acabamos de hacer.
Si el programa está bien hecho, no nos devolverá nada en consola al ejecutarse pero si por el contrario, la función devuelve algo diferente a lo que nos esperábamos, imprimirá un error en consola.
(Este ejemplo es muy sencillo pero nos servirá para ver cómo podemos realizar pruebas)

```
def suma(num1, num2):
    """
    Esta función se encarga de sumar dos números
    >>>suma(5, 10)
    15.0
    """

    return num1+num2

import doctest
doctest.testmod()
```

En este caso, al ejecutarse el programa no pasaría nada, ya que al esperar lo que se esperaba, no imprime nada en consola, por lo que funciona bien.

## Realizar varias pruebas

Es posible realizar varias pruebas dentro de la misma función:

```
def suma(num1, num2):
    “””
    >>>suma(5, 4)
    9
    >>>suma(10, 5)
    5
    “””
    return num1+num2

import doctest
doctest.testmod()
```

# Título III: Pruebas: doctest.testmod() con bucles o condicionales anidados

Podemos realizar pruebas usando bucles o condicionales.
(Este ejemplo es bastante sencillo y no sería necesario usarlas pero como ejemplo nos sirve para entender cómo se haría)

```
import math
def raizCuadrada(lista):
    “””
    Devuelve una lista con los resultados de la raíz cuadrada de
    la lista indicada

    >>> lista = []
    >>> for i in [4, 9, 16]:
    …          lista.append(i)  #*
    >>>raizCuadrada(lista)
    [2.0, 3.0, 4.0]
    “””
```

**return [math.sqrt(n) for n in lista]**

(*) Al haber código anidado dentro del comentario para el test, indicamos con los tres '.': … que la línea siguiente está anidada dentro de la línea anterior.

## Esperar un error

También podemos indicar al programa que esperamos recibir un error:

```
“””
>>>raizCuadrada([54, -90])
Traceback (most recent call last):
    …
ValueError: math domain error
“””
```

(Sustituir líneas por el error correspondiente en cada caso).

# Capítulo IX: Crear un archivo ejecutable

Para crear un archivo ejecutable, primero debemos instalar pyinstaller.

**Instalar pyinstaller**

Abrir cmd y escribir el comando:
**pip install pyinstaller**

1. Una vez que tengamos instalado el pyinstaller, nos dirigimos al directorio (o carpeta) donde tenemos nuestro archivo .py y copiamos la ruta.Nos dirigimos a cmd e introducimos: **cd ruta** (sustituyendo **ruta** por la ruta en cuestión).

2. Ejecutamos el comando: (**VER PUNTO 2 ENTERO**).
**pyinstaller nombre.py**
(sustituyendo **nombre** por el nombre de nuestro archivo .py)

*Si queremos que no se vea la pestaña de cmd cada vez que ejecutamos el .exe, agregar este atributo: **--windowed**

**Si queremos obtener un único archivo .exe que además se pueda ejecutar en todos los ordenadores sin necesidad de tener python instalado, agregamos también el siguiente atributo: **--onefile**

***Si queremos que nuestro .exe tenga un icono, debemos tener un archivo .ico en la misma carpeta que nuestro archivo .py y agregar el atributo: **--icon=./nombreicono.ico**

La instrucción final quedaría así:
**pyinstaller --windowed --onefile --icon=./nombrei.ico nombre.py**

3. Dentro de la carpeta dist>nombre> se encuentra nuestro .exe , aunque para que este funcione debe de ir acompañado por todos esos archivos, además de que se despliega una pestaña de cmd.
(Ver punto 2, * y ** para solucionarlo)

# Capítulo X: Sockets