



Índice

Tipo de lenguaje	3
Entorno de Desarrollo Integrado (IDE)	4
Sintaxis	6
Variables	7
Operadores	9
Strings	11
Tuplas	14
Listas (<i>vectores</i>)	15
Sets	17
Diccionarios (<i>JSON</i>)	18
Estructuras de control	20
Manejo de errores	22
Funciones y procedimientos	23
Introducción de datos por usuario	24
Funciones lambda o anónimas	25
Funciones generadoras	26
Módulos (<i>bibliotecas</i>)	27
Clases (objeto)	29
Paquetes	32
Archivos	34
Programación Orientada a Objetos	38
Encapsulamiento	39
Herencia simple y múltiple	40
Palabra reservada super	42
Polimorfismo y métodos abstractos	43
Función Filter	44
Función Map	46
Interfaces gráficas	47
Widgets	50
Grids	56
Ventanas emergentes	59
Explorador de archivos	60
Personalización de colores	61
Threads	62
Generación de un ejecutable	63
Bases de datos	64
Solicitudes HTTP	68



Tipo de lenguaje

Compilado	Interpretado
<ul style="list-style-type: none">• Debe ser compilado antes de ejecutarse• El compilador genera un binario ejecutable a partir del código• Se compila para una arquitectura y sistema operativo específico 	<ul style="list-style-type: none">• No requiere compilación• Un programa llamado “intérprete” se ocupa de traducir en tiempo real el código a medida que se quiere ejecutar.• Se necesita tener el intérprete instalado para poder ejecutar el código.

Python es un lenguaje **interpretado**, lo cual hace que el tiempo de ejecución sea más lento ya que requiere otro programa paralelo que vaya traduciendo el código, pero a su vez tiene mayor compatibilidad para todos los tipos de sistema operativo ya que no se compila para uno en específico.

Además, es un lenguaje **multiplataforma** ya que puede utilizarse tanto en plataformas Windows, Linux y Mac.

Lenguaje multiparadigma

- ✓ Imperativo (va de línea a línea) → como C
- ✓ Orientado a objetos
- ✓ Funcional
- ✓ Reflexivo

Tipo de tipado

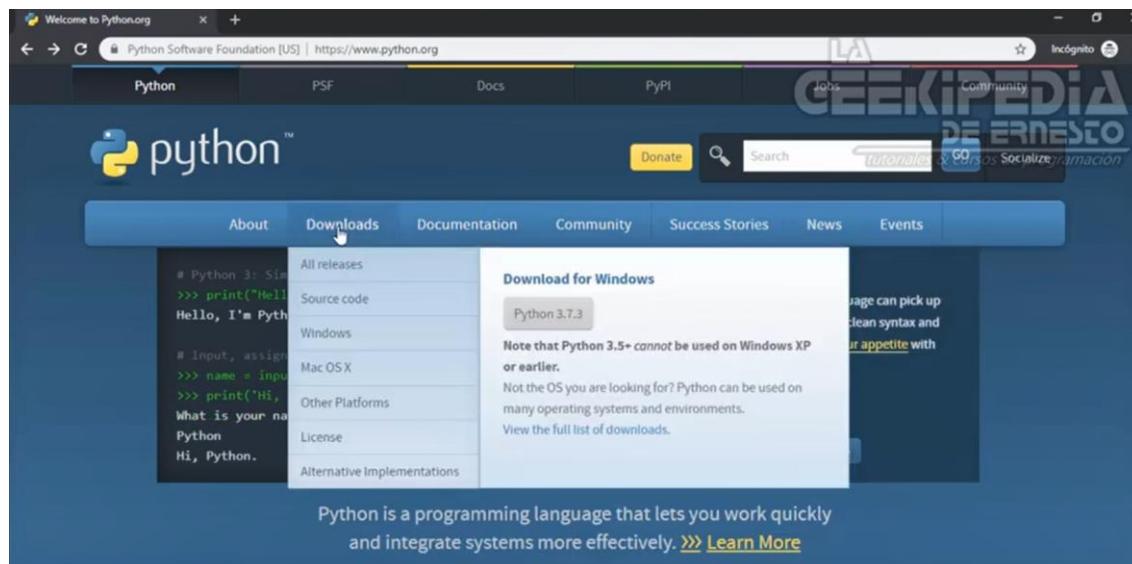
- ✓ Tipado fuerte Todas las variables tienen asignado un tipo de dato específico, y no se puede realizar operaciones entre variables de distinto tipo.
- ✓ Tipado dinámico Una variable puede tomar valores de distinto tipo. Una variable puede ser inicialmente un entero y luego puede ser un decimal.
- ✓ Tipado implícito Reconoce automáticamente el tipo de dato introducido.



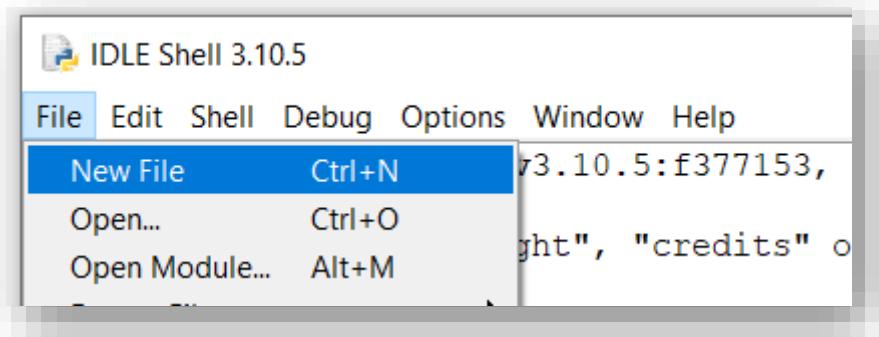
Entorno de desarrollo

Es un entorno de desarrollo integrado específico para programar en Python que sirve para Windows. Se puede descargar desde la página oficial de Python:

www.python.org



Una vez descargado el **IDLE**, se debe crear un nuevo archivo para comenzar a escribir el código.



Luego, una vez que se tenga creado un archivo .py (extensión de Python), se lo puede ejecutar con el comando `python` desde la consola del sistema.



En Linux, queda guardado con el nombre Python3. Se puede ver la versión instalada escribiendo en consola “python3 -v”

```
ernesto@ernesto-VirtualBox: ~
Archivo Editar Ver Buscar Terminal Ayuda
ernesto@ernesto-VirtualBox:~$ sudo apt install python3
[sudo] contraseña para ernesto:
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
python3 ya está en su versión más reciente (3.6.7-1~18.04).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 52 no actualizados.
ernesto@ernesto-VirtualBox:~$ sudo apt-get install idle
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
idle ya está en su versión más reciente (3.6.7-1~18.04).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 52 no actualizados.
ernesto@ernesto-VirtualBox:~$
```

Una buena IDE en Linux también puede ser ptthon.



Syntaxis

En Python no se utilizan llaves ({}) para separar fragmentos de código. En cambio, la **indentación** es obligatoria y funciona para la separación de fragmentos de código.

No se utiliza punto y coma (;) al final de las líneas.

Las constantes y el main no existen. Es de buena práctica utilizar una variable en mayúsculas como constante y no modificarla bajo ningún concepto.

No existen los punteros.



Variables

Las variables son un espacio en memoria donde se almacenan datos.

Las constantes no existen en Python. Es de buena práctica utilizar una variable en mayúsculas como constante y no modificarla bajo ningún concepto.

En Python, las variables detectan automáticamente el tipo de dato que está almacenando. Es decir, no es necesario definir el tipo de dato.

Tipos de datos básicos

- Lógicos (**bool**): pueden ser *True* o *False* (primera letra en mayúsculas).
 - Enteros (**int** o **long**): no tienen decimales.
- Numéricos
 - Reales (**float**): contienen decimales.
 - Complejos (**complex**): tienen una parte real y una imaginaria.
- Cadenas de texto (**strings**): se escriben entre comillas simples o dobles.
 - Si se utilizan comillas triples, se puede introducir saltos de línea sin usar /n

```
# Strings
print("Hello world")
print('Hello World')
print('''Hello World''')
print(""""Hello World""")
```

5 + 6j

Para crear una variable, se debe asignar un nombre sin espacios, seguido de un signo igual (=) y un dato con el cual inicializar.

```
numero = 1
```

Para convertir un tipo de dato numérico a un string se utiliza el método **str(numero)**.

```
numero = 1
str(numero)
# '1'
```

En Python no existe el tipo de dato char (un solo carácter).

Se puede saber qué tipo de dato tiene asignado una variable con la función '**type(nombre de la variable)**'.

```
numero_favorito = "doce"
type(numero_favorito)
# <class 'str'>
```



Tipos de datos de mayor complejidad

- **Tuplas**: agrupa datos por medio de paréntesis que no cambian con el tiempo, y no necesariamente deben ser de un mismo tipo de dato.

```
tupla = (3, True, "hola")
```

- **Listas (vectores)**: agrupa datos en una misma variable por medio de corchetes y no necesariamente deben ser de un mismo tipo de dato.

```
lista = [10, 4.5, 2+6j, "hola", True, (2, 5)]
```

- **Sets**: colección desordenada y sin índice.

```
set = {"Hola", 23, (1, 2)}
```

- **Diccionarios (structs)**: agrupa datos que pertenecen a una misma entidad.

```
diccionario = {
    "Nombre": "Guille",
    "Edad": 27
}
```

Se puede conocer todos los métodos asociados a un tipo de dato con el uso del método `dir(variable)`.

Una variable se puede eliminar mediante la palabra reservada `del`

```
del variable
```

Verificar tipo de variable

Se puede verificar en código que una variable es de determinado tipo de dato usando alguno de las siguientes formas.

<pre>texto = "Hola" entero = 1 decimal = 0.2 booleano = True tupla = ("una", "tupla") lista = ["una", "lista"] colección = {"un", "set"} diccionario = {"Nombre": "Guille"}</pre>	<pre>(type(texto) == str) (type(entero) == int) (type(decimal) == float) (type(booleano) == bool) (type(tupla) == tuple) (type(lista) == list) (type(colección) == set) (type(diccionario) == dict)</pre>	<pre>isinstance(texto, str) isinstance(entero, int) isinstance(decimal, float) isinstance(booleano, bool) isinstance(tupla, tuple) isinstance(lista, list) isinstance(colección, set) isinstance(diccionario, dict)</pre>
---	---	---

```
aString = "1234"
if aString.isdigit(): True
```



Operadores

Símbolo que indica que debe ser llevada a cabo una operación especificada sobre cierto número de operador (tipo de dato).

 **Asignación** = Asigna un valor a un operando.

 **Aritméticos**

❖ Suma	+	Suma dos números o concatena textos.
❖ Sustracción	-	Resta dos números.
❖ Multiplicación	*	Multiplica números.
❖ División	/	Divide números.
❖ División entera	//	Divide números y obtiene un entero.
❖ Resto	%	Divide números y se obtiene el resto de dicha división.
❖ Exponenciación	**	Aplica potencias.

 **Relacionales** Compara dos valores del mismo tipo y devuelve un valor lógico.

True o False

❖ Mayor estricto	>
❖ Menor estricto	<
❖ Mayor o igual	>=
❖ Menor o igual	<=
❖ Desigualdad	!=
❖ Igualdad	==

 **Lógicos**

❖ Conjunción	and	Si todo es verdadero, retorna verdadero.
❖ Disyunción	or	Si al menos uno es verdadero, retorna verdadero.
❖ Negación	not	Convierte lo verdadero en falso y viceversa.

En Python, no existe la operación ‘++’.



Operaciones con cadenas de texto

- Asignación (`+=`)

Se agrega texto a una cadena de texto ya existente.

```
texto = "Hola"  
texto += ", ¿cómo estás?"  
# 'Hola, ¿cómo estás?'
```

- Concatenación (`+`)

Une varias cadenas de texto para formar otra de mayor tamaño.

Para concatenar texto con números, es necesario usar el método `str`.

```
texto1 = "Hola"  
texto2 = ", ¿cómo estás?"  
textoFinal = texto1 + texto2  
# 'Hola, ¿cómo estás?'
```

```
saludo = "Hola, hoy es "  
numero = 12  
saludo + str(numero)  
# 'Hola, hoy es 12'
```

- Comparación (`==`)

Compara dos cadenas de caracteres y devuelve un valor booleano.

```
texto1 = "Hola"  
texto2 = "Holas"  
texto1 == texto2  
# False
```

- Extracción (`[n:m]`)

Saca fuera de la cadena de texto una porción de la misma según su posición dentro de ella.

Para esto hay que tener en cuenta cómo son las posiciones de los caracteres de un texto.

```
texto = "Hola Guillermo"  
extraccion = texto[5:14]  
# Guillermo
```





Strings

Son cadenas de caracteres, los cuales forman textos, y se pueden trabajar con diversas funciones.

- Longitud (`len`)

Función que permite conocer la longitud de una cadena de caracteres (o bien, el número de elementos de un objeto).

```
texto = "Hola"  
len(texto)  
# 4
```

- Búsqueda (`.find o .index`)

Devuelve la posición dentro de la cadena de texto donde se ubica la primera letra del texto a buscar (-1 si no existe).

```
texto = "Hola Guillermo"  
busqueda = texto.find("Guillermo")  
      ↘ 5
```

```
texto = "Hola Guillermo"  
busqueda = texto.index("G")  
      ↘ 5
```

- Reemplazar (`.replace`)

Reemplaza una subcadena de texto por otra.

```
texto = "Hola mundo"  
texto.replace("Hola", "Adiós")  
# 'Adiós mundo'
```

- Contar todos los caracteres que existen (`.count(char)`)
- Verificar si un texto empieza de determinada forma (`.startswith(text)`): True o False
- Verificar si un texto empieza de determinada forma (`.endswith(text)`): True o False
- Convertir todo a mayúscula (`.upper()`)
- Convertir todo a minúscula (`.lower()`)
- Cambiar mayúsculas por minúsculas y viceversa (`.swapcase()`)
- Forzar sólo la primer letra en mayúsculas (`.capitalize()`)
- Separar texto en vector (`.split`): si se usa vacío, separa por espacios en blanco.



Al trabajar con cadenas de caracteres, muchas veces es necesario eliminar espacios en blanco o algún carácter en específico, ya sea al inicio o al final de la cadena. Para ello, se pueden utilizar algunos de los siguientes métodos:

- **strip** Elimina caracteres especificados al inicio y al final del texto. Si no se le especifican caracteres, sólo elimina espacios en blanco o saltos de línea.
- **lstrip** Se utiliza para eliminar sólo caracteres al final de una cadena de caracteres.
- **rstrip** Se utiliza para eliminar sólo caracteres al final de una cadena de caracteres.

```
texto = "Hola Guillermo "
texto.strip() # 'Hola Guillermo'
texto.strip(" Ho") # 'la Guillerm'  
  
borra todos los caracteres  
especificados que encuentre tanto  
al inicio como al final de la  
cadena de texto
```

Concatenación en **Python3**

En Python3, se puede concatenar textos mediante el uso de `.format`.

Dicho método se puede utilizar de varias maneras:

```
nombre = "Guille"
edad = 27
"Hola {} tienes {} años".format(nombre, edad)
# 'Hola Guille tienes 27 años'
```

```
"Hola {nombre} tienes {edad} años".format(nombre = "Guille", edad = 27)
# 'Hola Guille tienes 27 años'
```

```
nombre = "Guille"
edad = 27
          0      1
"Hola {1} tienes {0} años".format(edad, nombre)
# 'Hola Guille tienes 27 años'
```

Otra forma de concatenar texto en Python 3.6, es mediante **f-strings**. Los cuales se utilizan de la siguiente manera:

```
nombre = "Guille"
edad = 27
f"Hola {nombre} tienes {edad} años"
# 'Hola Guille tienes 27 años'
```

Este método también permite agregar expresiones que sean válidas. Por ejemplo:

```
nombre = "Guille"
edad = 27
f"Tienes {20 + 7} años"
# 'Tienes 27 años'
```



Tupla

Conjunto de datos que no pueden ser modificados. Sus únicas funciones son **index** y **count**.

Si se crea una tupla con un solo elemento, se obtendrá un entero (**int**) o el tipo de dato que se haya introducido.

```
tupla = (1)  
type(tupla) # int
```

Si se desea forzar la creación de una tupla con un solo elemento, es necesario agregar una coma.

```
tupla = (1,)
```

Los tipo de datos Tupla se pueden utilizar en funciones donde no se conoce la cantidad de elementos que se pasarán por parámetro (como un parámetro rest).

```
def getCiudad(*ciudades):
```



El asterisco indica que se recibirán una cantidad desconocida de elementos en formato de tupla

De esta forma, se puede trabajar con una función generadora y tuplas.

```
# Función generadora con tupla como parámetro  
def getCiudad(*ciudades):  
    for elemento in ciudades:  
        yield elemento  
  
ciudades = getCiudad("Avellaneda", "Gerli", "Lanús")  
  
nombre1 = next(ciudades) # 'Avellaneda'  
nombre2 = next(ciudades) # 'Gerli'  
nombre3 = next(ciudades) # 'Lanús'
```



Listas (vectores)

Pueden tener distintos tipos de datos.

No es necesario indicar el tope del vector.

```
vector = [1, 1, 2, 3, 5, 8, 13]
```

Se utiliza con corchetes ([]) y se accede a los elementos de los vectores de la misma forma que en C, indicando el número de la posición del elemento.

Si se utiliza una posición negativa, recorre el vector desde el final.

```
# Recorrer un vector hasta la posición 3 (exclusive)
vector[:3]                                     >>> [1, 1, 2]

# Recorrer un vector desde la posición 3
vector[3:]                                     >>> [3, 5, 8]

# Recorrer un segmento del vector
vector[2:4]                                     >>> [2, 3]

# Recorrer los últimos 3 elementos del vector
vector[-3:]                                    >>> [5, 8, 13]

# Recorrer vector de dos en dos
vector[::-2]                                    >>> [1, 2, 5, 13]

# Recorrer vector hacia atrás
vector[::-1]                                    >>> [13, 8, 5, 3, 2, 1, 1]
```

- Longitud (`len`) Devuelve la cantidad de elementos.

```
len(vector) # 7
```

- Añadir elemento al final de la lista (`.append`)
- Añadir todos los elementos de otra lista (`.extend`)
- Añadir un elemento en una posición específica (`.insert`)

```
vector.insert(0, "Hola") # ["Hola", 1, 1, 2, 3, 5, 8, 13]
```

- Eliminar el último elemento de la lista (`.pop`)
- Eliminar un elemento de la lista (`.remove`)
- Eliminar todos los elementos de una lista (`.clear`)
- Invertir lista (`.reverse`)
- Ordenar lista (`.sort`)
- Contar la cantidad de veces que hay un elemento (`.count`)
- Copia una lista en otra lista (`.copy`)
- Buscar elemento (`.index`) o verificar si existe (`in`)

```
1 in vector # True
```

```
# Recorrer una lista
for elemento in vector:
    print(elemento)
```



Se puede crear una lista (**vector**) a partir de una tupla con **list**.

```
tupla = (3, 5, 2)  
list(tupla) # [3, 5, 2]
```

```
list(range(1, 100)) # [1, 2, 3, ..., 99]
```

Es posible buscar, dentro de una lista, el valor máximo de un elemento con la función **max**.

Se puede convertir una lista de strings en una cadena de la siguiente manera:

```
texto = ", ".join(lista)
```



Sets

Colección desordenada de elementos sin índice.

```
set = {"Hola", 23, (1, 2)}
```

Se puede buscar si un elemento existe dentro del set

```
"Hola" in set  
# True
```

- Añadir otro dato (`.add`): lo agrega al inicio de la colección ya que no tiene índice.
`set.add(3)`
`# {3, "Hola", 23, (1, 2)}`
- Remover un elemento (`.remove`)
`set.remove("Hola")`
- Remover todo (`.clear`)



Diccionario

```
#class <'dict'>
```

Similar a una lista (`vector`), pero en vez de tener posiciones ordenadas, se utiliza **clave – valor**. Como un objeto JSON.

```
puntos_campeonato = {  
    "Real Madrid": 39,  
    "Barcelona": 29,  
    "Sevilla": 31  
}  
  
Clave           Valor
```

La **clave** puede ser cualquier tipo de dato, excepto una lista (`vector`).
El **valor** puede ser cualquier tipo de dato, incluso un diccionario.

Para acceder al valor, se utiliza el nombre del diccionario con la clave entre corchetes.

```
puntos_campeonato["Real Madrid"]  
=> 39
```

Trabajar con diccionarios dentro de diccionarios, los cuales no necesariamente deben tener las mismas claves, y se realiza de la siguiente manera:

```
alumnos_de_la_uba = {  
    313411234: {  
        "nombre": "Guillermo",  
        "promedio": 8  
    }  
    512312561: {  
        "nombre": "Daniel",  
        "egresado": True  
    }  
}
```

```
alumnos_de_la_uba[313411234]["promedio"]
>>> 8
```

- Obtener la cantidad de claves que existen en un diccionario (`len`)
 - Obtener todas las claves de un diccionario (`.keys()`)
 - Obtener todos los elementos del diccionario (`.items()`)
 - Eliminar todos los elementos del diccionario (`.clear()`)

```
len(alumnos_de_la_uba)  
>>> 2
```



Lista (vector) de diccionarios

Se pueden crear diccionarios ordenados mediante el uso de listas (vectores).

```
equipos = [
    {
        "nombre": "Real Madrid",
        "puntos": 39,
        "partidos jugados": 9
    },
    {
        "nombre": "Barcelona",
        "puntos": 29,
        "partidos jugados": 5
    },
    {
        "nombre": "Sevilla",
        "puntos": 31,
        "partidos jugados": 7
    }
]
```

```
equipos.sort(key = lambda x: -x["puntos"])
```

↓
indica que se ordene inversamente según la clave "puntos"

```
max(equipos, key = lambda x: x["puntos"])
```

↓ **también existe min**
se obtiene el diccionario cuyo valor de la clave "puntos" sea el mayor de toda la lista de diccionarios

```
def comparador(elemento):
    return elemento["puntos"], elemento["partidos jugados"]
max(equipos, key = comparador)
```

→ De esta manera, se puede buscar en el diccionario, la clave que contenga el mayor valor. En caso que exista un empate, procede a comparar la siguiente clave



Estructuras de control

Estructuras condicionales

Permite establecer una condición para que se ejecute un bloque de código, el cual debe tener una correcta indentación, ya que es lo que Python detecta para detectar dichos bloques de código en reemplazo de las llaves.

```
if numero > 0:  
    # Hacer algo...  
elif numero == 0:  
    # Hacer otra cosa...  
else:  
    # ¿Qué más querés?  
  
saludo = "Hola"  
match saludo:# similar a Switch  
    case "chanu":  
        print(1)  
    case "Hola":  
        print(4)  
    case (a, b):  
        print(a + b)
```



Si se desea utilizar más de una condición en una misma estructura, se pueden utilizar los operadores lógicos.

```
if numero1 >= 5 and numero2 <= 3:  
    # Las dos condiciones se cumplen...  
  
if numero1 >= 5 or numero2 <= 3:  
    # Una de las condiciones se cumplen...
```

Si se desea buscar el contrario de una condición, se puede utilizar el operador lógico de la negación.

```
if not numero1 >= 5:  
    # La condición no se cumple...
```

Operadores ternarios

Los operadores ternarios permiten la asignación de un valor con una condición.

```
descripcion:str  
if (edad >= 18): descripcion = "Mayor de edad"  
else: descripcion = "Menor de edad"
```

se resume en

```
descripcion = "Mayor de edad" if edad >= 18 else "Menor de edad"
```

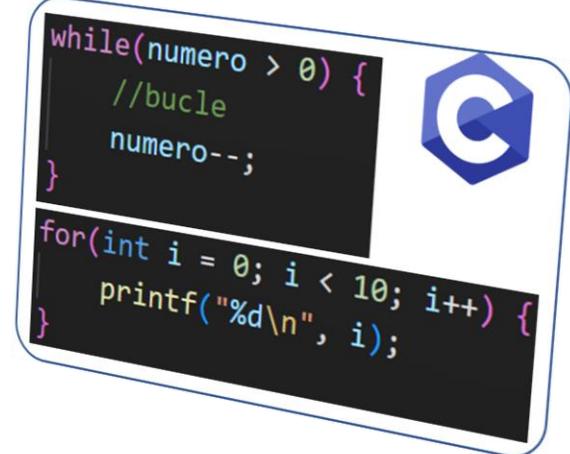


Estructuras iterativas

Permite realizar estructuras repetitivas mientras se cumpla una condición o un numero definido de veces.

```
while numero > 0:  
    # Bucle  
    numero -= 1
```

```
for i in range(10):  
    print(i)  
  
    ↴ salto de línea  
    automáticamente
```



```
while(numero > 0) {  
    //bucle  
    numero--;  
}  
for(int i = 0; i < 10; i++) {  
    printf("%d\n", i);  
}
```

En Python, se puede aclarar dentro del **range** los valores donde se desea iniciar y de a cuantos valores sumar/restar.

```
for i in range(10):# for(int i = 0; i < 10; i++)  
    print(i)  
  
for i int range(3, 10):# for(int i = 3; i < 10; i++)  
    print(i, end="")  
  
for i int range(3, 10, 2):# for(int i = 3; i < 10; i+=2)  
    print(i)
```

Sentencias de una estructura iterativa

- **break** Se utiliza para detener la ejecución de una iteración de forma total. Es decir, se continúa ejecutando el código que está a continuación de la estructura iterativa.
- **continue** Permite detener la iteración actual, es decir, comienza una nueva iteración siempre y cuando se siga cumpliendo la condición para la misma.



Manejo de errores

Las excepciones son el medio para tratar situaciones anómalas que pueden suceder, como por ejemplo:

- Intentar abrir un fichero que no existe para leerlo.
- Intentar dividir un numero sobre cero.

Si un código no se ejecuta de la forma prevista, se establece cómo debe responder el programa.

Para ello, se utilizan las palabras reservadas:

- **try** función que se intentará realizar.
- **except** lo que sucederá si surge un error.
- **finally** lo que sucederá independientemente de si hubo un error o no.

```
divisor = 10
dividendo = 0

try:
    resultado = divisor / dividendo
    # Una vez ocurrido el error, se ejecuta el bloque 'except'
    print("Este mensaje nunca será mostrado")

except Exception as error:
    print("Ocurrió un error: ", error)

finally:
    print("Este mensaje siempre será mostrado")
```



Ocurrió un error: division by zero
Este mensaje siempre será mostrado

Se puede generar un error personalizado con la palabra reservada **raise**.

```
raise Exception("Mensaje")
```

→ *Tipo de error*



Funciones y procedimientos

Las funciones y procedimientos permiten modularizar código.

```
# Procedimiento (sin return)
def saludar():
    print("Hola mundo!")

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

```
// Procedimiento
void saludar() {
    printf("Hola mundo!");
}

// Función
int suma(int a, int b) {
    return a + b;
```

Las variables que se describen dentro del paréntesis, se denominan parámetros, y son variables requeridas por el método (procedimiento o función) para realizar su tarea.

Estos parámetros pueden ser:

- **Valor:** una copia de una variable, es decir, no se modifica la variable original.
- **Referencia:** se modifica la variable original.

Las listas (**vectores**) se pasan por referencia, mientras que los tipos de datos más básicos se pasan por valor (por defecto).



Mostrar mensajes por pantalla al usuario

El método **print** se utiliza para mostrar un mensaje por pantalla al usuario, el cual puede formarse con distintas cadenas de texto.

```
print(texto1, texto2, texto3, ...)
```

Por defecto, agrega un espacio entre cada texto y, al final, agrega un salto de línea.

Sin embargo, este método tiene algunos parámetros que permiten modificar esto, y se utiliza de la siguiente manera:

- **end** Se utiliza para agregar cualquier cadena de caracteres al final de la salida e impresión en pantalla. Con este parámetro, se puede evitar el salto de línea automático.

```
print("Hola", end = "")
```

- **sep** Se utiliza para dar formato a las cadenas de caracteres que deben imprimirse en pantalla, modificando el **separador** entre las cadenas que se imprimirán.

```
print(str(numero1), str(numero2), sep=" - ")
```

Es importante recordar que la función **str** convierte un número en texto.

Introducción de datos por usuario

Mediante la función **input**, se puede pedir datos al usuario. Similar a “scanf” en C.

Siempre se obtiene como tipo de dato string.

Una vez obtenido el dato que ingresa el usuario, se imprime por pantalla automáticamente.

```
entrada = input("¿Cuál es tu nombre? ")
>>> ¿Cuál es tu nombre? #####
'#####5####'

entrada.replace("#", "")
>>> '5'

int(entrada.replace("#", ""))
>>> 5
```



Funciones lambda o anónimas

Las funciones anónimas o lambda se utilizan para resumir funciones sencillas.

```
def area_triangulo(base, altura):  
    return (base*altura)/2  
  
area_triangulo = lambda base,altura: (base*altura)/2  
  
area1 = area_triangulo(5,7) # area1 == 17.5
```



Dentro de estas funciones lambda no se pueden añadir estructuras de control ni iterativas. Sirven para utilizarse dentro de una porción de código y luego se “descarta”.



Funciones generadoras

Es similar a una función, con la diferencia que queda pausado con cada `return`, que en este caso se llamará `yield`, el cual genera un objeto iterable. Luego, al ser llamado de nuevo, se reanudará donde quedó pausado.

Un ejemplo de esto, es un generador de números pares.

Función “tradicional”

```
def numerosPares(limite):
    numero = 1
    lista_numeros = []
    while(numero < limite):
        lista_numeros.append(numero*2)
        numero += 1
    return lista_numeros
```

↓
Devuelve una lista (`vector`) completa con números pares.

Función generadora

```
def generarNumerosPares(limite):
    numero = 1
    while(numero < limite):
        yield numero*2
        numero += 1
```

↓
Devuelve un valor de la lista de números pares cada vez que se llame a la función (el proceso de calcular el siguiente valor también lo realiza durante la siguiente llamada)

Para hacer uso de una función generadora, es necesario llamarla desde una variable designada como “generador”.

```
obtenerPar = generarNumerosPares(10)
# <class 'generator'>
```

Una vez creado el generador, se puede obtener el siguiente valor generado mediante el método `next`.

```
par1 = next(obtenerPar) # 2
par2 = next(obtenerPar) # 4
par3 = next(obtenerPar) # 6
par4 = next(obtenerPar) # 8
```

~~#include "biblioteca.h"~~

Módulos (bibliotecas)

Se pueden importar bibliotecas o funciones específicas de una biblioteca.

```
# Usar funciones de una biblioteca
import biblioteca
biblioteca.saludar()
```

```
 biblioteca.py
1 def saludar():
2     print("Hola")
3
4 def saludar_a(nombre):
5     print("Hola " + nombre)
```

```
# Usar directamente una función de una biblioteca
from biblioteca import saludar_a
saludar_a("Guille")
```

Con un * se pueden usar todas las funciones existentes en la biblioteca

```
# Usar funciones de una biblioteca con un alias
import biblioteca as bib
bib.saludar_a("Guille")
```

Todo archivo de Python tiene una variable global predefinida llamada `__name__`. Dicha variable contiene el nombre del archivo Python, con excepción del archivo Python desde donde se está ejecutando el programa, el cual será nombrado `__main__`.

En una biblioteca, se pueden agregar instrucciones sobre qué funciones tienen, sus pre y post condiciones. Para ello, se le agrega un condicional que indique que su variable `__name__` sea el correspondiente al main.

```
 biblioteca.py
1 def saludar() -> None:
2     print("Hola")
3
4 def saludar_a(nombre: str) -> str:
5     print("Hola " + str(nombre))
6
7 if __name__ == "__main__":
8     print("Funciones:")
9     print("- saludar(): envía un saludo")
10    print("- saludar_a(nombre: str): saluda a alguien")
```



Algunas aclaraciones a modo de comentario (Python las ignora) que se pueden añadir a las funciones y procedimientos son indicar qué tipo de dato deben usarse y qué tipo de dato devuelve.

```
# Función con un solo tipo de variable
def saludar_a(nombre: str) -> str:
    print(str(nombre))
```

```
# Procedimiento (no devuelve nada)
def saludar() -> None:
    print("Hola mundo")
```

```
# Función con más de un tipo de variable
def suma(a: int|str, b: int|str) -> int|str:
    print(a + b)
```

```
# Función que devuelve más de un valor
def coordenada_aleatoria():
    return randint(0, 10), randint(0, 15)
#Ejemplo: (4, 12)
fila, columna = coordenada_aleatoria()
```

```
# Función con valor de parámetros por defecto
def armar_helado(gusto_de_arriba: str = "Chocolate", gusto_de_abajo: str = "Vainilla") -> str:
    print("Armaste un helado de " + gusto_de_arriba + " y " + gusto_de_abajo)

armar_helado()
>>> 'Armaste un helado de Chocolate y Vainilla'

armar_helado("Frutilla")
>>> 'Armaste un helado de Frutilla y Vainilla'

armar_helado(gusto_de_abajo = "Frutilla")
>>> 'Armaste un helado de Chocolate y Frutilla'

armar_helado("Frutilla", "Banana split")
>>> 'Armaste un helado de Frutilla y Banana split'
```



Clases (objeto)

Sirven para crear un tipo de dato propio y personalizado que, al instanciarlas en un código, se crea un objeto de ese tipo.

Para crear una clase, se debe proceder a crear un archivo Python nuevo (como un módulo). Dentro de este nuevo archivo, se comienza escribiendo la palabra reservada **class** seguido del nombre de la clase que queremos crear.

```
clase.py > ...
1   class ClassName:
2       pass
```

Nota: la palabra reservada **pass** sirve para indicar al programa Python que no se requiere acción alguna.

Para poder utilizar esta clase desde el código de otro archivo Python, se debe hacer de la siguiente manera:

```
from clase import ClassName
objeto = ClassName()
```

Instancia de clase

Atributos de una clase

Propiedades pertenecientes a una clase. Estos se declaran como variables luego de declarar la clase. Dichas variables pueden establecerse en la propia clase y modificarse tras instanciarla.

```
clase.py > ...
1   class Escuela:
2       nombre = ''
3       puntaje = 8
```

```
from clase import Escuela
objeto = Escuela()
objeto.nombre = "Dr. Ernesto Longobardi"
# nombre = 'Dr. Ernesto Longobardi'
# puntaje = 8
```

Métodos de una clase

Funciones definidas con la palabra reservada **def** y cuentan con el mismo formato que las funciones. Sin embargo, cuentan con una diferencia y es que deben contener obligatoriamente un argumento llamado **self**, que se refiere al objeto del método que está siendo llamado (al llamar al método, no es necesario pasar dicho argumento).

```
clase.py > ...
1   class Escuela:
2       nombre = ''
3       puntaje = 8
4
5       def imprimir_nombre(self):
6           print(self.nombre)
```

```
from clase import Escuela
objeto = Escuela()
objeto.nombre = "Dr. Ernesto Longobardi"
objeto.imprimir_nombre()
# 'Dr. Ernesto Longobardi'
```

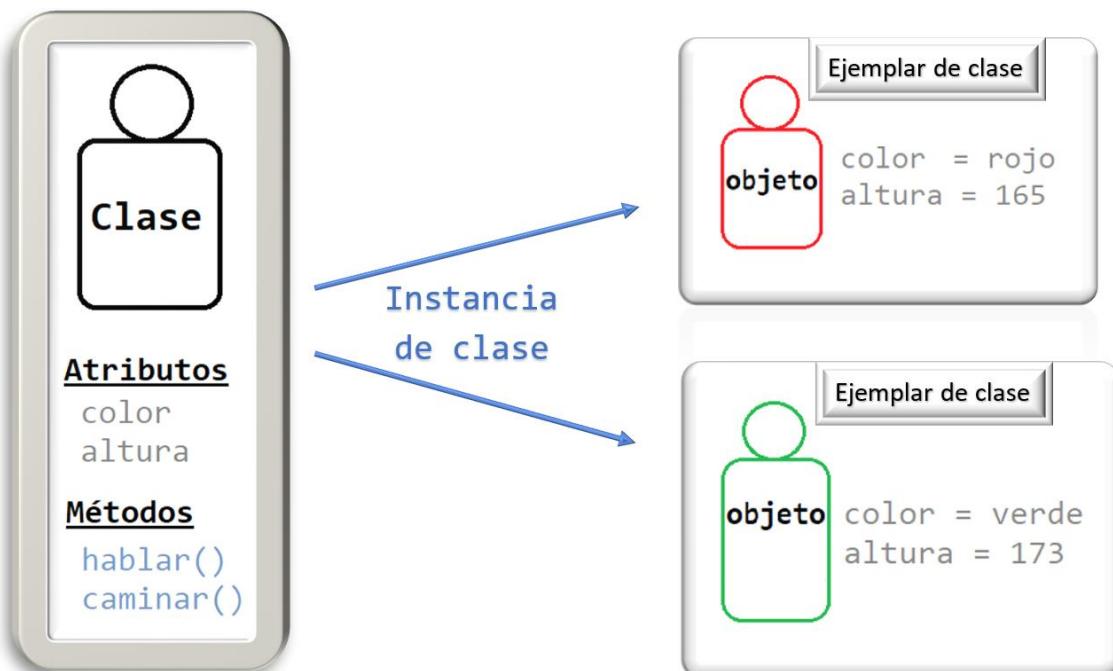


Inicialización

Las variables pueden inicializarse directamente en los renglones siguientes de la instancia de la clase en nuestro código. Sin embargo, existe una función específica en Python (llamado **constructor**) para hacerlo directamente desde la llamada de la clase, denominada `_init_`

Esta función debe recibir obligatoriamente al menos el argumento `self`, y el/los valores que recibirán las variables a inicializar desde la instancia de la clase.

```
clase.py > ...
1  class Escuela:
2      def __init__(self, n, p):
3          self.nombre = n
4          self.puntaje = p
from clase import Escuela
objeto = Escuela("Dr. Ernesto Longobardi", 10)
# nombre = 'Dr. Ernesto Longobardi'
# puntaje = 10
```





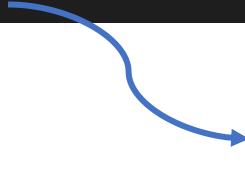
Método `str` de una clase

El método `__str__` en una clase permite enviar un String de la clase cuando es instanciada como tal.

```
class Persona:  
    # Constructor  
    def __init__(self, nombre, edad):  
        # Atributos:  
        self.nombre = nombre  
        self.edad = edad  
  
    def __str__(self):  
        return "{} tiene {} años".format(self.nombre, self.edad)
```

De esta forma, si se llama al ejemplar de clase, se devolverá el string definido.

```
guille = Persona("Guillermo", 27)  
  
print(guille)
```



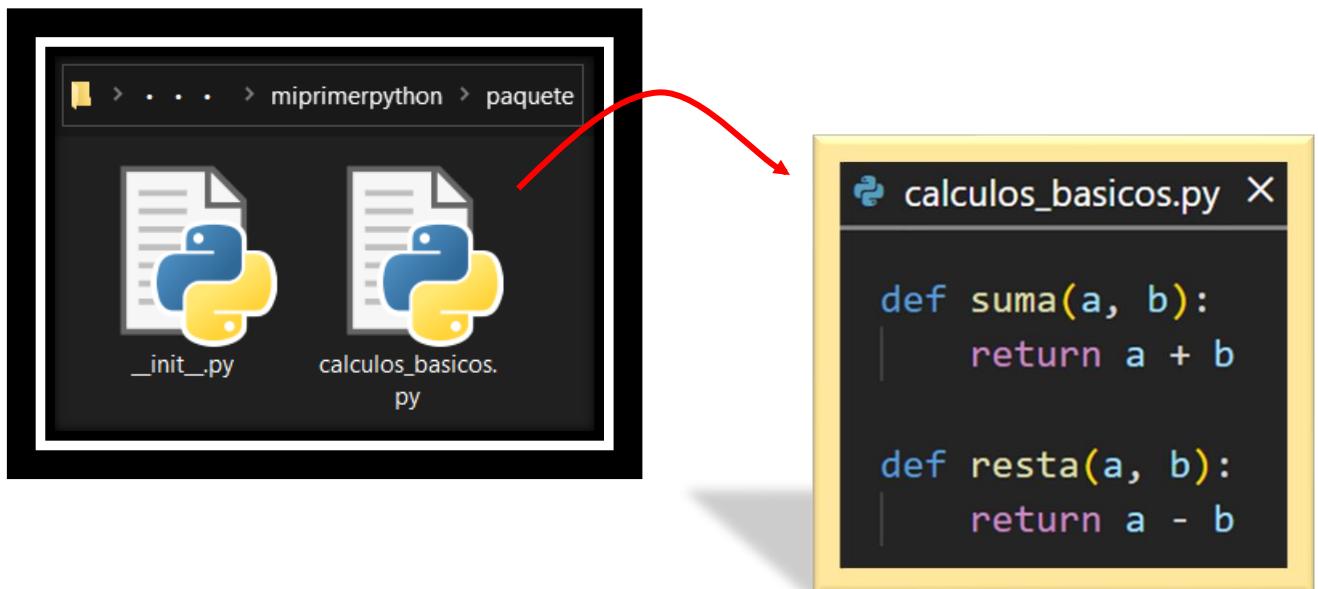
```
howto geek@ubuntu: ~  
Guillermo tiene 27 años
```



Paquetes

Directorio donde se almacenan módulos relacionados entre sí.

Para crear un paquete, es necesario crear una carpeta donde se alojarán todos los módulos requeridos y un archivo llamado `__init__.py`



Una vez creada la carpeta paquete, se la puede importar en nuestro código.

```
from paquete.calculos_basicos import *

suma(1,2)
>>> 3
```

Incluso se pueden crear paquetes dentro de paquetes, y se podrán utilizar siempre y cuando se tenga la carpeta `__init__.py` dentro.



Paquete distribuible

Un paquete se puede instalar dentro del propio Python del Sistema operativo del computador para poder utilizarlo sin necesidad de tenerlo en el mismo directorio de donde se desea utilizar. Para ello, es necesario crear un archivo llamado **setup.py** el cual contendrá la configuración del paquete.

The screenshot shows a code editor with an open file named `setup.py`. The file tree on the left shows a directory structure: `Python\calculos\redondeo_potencia` containing `__init__.py` and `redondeaPotencia.py`, and `calculos_generales.py`. The `setup.py` script itself defines a package named `paquetecalculos` with version `1.0`, description `"Paquete de redondeo y potencia"`, author `Juan`, and author_email `""`. It also specifies packages to be included: `["calculos", "calculos.redondeo_potencia"]`.

```
OPEN FILES
  • setup.py

FOLDERS
  ▾ Python
    ▾ calculos
      ▾ redondeo_potencia
        /* __init__.py
        /* redondeaPotencia.py
        /* __init__.py
        /* calculos_generales.py
      /* setup.py

setup.py
1 from setuptools import setup
2
3 setup(
4
5     name="paquetecalculos",
6     version="1.0",
7     description="Paquete de redondeo y potencia",
8     author="Juan",
9     author_email="",
10    url="",
11    packages=["calculos", "calculos.redondeo_potencia"]
12
13 )
```

A continuación, se debe utilizar el **Símbolo del sistema** y dirigirse a la carpeta donde se encuentra el archivo `setup.py` y el paquete para ejecutar el comando:
python setup.py sdist

The screenshot shows a Windows Command Prompt window titled "Administrador: Símbolo del sistema". The user navigates to the directory `C:\Users\Juan\Desktop\Python` and runs the command `python setup.py sdist`. The output shows the creation of a `dist` folder.

```
Administrator: Símbolo del sistema
Microsoft Windows [Versión 10.0.15063]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

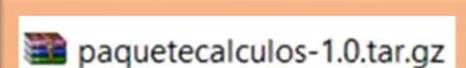
C:\WINDOWS\system32>cd..
C:\Windows>cd..
C:\>cd Users
C:\Users>cd Juanm
El sistema no puede encontrar la ruta especificada.
C:\Users>cd Juan
C:\Users\Juan>cd Desktop
C:\Users\Juan\Desktop>cd Python
C:\Users\Juan\Desktop\Python> python setup.py sdist
```

Ejecutado el comando, se creará una carpeta **dist** en el directorio.

Dicha carpeta, contiene el paquete distribuible creado.



Para instalar el paquete creado, es necesario utilizar nuevamente el Símbolo del sistema en el directorio donde se encuentra el paquete en formato tar.gz. con el comando: **pip3 install nombre-version.tar.gz**



Si se desea desinstalar el paquete, solo hay que usar el comando: **pip3 uninstall nombre**

The screenshot shows a Windows Command Prompt window titled "Administrador: Símbolo del sistema". The user navigates to the `dist` folder and runs the command `pip3 install paquetecalculos-1.0.tar.gz`. The output shows the successful installation of the package.

```
Administrator: Símbolo del sistema
C:\Users\Juan\Desktop\Python>cd dist
C:\Users\Juan\Desktop\Python\dist>pip3 install paquetecalculos-1.0.tar.gz
Processing c:\users\juan\desktop\python\dist\paquetecalculos-1.0.tar.gz
Installing collected packages: paquetecalculos ... done
Successfully installed paquetecalculos-1.0
C:\Users\Juan\Desktop\Python\dist> Pip3 uninstall paquetecalculos
```



Archivos

Python permite trabajar con archivos externos.

Se utiliza el método `open`, donde se incluye el nombre/ruta del archivo con el que se desea trabajar, seguido de un parámetro que indique si se trabajará en modo lectura (“`r`”), escritura borrando lo ya existente (“`w`”) o en modo append para agregar texto sin borrar lo anterior (“`a`”).

```
archivo = open("archivo.txt", "r")
```

Por defecto, se incluye “`r`” como parámetro por lo que no es necesario especificarlo.

```
# <class '_io.TextIOWrapper'>
```

Si el archivo no se logra abrir (porque no existe), el programa lanzará un error.

Es importante saber que, cuando se lee un archivo, existe algo similar a un puntero. Esto significa que una vez que se lee todo el archivo, no se puede volver a leer a menos que llevemos el “puntero” al inicio, eso se puede hacer mediante el método `.seek(0)`.

```
archivo.seek(0)
```

Además, es importante que, al terminar de trabajar con un archivo, éste debe cerrarse con el comando método `.close()`.

```
archivo.close()
```

Una vez abierto el archivo, se pueden realizar diversas funciones:

- Leer todo (`.read()`): se obtiene un **string** con todo el texto del archivo.
`contenido = archivo.read()`
- Leer todo separando en líneas (`.readlines()`): se obtiene una **lista** (`vector`) de strings con cada línea del archivo por separado. Cada string incluye el salto de línea `/n`.
`for linea in archivo.readlines():
 print(linea) # Imprime cada línea`
- Lee la siguiente línea según dónde esté ubicado el “cursor” (`.readLine()`): se obtiene un solo **string**.
`for linea in archivo.readline():
 print(linea) # Imprime cada letra de una línea`

Un ejemplo de lectura línea por línea de un archivo es el siguiente:

```
try:
    with open(path, "r") as file:
        for line in file:
            print(line)
except Exception as err: return err
```

with permite utilizar recursos de forma segura, cerrando el archivo sin necesidad de hacerlo explícitamente con `.close()`



```
archivo = open("archivo.txt", "a")
```

Para trabajar en modo escritura, se debe abrir el archivo en modo escritura. Si el archivo no existe, se creará uno nuevo.

No olvidar que, una vez que se termine de trabajar, se debe cerrar.

```
archivo.flush()  
archivo.close()
```

En este modo, se debe utilizar el método `.write(text)` para escribir el archivo. Dicho método devuelve el número de caracteres escritos.

```
archivo.write("Texto a escribir") # 16
```

Es importante destacar que se efectuarán los cambios en el archivo en el momento en que se llame al método `.close()`. Mientras tanto, el programa trabajará con un “caché” o “buffer”.

Si se desea efectuar los cambios antes de cerrar el archivo, se puede utilizar el método `.flush()`

```
archivo.flush()
```



Archivos CSV

Archivo que se utiliza para guardar datos en gran cantidad, separados por un delimitador.

Para trabajar con este tipo de archivos en Python, es necesario importar el módulo de CSV.

```
import csv
```

Nombre	Obra social	Categoría	Enfermedad	Peso	Edad
Guillermo	IOMA	Platino	Depresion	27	81
Daniela	OSDE	ASC10	Esquizofrenia	25	65
Lucas	No	No	Gastritis	18	76

Este módulo (**biblioteca**) contiene un método que permite leer un archivo csv, especificando su delimitador (lo que separa un dato de otro, en este caso un punto y coma).

```
archivo = open("pacientes.csv") # Modo lectura por defecto  
csv_reader = csv.reader(archivo, delimiter = ";")
```

Una vez inicializado el *reader* de CSV de Python, se puede utilizar el método **next** para leer una línea (fila) del CSV, creando una **lista** (**vector**) con el contenido de cada columna de la fila que esté leyendo. Es importante recordar que cada vez que se ejecute el método, el “cursor” se moverá a la siguiente línea.

```
next(csv_reader) # ['Guillermo', 'IOMA', 'Platino', 'Depresion', 27, 81]
```

Para leer todas las líneas del archivo CSV, se puede utilizar un **for** de la siguiente manera.

```
# Se obtiene una lista (vector) de la fila  
# en que se encuentre en cada iteración  
for linea in csv_reader:  
    print(linea) # linea es un vector  
  
    # ['Guillermo', 'IOMA', 'Platino', 'Depresion', 27, 81]  
    # ['Daniela', 'OSDE', 'ASC10', 'Esquizofrenia', 25, 65]  
    # ['Lucas', 'No', 'No', 'Gastritis', 18, 76]
```

Es muy importante no olvidar cerrar el archivo una vez que se finalice todo lo que se desea hacer.

```
archivo.close()
```



Archivos binarios

Archivo que se utiliza para guardar datos en formato binario para una mejor adaptación en caso de quererse compartir con otros sistemas operativos.

Para trabajar con este tipo de archivos, es necesario importar el módulo Pickle.

Para comenzar, se debe crear un archivo en modo escritura binaria. `import pickle`
Esto se realiza de la siguiente forma:

```
fichero_binario = open("archivo_binario", "wb")
```

Para volcar los datos dentro del archivo en formato binario, se utiliza el método `dump` del módulo importado.

```
lista_nombres = ["Guillermo", "Daniel", "Roberto"]
```

Una vez finalizado, se cierra el archivo.

```
fichero_binario.close()
```

Para leer un archivo en formato binario, se puede utilizar el método `load` del módulo tras haber abierto nuevamente el archivo en formato binario en modo lectura binaria.

```
fichero_binario = open("archivo_binario", "rb")
dato = pickle.load(fichero_binario)
fichero_binario.close()
```



Programación Orientada a Objetos

La programación orientada a objetos se enfoca en la creación de objetos (como clases), los cuales tienen sus propiedades/atributos (ej: color, peso, altura) y comportamientos/métodos (lo que es capaz de hacer, por ej: caminar, acelerar, etc).

Este paradigma de programación permite crear programas divididos en módulos o clases. Además que permite el control de errores en un programa y el uso de la herencia.

Algunos conceptos de la programación orientada a objetos son:

- Objeto
 - Clases
 - Instancia de clase
 - Modularización
 - Encapsulamiento
 - Herencia
 - Polimorfismo

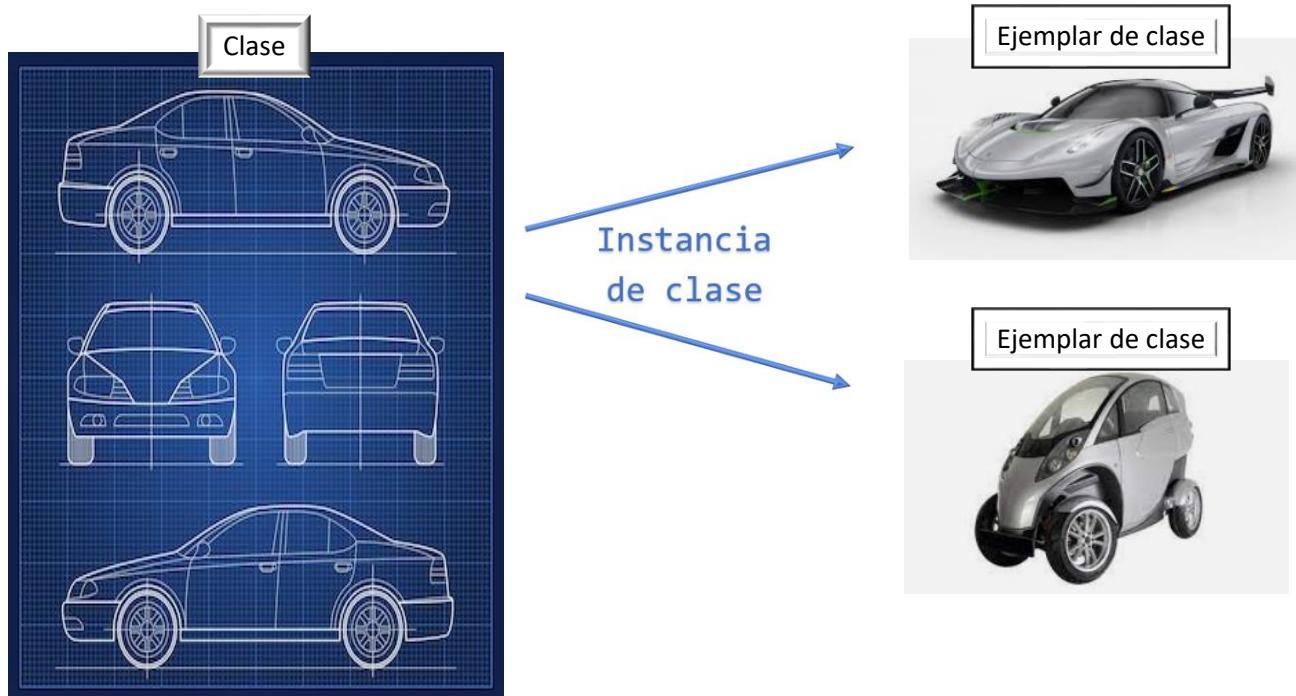
modelo que redacta las características comunes de un tipo de objeto.

un ejemplar perteneciente a una clase (un objeto).

creación de múltiples clases con las diversas funciones del programa.

permite que el funcionamiento interno de la clase no sea accesible desde fuera.

con **métodos de acceso** se pueden cambiar determinadas características que se necesiten para el correcto funcionamiento





Encapsulamiento

Permite que ciertas funcionalidades internas de una clase no sean accesibles desde fuera de la misma. Sólo se podrán acceder a las funciones o características que sean necesarias para el correcto funcionamiento de la clase mediante métodos de acceso.

Para hacer que un atributo o método de una clase sea privado (que no se pueda modificar desde fuera de la clase), se debe agregar **dos guiones bajos** al inicio de su nombre.

```
class Coche:  
    # Constructor  
    def __init__(self, modelo:str) -> None:  
        self.modelo = modelo  
        self.largoChasis = 250  
        self.anchochasis = 120  
        self.enMarcha = False  
        self.__ruedas = 4  
    # Métodos  
    def quitarRueda(self):  
        self.__ruedas -= 1  
  
    def estado(self):  
        print(f"El {self.modelo} de dimensiones {self.largoChasis}x{self.anchochasis}",  
              f" y de {self.__ruedas} ruedas está ", sep="", end="")  
        if(self.enMarcha):  
            print("en marcha.")  
        else:  
            print("apagado.")  
  
    → Estado inicial  
    → doble guión bajo establece el acceso privado
```

```
fiat600 = Coche("fiat600")# Ejemplar de clase  
fiat600.quitarRueda() → internamente quitó una rueda  
fiat600.estado()  
# 'El fiat600 de dimensiones 250x120 y de 3 ruedas está apagado'
```

```
renault12 = Coche("renault12")# Ejemplar de clase  
renault12.enMarcha = True → atributo no privado  
                                         → sí se modificó  
renault12.__ruedas = 8 → el atributo privado  
                                         → no se pudo modificar  
renault12.estado()  
# 'El renault12 de dimensiones 250x120 y de 4 ruedas está en marcha'
```



Herencia

Permite la reutilización de código de clases con características similares o comunes. Se trata de crear una clase denominada **superclase** o **clase padre** que contenga todas las características en común que contengan diversas **subclases** o **clases hijas**, las cuales tendrán comportamientos (métodos) y propiedades (atributos) únicas.

```
Super clase / Clase padre
class Vehiculos():

    # Constructor (función predefinida privada)
    def __init__(self, marca:str, modelo:str) -> None:
        self.marca = marca
        self.modelo = modelo
        self.enMarcha = False
        self.acelera = False
        self.frena = False

    # Métodos / Comportamientos
    def arrancar(self) -> None:
        self.enMarcha = True

    def acelerar(self) -> None:
        self.acelera = True

    def frenar(self) -> None:
        self.frena = True
```

Para hacer uso de la herencia, primero se debe crear una clase padre con su constructor (el cual también será heredado).

Luego, se creará la clase hija, la cual heredará todos los atributos y métodos de la clase padre. Para aplicar la herencia, se debe pasar como parámetro el nombre de la clase padre.

Una vez realizada la herencia, se puede trabajar con los atributos y métodos de la clase padre desde una instancia de clases de la clase hija, la cual debe tener como argumentos los de la clase padre.

```
# Sub-clase / Clase hija
class Moto(Vehiculos):

    haciendoWillie = False

    def hacerWillie(self):
        self.haciendoWillie = True
```

```
honda = Moto("Honda", "1995")

honda.acelerar() # honda.acelera == True
honda.hacerWillie() # honda.haciendoWillie == True
```

Si dentro de la clase hija hay un método / función con igual nombre que algún método / función de la clase padre, se utilizará el de la clase hija ya que es la clase que será instanciada finalmente.

```
# Sub-clase / Clase hija
class Tractor(Vehiculos):
    marcha = 0

    def arrancar(self):
        self.marcha = 2
```

```
maquina = Tractor("Volkswagen", "2010")

maquina.arrancar() # maquina.marcha == 2
# maquina.enMarcha no fue alterado
```



En Python también existe la **herencia múltiple**, es decir, una clase hija puede tener múltiples clases padre. Para ello, se deben crear las clases padre y, a la clase hija, agregarle como parámetro el nombre de todas las clases padres que vaya a tener.

```
Super clase / Clase padre
class Vehiculos():

    # Constructor (función predefinida privada)
    def __init__(self, marca:str, modelo:str) -> None:
        self.marca = marca
        self.modelo = modelo
        self.enMarcha = False
        self.acelera = False
        self.frena = False

    # Métodos / Comportamientos
    def arrancar(self) -> None:
        self.enMarcha = True

    def acelerar(self) -> None:
        self.acelera = True

    def frenar(self) -> None:
        self.frena = True
```

```
# Super clase / Clase padre
class VehiculoElectrico():

    # Constructor (función predefinida privada)
    def __init__(self) -> None:
        self.cargaElectrica = 100
        self.cargando = False

    def cargarEnergia(self):
        self.cargando = True
```

Es importante destacar que, cuando se crea la clase hija que contendrá como argumentos los nombres de las clases padres, se dará prioridad al primero que esté escrito.

Esto afecta principalmente a los métodos que las clases padres tengan en común. Es decir, si ambas clases padre tienen un método con un mismo nombre, en la clase hija se podrá utilizar el de la clase padre que esté escrito más a la izquierda en los argumentos de la clase hija.

```
# Sub-clase / Clase hija
class BicicletaElectrica(Vehiculos, VehiculoElectrico):
    pass
```

Al realizar la instancia de clases, se pasa por parámetro los de la clase padre que esté más a la izquierda en los argumentos de la clase hija.

En este caso, se hereda la función `__init__` de la clase padre `Vehiculos` por ser la primera en ser llamada

```
bicicleta = BicicletaElectrica("Toyota", "2020")
print(bicicleta.cargando) # ERROR -> No existe el atributo
```



Palabra reservada **super**

Esta instrucción permite ejecutar una función de la clase padre desde una clase hija.

```
# Super-clase / Clase padre
```

```
class Persona():

    # Constructor
    def __init__(self, nombre:str, edad:int, residencia:str):
        self.nombre = nombre
        self.edad = edad
        self.residencia = residencia

    # Métodos / Comportamientos
    def descripcion(self):
```

```
        print("Nombre: ", self.nombre,
              "Edad: ", self.edad,
              "Residencia: ", self.residencia)
```

```
Sub-clase / Clase hija
```

```
class Empleado(Persona):

    # Constructor
    def __init__(self, salario:int, antiguedad:int,
                 nombre:str, edad:int, residencia:str):
        super().__init__(nombre, edad, residencia)

        self.salario = salario
        self.antiguedad = antiguedad

    def descripcion(self):
        super().descripcion()

        print("Salario: ", self.salario,
              "Antiguedad: ", self.antiguedad)
```

```
# Instancia de clase
antonio = Empleado(1500, 7, "Antonio", 55, "Argentina")

antonio.descripcion()
>>> Nombre: Antonio Edad: 55 Residencia: Argentina
>>> Salario: 1500 Antiguedad: 7
```

Se puede saber si un objeto (ejemplar de clase) pertenece a una clase específica mediante la función **isinstance(objeto, clase)**

```
# Instancia de clase
antonio = Empleado(1500, 7, "Antonio", 55, "Argentina")
isinstance(antonio, Empleado) # True
isinstance(antonio, Persona) # True
```



Polimorfismo

Permite que una misma función pueda utilizar un método de una determinada clase según el ejemplar de clase que reciba como parámetro.

```
class Coche():
    def cantidadRuedas(self):
        return 4
```

```
class Moto():
    def cantidadRuedas(self):
        return 2
```

```
class Camion():
    def cantidadRuedas(self):
        return 6
```

```
# Función polimorfa
def ruedasDelVehiculo(vehiculoConRuedas):
    return vehiculoConRuedas.cantidadRuedas()
```

```
mi_vehiculo = Camion()
ruedasDelVehiculo(mi_vehiculo) # 6
```

En este caso, la función polimorfa recibe como parámetro un objeto de tipo Camión. Por lo tanto, utiliza la función de la clase Camión y devuelve la cantidad de ruedas = 6.

Métodos abstractos

Se utilizan para definir métodos que no están implementados y que, obligatoriamente, deben implementarse en clases hijas.

Para esto, deberemos importar los siguientes módulos y hacer que nuestra class abstracta (por tener métodos abstractos) reciba herencia de ABC.

Ahora, usando `@abstractmethod` podemos definir nuestros métodos abstractos.

```
from abc import ABC, abstractmethod
class Empresa(ABC):

    @abstractmethod
    def empleados(self):
        pass
```



Función Filter

Forma parte de un grupo de funciones llamado *funciones de orden superior*. Utiliza un paradigma de programación denominado *programación funcional*.

La función Filter verifica que los elementos de una secuencia cumplen una condición, devolviendo un iterador con los elementos que cumplen dicha condición.

`filter(function , collection_to_filter)`

La función devuelve True o False

El iterador que devuelve la función tiene un comportamiento similar al de una función generadora. Es decir, se podría usar el método **next**.

Es de especial utilidad el uso de las funciones lambda, por ejemplo, para la selección de números pares.

```
def esNumeroPar(numero):
    if numero % 2 == 0:
        return True }
```

lambda numero: numero % 2 == 0

Con la función que será usada como filtro establecida, se la puede en la función Filter de las siguientes maneras:

```
iterador = filter(esNumeroPar, listaNumerosAleatorios)  sino, con Lambda:
iterador = filter(lambda numero: numero % 2 == 0, listaNumerosAleatorios)
```

Un ejemplo de uso es el siguiente:

```
numerosAleatorios = [5,7,22,53,66,97,104]

iterador = filter(lambda numero: numero % 2 == 0, numerosAleatorios)

for numeroPar in iterador:
    # Se ejecuta una iteración con cada valor filtrado (22, 66, 104)
```

También se puede obtener una lista de todos los elementos que hayan pasado el filtro, de la siguiente manera:

`list(iterador)`



Las funciones Filter suelen utilizarse para filtrar objetos, por ello se la puede utilizar para filtrar ejemplares de clase.

```
class Empleado:  
    # Constructor  
    def __init__(self, nombre, cargo, sueldo):  
        # Atributos:  
        self.nombre = nombre  
        self.cargo = cargo  
        self.sueldo = sueldo  
  
    # Métodos:  
    def __str__(self):  
        return "{} ({}) gana ${} al mes".format(self.nombre, self.cargo, self.sueldo)
```

```
listaEmpleados = [  
    Empleado("Guillermo", "Enfermero", 80000),  
    Empleado("Marcela", "Enfermera", 120000),  
    Empleado("Francisco", "Tec. Laboratorio", 60000),  
    Empleado("Sol", "Aux. Farmacia", 45000),  
    Empleado("Romina", "Aux. Limpieza", 27000)  
]  
iterador_sueldosAltos = filter(lambda empleado: empleado.sueldo > 50000, listaEmpleados)  
  
for empleadoAltoSalario in iterador_sueldosAltos:  
    print(empleadoAltoSalario)
```



```
howto geek@ubuntu: ~  
Guillermo (Enfermero) gana $80000 al mes  
Marcela (Enfermera) gana $120000 al mes  
Francisco (Tec. Laboratorio) gana $60000 al mes
```



Función Map

Forma parte de un grupo de funciones llamado *funciones de orden superior*. Utiliza un paradigma de programación denominado *programación funcional*.

La función map aplica una función iterable a cada elemento de una lista iterable (listas, tuplas, etc.) devolviendo una lista con los resultados.

```
map(function, collection_to_filter)
```

Siguiendo con el ejemplo anterior (empleados):

Se puede mapear una lista de empleados.

```
listaEmpleadosBruto = [
    Empleado("Guillermo", "Enfermero", 80000),
    Empleado("Marcela", "Enfermera", 120000),
    Empleado("Francisco", "Tec. Laboratorio", 60000),
    Empleado("Sol", "Aux. Farmacia", 45000),
    Empleado("Romina", "Aux. Limpieza", 27000)
]
```

```
class Empleado:
    # Constructor
    def __init__(self, nombre, cargo, sueldo):
        # Atributos:
        self.nombre = nombre
        self.cargo = cargo
        self.sueldo = sueldo

    # Métodos:
    def __str__(self):
        return "{} ({}) gana ${} al mes".format(self.nombre, self.cargo, self.sueldo)
```

Primero, se establece la función que se utilizará, por ejemplo, cobrar impuestos al sueldo:

```
PORCENTAJE_IMPUESTOS = 21
PORCENTAJE_IMPUESTOS_BAJOS = 17
SUELDO_ALTO = 10000

def restarImpuestosAlSueldo(empleado):
    if empleado.sueldo >= SUELDO_ALTO:
        empleado.sueldo -= int(empleado.sueldo*PORCENTAJE_IMPUESTOS/100)
    else:
        empleado.sueldo -= int(empleado.sueldo*PORCENTAJE_IMPUESTOS_BAJOS/100)
    return empleado
```

Luego, se puede aplicar en una función map de la siguiente manera:

```
listaEmpleadosNeto = map(restarImpuestosAlSueldo, listaEmpleadosBruto)
```

Se puede recorrer la lista de la misma forma que se hacía en Filter:

```
for empleado in listaEmpleadosNeto:
    print(empleado)
```



```
Guillermo (Enfermero) gana $63200 al mes
Marcela (Enfermera) gana $94800 al mes
Francisco (Tec. Laboratorio) gana $47400 al mes
Sol (Aux. Farmacia) gana $35550 al mes
Romina (Aux. Limpieza) gana $21330 al mes
```



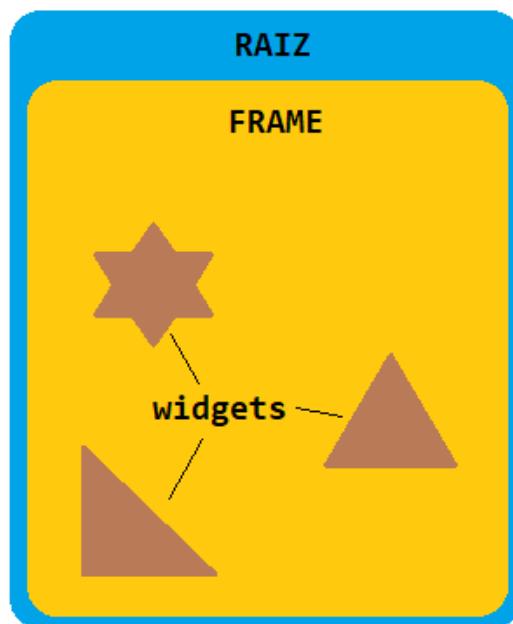
Interfaces gráficas

Una GUI es un intermediario entre el programa y el usuario formada por un conjunto de gráficos como ventanas, botones, menús y casillas de verificación.

Una biblioteca / módulo útil para la creación de interfaces gráficas en Python es Tkinter, la cual es un “puente” con TCL/TK.

Una interfaz gráfica en Python (TK) está compuesta por:

1. **Raíz (ventana)** base de toda la interfaz gráfica
2. **Frame** organizador de elementos
3. **Widgets** elementos de la interfaz (botones, menús, etc)



Si se desea que nuestro archivo Python con interfaz gráfica se abra sin la consola por detrás, se debe cambiar la extensión del archivo por [.pyw](#)



Raiz



Para que una ventana se mantenga en pantalla, debe estar en un estado similar a un bucle infinito para poder estar a “la escucha” de la interacción del usuario. Esta función nos la brinda Tkinter con **mainloop** y debe ubicarse siempre al final.

Además, el tamaño de la raíz siempre se adaptará al tamaño de su contenido.

```
from tkinter import *

raiz = Tk()

...
raiz.mainloop()
```

Algunos métodos que se pueden utilizar sobre la raíz son:

```
from tkinter import *
raiz=Tk()
raiz.title("Ventana de pruebas")
raiz.resizable(True,False)
raiz.iconbitmap("gato.ico")
raiz.geometry("650x350")
raiz.config(bg="blue")
raiz.mainloop()
```

- **.title("nombre")**
- **.resizable(width:bool , height:bool)**
- **.iconbitmap("path.ico")**
- **.geometry("WxH")**
- **.config**

establece un nombre a la ventana
determina si se puede alterar el tamaño
establece un ícono (coincide con empaquetado)
establece el tamaño de la ventana
permite establecer varias configuraciones

bg → color de fondo

Para cerrar la interfaz gráfica de forma programática, se puede usar el método:

```
raiz.destroy()
```



Frame()

Frame

Para ubicar el frame dentro de la raíz, es necesario empaquetarla dentro de dicha raíz con el método `.pack()`. Además, se puede establecer su tamaño con `.config`

Por defecto, un widget (el frame es un tipo de widget) se ancla en la parte superior y se centra horizontalmente en la ventana raíz.

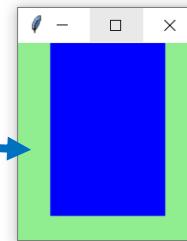
```
from tkinter import *

raiz = Tk()
raiz.config(bg="lightgreen")

frame = Frame()
frame.pack()

frame.config(bg="blue")
frame.config(width="100", height="150")

raiz.mainloop()
```



Dentro del método `.pack`, se puede establecer el comportamiento por defecto del frame.

- | | | | |
|-----------------------|--|---|---|
| ➤ <code>side</code> | <code>"left, right, top o bottom"</code> | establece hacia cuál borde se va a anclar | } |
| ➤ <code>anchor</code> | <code>"w", "n", "e", "s"</code> | establece hacia qué punto cardinal se anclará | |
| ➤ <code>fill</code> | <code>"x", "y", "both", "none"</code> | establece que el frame se rellene en la raíz | |
| ➤ <code>expand</code> | <code>"True", "False"</code> | establece que el frame pueda expandirse | |

Además, se puede aplicar las configuraciones al frame con el método `.config`

- | | | |
|-----------------------|------------------------------|--|
| ➤ <code>bg</code> | <code>"color"</code> | establece el color de fondo del frame |
| ➤ <code>width</code> | <code>"tamaño"</code> | establece el ancho del frame |
| ➤ <code>height</code> | <code>"tamaño"</code> | establece el alto del frame |
| ➤ <code>bd</code> | <code>"tamaño borde"</code> | establece el tamaño del borde del frame |
| ➤ <code>relief</code> | <code>"tipo de borde"</code> | establece un tipo de borde al frame |
| ➤ <code>cursor</code> | <code>"tipo puntero"</code> | establece la forma del cursor al estar en el frame |

Es posible establecer un tamaño mínimo a la ventana, ajustado al contenido del frame. Para esto, se debe escribir el siguiente código:

```
raiz = Tk()

frame = Frame(raiz)

...
raiz.update()
raiz.minsize(frame.winfo_width(), frame.winfo_height())

raiz.mainloop()
```



Widgets

Por defecto, un widget se ancla en la parte superior y se centra horizontalmente en su contenedor (frame o widget donde estará contenido).

- **Label:** texto no interactuable.

Label(contenedor, opciones)

Las opciones posibles de configuración son:

➤ text	texto que se muestra en el label
➤ anchor	posición del texto (centrado por defecto)
➤ width	ancho del label (en caracteres)
➤ height	alto del label (en caracteres)
➤ bg	color del fondo
➤ bd	grosor del borde (2 px por defecto)
➤ font	fuente del texto y tamaño de letra font=("Comic Sans MS", 18)
➤ fg	color del texto
➤ justify	justificación del texto
➤ image	muestra una imagen en vez de texto (png o gif)
➤ bitmap	muestra un grafico (mapa de bits)

El método **.place** permite ubicar el widget en una determinada posición respecto de su contenedor, ignorando el anclaje por defecto.

pyLabel.place(x = 100, y = 200)

Distancia en px desde
el borde izquierdo

Distancia en px desde
el borde superior

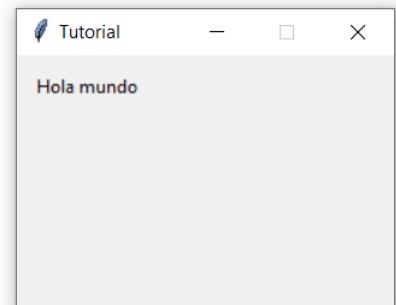
```
from tkinter import *

raiz = Tk()
raiz.title("Tutorial")
raiz.resizable(False, False)

frame = Frame(width=250, height=170)
frame.pack()

pyLabel = Label(frame, text="Hola mundo")
pyLabel.place(x = 10, y = 10)

raiz.mainloop()
```



```
photo = PhotoImage(file="foto.png")
Label(frame, image=photo).place(x = 10, y = 10)
```



- **Text:** texto multilínea

Text(contenedor)

Crea un cuadro de texto multilínea de gran tamaño.

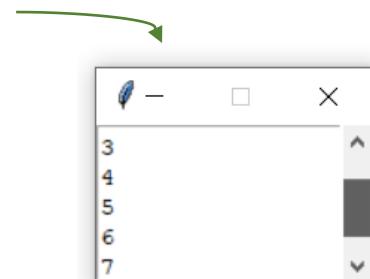
Recordar que se puede arreglar el tamaño con **width** y **height**.

Además, puede ser útil agregar una **barra de scroll**:

```
pyTextArea = Text(frame, width=16, height=5)
pyTextArea.grid(row=0, column=0) # Posiciona el widget en una tabla

scrollbarVertical = Scrollbar(frame, command=pyTextArea.yview)
scrollbarVertical.grid(row=0, column=1, sticky="nsew")
# Posiciona la barra una columna a la derecha del textArea

pyTextArea.config(yscrollcommand=scrollbarVertical.set)
# Establece que la barra "siga" al texto
```



Para una barra horizontal, se realiza igual pero cambiando **yview** por **xview**, **yscrollcommand** por **xscrollcommand**, se agrega la configuración **orient="horizontal"**, y además, es necesario agregar **wrap=NONE** a la configuración del widget para que el texto no se vaya automáticamente hacia la siguiente línea. **textArea = Text(frame, wrap=NONE)**

pyTextArea.get("1.0", "end")

Se obtiene el texto completo

pyTextArea.insert("end", texto)

Se añade texto al final del texto ya existente

pyTextArea.config(state="disabled")

Hace que el texto sea de sólo lectura
(requiere estado "normal" para escribirse)

- **Entry:** caja de texto

Entry(contenedor, opciones)

Puede ser de utilidad esconder el texto que se escribe (por ejemplo, para un password). Para ello, se puede utilizar la configuración: **.config(show="*")**

Se le puede establecer una variable de texto (para modificarlo u obtener su texto) con el parámetro **textvariable**. Previamente a esto, es necesario declarar una variable de tipo StringVar.

nombre = StringVar()

pyTextField = Entry(frame, textvariable = nombre)

Entones, con las siguientes instrucciones se puede establecer u obtener el texto de una caja de texto:

nombre.set("text")

otraVariable = nombre.get() 51



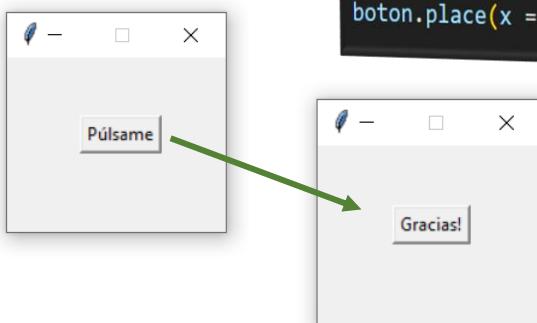
- **Button:** botón

```
Button(contenedor, text="", command=function)
```

Permite agregar un botón a la interfaz, el cual puede ejecutar una acción al pulsarse.

```
def botonActionPerformed():
    boton.config(text="Gracias!")

boton = Button(frame, text="Púlsame", command=botonActionPerformed)
boton.config(cursor="hand2") # Puntero se convierte en mano
boton.place(x = 50, y = 40)
```



Si se quisiera pasar un valor por parámetro, se debe usar una función anónima (lambda)

```
command=lambda:botonActionPerformed(3)
```

Si se desea que el botón interactúe con otro widget, se puede hacer de la siguiente manera:

```
from tkinter import *

raiz = Tk()
raiz.title("")
raiz.resizable(False, False)

frame = Frame(width=150, height=120)
frame.pack()

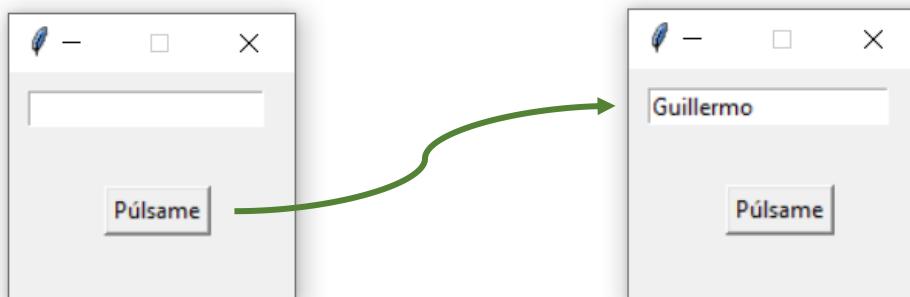
# Variable
nombre = StringVar() # Definida como tipo de dato string

# Caja de texto
pyTextField = Entry(frame, textvariable = nombre)
pyTextField.place(x = 10, y = 10)

# Botón
def botonActionPerformed():
    nombre.set("Guillermo") # con .get se obtendría el texto que ya hay

boton = Button(frame, text="Púlsame", command=botonActionPerformed)
boton.config(cursor="hand2") # Puntero se convierte en mano
boton.place(x = 50, y = 60)

raiz.mainloop()
```





- **RadioButton:** botones de selección para preguntas de respuesta única.

Requiere crear una variable que almacene cuál radiobutton está activo, esto se establece con el parámetro **variable** y **value**.

```
# Variable
varOpcion = IntVar()

# Funcion
def rbActionPerformed():
    print("Pulsaste el radio button: " + str(varOpcion.get()))

# RadioButtons
Radiobutton(frame, text="Masculino", variable=varOpcion, value=1,
command=rbActionPerformed).place(x=10, y=10)

Radiobutton(frame, text="Femenino", variable=varOpcion, value=2,
command=rbActionPerformed).place(x=10, y=30)
```



- **CheckButton:** casillas de verificación.

Requiere crear variables que almacenen si el check button está presionado o no. Para ello se utilizan los parámetros **variable**, **onvalue** y **offvalue**.

```
# Variables
aprendi_C = IntVar()
aprendi_Java = IntVar()
aprendi_Python = IntVar()

# Funciones
def cbActionPerformed():
    if aprendi_C.get() == 1: print("Sabes C")
    if aprendi_Java.get() == 1: print("Sabes Java")
    if aprendi_Python.get() == 1: print("Sabes Python")

#CheckButtons
Checkbutton(frame, text="C", variable=aprendi_C, onvalue=1, offvalue=0,
command=cbActionPerformed).place(x=10, y=10)

Checkbutton(frame, text="Java", variable=aprendi_Java, onvalue=1, offvalue=0,
command=cbActionPerformed).place(x=10, y=40)

Checkbutton(frame, text="Python", variable=aprendi_Python, onvalue=1, offvalue=0,
command=cbActionPerformed).place(x=10, y=70)
```



Valor que obtendrá la variable
si el checkbutton está activo

Valor que obtendrá la variable
si el checkbutton está inactivo



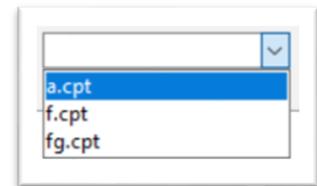
- **ComboBox**: lista desplegable

Para crear una lista desplegable, es necesario importar ttk de tkinter

```
from tkinter import ttk
```

Luego, se procede a crear la lista desplegable de la siguiente manera:

```
pyComboBox = ttk.Combobox(frame, state="readonly")
pyComboBox.place(x=10, y=10)
pyComboBox ['values'] = lista
```



Se puede obtener el **index** de la lista seleccionada con el método:

```
pyComboBox.current()
```

Si se desea establecer un seleccionado en particular como texto, se puede usar el método: `pyComboBox.set("")`

Se puede obtener el texto del seleccionado con el método: `pyComboBox.get()`

Para detectar cuando un elemento de la lista es seleccionado, se utiliza **bind**.

```
fileNameList.bind("<<ComboboxSelected>>", onListElementSelected)
```

```
def onListElementSelected(event):
```



- **Menu**

Para crear un menú, es necesario trabajar con la *raíz* de la interfaz gráfica.

1. Declarar la barra de menú y configurarla en la *raíz*

```
barraMenu = Menu(raiz)
raiz.config(menu=barraMenu)
```

2. Declarar los menú que estarán en la barra (*especificando que pertenecen a ella*)

```
menuArchivo = Menu(barraMenu, tearoff=0)
menuEdicion = Menu(barraMenu, tearoff=0)
menuHerramientas = Menu(barraMenu, tearoff=0)
menuAyuda = Menu(barraMenu, tearoff=0)
```

→ *elimina elemento vacío*

3. Añadir los menú declarados a la barra de menú → *nombre del menú*

```
barraMenu.add_cascade(label="Archivo", menu=menuArchivo)
barraMenu.add_cascade(label="Edición", menu=menuEdicion)
barraMenu.add_cascade(label="Herramientas", menu=menuHerramientas)
barraMenu.add_cascade(label="Ayuda", menu=menuAyuda)
```

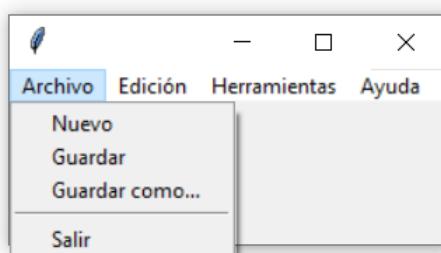
4. Añadir los elementos a cada menú creado

```
menuArchivo.add_command(label="Nuevo")
menuArchivo.add_command(label="Guardar")
menuArchivo.add_command(label="Guardar como...")
menuArchivo.add_separator()
menuArchivo.add_command(label="Salir", command=lambda:raiz.destroy())

menuEdicion.add_command(label="Copiar")
menuEdicion.add_command(label="Cortar")
menuEdicion.add_command(label="Pegar")

menuHerramientas.add_command(label="Borrar todo")

menuAyuda.add_command(label="Licencia")
menuAyuda.add_command(label="Acerca de..")
```





Grids

Disposición del espacio de un widget como una grilla (tabla).

El espacio que ocupa una columna o una fila se ajusta por defecto al espacio que ocupe el widget de mayor tamaño sobre esa fila o columna.

```
pyLabel1 = Label(frame, text="Nombre:")
pyLabel1.grid(row=0, column=0)

pyTextField1 = Entry(frame)
pyTextField1.grid(row=0, column=1)

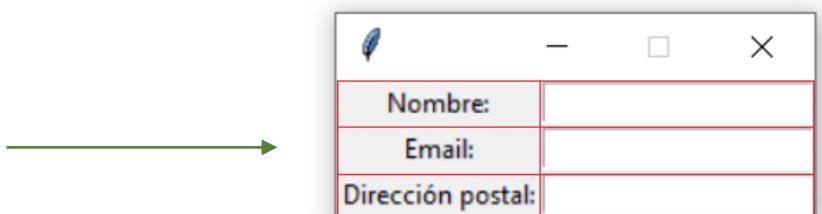
pyLabel2 = Label(frame, text="Email:")
pyLabel2.grid(row=1, column=0)

pyTextField2 = Entry(frame)
pyTextField2.grid(row=1, column=1)

pyLabel3 = Label(frame, text="Dirección postal:")
pyLabel3.grid(row=2, column=0)

pyTextField3 = Entry(frame)
pyTextField3.grid(row=2, column=1)
```

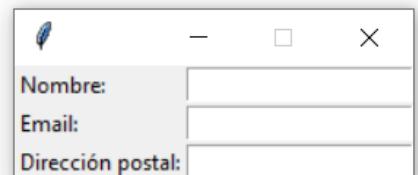
	column 0	column 1	column 2	column 3
row 0				
row 1				
row 2				
row 3				



Es importante recordar que, por defecto, un widget se ancla en la parte superior y centrado horizontalmente a su contenedor. Esto se puede modificar con el parámetro `sticky`.

```
pyLabel.grid(row=0, column=0, sticky="w")
```

Como valor usa los puntos cardinales
En caso de usar “nsew”, ocupará toda la celda



Box model

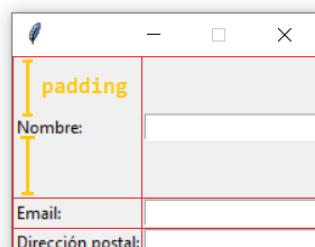
Es importante saber que cada `widget` tiene un contenido (ej: un texto), el cual está dentro de una “caja” delimitada por un borde.



El espacio entre el `borde` y el `contenido` de un `widget` se denomina `padding`. Además, el espacio que deja no disponible un `widget` por fuera de su “caja” se denomina `margin`.

En Python, el padding se puede controlar mediante los argumentos `padx` y `pady`.

```
pyLabel.grid(row=0, column=0, sticky="w", pady=40)
```





Si se desea “combinar celdas” de la grilla, se puede usar el parámetro `columnspan` del grid.

```
from tkinter import *

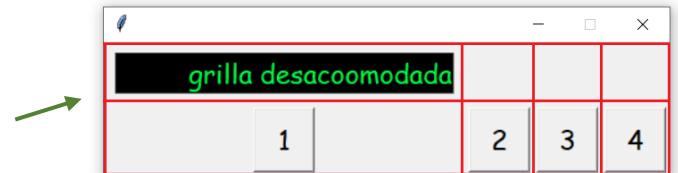
raiz = Tk()
raiz.title("")
raiz.resizable(False, False)

frame = Frame(raiz)
frame.pack()

Button(frame, text="1", width=3).grid(row=1, column=0)
Button(frame, text="2", width=3).grid(row=1, column=1)
Button(frame, text="3", width=3).grid(row=1, column=2)
Button(frame, text="4", width=3).grid(row=1, column=3)

pyTextField = Entry(frame)
pyTextField.grid(row=0, column=0, padx=10, pady=10)
pyTextField.config(background="black", fg="#03f943", justify="right")

raiz.mainloop()
```



Define la cantidad de columnas que ocupará el *widget*





Para conseguir un buen **responsive design** con el formato de grillas (grid), se puede aplicar `grid_rowconfigure` y `grid_columnconfigure` al widget/frame que contiene la grilla (grid). El primer parámetro de ambos métodos es el número de fila / columna, el segundo parámetro es la importancia de crecimiento respecto a otras (si es que hay) y el valor máximo es 1.

Es importante permitir que el frame se pueda expandir y llenar en la raíz con los parámetros `fill` y `expand` dentro del `pack`.

También es necesario que, el widget que se estirará para adaptarse a un nuevo tamaño de ventana, tenga su parámetro de grilla `sticky` en "nsew".

```
from tkinter import *

raiz = Tk()
raiz.title("Tutorial")

frame = Frame(raiz)
frame.pack(fill="both", expand=True)

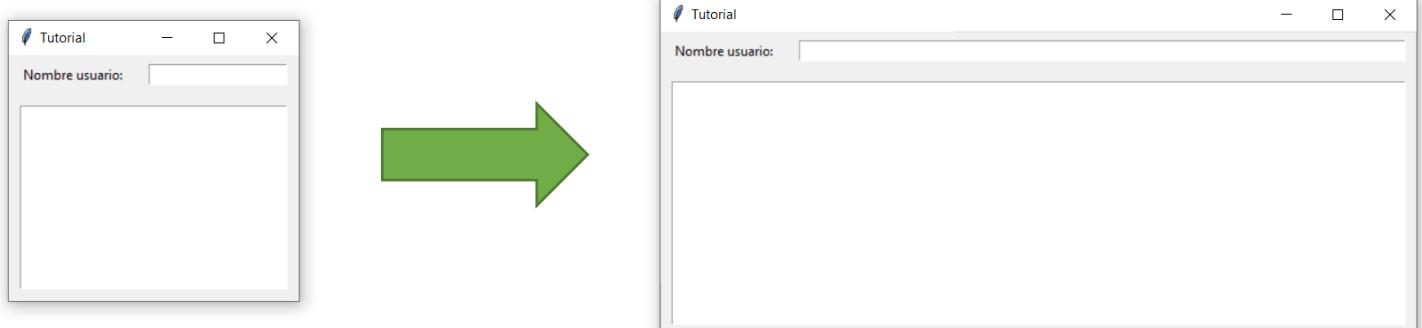
user_name = StringVar()

# Creando widgets en grid
Label(frame, text="Nombre usuario:").grid(row=0, column=0, sticky="e", padx=10, pady=7)
Entry(frame, textvariable=user_name).grid(row=0, column=1, sticky="we", padx=10, pady=7)
textArea = Text(frame, width=10, height=10)
textArea.grid(row=1, column=0, columnspan=2, sticky="nsew", padx=10, pady=10)

# Configurando la fila y columna que se adaptarán al tamaño del frame
frame.grid_rowconfigure(1, weight=1)
frame.grid_columnconfigure(1, weight=1)

# Estableciendo el tamaño mínimo de la interfaz
raiz.update()
raiz.minsize(frame.winfo_width(), frame.winfo_height()) # winfo obtiene tamaño actual

raiz.mainloop()
```





Ventanas emergentes

Para ejecutar ventanas emergentes, es necesario importar el módulo **messagebox**

```
from tkinter import messagebox
```

```
def mostrarInfo():
    messagebox.showinfo("Titulo", "Información")
    # Se reproduce sonido
    # Botón: Aceptar

def mostrarAviso():
    messagebox.showwarning("Titulo", "Aviso")
    # Se reproduce sonido
    # Botón: Aceptar

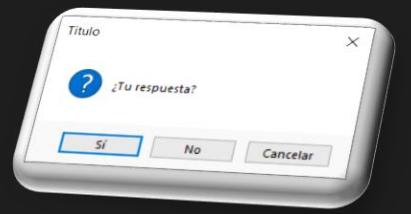
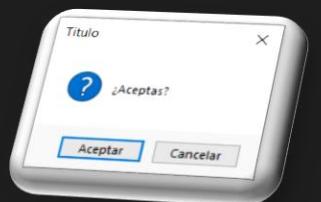
def mostrarError():
    messagebox.showerror("Titulo", "Error")
    # Se reproduce sonido
    # Botón: Aceptar

def mostrarPregunta_SiNo():
    respuesta = messagebox.askquestion("Titulo", "¿Pregunta?")
    # No se puede cerrar por la x
    # No se reproduce sonido
    # Botones: Si - No
    if respuesta == "yes":
        pass

def mostrarPregunta_AceptarCancelar():
    acepta = messagebox.askokcancel("Titulo", "¿Aceptas?")
    # No se reproduce sonido
    # Botones: Aceptar - Cancelar
    if acepta == True:
        pass

def mostrarPregunta_ReintentarCancelar():
    reintenta = messagebox.askokcancel("Titulo", "¿Reintentarás?")
    # No se reproduce sonido
    # Botones: Reintentar - Cancelar
    if reintenta == True:
        pass

def mostrarPregunta_SiNoCancelar():
    respuesta = messagebox.askyesnocancel("Titulo", "¿Tu respuesta?")
    # Botones: Si - No - Cancelar
    if respuesta == True: # Si
        pass
    elif respuesta == None: # Cancelar
        pass
```





Explorador de archivos

Para buscar archivos externos, es necesario importar el módulo **filedialog**.

Function	Parameters	Purpose
.askopenfilename	Directory, Title, Extension	To open file: Dialog that requests selection of an existing file.
.asksaveasfilename	Directory, Title, Extension)	To save file: Dialog that requests creation or replacement of a file.
.askdirectory	None	To open directory

Para seleccionar un archivo, se usa el método:

filedialog.askopenfilename()

Como parámetro, se puede usar:

- title establece el título de la ventana de búsqueda
- initialdir establece dónde comenzará la ventana de búsqueda (“Mis documentos” por defecto)
- filetypes determina el tipo de archivo que se busca (en formato **tupla**), mínimo 2 tipos

```
def abrirArchivo():
    filePath = filedialog.askopenfilename(title="Abrir archivo", initialdir="C:",
                                           filetypes=(
                                               ("Ficheros de Excel", "*.xlsx"),
                                               ("Todos los archivos", "*.*"))
                                           )
    print(filePath) # Devuelve la ruta del archivo seleccionado
```

tkinter.scrolledtext
Text widget with a vertical scroll bar built in.

tkinter.colorchooser
Dialog to let the user choose a color.

tkinter.commondialog
Base class for the dialogs defined in the other modules listed here.

tkinter.filedialog
Common dialogs to allow the user to specify a file to open or save.

tkinter.font
Utilities to help work with fonts.

tkinter.messagebox
Access to standard Tk dialog boxes.

tkinter.simpledialog
Basic dialogs and convenience functions.



Personalización de colores en eventos

Por defecto, TkInter establece los colores y estilos por defecto de cada widget cuando se realiza un evento (como presionar un botón, un radiobutton, etc).

Esto perjudica en los casos en que el color del fondo del widget es distinto del establecido por defecto.

Para prevenir estos cambios notorios de estilos, es necesario utilizar los eventos de **bind**.

```
# Submit
def submitActionPerformed(): pass
submitButton = Button(submitFrame, relief=RAISED, bg=BUTTON_BACKGROUND_COLOR, fg=FOREGROUND_COLOR)
submitButton.grid(row=0, column=0, sticky="nsew", pady=14)
submitButton.bind(
    "<ButtonPress-1>", # evento de ser presionado
    Lambda e: (
        submitButton.config(state="disabled"), # botón deshabilitado
        submitButton.config(
            fg=BACKGROUND_COLOR_SELECTED,
            bg=BUTTON_BACKGROUND_COLOR_SELECTED,
            relief=SUNKEN # apariencia de oprimido
        )
    )
)
submitButton.bind(
    "<ButtonRelease-1>", # evento de ser soltado
    Lambda e: (
        submitButton.config(state="normal"), # botón habilitado
        submitButton.config(
            fg=foreground_color,
            bg=BUTTON_BACKGROUND_COLOR,
            relief=RAISED # apariencia 3D
        ) submitActionPerformed()
    )
)
FOREGROUND_COLOR = "white"
BUTTON_BACKGROUND_COLOR = "#1a438b"
BUTTON_BACKGROUND_COLOR_SELECTED = "#143266"
```



Threads

Permite la ejecución de procesos en segundo plano sin que el programa se quede congelado mientras ese proceso está ejecutándose.

Requiere importar el módulo **threading**:

```
import threading
```

```
def unaFuncion(arg1, arg2):
    print(arg1, arg2)

param1 = "Guille"
param2 = 28
thread = threading.Thread(
    target=unaFuncion,
    args=(param1, param2)
)
thread.start()
```



Generación de un ejecutable

Para generar un ejecutable de un programa creado en Python, es necesario instalar **pyinstaller** desde el **Símbolo del sistema** (ejecutado en modo administrador).

```
C:\ Administrador: Símbolo del sistema
Microsoft Windows [Versión 10.0.19044.1766]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Windows\system32 pip install pyinstaller
```

Luego, una vez que finalice la instalación, hay que dirigirnos al directorio donde se encuentra nuestro programa con el comando **cd** (recordar que con **cd ..** se vuelve una carpeta hacia atrás).

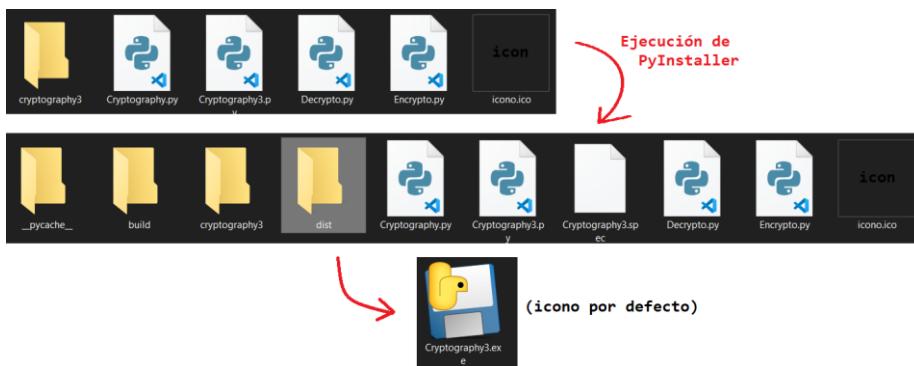
Una vez que se está en el directorio de nuestro programa, se puede crear el ejecutable utilizando el comando: **pyinstaller NOMBRE_DE_NUESTRO_PROGRAMA.py**

- ⊕ Modificadores de instrucción (seguido del comando **pyinstaller**)
 - windowed** el programa creado se ejecutará sin la consola detrás.
 - onefile** se creará un solo ejecutable dentro de la carpeta dist, el cual será más pesado y tardará un poco más en arrancar pero no necesita que esté Python instalado para usarse
 - icon=PATH** el ejecutable creado tendrá el ícono especificado en el PATH.

Si se busca que el ejecutable generado tenga un ícono, es necesario que dicho ícono se encuentre en formato **.ico**

El PATH de un ícono que se encuentra en el mismo directorio que nuestro programa a compilar será: **./ICON_NAME.ico**

```
C:\Users\Guill\Desktop\Proyectos\Cryptography>cd "C:\Users\Guill\Desktop\Proyectos\Cryptography"
C:\Users\Guill\Desktop\Proyectos\Cryptography>py -3.x -m PyInstaller --windowed --onefile --icon=icono.ico Cryptography3.py
```





Bases de datos

Python es capaz de trabajar con diversas bases de datos mediante módulos que pueden instalarse para importar en el código.

- SQL Server
- MySQL
- SQLite
- PostgreSQL
- Oracle

```
import pyodbc as sql # pip install pyodbc  
import MySQLdb as sql # pip install mysqlclient  
import sqlite3 as sql # pip install sqlite3  
import psycopg2 as sql # pip install psycopg2
```

Para comenzar a usar la Base de Datos, se necesita tener un Host y una Base de Datos creada.

Luego, se debe establecer la conexión a la Base de Datos.

Para poder manejar las peticiones (query) a la Base de Datos, es necesario crear un “cursor” que las gestione.

En caso de que no se pueda establecer la conexión, se lanzará un error el cual puede ser manejado.

Es importante cerrar las conexiones al finalizar para evitar corrupción de datos.

```
try:  
    conexion = sql.connect(  
        host = host,  
        database = dbname,  
        user = userName,  
        password = userPassword,  
        port = 5432 # Postgre  
    )  
  
    cursor = conexion.cursor()  
  
    # Aquí se ejecutarán las querys #  
  
except (Exception, sql.DatabaseError) as error:  
    print("Error al crear la tabla: ", error)  
  
finally: # Cerrar conexiones al finalizar  
    if cursor: cursor.close()  
    if conexion: conexion.close()
```

Puede ser de utilidad tener una variable con una tupla de las columnas de la tabla de la base de datos.

```
columns = ("id", "name", "age")
```



Creación de una Tabla

```
# Crear Tabla en PostgreSQL
cursor.execute(f'''
    CREATE TABLE {tablename} (
        id SERIAL PRIMARY KEY,
        name TEXT,
        age INTEGER
    )
''')
conexion.commit()
```

```
# Crear Tabla en MySQL
cursor.execute(f'''
    CREATE TABLE {tablename} (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER
    )
''')
conexion.commit()
```

```
#Establecer la conexión (si no existe, la crea)
conexion = sql.connect('mi_database.db')
cursor = conexion.cursor()

# Crear Tabla en SQLite3
cursor.execute(f'''
    CREATE TABLE {tablename} (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER
    )
''')
conexion.commit()
```

```
# Establecer conexión
conexion = sql.connect(
    driver = '{SQL Server Native Client 11.0}',
    server = host,
    database = dbname,
    uid = userName,
    pwd = userPassword
)
cursor = conexion.cursor()

# Crear Tabla en SQL Server
cursor.execute(f'''
    CREATE TABLE {tablename} (
        id INT PRIMARY KEY IDENTITY,
        name NVARCHAR(255),
        age INT
    )
''')
conexion.commit()
```

Ler una tabla

```
# Leer Tabla
query = f"SELECT * FROM {tablename}"
cursor.execute(query)
tabla = cursor.fetchall()

# Imprimir contenido de la tabla
for registro in tabla:
    for i in range(len(columns)):
        print(f'{columns[i]}: {registro[i]}')
```



Insertar una fila en la tabla

Agrega una fila en la tabla, iniciando con el ID en **1** (no existe id 0).

```
# Funcion útil para simplificar la query
allColumns = Lambda columns:str(columns).replace("'",'').lstrip("(id, ").rstrip(")")

# Insertar fila en una tabla
query = f"INSERT INTO {tablename} ({allColumns(columns)}) VALUES (%s, %s)"
cursor.execute(query, (
    "Guillermo",
    25
))
```

Aclaración: En SQLite y SQL Server se reemplaza **%s** por **?**

Eliminar una fila de una tabla

```
# Eliminar una fila por id
id_a_eliminar = 1
query = f"DELETE FROM {tablename} WHERE (id = %s)"
cursor.execute(query,
    (id_a_eliminar, ))
```

Editar una fila de una tabla

```
# Editar una fila por id
id_a_editar = 1
query = f"UPDATE {tablename} SET name = %s, age = %s WHERE (id = %s)"
cursor.execute(query,
    (
        "Nicolas", # new name
        "28",       # new age
        1           # id que se editará
    )
)
conexion.commit()
```



Ver los usuarios de la base de datos

```
# PostgreSQL users
query = "SELECT username FROM pg_user"
cursor.execute(query)
users = cursor.fetchall()
for user in users:
    print(user[0])
```

Ver permisos de usuarios en una tabla

```
# PostgreSQL permission
query = f"SELECT grantee, privilege_type FROM information_schema.table_privileges WHERE table_name = '{tablename}'"
query = f"SELECT relname, relacl FROM pg_class WHERE relname = '{tablename}'"
cursor.execute(query)
results = cursor.fetchall()
for result in results:
    print(f"User/Rol: {result[0]}, Permiso: {result[1]}")
```

Crear usuario

```
# PostgreSQL create user
query = f"CREATE USER {username} WITH PASSWORD '{password}';"
cursor.execute(query)
conexion.commit()
```

Dar permisos a un usuario en una tabla específica

Para dar permisos, se utiliza el comando **GRANT** seguido de los permisos que se desean dar: **select**, **insert**, **update** y/o **delete**.

```
# PostgreSQL grant permission
query = f"GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO {username}"
query = f"GRANT INSERT, UPDATE, DELETE ON TABLE {tablename} TO {username}"
query = f"GRANT ALL PRIVILEGES ON TABLE {tablename} TO {username}"
cursor.execute(query)
conexion.commit()
```



Solicitudes HTTP

El módulo **requests** permite utilizar métodos como **fetch** de JavaScript, el cual se utiliza para obtener información de una página web o API.

```
pip install requests
```

```
import requests

url = 'https://ejemplo.com/api'
response = requests.get(url)

if response.status_code == 200: # success
    datos = response.text

else:
    print("Status ", response.status_code)
```

Método GET

```
params = {                      # dictionary
    'param1': 'valor1',
    'param2': 'valor2'
}
response = requests.get(url, params=params)
```

Método POST

```
json = {                      # dictionary
    'clave1': 'valor1',
    'clave2': 'valor2'
}
response = requests.post(url, data=json)
```