Formación profesional en CePETel 2023

Desde la Secretaria Técnica del Sindicato CePETel convocamos a participar en el siguiente curso de formación profesional:

Programación Orientada a Objetos con PYTHON

Clases: 12 clases de 3 hs c/u de 18:00 a 21:00 hs.
Días que se cursa: miércoles 22 y 29 de marzo; 5, 12, 19 y 26 de abril; 3, 10, 17, 24 y 31 de mayo; y 7 de junio.
Modalidad: a distancia (requiere conectarse a la plataforma Zoom en los días y horarios indicados precedentemente).
Docente: Nelson Villalba

La capacitación es:

- **Sin cargo para afiliados y su grupo familiar directo.**
- **Sin cargo para encuadrados con convenio CePETel.**
- **Con cargo al universo no contemplado en los anteriores.**

Informes: enviar correo a tecnico@cepetel.org.ar

Inscripción (hasta el 16 de marzo): ingresar al formulario (se recomienda utilizar una cuenta de correo personal)

https://forms.gle/smE237X5nf8dTPdy7

Programa

Unidad 01: Algoritmos
Formulación de problemas.
Etapas en la resolución de problemas computacionales.
Algoritmo.
Definición.
Características.

Unidad 02: Introducción al Lenguaje Python
Introducción al Python.
Instalación del programa.
Utilización de Ipython - Google Colab.

Unidad 03: Tipos de datos, Operadores, Condiciones
Tipos de datos: int, float, string, boolean.
Variables y constantes.

Ing. Daniel Herrero – Secretario Técnico – CDC

Escritura y Lectura de datos (mostrar y solicitar datos), conversiones de datos.
Operadores de asignación.
Operadores aritméticos, relacionales y lógicos.
Condiciones.
Actualizar variables.

Unidad 04: Estructuras de control selectivas o condicionales y repetitivas

Estructuras de control:
- Selectivas/simples.
- Dobles y múltiples if-else-elif.

Estructuras repetitivas: for y while.
Contadores y Acumuladores.

Unidad 05: Estructura de Datos Complejos

Tipos de datos complejos: listas, tuplas, diccionarios, conjuntos.
Métodos de listas.
Manejo de cadenas de caracteres.

Unidad 06: Funciones

Funciones.
Parámetros.
Ámbitos de las variables.
ARGS vs KWARGS.
Librerías de terceros.
Funciones de primer nivel (build in).
Cómo importar Librerías (tkinter).
Librería iDatetime (Fechas).

Unidad 08: Programación Orientada a Objetos. Clases y Objetos

Definiciones.
Abstracción.
Unidad y Objetos.
Atributos y Métodos.
Constructores.
Encapsulamiento.
Métodos de Acceso Setter y Getter.

Unidad 09: Herencia y Polimorfismo

Objetos dentro de Objetos
Herencia.
Polimorfismo.
Listas de Objetos.
Objetos Relacionados.

Unidad 10: Manejo de Excepciones y Ficheros

Introducción a las excepciones.
Múltiples excepciones, invocación de excepciones.
Creación de excepciones propias.
Ficheros de texto, ficheros y objetos.
App con datos persistentes.

Ficheros CSV y ficheros JSON. Uso de JSON.

Unidad 11: Bases de datos

Base de Datos: Concepto.
Base de Datos Relacional.
Lenguaje SQL.

Unidad 12: Bases de datos (continuación)

CRUD.
Clave primaria.
SQL segunda parte.
Clave foránea.
SQLITE
BD + Python

Acerca del docente

Nelson Villalba es Analista de Sistemas egresado del I.S.F.P. Padre Jose Frassineti – C.E. Loreto (Avellaneda).

A la fecha se desempeña en Telefónica de Argentina S.A. como Supervisor de Obras Interior Región NOA con tareas de: Proyecto, seguimiento, supervisión y aceptación de sistemas satelitales; Supervisión de instalación, puesta en marcha, integración y aceptación de sistemas de la red celular 2G, 3G y 4G de Movistar.

Anteriormente y en la misma empresa realizó tareas de: Proyecto, seguimiento, supervisión y aceptación de sistemas de radio y TV satelitales, redes informáticas satelitales y sistemas de gestión y control informáticos asociados; Proyecto, seguimiento, supervisión y aceptación de sistemas de radio enlaces terrestres y satelitales y redes informáticas de gestión.

En 2019 dictó de manera presencial en nuestra sede de Rocamora 4029 el curso Introducción a las Redes Satelitales.

Unidad 01

Formulación de problemas.

Etapas en la **resolución** de problemas computacionales.

Algoritmo

Definición

Características

¿QUÉ ES PROGRAMAR?

Programar es definir instrucciones para ser ejecutadas por un ordenador, el objetivo es resolver un problema.

Si deseamos aprender a programar debemos saber lo siguiente:

Los programadores son personas que resuelven problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático.

```
La resolución de un problema exige el diseño de un algoritmo.
```

¿QUÉ ES UN ALGORITMO?

algoritmo ≠ **programa**

Es un conjunto de pasos ordenados y finitos que permiten resolver un problema o tarea específica.

Es una lista de instrucciones "escritas" en un lenguaje de programación, que una computadora interpreta para resolver un problema.

IMPORTANCIA DEL ALGORITMO

- Sin algoritmo no puede existir un programa.
- Un algoritmo es independiente del lenguaje de programación y de la computadora que lo ejecuta.
- Un mismo algoritmo puede expresarse en lenguajes diferentes de programación y ejecutarse en computadoras distintas.

PARTES DE UN ALGORITMO INFORMÁTICO

Las **tres partes de un algoritmo** son:

1. **Input (entrada):** Información que damos al algoritmo con el que va a trabajar para ofrecer la solución esperada.
2. **Proceso:** Conjunto de pasos para que, a partir de los datos de entrada, llegue a la solución de la situación.
3. **Output (salida):** Resultados, a partir de la transformación de los valores de entrada durante el proceso.

De este modo, un algoritmo informático parte de un estado inicial y de unos valores de entrada, sigue una serie de pasos sucesivos y llega a un estado final en el que ha obtenido una solución.

CARACTERÍSTICAS DE ALGORITMOS

Asimismo, los algoritmos presentan una serie de *características comunes,* que son:

- *Precisos.* Objetivos, sin ambigüedad.
- *Ordenados.* Presentan una secuencia clara y precisa para poder llegar a la solución.
- *Finitos.* Contienen un número determinado de pasos.
- *Concretos.* Ofrecen una solución determinada para la situación o problema planteados.
- *Definidos.* El mismo algoritmo debe dar el mismo resultado al recibir la misma entrada.

ALGUNOS ALGORITMOS FAMOSOS (E INFLUYENTES)

Aunque todo esto te suene formal, e incluso aburrido, el ingenio de programadores de todo el mundo ha conseguido que algunos algoritmos se hayan hecho famosos.

- **PageRank, de Google**

Uno de los más utilizados del mundo. Se trata del conjunto de algoritmos que utiliza Google para determinar la importancia de los documentos indexados por su motor de búsqueda.
Dicho de otro modo, cuando realizas una búsqueda en Google, es uno de los elementos que **decide el orden en el que se te muestran los resultados**.

ALGUNOS ALGORITMOS FAMOSOS (E INFLUYENTES)

- **El algoritmo del Timeline de Facebook.**

Se trata de otro algoritmo que influye en nuestra vida mucho más de lo que creemos.
El conjunto de algoritmos que alimentan el Timeline de Facebook determina los contenidos que se nos muestran en el espacio más frecuentado de la red social. Así, en base a una serie de parámetros (gustos personales, respuesta a contenidos anteriores, etc), los algoritmos deciden cuál es contenido que nos va a mostrar la red social y en qué orden lo hará.

- **Algoritmos de Trading de Alta Frecuencia.**

Mueven miles de millones de dólares en los mercados cada día. Se trata de algoritmos utilizados por muchas de las más importantes entidades financieras del mundo, que lanzan órdenes al mercado en función del beneficio que éstos prevén obtener, según las condiciones de mercado dadas en cada momento.

Recomendación

El código de la discordia

Un grupo de jóvenes sostiene una batalla judicial larga y cuesta arriba contra Google. Ellos afirman ser los creadores del algoritmo que dio origen a Google Earth.

Ejemplo:

Un procedimiento que realizamos varias veces al día consiste en lavarnos los dientes. Veamos la forma de expresar este procedimiento como un Algoritmo:

INICIO del ALGORITMO

1. Tomar la crema dental.
2. Destapar la crema dental.
3. Tomar el cepillo de dientes.
4. Aplicar crema dental al cepillo.
5. Tapar la crema dental.
6. Abrir la llave del lavamanos.
7. Remojar el cepillo con la crema dental.
8. Cerrar la llave del lavamanos.
9. Frotar los dientes con el cepillo.
10. Abrir la llave del lavamanos.
11. Enjuagarse la boca.
12. Enjuagar el cepillo.
13. Cerrar la llave del lavamanos.
14. Cerrar la llave del lavamanos.

FIN DEL ALGORITMO

Ejemplo:

- Nos piden crear un algoritmo en el cual pueda sumar dos números y mostrar el resultado de esa suma.
- y también programarlo en ……..
- Y finalmente entregarlo a mi jefe/cliente.…
-

Rescribimos el problema:

Tengo que crear un algoritmo en el cual pueda sumar dos números y mostrar el resultado de esa suma

Dados 2 números reales, calcular y mostrar el resultado de la suma entre ellos.

¿Cómo sería el proceso?

1. Definición del problema.
2. Análisis del problema.
3. Diseño y técnicas para la formulación de un algoritmo.
4. Codificación.
5. Prueba y Depuración.
6. Documentación.
7. Mantenimiento.

¿Cómo sería el proceso?

Definición del problema: Dados 2 números reales, calcular y mostrar el resultado de la suma entre ellos.

Análisis del problema: Necesitamos dos números reales (entrada), los cuales vamos a sumar (proceso), y mostrar el resultado (salida).

Diseño del algoritmo: Cheems va a ingresar un número (llamado num1), luego el segundo número (llamado num2), después vamos a calcular la suma (num1 + num2), y finalmente vamos a mostrar el resultado.

¿Cómo sería el proceso?

Codificación

```
num1=int(input("ingrese primer numero:"))
num2=int(input("ingrese primer numero:"))
suma=num1+num2
print("el resultado es", suma)
```

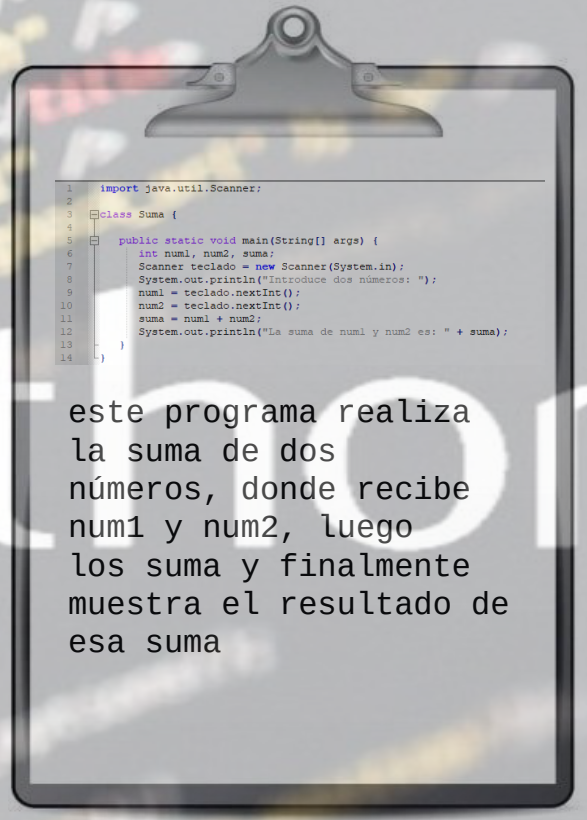
Prueba y Depuración

num1	num2	suma	salida
3	5	8	8

¿Cómo sería el proceso?

Documentación

Vamos a crear una anotación sobre qué es lo que realiza nuestro programa, que en este caso, simplemente suma dos números



este programa realiza la suma de dos números, donde recibe num1 y num2, luego los suma y finalmente muestra el resultado de esa suma

Mantenimiento

Si a futuro queremos mejorar nuestro programa, añadirle más funcionalidades o que realice otras operaciones, podemos hacerlo, siempre dependerá del contexto donde se lo utilice y el problema a resolver.

Ejemplo 1:

Calcular el área y el perímetro de un rectángulo, para lo cual se deben ingresar el valor del lado A y el valor del lado B, ambos números reales.

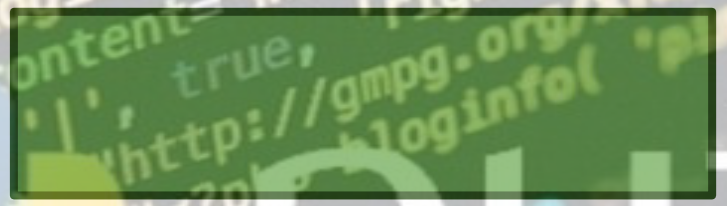
1. Definición del problema

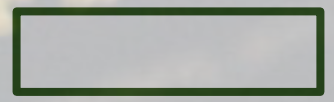
➔¿El enunciado es preciso?
➔¿Es posible calcular el área y perímetro de un rectángulo?

2. Análisis del problema

➔¿Cuáles son los datos de entrada y salida?
➔¿Son suficientes los datos de entrada para resolver el problema?
➔?Conocemos las fórmulas para obtener el área y el perímetro?
➔¿Somos capaces de ejemplificar con algunos valores de entrada cuáles serían los valores de salida?
➔¿Qué tipo de instrucciones necesitamos para diseñar el algoritmo?

Tomemos nota de lo que tenemos y lo que sabemos.





perímetro = 2a + 2b

area = b * h

Datos de Entrada: Dos números reales llamados LADO_A y LADO_B. Datos de Salida: Valor de AREA y PERIMETRO, que serán reales.

Caso de Prueba 1:

```
Datos de Entrada: LADO_A = 4; LADO_B = 8
Datos de Salida: AREA = 32
Datos de Salida: PERIMETRO = 24
```

Caso de prueba 2:

```
Datos de Entrada: LADO_A = 3; LADO_B = 7
Datos de Salida: AREA = 21
Datos de Salida: PERIMETRO = 20
```

Necesitamos usar:

- *Instrucciones para el ingreso de datos*
- *Instrucciones para la asignación de datos*
- *Instrucciones para la salida de datos*

3. Diseñar el algoritmo

1. Ingresar un valor para LADO_A
2. Ingresar un valor para LADO_B
3. Asignar a ÁREA el resultado de: LADO_A * LADO_B.
4. Asignar a PERÍMETRO el resultado de: 2*LADO_A + 2*LADO_B.
5. Mostrar el valor de ÁREA.
6. Mostrar el valor de PERÍMETRO.

Fin.

4. Prueba de Escritorio

LADO_A	LADO_B	ÁREA	PERÍMETRO	SALIDA
4	8	32	24	**32** **24**
3	7	21	20	**21** **20**
8,5	2	17	21	**17** **21**

El sueldo de un vendedor es la suma de un monto fijo pagado por el gerente más el 20% de sus ventas mensuales. Teniendo como dato el sueldo fijo y el monto total de la venta del mes del vendedor, calcule y muestre el salario final que recibirá.

- ¿Sabemos obtener un porcentaje?
- ¿Podemos escribir una expresión algebraica que calcule el salario final?

Datos de Entrada: Dos valores reales llamados sueldo_inicial y monto_ventas
Datos de Salida: Valor del sueldo_final que será un número real.

Caso de Prueba 1:

Datos de Entrada:
sueldo_inicial = 10000
monto_ventas = 56000

Datos de Salida:
sueldo_final = 21200

Caso de Prueba 2:

Datos de Entrada:
sueldo_inicial = 12000
monto_ventas = 90000

Datos de Salida:
sueldo_final = 30000

Necesitamos usar:

- *Instrucciones para el ingreso de datos*
- *Instrucciones para la asignación de datos*
- *Instrucciones para la salida de datos*

3. Diseñar el algoritmo

1. Ingresar un valor para sueldo_inicial
2. Ingresar un valor para monto_ventas
3. Asignar a sueldo_final el resultado de: sueldo_inicial + (monto_ventas*20/100).
4. Mostrar el valor de sueldo_final.

   Fin.

4. Prueba de Escritorio

sueldo_inicial	monto_ventas	sueldo_final	SALIDA
10000	56000	21200	**21200**
12000	90000	30000	**30000**
9500	25800	14660	**14660**

El sueldo de un vendedor es la suma de un monto fijo pagado por el gerente más un porcentaje de sus ventas mensuales en caso que superen un monto determinado.
Si el monto de sus ventas fue menor a $20000, no recibe porcentaje, en caso contrario recibe el 20% de ese monto.
Teniendo como dato el sueldo fijo y el monto de la venta mensual del vendedor, calcule y muestre el salario final que recibirá.

- ¿Reconocemos las diferencias entre este ejemplo y el anterior?
- La condición para recibir el porcentaje de ventas es simple o compuesta?

Datos de Entrada: 2 valores reales llamados sueldo_inicial y monto_ventas
Datos de Salida: Valor del sueldo_final que será un número real.

<u>Caso de Prueba 1:</u>

Datos de Entrada:
sueldo_inicial = 10000
monto_ventas = 56000

Datos de Salida:
sueldo_final = 21200

sí recibe porcentaje

<u>Caso de Prueba 2:</u>

Datos de Entrada:
sueldo_inicial = 10000
monto_ventas = 15000

Datos de Salida:
sueldo_final = 10000

no recibe porcentaje

<u>Caso de Prueba 3:</u>

Datos de Entrada:
sueldo_inicial = 10000

monto_ventas = 20000

Datos de Salida:
sueldo_final = ...

¿?

Necesitamos usar:

- *Instrucciones para el ingreso de datos, de alternativa/condición*
- *Instrucciones para la asignación de datos, para la salida de datos*

3. Diseñar el algoritmo

1. Ingresar un valor para sueldo_inicial
2. Ingresar un valor para monto_ventas
3. si (monto_ventas < 20000) entonces
   a) Asignar a sueldo_final el resultado de: sueldo_inicial

sino

   a) Asignar a sueldo_final el resultado de: sueldo_inicial + (monto_ventas*20/100)

1. Mostrar el valor de sueldo_final

Fin

4. Prueba de Escritorio

1. Ingresar un valor para sueldo_inicial
2. Ingresar un valor para monto_ventas
3. si (monto_ventas < 20000) entonces
   a) Asignar **a sueldo_final el resultado de: sueldo_inicial**

   sino
   a) Asignar a sueldo_final el resultado de: sueldo_inicial + (monto_ventas*20/100)
1. Mostrar el valor de sueldo_final

   Fin

sueldo_inicial	monto_ventas	sueldo_final	SALIDA
20000	25000	25000	25000
15000	20000	19000	19000
15000	10000	15000	15000

Unidad 02

Introducción al Python

Instalación del programa

Utilización de Ipython - Google Colab

¿Que es Python?

Python es un lenguaje de programación de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender, y fue creado a principios de los noventa por Guido van Rossum en los Países Bajos.

Es un lenguaje de alto nivel, que permite procesar fácilmente todo tipo de estructuras de datos, tanto numéricas como de texto.

Es software libre, y está implementado en todas las plataformas y sistemas operativos habituales.

Características básicas

Las características del lenguaje de programación Python se resumen a continuación:

- Es un lenguaje interpretado, no compilado que usa tipado dinámico, fuertemente tipado (el tipo de valor no cambia repentinamente).
- Es multiplataforma, lo cual es ventajoso para hacer ejecutable su código fuente entre varios sistema operativos.
- Es un lenguaje de programación multiparadigma, el cual soporta varios paradigmas de programación como orientación a objetos, estructurada, programación imperativa y, en menor medida, programación funcional.
- En Python, el formato del código (p. ej., la indentación) es estructural.

El Zen de Python

Zen: Sistema filosófico budista que tuvo su origen en China en el siglo VI; se caracteriza por potenciar la meditación metafísica utilizando técnicas lógicas especiales (como las paradojas) y el ejercicio físico arduo, con el fin de conseguir la iluminación que revela la verdad.

El Zen de Python es una colección de 20 principios de software que influyen en el diseño del Lenguaje de Programación Python, de los cuales 19 fueron escritos por Tim Peters en junio de 1999:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.

El Zen de Python

- Los errores nunca deberían dejarse pasar silenciosamente.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Ventajas

- Simplificado y rápido: Este lenguaje simplifica mucho la programación “hace que te adaptes a un modo de lenguaje de programación, Python te propone un patrón”. Es un gran lenguaje para scripting, si usted requiere algo rápido (en el sentido de la ejecución del lenguaje), con unas cuantas líneas ya está resuelto.

- Elegante y flexible: El lenguaje le da muchas herramientas, si usted quiere listas de varios tipo de datos, no hace falta que declares cada tipo de datos. Es un lenguaje tan flexible usted no se preocupa tanto por los detalles.

- Programación sana y productiva: Programar en Python se convierte en un estilo muy sano de programar: es sencillo de aprender, direccionado a las reglas perfectas, le hace como dependiente de mejorar, cumplir las reglas, el uso de las lineas, de variables”. Ademas es un lenguaje que fue hecho con productividad en mente, es decir, Python le hace ser mas productivo, le permite entregar en los tiempos que me requieren.

Ventajas

- Ordenado y limpio: El orden que mantiene Python, es de lo que más le gusta a sus usuarios, es muy legible, cualquier otro programador lo puede leer y trabajar sobre el programa escrito en Python. Los módulos están bien organizados, a diferencia de otros lenguajes.

- Portable: Es un lenguaje muy portable (ya sea en Mac, Linux o Windows) en comparación con otros lenguajes. La filosofía de baterías incluidas, son las librerías que más usted necesita al día a día de programación, ya están dentro del interprete, no tiene la necesidad de instalarlas adicionalmente con en otros lenguajes.

- Comunidad: Algo muy importante para el desarrollo de un lenguaje es la comunidad, la misma comunidad de Python cuida el lenguaje y casi todas las actualizaciones se hacen de manera democrática.

Desventajas

- Curva de aprendizaje: La “curva de aprendizaje cuando ya estás en la parte web no es tan sencilla”.
- Hosting: La mayoría de los servidores no tienen soporte a Python, y si lo soportan, la configuración es un poco difícil.
- Librerías incluidas: Algunas librerías que trae por defecto no son del gusto de amplio de la comunidad, y optan a usar librerías de terceros.

Python

- **ES UN LENGUAJE DE PROPÓSITO GENERAL.**
  - Seguridad Informática
  - Desarrollo Web
  - Videojuegos

- **Es multiparadigma:** Son métodos aplicables en todos los niveles del diseño de programas para resolver problemas computacionales:
  - Programación Estructurada
  - Programación Orientada a Objeto

Python

- **Es un lenguaje**
  - Interpretado (no se compila)
  - Dinámico (sus variables cambian en tiempo de ejecución)
  - Multiplataforma (windows, linux, mac)

- Es **Case-Sensitive:** reconoce entre mayúsculas y minúsculas. No es lo mismo decir a = 5 que A = 5. Para PYTHON son dos elementos diferentes.

- Es fácil de **implementar** a la hora de aprender.

Colab (Ipython/Jupyter notebooks)

En este curso usaremos Google Colab para desarrollar los trabajos prácticos. Pero, **¿qué es Google Colab?.**

Primero debemos tener en cuenta lo siguiente

IPython

IPython es un shell interactivo que añade funcionalidades extra al modo interactivo incluido con Python, como resaltado de líneas y errores mediante colores, una sintaxis adicional para el shell, autocompletado mediante tabulador de variables, módulos y atributos; entre otras funcionalidades.

Colab (Ipython/Jupyter notebooks)

Jupyter Notebooks

Jupyter Notebook es un entorno informático interactivo basado en la web para crear documentos de Jupyter notebook, que a su vez utiliza **IPython** para ejecutar el código. Está pensado para soportar muchos lenguajes, principalmente Julia, Python, R (de allí su nombre) y Scala, enfocado en las áreas de data science y machine learning.

Colab (Ipython/Jupyter notebooks)

Google Colaboratory

Colab, o "Colaboratory", es una implementación de Jupyter Notebooks creada por Google. Permite escribir y ejecutar código de Python en el navegador, utilizando los servidores de Google. Solo se necesita tener una cuenta gratuita de Google (Gmail).

Algunas de sus ventajas son:

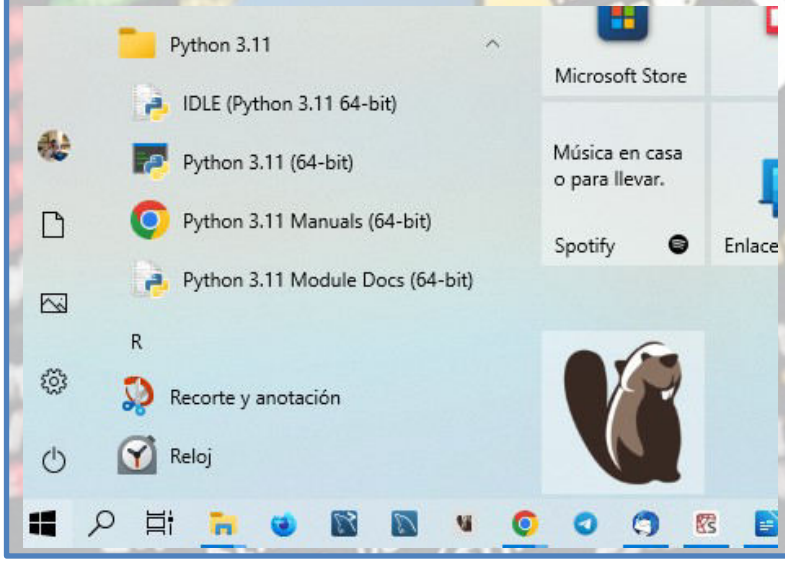
- Sin configuración requerida
- Acceso gratuito a GPU
- Facilidad para compartir

La pagina para acceder a Colab es:

https://colab.research.google.com/

Instalacion del software

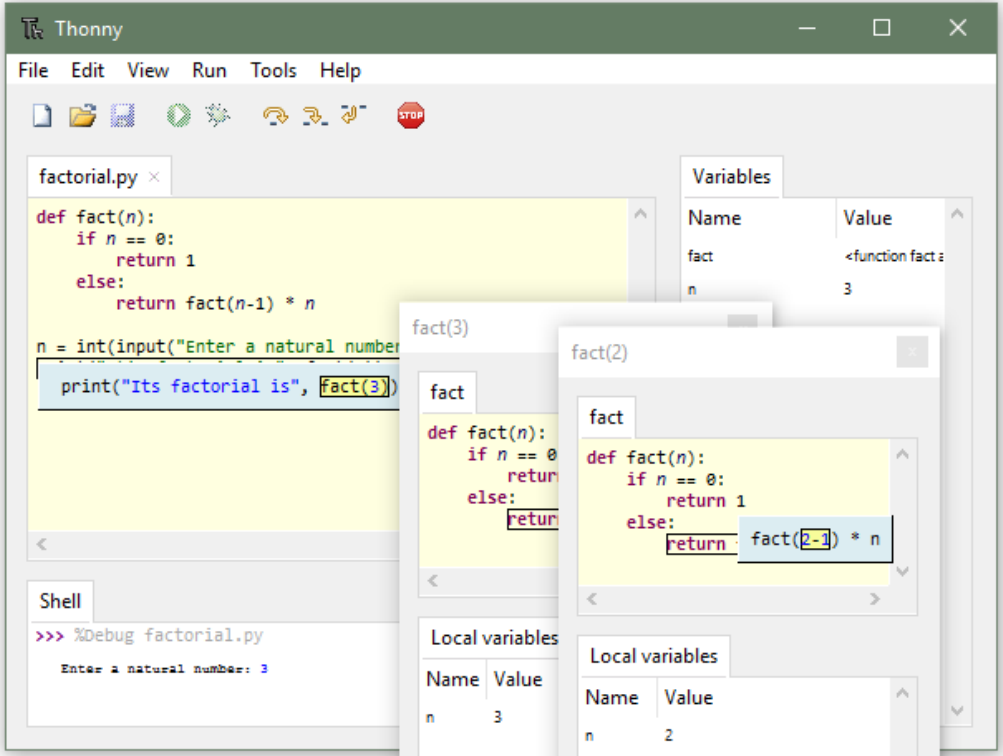
Para instalar Python en Windows debes ir a la sección de descargas de la web oficial y seleccionar la última versión. La versión mas reciente es la 3.11.2.
https://www.python.org/downloads/
Una vez descargado, ejecutamos el instalador y seguimos las instrucciones para su instalación Una vez terminado el proceso vamos a Inicio y buscamos la carpeta con los accesos directos a las aplicaciones que se instalaron:



1) **IDLE:** (Integrated Development and Learning Environment) es un Entorno Integrado de Desarrollo y Aprendizaje. Nos permite ejecutar codigo en forma directa o crear un script que luego se puede ejecutar desde aquí mismo y se pueden guardar los archivos fuente.
2) **Python 3.11:** es una consola de Python que permite ejecutar codigo, pero es mas primitivo ya que no tiene ninguna ayuda a la sintaxis (intellisense o auto completado). Se pueden ejecutar bloques de codigo pero es muy incomodo porque si hay un error hay que volver a escribir todo de nuevo.
3) **Python 3.11 Manuals:** este acceso directo nos lleva a una pagina en donde podemos encontrar los manuales, tutoriales, ayuda y toda la documentacion de Python.
4) **Python 3.11 Module Docs:** este acceso nos lleva a una pagina en donde se puede encontrat toda la documentacion de los Modulos de Python.

Instalacion del software

En la diapositiva anterior hemos visto como instalar el programa, y junto con el algunas utilidades.
Para la creación, edición y ejecución de código con esto seria suficiente, pero existe la posibilidad de instalar otros IDE (Entorno de Desarrollo Integrado) que nos podrían facilitar mas el trabajo.
En este curso vamos a utilizar el IDLE y la consola que viene con el Python para familiarizarnos con su uso, pero las practicas las vamos a efectuar con el Google Colab.
Tambien les voy a sugerir que instalen un IDE muy bueno y liviano con prestaciones superiores al IDLE del Python.
Este IDE que les mencione se llama ***Thonny,*** y fue desarrollado en la Universidad de Tartu (Estonia) con la ayuda de la comunidad de código abierto, la Raspberry Pi Foundation y Cybernetica AS.
Pueden descargarlo de: https://thonny.org/
Es muy sencillo de instalar y realmente es muy bueno, liviano y tiene muchas utilidades interesantes, entre ellas la posibilidad de efectuar ***debug*** del código, aunque el IDLE del Python también tiene la posibilidad de hacer debug pero es muy engorroso y poco claro.
Hay otras IDE que también dan algunas herramientas interesantes, pero esta para mi es la mejor y mas fácil de utilizar.


Thonny
File Edit View Run Tools Help
factorial.py
def fact(n):
if n == 0:
return 1
else:
return fact(n-1) * n
n = int(input("Enter a natural numbe
print("Its factorial is", fact(3))
Variables
Name Value
fact <function fact a
n 3
Shell
>>> %Debug factorial.py
Enter a natural number: 3
fact(3)
fact
Local variables
Name Value
n 3
fact(2)
fact
def fact(n):
if n == 0:
return 1
else:
return fact(2-1) * n
Local variables
Name Value
n 2

Unidad 03

Tipos de datos (int, float, str, bool).

Variables y constantes.

Escritura y lectura de datos (mostrar y solicitar datos), conversiones de datos.

Operadores de asignación.

Operadores aritméticos, relacionales y lógicos.

Condiciones.

Tipos de datos en python

TIPOS DE DATOS SIMPLE EN PYTHON

Número Entero (int)
Este tipo de dato se corresponde con números enteros, es decir, sin parte decimal.

Número real (float)
Este tipo de dato se corresponde con números reales con parte decimal. Cabe destacar que el separador decimal en Python es el punto (.) y no la coma (,).

Booleano (bool)
Este tipo de dato reconoce solamente dos valores: Verdadero (True) y Falso (False).
La primera letra tiene que ir en mayúscula (True, False).

Cadena de Texto (str)
Este tipo de datos se corresponde con una cadena de caracteres o conjunto de caracteres.
En **python** las cadenas se encierran en comillas simples o dobles. Este tipo de datos se corresponde con una cadena de caracteres o conjunto de caracteres.

Numero Complejo (complex)
Este tipo de datos se corresponde con un numero complejo en su forma binomial $(a + jb)$.

Ejemplos de tipos de datos en Python

Tipos de Datos	Ejemplos	Python Type
Números Enteros	1, 2, 0, -1, -14	int
Números Reales	1.34 3.14 -0.0022	float
Números Complejos	(2+j5), (6-j9)	complex
Lógicos	True, False	bool
Cadenas	“Hola como estas”	str

Variables

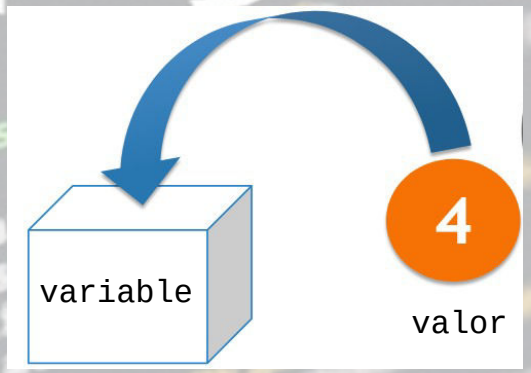
Las variables se pueden entender como contenedores en las que se guardan los datos, **pero en Python las variables son "etiquetas" que permiten hacer referencia a los datos.**

Solo podemos guardar un solo dato o valor en una variable.

las variables pueden ir cambiando su valor a lo largo de la ejecución del programa.

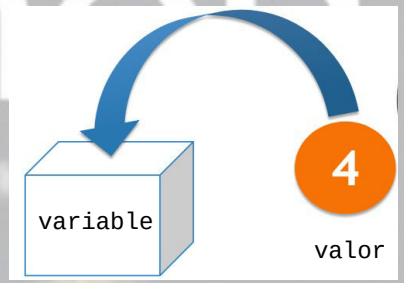
Para almacenar un dato o valor en una variable usamos el

operador de asignación " = "



Ejemplo de variables

```
apellido='ballesteros'
nombre='cristian'
edad=25
#letra='a'
precio=389.99
habilitar = True
```



Constantes

Las constantes son contenedores que guardan un dato o valor que no puede cambiar a lo largo del programa. En todos los lenguajes podrás ver el tema de constantes pero en **python no hay constantes, así que sólo queda declarar una variable** con un nombre característico en mayúscula, asignarle un valor y nunca modificarlo.

Por convencion en Python las constantes se declaran con mayusculas.

PI = 3.1416

E = 2.7172

Q = 1.602 *10^{-19}

Escritura de Datos: print(arg1,arg2,..,..)

Luego de aprender a crear tus primeras variables y cargarlas con datos. Lo primero que haremos es imprimirlas para ver si el contenido guardado es el deseado, al proceso de querer mostrar algo por consola lo llamamos **escritura de datos.**
Para la escritura de datos usaremos **print()**
Puedes imprimir cualquier tipo de dato, no es necesario que todos sean del mismo tipo de dato.

Importante: para mostrar más de un valor a la vez usaremos coma.

Escritura de Datos

```
#variables
apellido='ballesteros'
nombre='cristian'
edad=25
letra='a'
precio=389.99
habilitar = True

#imprimir o mostrar
print(apellido)
print(edad)
print(habilitar)
print('El apellido es:', apellido, ' el nombre es:', nombre )
print(edad , ' año de edad')
print('el valor de la letra es:',letra)
```

```
ballesteros
25
True
El apellido es: ballesteros  el nombre es: cristian
25  año de edad
el valor de la letra es: a
```

Actividad:

1) Utilizando la consola de Python escribamos estas lineas de programa.
2) Creamos un script con el IDLE del Python y lo ejecutamos.
3) Creamos un archivo Colab y escribimos el codigo y lo ejecutamos.
4) Abrimos el IDE Thonny y hacemos lo mismo.

Escritura de Datos

```python
#variables
apellido='ballesteros'
nombre='cristian'
edad=25
letra='a'
precio=389.99
habilitar = True

#imprimir o mostrar
print(apellido)
print(edad)
print(habilitar)
print('El apellido es:', apellido, ' el nombre es:', nombre )
print(edad , ' año de edad')
print('el valor de la letra es:',letra)
```

```
ballesteros
25
True
El apellido es: ballesteros  el nombre es: cristian
25  año de edad
el valor de la letra es: a
```

 # Codigo:

```
# variables
apellido='ballesteros'
nombre='cristian'
edad=25
letra='a'
precio=389.99
habilitar = True

# imprimir o mostrar
print(apellido)
print(nombre)
print(edad)
print(letra)
print(precio)
print(habilitar)
print("Mi nombre es: " + nombre + "; mi apellido es: " + apellido + " y mi edad es: " + str(edad))
#Otra forma:
print("Mi nombre es:",nombre, "; mi apellido es:", apellido, "y mi edad es:", str(edad))
print(f"Mi nombre es: {nombre} ; mi apellido es: {apellido } y mi edad es: {edad}")
```

Concatenación de str: “+”

Para concatenar variables o datos usaremos el operador **“ + ” que representa una unión.**

Unir variables o datos de tipo str no generará problemas. Pero intentar unir un str y un int nos dará un error. Esto también puede pasar con cualquier otro tipo de dato que no sea str.
Veamos unos ejemplos:

```
apellido='ballesteros'
nombre='cristian'
edad=25
#concatener un str y un str
print(' me llamo:'+ nombre + ', '+apellido)
#concatener un str y un int
print('me llamo '+ nombre + ' y tengo '+edad + ' años')
```

```
 me llamo:cristian, ballesteros
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-f8852fef3603> in <module>()
      4 edad=25
      5 print(' me llamo:'+ nombre + ', '+apellido)
----> 6 print('me llamo '+ nombre + ' y tengo '+edad + 'años')

TypeError: can only concatenate str (not "int") to str

SEARCH STACK OVERFLOW
```

Concatenación de str:

Para solucionar el error **TypeError: can only concatenate str (not "int") to str.**
La solución es fácil usaremos la función **str()** para poder convertir un tipo de datos que no sea cadena. A esta conversión se la llama **cast** o **casting,** es decir convertir un tipo de dato a otro.

```
apellido='ballesteros'
nombre='cristian'
edad=25
#concatener un str y un str
print(' me llamo:'+ nombre + ', '+apellido)
#concatener un str y un int
print('me llamo '+ nombre + ' y tengo '+ str(edad) + ' años')
```

```
 me llamo:cristian, ballesteros
me llamo cristian y tengo 25años
```

f-Strings en Python

En Python, una cadena de texto normalmente se escribe entre **comillas dobles (" ") o comillas simples (' ')**. Para crear **f-strings,** solo tienes que agregar la letra **f o F** mayúscula antes de las comillas.

Por ejemplo, "esto" es una cadena de texto normal y f"esto" es una f-string.

Cómo imprimir variables usando f-strings en Python Si quieres mostrar variables utilizando **f-strings**, solo tienes especificar el nombre de las variables entre llaves **{}**. Y al ejecutar tu código, todos los nombres de las variables serán reemplazados con sus respectivos valores.

En caso de tener múltiples variables en tu cadena de texto, cada variable necesita llaves propias **{}**

```
f"Esta es una f-string {nombre_var} y {nombre_var2}."
```

```
lenguaje = "Python"
escuela = "1000programadoresSalteños"
print(f"Estoy aprendiendo {lenguaje} en {escuela}.")
```

```
Estoy aprendiendo Python en 1000programadoresSalteños.
```

Lectura de Datos: input()

La lectura de datos consiste ya no en guardar datos manualmente en las variables, sino que las variables guardan datos o valores diferentes solicitados a un usuario. Usuario es la persona que usa una App, una Web, o programa y nos brinda los datos que necesitamos. La lectura de datos hace que nuestro programa sea más dinámico ya que los usuarios piensan diferente e ingresan datos diferentes en nuestras variables.

Para la lectura de datos ingresados por consola usaremos input()

Ejemplo de lectura de Datos

```
#lectura de datos por consola
apellido=input('Ingrese apellido:')
nombre=input('Ingrese nombre:')
letra=input('Ingrese letra:')

#imprimir variables en python
print(apellido)
print(nombre)
print(letra)
```

Actividad:

1) Utilizando la consola de Python escribamos estas lineas de programa.
2) Creamos un script con el IDLE del Python y lo ejecutamos.
3) Creamos un archivo Colab y escribimos el codigo y lo ejecutamos.
4) Abrimos el IDE Thonny y hacemos lo mismo.

Lectura de Datos + Conversiones

Tener en Cuenta que todo lo que se solicita al usuario es de tipo str es un texto. Por lo cual si quieres la edad de una persona tendría que ser un int lo cual nos llevaría a un casting o conversiones de tipos de datos.

Para convertir la entrada de datos debemos **usar int (entrada) o float (entrada) o bool (entrada) la entrada son los input** que pedíamos en el ejemplo anterior.

Los tipos de entrada que sean de tipo **str no es necesario convertirlos**.

Lectura de Datos + Conversiones

Las variables **float** tienen muchos dígitos en la parte decimal, por lo cual si quieres que aparezcan con 2 o 3 dígitos luego de la coma puedes usar **f"{variable: .cantidadDeDecimalesf}"** o round **(variable,cantidadDeDecimales)**

```
precio=float(input('Ingrese precio:'))
print('el precio ingresado es:',  f"{precio:.3f}" )
print('el precio ingresado es:',  round(precio,3) )
```

```
Ingrese precio:132.56446
el precio ingresado es: 132.564
el precio ingresado es: 132.564
```

Lectura de Datos + Conversiones

```python
#lectura de datos por consola
apellido=input('Ingrese apellido:')
nombre=input('Ingrese nombre:')
letra=input('Ingrese letra:')
edad=int(input('Ingrese edad:'))
precio=float(input('Ingrese precio:'))
habilitar=bool(input('Ingrese habilitar:'))

#imprimir variables en python
print(apellido)
print(nombre)
print(letra)
print(edad)

print('el precio es:', precio)
print('el precio ingresado es:',  f"{precio:.3f}" )
print('el precio ingresado es:',  round(precio,3) )
```

```
Ingrese apellido:ballesteros
Ingrese nombre:cristian
Ingrese letra:r
Ingrese edad:25
Ingrese precio:100
Ingrese habilitar:True
ballesteros
cristian
r
25
el precio es: 100.0
el precio ingresado es: 100.000
el precio ingresado es: 100.0
```

Comentarios en Python

Los comentarios en Python, al igual que sucede en otros lenguajes de programación, sirven para explicar a las personas que puedan leer el programa en el futuro, qué es lo que hace el programa, así como explicar algunas partes del código. Estos comentarios son ignorados por las computadoras cuando ejecutan el código. Escribir comentarios en Python, aunque requiere un esfuerzo, es una buena práctica y compensará con creces ese esfuerzo en el futuro.
En Python los comentarios se pueden poner de dos formas: Escribiendo el **símbolo #** delante de la línea de texto donde está el comentario.

Escribiendo triple comilla doble (""") al principio y al final del comentario (que puede ocupar más de una línea).

```
#aqui va tu comentario en la misma linea

"""Este es un comentario multilínea.
Podemos escribir tantas líneas queramos a modo de documentación."""
```

Operadores aritméticos

Los operadores aritméticos basicos y que ya vienen incluidos en la librería basica del Python son los siguientes:

OPERADOR	DESCRIPCIÓN	USO
+	Realiza Adición entre los operandos	12 + 3 = 15
-	Realiza Substracción entre los operandos	12 - 3 = 9
*	Realiza Multiplicación entre los operandos	12 * 3 = 36
/	Realiza División entre los operandos	12 / 3 = 4
%	Realiza un módulo entre los operandos	16 % 3 = 1
**	Realiza la potencia de los operandos	12 ** 3 = 1728
//	Realiza la división con resultado de número entero	18 // 5 = 3

Orden de operaciones en Python

Precedence Level	Operator	Explanation
1 (highest)	( )	Parentheses
2	**	Exponentiation
3	-a, +a	Negative, positive argument
4	* , / , // , %, @	Multiplication, division, floor division, modulus, at
5	+, -	Addition, subtraction
6	< , <=, >, >=, ==, !=	Less than, less than or equal, greater, greater or equal, equal, not equal
7	not	Boolean Not
8	and	Boolean And
9	or	Boolean Or

Ejemplo de uso de Operadores aritméticos

```python
a=int(input('Ingrese primer valor:'))
b=int(input('Ingrese segundo valor:'))

suma=a+b
resta=a-b
producto=a*b
division=a/b
resto=a%b
potencia=a**b
divisionEntera=a//b

print(suma)
print(resta)
print(producto)
print(division)
print(resto)
print(potencia)
print(divisionEntera)

#ejemplo de la raiz cuadrada usando math sqrt
import math
print(math.sqrt(16))
```

```
Ingrese primer valor:8
Ingrese segundo valor:2
10
6
16
4.0
0
64
4
4.0
```

Condiciones

Una condición es aquella que nos da como resultado **True o False.** Si la condición es True entonces la afirmación es verdadera, en cambio si nos devuelve False la condición no se cumple.

Para armar condiciones **simples** usamos **operadores relacionales** y para armar condiciones un poco más complejas usaremos los operadores relacionales y los **operadores lógicos para unir más de una condición.**

Ejemplo de algunas condiciones que podemos probar en un print por el momento:

```
#mas ejemplos
print(2>3)
print(2==2)
print(24<50)
a=10
print(a>10)
```

```
False
True
True
False
```

Operadores relacionales

Con los operadores relacionales podremos comparar al menos **dos elementos,** y dará un resultado final True o False. Nos permiten armar condicionales simples. Los elementos a comparar deben ser **homogéneos** o sea del mismo tipo.

OPERADOR	DESCRIPCIÓN	USO
>	Devuelve True si el operador de la izquierda es mayor que el operador de la derecha	12 > 3 devuelve True
<	Devuelve True si el operador de la derecha es mayor que el operador de la izquierda	12 < 3 devuelve False
==	Devuelve True si ambos operandos son iguales	12 == 3 devuelve False
>=	Devuelve True si el operador de la izquierda es mayor o igual que el operador de la derecha	12 >= 3 devuelve True
<=	Devuelve True si el operador de la derecha es mayor o igual que el operador de la izquierda	12 <= 3 devuelve False
!=	Devuelve True si ambos operandos no son iguales	12 != 3 devuelve True

Ejemplo de uso de Operadores relacionales

```
#tambien la variable a y b se pueden pedir por consola con un input
a=10
b=20
#comparamos dos variables de tipo int
print(a==b)
print(a!=b)
print(a>b)
print(a<b)
print(a>=b)
print(a<=b)
#tambien puedes comparar una variable con un valor
print(a>10)
#tambien puedes comparar string
apellido='ballesteros'
apellido2='balles'
print(apellido==apellido2)
```

```
False
True
False
True
False
True
```

Operadores lógicos

Nos permiten **unir más de una condición en una sola condición más grande**, por lo cual las condiciones ya no son simples sino un poco más complejas.

Operador	ejemplo
not	not condición
and	condicion1 and condicion2
or	condicion1 or condicion2

TABLA de verdad: Operadores Lógicos

El operador lógico NOT

x	resultado
true	false
false	true

El operador lógico OR

x	y	resultado
true	true	true
true	false	true
false	true	true
false	false	false

El operador lógico AND

x	y	resultado
true	true	true
true	false	false
false	true	false
false	false	false

```
# agregar la tabla de verdad
a=10
b=20
c=10
print( not(a==b) )
print( a==b and b==c)
print( a==c or c>b)
```

```
True
False
True
```

Actualizar variables

Actualizar una variable consiste en utilizar el valor de la variable y **aumentar, restar, multiplicar** o realizar cualquier operación y guardar el resultado obtenido sobre la misma variable. La actualización se utiliza mucho en el tema de contadores y acumuladores que veremos más adelante.

Ejemplo de actualización:

```
#la variable vale 100
precio=100
print('precio actual:', precio)
#pasan 2 dias y vale 50 pesos mas
#la variable se esta actualizando en 50 mas
precio=precio+50
print('precio actual despues de sumar 50:', precio)
```

```
precio actual: 100
precio actual despues de sumar 50: 150
```

Ejemplo de actualización de variables

```
#ejemplo contar personas
contadorPersonas=0
print('contador:',contadorPersonas)
contadorPersonas= contadorPersonas+1
print('contador de personas incrementado en 1:', contadorPersonas)
```

```
contador: 0
contador de personas incrementado en 1: 1
```

```
#sumar sueldo
suma=0
print('suma:',suma)
#incremento mi sueldo de 10mil
suma= suma+10000
print('suma:', suma)
```

```
suma: 0
suma: 10000
```

Operadores de asignación

OPERADOR	DESCRIPCIÓN
=	a = 5. El valor 5 es asignado a la variable a
+=	a += 5 es equivalente a a = a + 5
-=	a -= 5 es equivalente a a = a - 5
*=	a *= 3 es equivalente a a = a * 3
/=	a /= 3 es equivalente a a = a / 3
%=	a %= 3 es equivalente a a = a % 3
**=	a **= 3 es equivalente a a = a ** 3
//=	a //= 3 es equivalente a a = a // 3

Luego de haber aprendido la actualización de variables, ahora podremos entender mejor los operadores de asignación, los operadores de asignación son **actualizaciones que realizamos a las variables pero de una manera más corta en cuanto a código.**

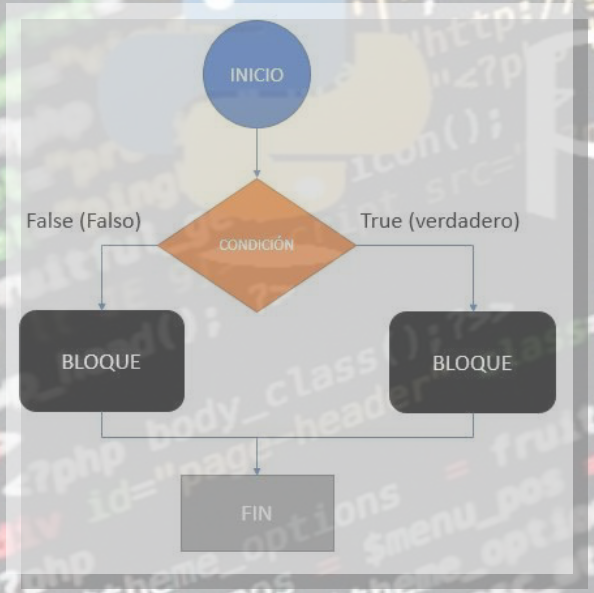
Unidad 04.1

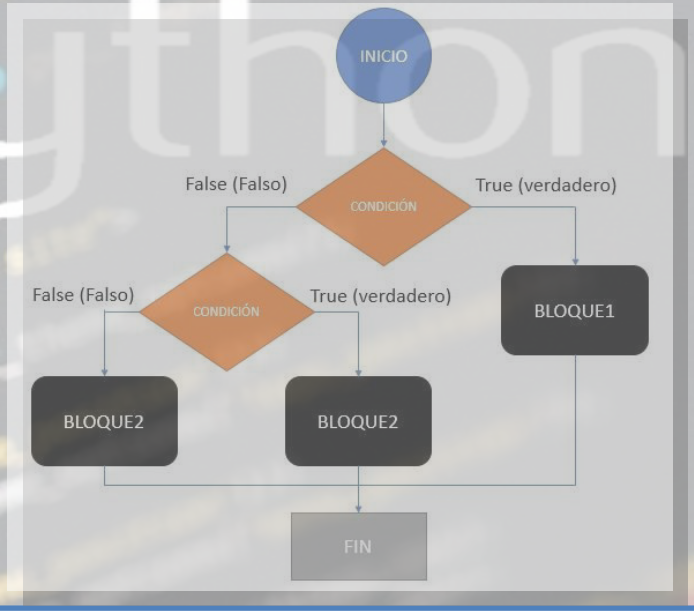
Estructuras de control:

- **Selectivas / simples.**
- **Dobles y múltiples if-else elif.**

Estructuras selectivas o condicionales

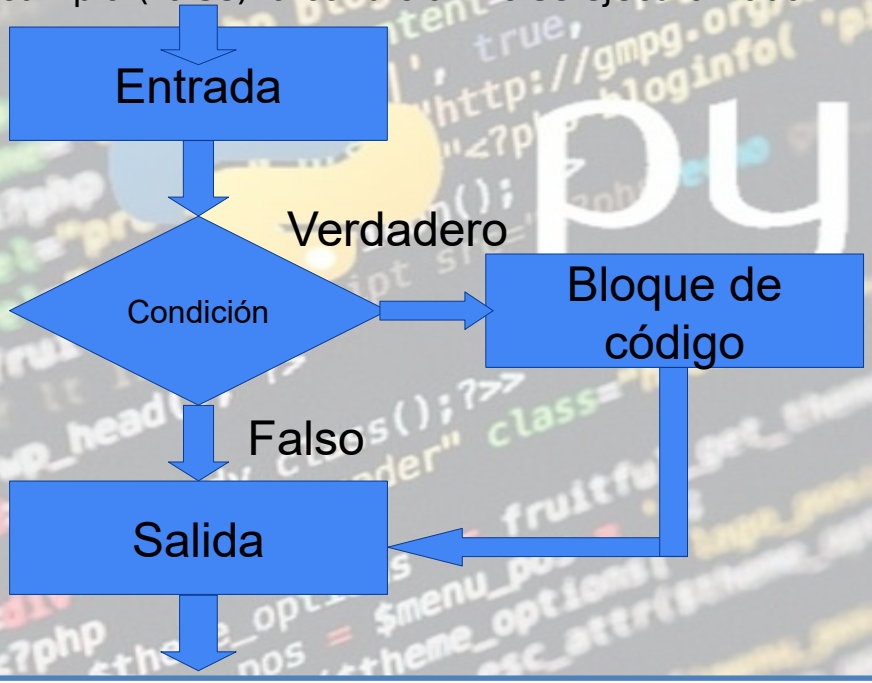
En un programa es frecuente que haya que tomar decisiones de acuerdo al valor que tome una proposicion. Por lo general usaremos una estructura que nos permita armar caminos de acuerdo a una condición y en cada uno de estos caminoa habra un bloque de codigo que se ejecutara de acuerdo a esa condicion que se haya cumplido.

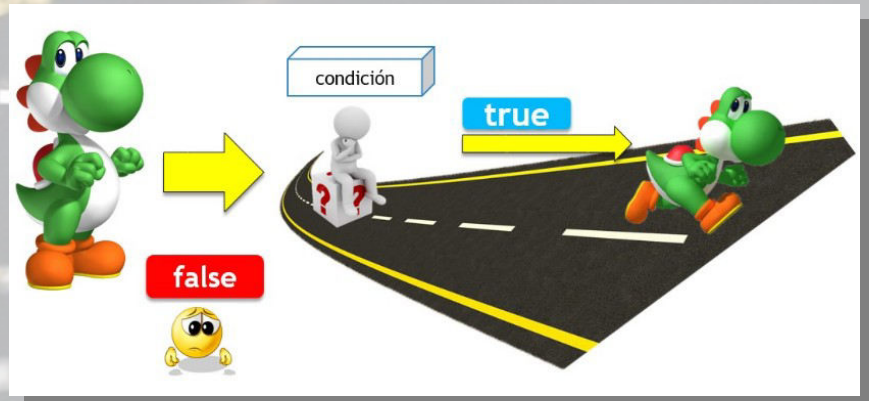




Estructuras selectiva simple if

Consiste en crear una condición: si la condición es verdadera o se cumple (True) se entra en el camino y se ejecutan las acciones para esa condicion. En caso que sea falsa o que no se cumpla (False) la condición no se ejecuta nada.





En Python se escribiría así:

```
if condicion:
    acciones
```

Estructuras selectiva simple if

```
edad = int(input('Ingrese su edad:'))
if edad>=18:
    print('es mayor de edad')
```

```
Ingrese su edad:20
es mayor de edad
```

```
numero = int(input('Ingrese su numero:'))
if numero>0:
    print('es positivo')
```

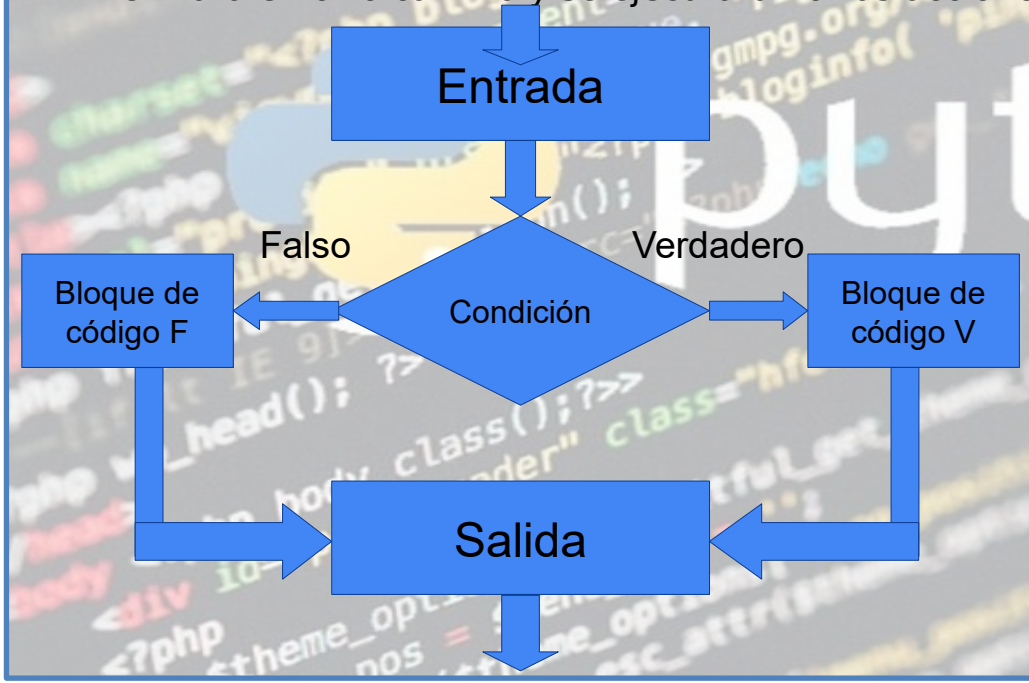
```
Ingrese su numero:5
es positivo
```

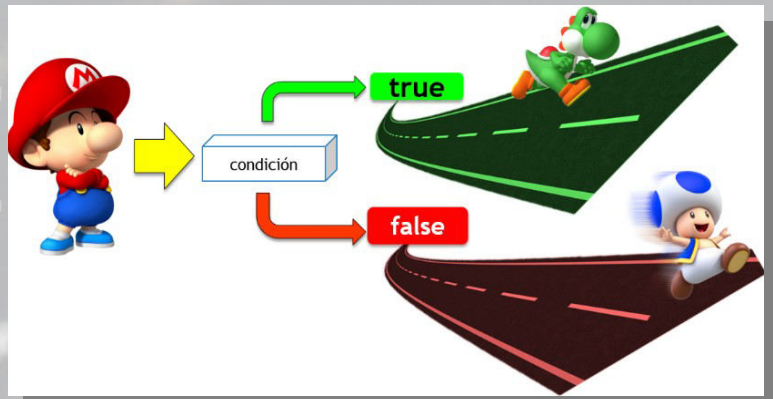
```
numero = int(input('Ingrese su numero:'))
if numero>=1 and numero<=10:
    print('esta entre 1 y 10')
```

```
Ingrese su numero:7
esta entre 1 y 10
```

Estructuras selectiva doble if/else

En este caso si la condición es verdadera o se cumple (True) se entra en un camino y se ejecutan las acciones para esa condicion. En caso que sea falsa o que no se cumpla (False) entrara en otro camino y se ejecutaran otras acciones para esa otra condicion.





En Python se escribiría así:

```
if condicion:
    acciones_camino_if
else:
    accciones_camino_else
```

Estructuras selectiva doble if

```
edad = int(input('Ingrese su edad:'))
if edad>=18:
    print('es mayor de edad')
else:
    print('no es mayor de edad')
```

```
Ingrese su edad:19
es mayor de edad
```

```
numero = int(input('Ingrese su numero:'))
if numero>=1 and numero<=10:
    print('esta entre 1 y 10')
else:
    print('no esta entre 1 y 10')
```

```
Ingrese su numero:12
no esta entre 1 y 10
```

```
numero = int(input('Ingrese su numero:'))
if numero%2==0:
    print('es par')
else:
    print('no es par o es impar')
```

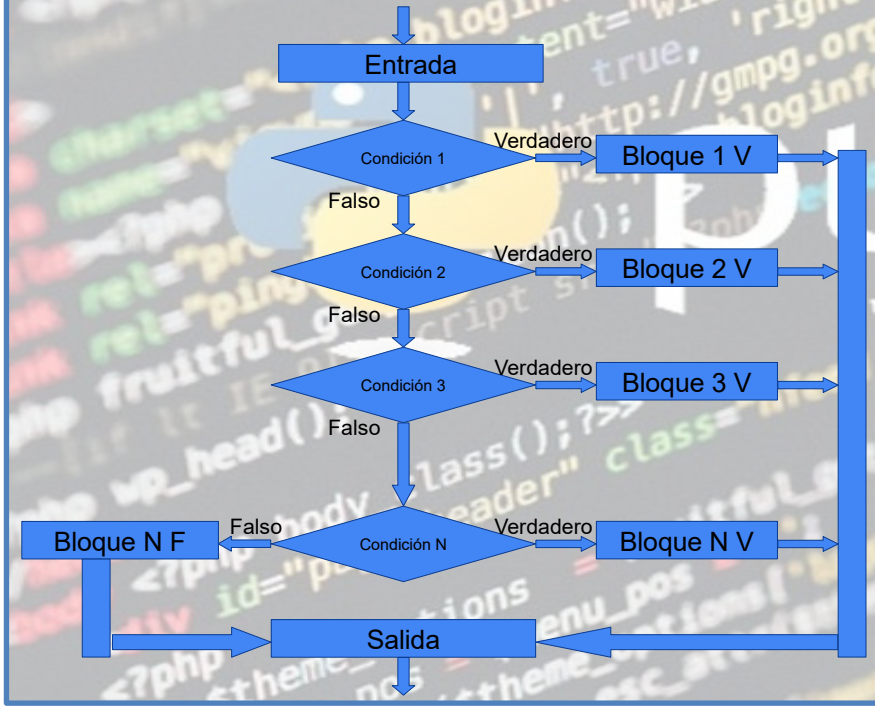
```
Ingrese su numero:7
no es par o es impar
```

```
apellido=input('Ingrese apellido:')
if apellido == 'mamani':
    print('es mamani')
else:
    print('es un apellido que no es mamani')
```

```
Ingrese apellido:mendez
es un apellido que no es mamani
```

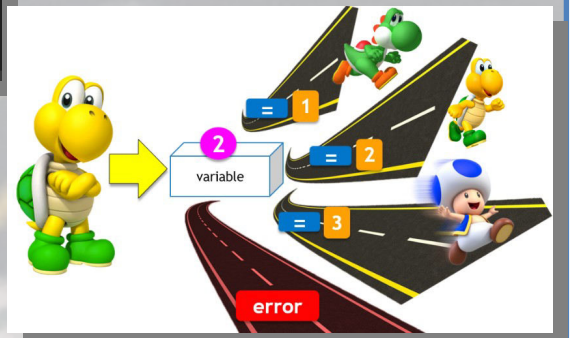
Estructuras selectiva múltiple if/elif/else

Se trata de una estructura selectiva múltiple o una estructura de if anidados donde podremos armar más de dos caminos. Y dentro de un if o else se pueden poner otro if.



En Python se escribiría así:

```
if condicion1:
  #acciones condicion1
elif condicion2:
  #acciones condicion2
elif condicion3
  #acciones condicion3
else:
  #acciones else
```



Estructuras selectiva múltiple if/elif/else

```
 #en este ejemplo dentro de un else hay otro if
numero = int(input('Ingrese un numero:'))
if numero>0:
    print('es positivo')
else:
    #neg cero
    if numero==0:
        #es cero
        print('es cero')
    else:
        #negativo
        print('negativo')
```

```
Ingrese un numero:-5
negativo
```

```
#en este ejemplo el dentro de un if hay otro if
numero = int(input('Ingrese un numero:'))
if numero>=1 and numero<=3 :
    print('esta entre 1y 3')
    if numero==1:
      print('uno')
    else:
        if numero==2:
            print('dos')
        else:
            if numero==3:
                print('tres')

else:
    print('No es 1, 2 ni 3')
```

```
Ingrese un numero:2
esta entre 1y 3
dos
```

Estructuras selectiva múltiple if/elif/else

```
#este es una selectiva multiple
opcion=input('Ingrese una opcion:')
if opcion=='a':
    print('aqui va la suma')
elif opcion=='b':
      print('aqui va la resta')
elif opcion=='c':
      print('aqui va la factorial')
elif opcion=='d':
      print('fin de la app')
else:
      print('opcion no valida')
```

```
Ingrese una opcion:c
aqui va la factorial
```

```
# Calculadora
operador1 = int(input('Ingrese el primer valor: '))
operador2 = int(input('Ingrese el segundo valor: '))
operacion = input('Selecione operacion (+, -, *, /): ')
resultado = 0
if (operacion == '+') :
    resultado = (operador1 + operador2)
elif (operacion == '-') :
    resultado = (operador1 - operador2)
elif (operacion == '*') :
    resultado = (operador1 * operador2)
elif (operacion == '/') :
    if (operador2 == 0) :
        resultado = 'Error no se puede dividir por cero'
    else :
        resultado = (operador1 / operador2)
else :
    print('No se ingreso una operacion valida')
print('Resultado = ',resultado)
```

Indentación en la estructura de control

Python utiliza la ***indentación (o tabulación)*** para delimitar la estructura permitiendo establecer bloques de código. No existen comandos para finalizar las líneas ni llaves con las que delimitar el código. **Los únicos delimitadores existentes son los dos puntos (:) y la indentación del código.**

La indentación es un espacio en blanco o sangrado con lo que se indicaría el inicio del bloque, si en las posteriores líneas no introdujeramos el sangrado, significa el final de dicho bloque de código, con lo cual para finalizar un bloque de código, sólo tenemos que dejar de introducir el sangrado, no tenemos que usar ninguna llave ni símbolo.

Esta regla se aplica para las estructuras de control IF – IF ELSE – IL ELIF

indentación

```
edad = int(input('Ingrese su edad:'))
if edad>=18:
    print('es mayor de edad')
else:
    print('no es mayor de edad')
```

```
Ingrese su edad:19
es mayor de edad
```

bloque de código

Unidad 04.2

Estructuras repetitivas: for y while. Contadores y acumuladores.

Estructuras repetitivas

Permiten repetir una serie de acciones, según se cumpla una condición.
Cada repetición se llama **bucle o ciclo**.
Es una secuencia de instrucciones de código que se ejecuta repetidas veces. Hasta cumplir una condición.

Range

range: representa una secuencia de números enteros. Se lo puede utilizar pasándole un único valor de **stop**. Genera una secuencia de números enteros consecutivos entre 0 y stop-1.

```
#range solo
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si se utiliza pasándole dos argumentos, **start** y **stop**, genera una secuencia de números enteros consecutivos entre start y stop-1.

```
#genera numero desde 1 hasta 9
list(range(1,10))
```

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

La inclusión de un tercer argumento **step** (salto) fuerza a que la secuencia de números se genere con un salto entre un número y el siguiente.

```
#generar numeros 1, 3, 5, 7, 9 numeros impares entre 1 y 10
list(range(1,10,2))
```

```
[1, 3, 5, 7, 9]
```

Ciclo for

La sentencia for repite un bloque de código para un conjunto de valores.

La sentencia for en Python itera sobre los ítems de un conjunto de datos (cd)
Su sintaxis es:

```
for <nombre> in <cd>:
        <bloque de código>
```

<nombre> es una variable, que creamos nosotros,
que va tomando todos los valores de <cd>, uno por uno,
en cada iteración. La idea es usar el valor que toma
la variable dentro del bloque de código.
La cantidad de iteraciones viene dada por la cantidad
de elementos del conjunto de datos.

Ciclo for

ejemplos:

```
#mostrar un mensaje 3 veces
for i in range(1,4):
    print('hola python')
```

```
hola python
hola python
hola python
```

```
#podesmos generar numeros
# desde 1 hasta 10 pero
# que i se incremente de a 2
for i in range(1,11,2):
    print(i)
```

```
1
3
5
7
9
```

Ciclo for

¿Puedo utilizar un for dentro de un for?

Respuesta:

si -> Esto se llama **CICLO ANIDADO**

```
N = int(input('ingrese la cantidad de alumnos que desea:'))
for i in range(1,N+1):
    print('alumno:', i)
    cantidadNotas = int(input('ingrese la cantidad de notas:'))
    for i in range(1,cantidadNotas+1):
       nota=int(input('ingrese nota:'))
       print('su nota es:', nota)
```

```
ingrese la cantidad de alumnos que desea:2
alumno: 1
ingrese la cantidad de notas:3
ingrese nota:10
su nota es: 10
ingrese nota:8
su nota es: 8
ingrese nota:9
su nota es: 9
alumno: 2
ingrese la cantidad de notas:1
ingrese nota:4
```

Contadores

Es la acción de contar

Los contadores son siempre variables enteras.

Se inicializan en cero generalmente.

Usualmente se incrementa de a 1 pero también puede haber un incremento mayor.

```
#1 - se crea una variable entera en cero
cont = 0
#2 - se utiliza cuando deseamos contar
cont = cont +1
#3 - se muestra al final o cuando se desee mostrar el valor del contador
print(cont)
```

Contadores

Ejemplo de contador

```python
#un solo contador para numeros positivos
contPos=0
N = int(input('ingrese la cantidad de repeticiones que desea:'))
for i in range(1,N+1):
    num = int(input('ingrese un numero:'))
    if num>0:
       print(num,' es positivo')
       contPos=contPos+1
    else:
      if num<0:
          print(num,' es negativo')
      else:
          print(num, ' es cero')

print('la cantidad de positivos es:',contPos)
```

```
ingrese la cantidad de repeticiones que desea:3
ingrese un numero:10
10  es positivo
ingrese un numero:20
20  es positivo
ingrese un numero:-7
-7  es negativo
la cantidad de positivos es: 2
```

Acumuladores

Un acumulador es una variable, no necesariamente entera, pero sí numérica.

Su objetivo es “acumular”, es decir: acopiar, almacenar, añadir un cierto valor.

La diferencia con una variable cualquiera es que el acumulador agrega un nuevo valor al que ya tiene

```
#1 - se crea una variable en cero
suma = 0
#2 - se utiliza cuando deseamos almacenar algun valor en el acumulador
#valor seria la variable o el dato que queremos que se sume al acumulador
suma = suma + valor
#3 - se muestra al final o cuando se desee mostrar el valor del acumulador
print(suma)
```

Acumuladores

Ejemplo de un acumulador

```
#puedo crear mas de un acumulador para diferente cuentas
sumaPos=0
N = int(input('ingrese la cantidad de repeticiones que desea:'))
for i in range(1,N+1):
    num = int(input('ingrese un numero:'))
    if num>0:
       print(num,' es positivo')
       sumaPos=sumaPos+num #se esta acumulando el num
    else:
      if num<0:
          print(num,' es negativo')
      else:
          print(num, ' es cero')

print('la suma de positivos es:',sumaPos)
```

```
ingrese la cantidad de repeticiones que desea:3
ingrese un numero:10
10  es positivo
ingrese un numero:20
20  es positivo
ingrese un numero:-5
-5  es negativo
la suma de positivos es: 30
```

Ciclo while

La sentencia o ciclo while es una sentencia de control de flujo que se utiliza para ejecutar un bloque de código de forma repetitiva mientras se cumpla una condición determinada.

La estructura de esta sentencia es la siguiente:

```
while <condición>:
    <bloque de código>
```

De la misma forma que la sentencia if, si la condición evalúa a True se ejecutará el bloque de código. Sino, se ignorará. La diferencia está en que al terminar de ejecutar el bloque, automáticamente se volverá a comprobar la condición, si continúa siendo verdadera se ejecutará el bloque nuevamente y así sucesivamente. Cuando la condición evalúe a False, se ignorará el bloque de código y se continuará la ejecución normal.
Cada ejecución del bloque de código se denomina iteración.

Ciclo while

Ejemplo de while

```
#repetir accciones a pedido del operador
respuesta = 'si'
while respuesta =='si':
    print('hola python  - while')
    respuesta = input('desea continuar si o no ? ')
```

```
hola python  - while
desea continuar si o no ? si
hola python  - while
desea continuar si o no ? no
```

Unidad 05

Tipos de datos complejos(listas, tuplas, diccionarios, conjunto).

Métodos de listas

Manejo de cadenas de caracteres.

Mutabilidad en Python

- **Mutables:** estructuras de datos que permiten modificar su contenido.
  - Listas
  - Diccionarios
  - Sets
- **Inmutables:** estructuras de datos que no permiten modificar su contenido.
  - Números
  - Cadenas
  - Tuplas

Mutabilidad en Python

Las variables son nombres, no lugares. Cuando asignamos un valor a una variable, lo que realmente está ocurriendo es que se hace **apuntar** el nombre de la variable a una zona de memoria en la que se representa el objeto (con su valor).
Si realizamos la asignación de una variable a un valor lo que está ocurriendo es que el nombre de la variable es una **referencia** al valor, no el valor en sí mismo:

```
>>> a = 5
```

Si ahora «copiamos» el valor de a en otra variable b se podría esperar que hubiera otro espacio en memoria para dicho valor, pero como ya hemos dicho, son referencias a memoria:

```
>>> b = a
```

La función id() nos permite conocer la dirección de memoria de un objeto en Python. A través de ella podemos comprobar que los dos objetos que hemos creado «apuntan» a la misma zona de memoria:

```
>>> id(a)
140717968126888
>>> id(b)
140717968126888
```

Se dice, por ejemplo, que un entero es **inmutable** ya que a la hora de modificar su valor obtenemos una nueva zona de memoria, o lo que es lo mismo, un nuevo objeto:

```
>>> a = 7
>>> id(a)
140717968126952
```

Mutabilidad en Python

La característica de que los nombres de variables sean referencias a objetos en memoria es la que hace posible diferenciar entre **objetos mutables e inmutables:**

Inmutable	Mutable
bool	list
int	set
float	dict
str	
tuple	

Importante: El hecho de que un tipo de datos sea inmutable significa que no podemos modificar su valor «in-situ», pero siempre podremos asignarle un nuevo valor (hacerlo apuntar a otra zona de memoria).

Mutabilidad en Python - Ejemplo

- Estructuras de datos mutables: lista (list())

```
Lista=[1,2,3,50]
print(lista)                    Salida: [1,2,3,50]
print(id(lista))                Salida: 1808543149568

lista.append(5)
print(lista)                    Salida: [1,2,3,50,5]
print(id(lista))                Salida: 1808543149568
```

Nota: id() es una función que se que se encuentra en la librería estándar de Python. Esta función retorna la «identidad» de una “variable”. Este consiste en un entero que está garantizado que es único y constante para esta “variable” durante toda su existencia.

Mutabilidad en Python - Ejemplo

- Estructuras de datos mutables: diccionario (dict())

```
miDic = {1: 'uno', 2: 'dos', 3: 'tres'}
print(miDic)
Salida: {1: 'uno', 2: 'dos', 3: 'tres'}
print(id(miDic))          Salida: 1808531933504

miDic[4] = 'cuatro'   # Agregamos un par (clave,valor)
print(miDic)
Salida: {1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro'}
print(id(miDic))          Salida: 1808531933504
```

Mutabilidad en Python - Ejemplo

- Estructuras de datos mutables: conjunto (set())

```
miConj = {'a', 'b', 'c', 4, 9}
print(miConj)
Salida: {'a', 'b', 'c', 4, 9}
print(id(miConj))          Salida: 1808552023584

miConj.add('w')          # Agregamos un elemento nuevo
print(miConj)
Salida: {'w', 4, 'c', 9, 'a', 'b'}
print(id(miConj))          Salida: 1808552023584
```

Mutabilidad en Python - Ejemplo

- Estructuras de datos NO mutables (Inmutables): número (int, float, bool)

```
miNumero = 3.14
print(miNumero)
Salida: 3.14
print(id(miNumero))          Salida: 1808551827504

miNumero = 2.71
print(miNumero)
Salida: 2.71
print(id(miNumero))          Salida: 1808531235760
```

Nota: los id() o la «identidad» de una “variable” inmutable son diferentes cuando se cambia su valor.

Mutabilidad en Python - Ejemplo

- Estructuras de datos NO mutables (Inmutables): string

```
miString = 'Python es interpretado'
print(miString)
Salida: Python es interpretado
print(id(miString))          Salida: 1808552076320

miString = 'Python no es compilado'
print(miString)
Salida: Python no es compilado
print(id(miString))          Salida: 1808552077840
```

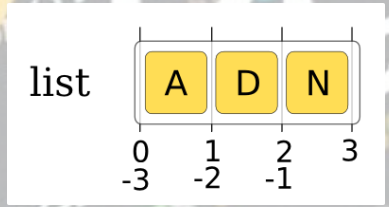
Tipos de Datos Estructurados

Los datos estructurados en Python son aquellos que **permiten que una variable pueda almacenar "más de un valor"**. Es decir, combinamos los tipos básicos de formas más complejas.

- Listas: list()
- Tuplas: tuple()
- Diccionarios: dict()
- Conjuntos: set()
- Ficheros: files

Listas (list)

Las listas son tipos de datos usados para almacenar listas de elementos.
Las listas permiten almacenar objetos mediante un orden definido y con posibilidad de duplicados. Las listas son **estructuras de datos mutables**, lo que significa que podemos añadir, eliminar o modificar sus elementos.
Se accede a cada elemento de la lista por medio de un índice. El primer elemento de la lista inicia en 0.



Las listas son dinámicas:

- No es necesario indicarles un tamaño fijo.
- Puede contener distintos tipos de datos incluso listas, tuplas y otros:
  lista = [10, 20, 30, 40, 50, 60, 70, [1, 2, 3], miTupla]

Listas (list) - Definición

Las listas se pueden construir de diferentes formas:

- Usando corchetes, separando los elementos con comas:

```
lista=['a', 'b', 'c', 3, 7.33, True, 'hola']
lista=['a']
lista=[ ]
```

- Usando el constructor list():

```
lista=list()
lista=list('hola')
print(lista) →→→ Salida: ['h', 'o', 'l', 'a']
```

Listas (list) - Composición

Cantidad de elementos: 5

```
Lista = ["Hola", 12, 5.0, True, False]

Indices (+)  [  0     1   2     3     4  ]
Indices (-)  [ -5    -4  -3    -2    -1  ]
```

Índices o posiciones: son valores numéricos enteros que permiten recorrer los elementos de una lista, y pueden ser positivos y negativos.

Listas (list) - Manejo de Índices

```
lista=["Hola", 12, 5.0, True, False]
```

```
print(lista[0])          salida:"Hola"
print(lista[-5])         salida:"Hola"
print(lista[4])          salida:False
print(lista[-1])         salida:False
```

SI apuntamos a un indice fuera de rango dara error:

```
print(lista[10])         salida:IndexError:
                                list index out of range
```

Listas (list) – Uso del lazo for ... in .. list

Dos maneras de recorrer una lista con for:

```
lista=["Hola", 12, 5.0, True, False]
```

```
for elemento in lista:
    print(elemento)
```

Salida:

"Hola"
12
5.0
True
False

```
for i in range(len(lista)):
    print(lista[i])
```

Salida:

"Hola"
12
5.0
True
False

Listas (list) – Métodos (Operaciones)

- **lista.append(**x**)**: Agrega el elemento **x** al final de la lista.
- **lista.insert(**i, x**)**: Inserta el elemento **x** en una posición **i** de la lista.
- **lista.remove(**x**)**: Quita el primer ítem de la lista cuyo valor sea **x**.
- **lista.index(**x**)**: Retorna el índice (posición) del elemento x.
- **lista.reverse()**: Invierte los elementos de la lista in situ y la guarda.
- **lista.copy()**: retorna una copia de todos los elemento de la lista.
- **lista.count(x)**: retorna el numero de ocurrencias del valor x.
- **lista.clear()**: remueve todos los elemento de la lista.
- **lista.pop(i)**: remueve el elemento con el indice i y retorna su valor.

Listas (list) – Slicing (“rebanar”)

El **slicing** es una característica de las estructuras de datos que nos permite acceder a una **“porción”** definida de ellas: ***lista[inicio,fin,paso]***

```
lista = [50, 70, 30, 20, 90, 10, 50]
print(lista[1:5])

Salida: [70, 30, 20, 90]
```

El **slicing** recorre los elementos desde la posición inicial hasta N – 1. Si no se indica el paso por defecto es 1.

Posición inicial 1

Posición final 4 ya que (5-1 = 4)

Listas (list) – Slicing (“rebanar”)

El slicing también nos permite realizar saltos

```
lista = [50, 70, 30, 20, 90, 10, 50]

#posiciones          0   1   2   3   4   5   6
#pos de salto slicing  3

print(lista[1:5:3])

Salida: [70, 90]
```

¿Qué muestra `print(lista[1:5:6)`?

```
Salida: [70]
```

Listas (list) – Slicing (“rebanar”)

Ejercicio: invertir la lista usando **SLICING**

```
lista = [10, 20, 30, 40, 50, 60, 70]
```

Resultado esperado:

```
lista = [70, 60, 50, 40, 30, 20, 10]
```

```
Solucion: lista[::-1]
```
NOTA: la lista NO se modifica.

***Ayuda: hacer saltos de (-1)**

Tuplas (tuple)

- Una **tupla** es un contenedor de elementos entre paréntesis separados por comas.
- **Las tuplas son inmutables.**
- Normalmente contienen una secuencia heterogénea de elementos que son accedidos por medio de un índice.
- Pueden contener cualquier tipo de datos: números, caracteres, cadenas, booleanos, listas, tuplas.

```
miTupla = (1, 2.4, 'hola', True, 'g', lista2, ('a','b'))
print(miTupla)
```

Salida:

```
(1, 2.4, 'hola', True, 'g', [1, 2, 3], ('a', 'b'))
```

Tuplas (tuple) - Definición

Las tuplas se pueden construir de diferentes formas:

- Definiendo una secuencia de elementos separados por comas.

```
tupla1 = 1,2,3
```

- Usando el constructor **tuple**()

```
tupla2 = tuple("hola")
ó
tupla2 = tuple('hola')
guarda: ('h', 'o', 'l', 'a')
```

- Imprimiendo un elemento indicando el indice:

```
print(tupla2[2])
salida: l
```

Tuplas (tuple) - Métodos

tupla=1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6

tupla.count(valor) retorna las veces que aparece valor

```
tupla.count(1)      retorna 3
tupla.count(4)      retorna 1
```

tupla.index(valor) retorna el primer índice en que aparece valor.

```
tupla.index(3)      retorna 2
tupla.count(5)      retorna 10
```

Conjuntos (set)

El tipo de dato ***conjunto*** o ***set*** es un contenedor encerrado por llaves {} **no ordenado** de **distintos elementos,** no admite elementos repetidos, o sea que si intentamos agregar un elemento repetido este sera ignorado y solo guardara uno de ellos.

```
miConj = {'a', 'b', 'c', 'b', 'a', 'd'}
print(miConj)                Salida: {'a', 'b', 'd', 'c'}
```

Como es un contenedor sin orden, los conjuntos no registran ni la posición ni el orden de inserción de los elementos. Por lo tanto, este tipo de dato no soporta indexado, ni operaciones de rebanadas (Slicing), ni otras capacidades propias de listas.

Conjuntos (set) - Definición

Los conjuntos (***Sets***) se pueden construir de diferentes formas:

- Usando una lista de elementos separados por coma entre llaves: {'jack', 'sjoerd'}

- Usando el constructor **set()** enviando como argumento una lista, una tupla, una cadena:

```
set(['a', 'b', 'foo'])
set('foobar')
```

Conjuntos (set) - Ejemplos

```
conjunto={'a', "hola", 3, 4, True}
conjunto
{True, 'hola', 3, 4, 'a'}

lista=["hola", 2, 4, True, 'a', 'b', 'c']
lista
['hola', 2, 4, True, 'a', 'b', 'c']

conjunto1=set(lista)
conjunto1
{True, 2, 4, 'c', 'b', 'hola', 'a'}
```

```
tupla=["hola", 2, 4, True, 'a', 'b', 'c']
tupla
['hola', 2, 4, True, 'a', 'b', 'c']

conjunto2=set(tupla)
conjunto2
{True, 2, 4, 'c', 'b', 'hola', 'a'}
```

Conjuntos (set) - Operaciones

- **Cardinalidad**: Retorna el número de elementos en el conjunto.
  len(conjunto**)**

- **Pertenencia**: Comprueba que el elemento x está incluido en conjunto.
  x **in** conjunto
  Si uno de los elementos del conjunto es True, al hacer 1 in conjunto dará True aunque 1 no forme parte del set, y si hay algún False al hacer 0 in conjunto dará True aunque no haya ningún 0.
  Ejemplo:
  conjunto={'a', "hola", 3, 4, True} → 1 in conjunto retornara True
  conjunto={'a', "hola", 3, 4, False} → 0 in conjunto retornara True

- **No Pertenencia**: Comprueba que el elemento x NO está incluido en conjunto.
  x **not in** conjunto

Conjuntos (set) - Operaciones

- **Agregar un elemento:** Añade al conjunto el elemento ***elem***.
  - conjunto**.add(‘*elem*’)**
- **Eliminar un elemento:** Elimina del conjunto el elemento ***elem*** si está incluido.
  - conjunto**.discard(‘*elem*’)**

Diccionarios (dict)

- Un diccionario puede verse como un contenedor de pares ***clave:valor*** con el requerimiento de que las claves sean únicas (dentro de un diccionario).

- Los diccionarios son indexados por claves, es decir, los diccionarios asocian ***claves*** a los ***valores*** y de esta forma se tiene acceso a los ***valores*** almacenados mediante las ***claves***.

Diccionarios (dict)

Los diccionarios se pueden construir de diferentes formas:

- La más simple es encerrar una secuencia de pares clave: valor separados por comas entre llaves {}:

- **`d = {'Clave1': 'Valor1',1: 'hola', 89: 'Pythonista', 'c': 27}`**

- Usando el constructor de la clase dict():

```
d2 = dict(clave='valor', dos=2, tres=3)
d1 = {'uno': 1, 'dos': 2, 'tres': 3}
d2 = dict({'uno': 1, 'dos': 2, 'tres': 3})
d3 = dict([('uno', 1), ('dos', 2), ('tres', 3)])
```

Diccionarios (dict) - Operaciones

Acceder a un elemento de un diccionario.

El acceso se realiza mediante indexación de la clave.
Para ello, simplemente encierra entre corchetes la clave del elemento **miDic[clave]**.
En caso de que la clave no exista, se lanzará la excepción KeyError.

```
miDic = {'uno': 1, 'dos': 2, 'tres': 3}
print(miDic['dos'])
```

Salida: 2

Diccionarios (dict) - Operaciones

Acceder a un elemento de un diccionario.

Usando el método ***get(clave[, valor por defecto])***.
Este método devuelve el valor correspondiente a la clave. En caso de que la clave no exista no lanza ningún error, sino que devuelve el segundo argumento valor por defecto. Si no se proporciona este argumento, se devuelve el valor None.

```
d = {'uno': 1, 'dos': 2, 'tres': 3}
print(d.get('uno'))
Salida: 1
```

Diccionarios (dict) - Operaciones

Recorrer clave del diccionario

```
diccionario = {
  'clave 1': 'valor 1',
  'clave 2': 'valor 2',
  'clave 3': 'valor 3'
}
for clave in diccionario.keys():
  print(clave)
```

Diccionarios (dict) - Operaciones

Recorrer valores del diccionario

```
diccionario = {
   'clave 1': 'valor 1',
   'clave 2': 'valor 2',
   'clave 3': 'valor 3'
}
for valor in diccionario.values():
   print(valor)
```

Diccionarios (dict) - Operaciones

Recorrer items del diccionario

```
diccionario = {
   'clave 1': 'valor 1',
   'clave 2': 'valor 2',
   'clave 3': 'valor 3'
}
for clave, valor in diccionario.items():
    print(valor)
    print(valor)
```

Diccionarios (dict) - Ejemplos

Ejemplo 1: Diseñe un diccionario que describa cuáles días de la semana son laborables y cuáles no.

Definición del diccionario:

```
dias_semana = dict()
dias_semana = {
  "Lun" : "Trabajo",
  "Mar" : "Trabajo",
  "Mie" : "Trabajo",
  "Jue" : "Trabajo",
  "Vie" : "Trabajo",
  "Sab" : "Fiesta",
  "Dom" : "Fiesta"
}
```

Diccionarios (dict) - Ejemplos

Ejemplo 2: Con el diccionario diseñado en el ejemplo anterior. Muestra los días laborables.

```
for k,v in dias_semana.items():
  if (v == "Trabajo"):
    print(k)
```

```
Salida:
Lun
Mar
Mie
Jue
Vie
```

dias_semana.items()
representa la lista de todas tuplas formadas por los pares clave:valor

V == ”Trabajo”
separa el valor deseado

print(k)
imprime la clave de las tuplas que cumplen la condición

Diccionarios (dict) - Ejemplos

Ejemplo 3: Con el diccionario diseñado en el ejemplo anterior, muestre los días laborables en una lista.

```
for k,v in dias_semana.items():
  if (v == "Trabajo"):
    lista.append(k)
print(k)
```

Salida:

```
['Lun', 'Mar', 'Mie', 'Jue', 'Vie']
```

lista= list()
inicializa una lista vacía

lista.append(k)
agrega elementos a la lista, en este caso, los valores de clave del diccionario que cumplen la condición

print(lista)
imprime la lista de claves creada

str - Manejo de caracteres

Los tipos de datos **str** incluyen variados métodos para analizar, transformar, separar y unir el contenido de las cadenas de caracteres.

- **upper():** Devuelve la cadena con todos sus caracteres a mayúscula.

```
cadena = "Hola Mundo"
print(cadena.upper())      Salida: "HOLA MUNDO"
```

- **lower():** Devuelve la cadena con todos sus caracteres a minúscula.

```
cadena = "Hola Mundo"
print(cadena.lower())      Salida: "hola mundo"
```

str - Manejo de caracteres

- **capitalize():** Devuelve la cadena con su primer carácter en mayúscula.

```
cadena = "hola Mundo"
print(cadena.capitalize())      Salida: "Hola mundo"
```

- **split():** Separa la cadena en subcadenas a partir de sus espacios y devuelve una lista.

```
cadena = "Hola Mundo"
print(cadena.split())      Salida: ["hola", "mundo"]
```

- **title():** Devuelve la cadena con el primer carácter de cada palabra en mayúscula.

```
cadena = "hola mundo"
print(cadena.title())      Salida: "Hola Mundo"
```

str - Manejo de caracteres

- **count():** Devuelve una cuenta de las veces que aparece una subcadena en la cadena.

```
cadena = "abcd abef abgh"
print(cadena.count('ab')) Salida: 3
```

- **find():** Devuelve el índice en el que aparece la subcadena (-1 si no aparece).

```
cadena = "Hola Mundo"
print(cadena.find())      Salida: 7
```

- **join():** Une todos los caracteres de una cadena utilizando un carácter de unión.

```
cadena1 = "hola mundo"
cadena2 = "####"
print(cadena1.join(cadena2))
Salida: #Hola Mundo#Hola Mundo#Hola Mundo#
```

str - Manejo de caracteres

- **replace():** Reemplaza una subcadena de una cadena por otra y la devuelve.

```
cadena1 = "hola mundo"
cadena2 = "##"
print(cadena.replace(' ', cadena2))        Salida: Hola##Mundo
```

- **Slicing de str** Devuelve una subcadena entre extremos y pasos indicados:

subcadena = cadena[inicio:fin:salto]

Inicio: posición inicial desde donde se recorta la subcadena.

fin: posición final (menos 1) hasta donde se recorta la subcadena.

salto: salto entre posiciones sucesivas.

str - Manejo de caracteres

Ejemplo:

cadena1 = ‘metereologia’

subcadena1 = cadena[1:10:3]

print(subcadena1)

Salida: erl

cadena2 = ‘entretenimientos’

subcadena2 = cadena[-2:-14:-4]

print(subcadena2)

Salida: oie

Unidad 06

Funciones.
Parámetros.
Ámbitos de las variables.
ARGS vs KWARGS.
Librerías de terceros.

Modularidad - Concepto

Para resolver problemas complejos y/o de gran tamaño es conveniente aplicar una técnica de diseño de algoritmos. Una de las más utilizadas es la conocida como **“Divide y Vencerás”**. Esta técnica consiste en descomponer el problema en problemas más pequeños (subproblemas), hasta que el problema original quede reducido a un conjunto de actividades básicas que no se puede descomponer o que no es conveniente descomponer.

Modularidad - Concepto

La solución de un subproblema se la denomina **módulo**.

Utilizando esta técnica la solución de un problema se expresa como un programa que quedará formado por una serie de módulos, donde cada módulo realiza una tarea concreta de la tarea total.

Este diseño produce beneficios como:

- **producir programas más fáciles de escribir**
- **más fáciles de mantener**
- **se pueden realizar pruebas independientes.**

Funciones - Concepto

Python define un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones, se denominan ***nativas o predefinidas o integradas***. Ya hemos utilizado algunas de ellas, como la función **len()**, que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario, una cadena o un conjunto. También hemos visto la función **print()**, que muestra por consola un texto. Sin embargo, como programadores, podemos definir nuestras propias funciones para estructurar el código de manera que sea más legible y para reutilizar aquellas partes que se repiten a lo largo de una aplicación. Esto es una tarea fundamental a medida que va creciendo el número de líneas de un programa.

El diseño de funciones parte de los siguientes principios:

- **El principio de reusabilidad:** si tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Nos evitaría tener código repetido y modificarlo sería más fácil, ya que bastaría con cambiar la función una vez.
- **Y el principio de modularidad:** en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen fragmentos en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer.

Funciones - Concepto

Una función es un grupo de instrucciones que resuelven un problema muy concreto.

Dividir y organizar el código en partes más sencillas y mantenibles

principio de modularidad

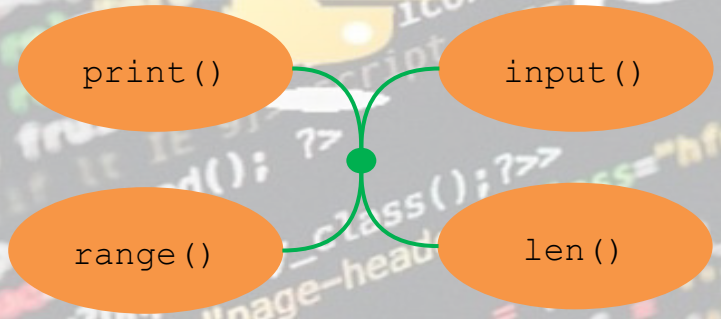
Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

principio de reusabilidad

Funciones - Tipos

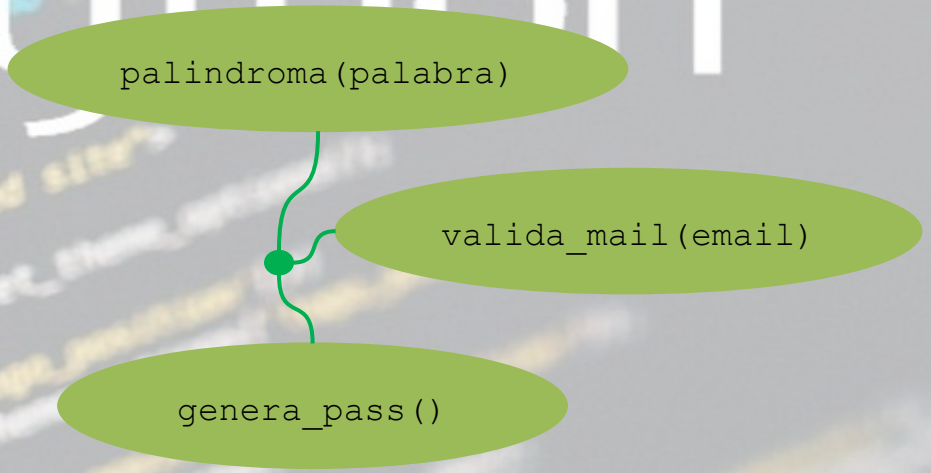
Funciones integradas

conjunto de funciones que provee el lenguaje



Funciones definidas por el programador

conjunto de funciones que deben definirse antes de ser usadas



Funciones de Primer Nivel

Python nos brinda un conjunto de funciones integradas.

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Funciones de Primer Nivel

Ejemplos:

```
num = 5
print(type(num))
print(len([1,2,3]))
print(round(3.141516,2))
```

Salida:

```
<class 'int'>
3
3.14
```

```
num1 = 4
num2 = 7
print(num1 + num2)
print('Los numeros son el:', num1, 'y el:', num2)
print(round(num1/num2,3))
```

Salida:

```
11
Los numeros son el: 4 y el: 7
0.571
```

Funciones - Definidas

Estructura para definir una función propia en Python

```
def nombre (parametro1,parametro2…,parametroN):
    instruccion1
    instruccion2
        …
    instruccionM
    return valor
```

def: palabra reservada que se utiliza para definir una función.

parámetro: valores que necesita la función para poder realizar la tarea para la que fue definida. Una función puede no necesitar valores para realizar la tarea.

return: palabra reservada para devolver un resultado (su utilización es opcional)

Funciones - Definidas

Estructura para definir una función propia en Python

Para definir una función se utiliza la palabra reservada **def**. A continuación el nombre o identificador de la función que es el que se utiliza para invocarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. La cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (identado) que es el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado. En último lugar y de manera opcional, se añade la instrucción con la palabra reservada return para devolver un resultado.

Para usar o invocar a una función, simplemente hay que escribir su nombre pasando los argumentos necesarios según los parámetros que defina la función.

Por lo tanto, los identificadores que se colocan al DEFINIR la función se denominan PARÁMETROS y los identificadores que se utilizan en la INVOCACIÓN de la función se llaman ARGUMENTOS. Es decir que los argumentos son los valores que se actualizan en el cuerpo de la función a través de los parámetros. Veamos un ejemplo, donde simulamos una calculadora que realiza las operaciones básicas, el usuario ingresa dos operandos y puede elegir cuál operación realizar y además decidir si quiere realizar otro cálculo o no.

Funciones - Definidas

Ejemplo 1: definir una función llamada “contar_letras” que devolverá la cantidad de letras de una cadena omitiendo los espacios en blanco.

```
def contar_letras(palabra):
    contar = 0
    for i in range(len(palabra)):
        if palabra[i] != ' ':
            contar = contar + 1
    return contar
palabra = "Esta es una cadena de prueba"
nro_letras = contar_letras(palabra)
print(nro_letras)
# Nos muestra 23
```

Funciones - Definidas

Ejemplo 2: definir una función llamada “bolillero” que imprima 10 números aleatorios enteros comprendidos entre 1 y 100.

```
import random
def bolillero():
    for i in range(10):
        print(random.randint(1,100))
# Ejecutamos:
bolillero()
```

Funciones - Definidas

Ejemplo 3: diseñar una calculadora con las 4 operaciones basicas.

```python
# Sección 1 ################################################
def sumar(a,b):          #Definimos la función sumar
  x = a + b
  return x
def restar(a,b):             #Definimos la función restar
  x = a - b
  return x
def multiplicar(a,b):        #Definimos la función multiplicar
  x = a * b
  return x
def dividir(a,b):            #Definimos la función dividir
  x = a / b
  return x
# Sección 2 ################################################
operar = True
while operar:             #Creamos un ciclo
  a = int(input("Ingresa el primer numero: \n"))
  b = int(input("Ingresa el segundo numero: \n"))
  op = input ("""Elegir cálculo
        1- Sumar
        2- Restar
        3- Multiplicar
        4- Dividir \n""")
```

```python
# Sección 3 ################################################
  if op == '1':      #Llamamos a función sumar
    res = sumar(a,b)
    print("El resultado es: ",res)
  elif op == '2':        #Llamamos a función restar
    res = restar(a,b)
    print("El resultado es: ",res)
  elif op == '3':        #Llamamos a función multiplicar
    res = multiplicar(a,b)
    print("El resultado es: ",res)
  elif op == '4':        #Llamamos a función dividir
    res = dividir(a,b)
    print("El resultado es: ",res)
  else:
    print("""Número de opción erróneo""")
# Sección 4 ################################################
  s = input("Quieres hacer otra operación? S/N \n")
  if (s.upper() == 'N'):
    operar = False       # Para finalizar el ciclo
print("FIN de calculadora")     # Mensaje luego que corta el ciclo

```

Funciones - Definidas

Ejemplo 3:

En la sección 1 se definen las cuatro funciones correspondientes al cálculo que habrá disponible, pero aún no son llamadas, o sea no son utilizadas.

En la sección 2 se crea un bucle o ciclo (que se repite mientras el usuario ingresa una opción distinta a “N”). Se solicitan los valores para ser operados y la operación deseada.

En la sección 3 se determina cuál operación eligió el usuario y se invoca a una de las cuatro funciones diseñadas.

En la sección 4 se da la posibilidad al usuario de realizar otra operación con otros elementos. Si ingresa “N”, se detiene el ciclo y termina el programa, si ingresa cualquier otro carácter el ciclo vuelve a iterar.

Argumentos de entrada

Empecemos por una función sencilla sin parámetros de entrada ni salida.

```
1  import random
2  def bolillero():
3      for i in range(10):
4          print(random.randint(1,100), end=' ')
5  # Ejecutamos:
6  bolillero()
```

```
63 8 73 99 63 51 84 48 83 48
```

Definir una función llamada “bolillero” que imprime 10 números aleatorios enteros comprendidos entre 1 y 100.
En la linea 6 invocamos a la funcion bolillero pero no pasamos ningun parametro.
Tampoco la funcion tiene un return, directamente desde la funcion se imprimen los numeros aleatorios.

Argumentos de entrada

Ahora vemos ejemplos de funciónes con parámetros de entrada.

```
1  def di_hola(nombre):        #Definición de la función
2      print("Hola", nombre)
3
4
5  di_hola("Elena")            #Invocación de la función
6
```

```
Hola Elena
```

```
1  def resta(a, b):            #Definición de la función
2      return a-b
3
4
5  print(resta(5, 3))      #Invocación 1 de la función
6  print(resta(3, 5))      #Invocación 2 de la función
7
```

```
2
-2
```

Por defecto, los valores de los argumentos se asignan a los parámetros en el mismo orden en el que se los pasa al llamar a la función. Luego veremos que esto puede cambiar. Veamos un ejemplo de en qué afecta el orden de los parámetros.

Funciones - Parámetros

- **Parámetros por valor:** lo que hace es copiar el valor de las variables en los respectivos parámetros. Una modificación del valor del parámetro, no afecta a la variable externa correspondiente.

- **Parámetros por referencia**: Lo que hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Una modificación del valor en el parámetro afectará a la variable externa correspondiente.

En Python, no aplica este concepto de forma literal, Ya que no realiza copias, sino crea NUEVAS VARIABLES o hace referencia a la varible si esta es mutable

Funciones - Parámetros

```python
def suma(a,b):
    a = a + 1
    b = b + 1
    print(f"Valores de a y b dentro de la función: {a}, {b}")
    print(f"Id de las variables a y b dentro de la función: {id(a)}, {id(b)}")
    return a+b

a = 5
b = 6
print(f"Valores de a y b fuera de la función: {a}, {b}")
print(f"Id de las variables a y b fuera de la función: {id(a)}, {id(b)}")
print(f"resultado de la función: {suma(a,b)}")
print("\nVemos como quedan las variables después de ejecutar la función")
print(f"Valores de a y b función de la función despues de sumar: {a}, {b}")
print(f"Id de las variables a y b fuera de la función despues de sumar: {id(a)}, {id(b)}")

#Van a notar que los identificadores son distintos,
#no hace una copia del valor sino que crea un identificador nuevo apuntando a ese valor
```

```
Valores de a y b fuera de la función: 5, 6
Id de las variables a y b fuera de la función: 140490681866672, 140490681866704
Valores de a y b dentro de la función: 6, 7
Id de las variables a y b dentro de la función: 140490681866704, 140490681866736
resultado de la función: 13

Vemos como quedan las variables después de ejecutar la función
Valores de a y b función de la función despues de sumar: 5, 6
Id de las variables a y b fuera de la función despues de sumar: 140490681866672, 140490681866704
```

Python define variables nuevas dentro de una función, cuando el tipo de dato no es mutable

Funciones - Parámetros

Presentamos estos conceptos por que existen en otro lenguaje de programación como **JAVA**, pero Python no lo aplica de forma literal.

```
def suma(a, b):
    a = a + 1
    b = b + 1
    print(f"Valores de a y b dentro de la función: {a}, {b}")
    print(f"Id de las variables a y b dentro de la función: {id(a)}, {id(b)}")
    return (a + b)

a = 5
b = 6
print(f"Valores de a y b fuera de la función: {a}, {b}")
print(f"Id de las variables a y b fuera de la función: {id(a)}, {id(b)}")
print(f"Resultado de la función: {suma(a, b)}")
print("\nVemos como quedan las variables después de ejecutar la función")
print(f"Valores de a y b función de la función despues de sumar: {a}, {b}")
print(f"Id de las variables a y b fuera de la función despues de sumar: {id(a)}, {id(b)}")

#Van a notar que los identificadores son distintos,
#no hace una copia del valor sino que crea un identificador nuevo apuntando a ese valor
```

Funciones - Parámetros

Presentamos estos conceptos por que existen en otro lenguaje de programación como **JAVA**, pero Python no lo aplica de forma literal.

 # Resultados:

Valores de a y b fuera de la función: 5, 6
Identificador de las variables a y b fuera de la función: 140729307820968, 140729307821000
Valores de a y b dentro de la funsión: 6, 7
Identificador de las variable a y b dentro de la función: 140729307821000, 140729307821032
resultado de la función: 13

Vemos como quedan las variables después de ejecutar la función
Valores de a y b función de la función despues de sumar: 5, 6
Identificador de las variables a y b fuera de la función despues de sumar: 140729307820968, 140729307821000

Python define variables nuevas dentro de una función, cuando el tipo de dato no es mutable

Funciones – Parámetros (ejemplo)

```
def suma(lista2):
    lista2[0]=lista2[0]+1
    lista2[1]=lista2[1]+1
    print(f"Valores de lista2 dentro de la función: {lista2}")
    print(f"Identificadores de lista2 dentro de la función: {id(lista2)}")
    return (lista2[0]+lista2[1])

lista=[5,8]
print(f"Valores de la lista fuera de la función: {lista}")
print(f"Identificadores de la lista fuera de la función: {id(lista)}")
print(f"resultado de la función: {suma(lista)}")
print("\nVemos como quedan las variables después de ejecutar la función")
print(f"Valores de la lista fuera de la función: {lista}")
print(f"Identificadores de las variables lista fuera de la función: {id(lista)}")
```

```
Valores de la lista fuera de la función: [5, 6]
Identificadores de la lista fuera de la función: 1271218716608
Valores de lista2 dentro de la función: [6, 7]
Identificadores de lista2 dentro de la función: 1271218716608
resultado de la función: 13
Vemos como quedan las variables después de ejecutar la función
Valores de la lista fuera de la función: [6, 7]
Identificadores de las variables lista fuera de la función: 1271218716608
```

Ámbito de una variable

- Es el contexto en el que existe una variable (existe en dicho ámbito a partir de que se crea y deja de existir cuando desaparece su ámbito).
- Las variables son accesibles sólo desde su propio ámbito (LOCALES), pero con alguna excepción, como veremos más adelante.

Los principales tipos de ámbitos en Python son dos:

- **Ámbito local:** corresponde con el ámbito de una función, que existe desde que se invoca a una función hasta que termina su ejecución. En un programa, el ámbito local corresponde con las líneas de código de una función. Dicho ámbito se crea cada vez que se invoca a la función. Cada función tiene su ámbito local. No se puede acceder a las variables de una función desde fuera de esa función o desde otra función.
- **Ámbito global:** corresponde con el ámbito que existe desde el comienzo de la ejecución de un programa. Todas las variables definidas fuera de cualquier función corresponden al ámbito global, que es accesible desde cualquier punto del programa, incluidas las funciones. Si desde un módulo A importamos un módulo B mediante un import, desde A podremos acceder a las variables globales de B.

Ámbito de una variable - Ambito local

Variables Locales: las variables locales son aquellas definidas dentro de una función. Los parámetros de una función también son considerados como variables locales.

- **Desde el cuerpo principal del programa no se puede acceder a las variables locales de ninguna función. Tampoco es posible acceder a las variables locales de una función desde otra.**

Ámbito de una variable - Ambito local

```
def saludar():
    saludo = '¡Hola!' #esta variable local pertenece al
                      #ámbito local de la función saludar
    print(saludo)

def sumar(sumando1, sumando2):    #sumando1, sumando2 y suma
    suma = sumando1 + sumando2    #son variables locales
    return suma                   #que corresponden al ámbito
                                   #local de la función sumar

saludar() # invoca a la funcion saludar()

print(sumar(4, 5)) # invoca a la funcion sumar()
print(saludo) #da error porque saludo no es visible fuera de la
funcion
```

Ámbito de una variable - Ambito local

El error NameError

Si intentamos acceder al valor de una variable local desde el cuerpo principal del programa o, en general, a una variable que no ha sido definida obtendremos un error típico de Python: NameError.

```
def saludar():
    saludo = '¡Hola!'
    print(saludo)

saludar() # invoca a la funcion saludar()
print(saludo) #da error porque saludo no es visible fuera de la funcion
```

```
¡Hola!
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-6cf96d515b5b> in <module>
      4
      5 saludar()
----> 6 print(saludo)

NameError: name 'saludo' is not defined
```

Ámbito de una variable - Ambito global

Variables Globales: las variables globales son aquellas definidas en el cuerpo principal del programa fuera de cualquier función.
Son accesibles desde cualquier punto del programa, incluso desde adentro de las funciones. También se puede acceder a las variables globales de otros programas o módulos importados.
Por esto decimos que las variables globales son aquellas de ámbito global.

```
def saludar():
    print(saludo)

saludo = '¡Hola, Mundo!' #variable global definida en el
                         #cuerpo principal del programa
saludar()
```

Ámbito de una variable - Ambito global

Cómo modificar una variable global desde una función

```
contador=10
def reiniciar(contador):
    print(f"Dentro de la función antes de poner en 0, contador={contador} y su id={id(contador)}")
    contador=0
    print(f"Dentro de la función después de poner en 0, contador={contador} y su id={id(contador)}")

print(f"Contador antes de la función es {contador} y su id={id(contador)}")

reiniciar(contador)

print(f"Contador después de la función es {contador} y su id={id(contador)}")
```

```
Contador antes de la función es 10 y su id=11126976
Dentro de la función antes de poner en 0, contador=10 y su id=11126976
Dentro de la función después de poner en 0, contador=0 y su id=11126656
Contador después de la función es 10 y su id=11126976
```

Ámbito de una variable - Ambito global

Al ejecutar el código lo esperable sería obtener por pantalla los valores 10 y 0, pero en su lugar tenemos lo siguiente:

- Primer print Contador antes es 10
- Segundo print Contador después es 10

Se debe a un mecanismo de protección para evitar modificar sin querer una variable global.

DEBEMOS TENER CUIDADO CON LOS TIPOS DE DATOS MUTABLES, YA QUE POR SU NATURALEZA ELLOS SI SE MODIFICAN.

Ámbito de una variable - Ambito global

Para tener acceso de modificación es necesario utilizar el ***modificador global***. Con esta declaración le estamos diciendo a Python que sabemos que vamos a utilizar una variable global y que queremos modificarla. Veamos el ejemplo corregido:

```
contador = 10

def reiniciar_contador():
        global contador
        contador = 0

print(f'Contador antes es {contador}')
print(f'Id antes es {id(contador)}')
reiniciar_contador()
print(f'Contador después es {contador}')
print(f'Id despues es {id(contador)}')
```

Salida:

```
Contador antes es 10
Id antes es 140273594067536
Contador después es 0
Id despues es 140273594067216
```

Funciones - Parámetros – Otros
ARGS - KWARGS

Python nos permite crear funciones que acepten un **número indefinido de parámetros** sin necesidad de que todos ellos aparezcan en la cabecera de la función. Los operadores * y ** son los que se utilizan para esta funcionalidad.

En realidad no es necesario usar los nombres ***args*** o ***kwargs***, ya que se trata de una mera convención entre programadores. Sin embargo lo que si es necesario es usar es el asterisco simple * o doble **. Es decir, se podría escribir ***variable*** y ****variables***.

Empecemos viendo el uso de *args.

Funciones - Parámetros – ARGS

El parámetro especial *args en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

- *El símbolo '*' indica el tipo parámetro.*
- *El nombre **args** se usa por convención.*
- *El parámetro recibe los argumentos como una **tupla**.*
- *Es un parámetro opcional.*
- *El número de argumentos al invocar a la función es variable.*
- *Son parámetros posicionales, es decir su valor depende de la posición en la que se pasen a la función.*

Funciones - ARGS - Ejemplos

Definición de la función

```
def fun(*valores):
    print(valores)
```

Invocación de la función

```
X=5
Y=8
fun(x, y)
```

(5, 8)

```
Z=4
fun(x, y, z)
```

(5, 8, 4)

Funciones - ARGS - Ejemplos

La siguiente función toma dos parámetros y devuelve la suma de los mismos:

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

Sum(2, 3) # Ejecutamos sum
Salida: 5

Pero, ¿si necesitamos sumar un valor más?
Sum(2, 3, 4) # Ejecutamos sum con tres valores
Salida: **nos da un error**

```
TypeError                                 Traceback (most recent call
last)
<ipython-input-12-fc2d57a9194a> in <module>()
----> 1 sum(2, 3, 4)

TypeError: sum() takes 2 positional arguments but 3 were given
```

Funciones - ARGS - Ejemplos

Por supuesto que esperábamos que la llamada falle. Una posible solución sería agregar parámetros, pero tampoco sabríamos cuántos. Esto se soluciona en Python con el uso de ***args** en la definición de esta función. De este modo, podemos pasar tantos argumentos como queramos.
Primero hay que rediseñar la **función sum** y con esa nueva implementación, podemos llamar a la función con cualquier número variable de valores dentro de una tupla: ***tupla_args(arg1,….,argn)***

```
def sum(*args):
    value = 0
    for n in args:
        value += n
    return value
```

#Invocaciones de la función
print(sum())
print(sum(4,5))
print(sum(4,5,24,11))
print(sum(1,2,3,4,5,6))

Salidas:
0
9
44
21

El comportamiento de la función siempre es el mismo, con independencia del número de argumentos que pasamos.

Con esto resolvemos nuestro problema inicial, en el que necesitábamos un número variable de argumentos.

Funciones - ARGS - Ejemplos

```
def test_var_args(f_arg, *argv):
    print("primer argumento normal:", f_arg)
    for arg in argv:
        print("argumentos de *argv:", arg)

test_var_args('python', 'foo', 'bar')
```

Y la salida que produce el código anterior al llamarlo con 3 parámetros es la siguiente:

```
primer argumento normal: python
argumentos de *argv: foo
argumentos de *argv: bar
```

Funciones - Parámetros - KWARGS

El parámetro especial **kwargs en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

- *El símbolo **'**'** indica el tipo parámetro*
- *El nombre kwargs se usa por convención.*
- *El parámetro recibe los argumentos como un **diccionario**.*
- *Al tratarse de un diccionario, el orden de los parámetros no importa.*
- *Los parámetros se asocian en función de las claves del diccionario.*

Funciones - KWARGS - Ejemplos

Definición de la función

```
def fun(**valores):
  print(valores)
```

Invocación de la función

```
fun(x=5, y=8)
```

```
{'x': 5, 'y': 8}
```

```
fun(x=5, y=8, z=4)
```

```
{'x': 5, 'y': 8, 'z': 4}
```

Funciones - KWARGS - Ejemplos

Veamos un ejemplo sencillo de cómo ** se asocia a un diccionario

```
def fun(**params):
    print(params)

fun(x=2)
fun(x=5, y=8)
fun(x=5, y=8, z=4)
```

Salidas:

```
{'x': 2}
{'x': 5, 'y': 8}
{'x': 5, 'y': 8, 'z': 4}
```

Es decir, dentro de la función no solo tenemos acceso a la variable como con * args, sino que también tenemos acceso a un nombre o key asociado.

Funciones - KWARGS - Ejemplos

También es posible que la cabecera de una función utilice uno o varios argumentos posicionales, seguidos del operador * o **. Esto nos proporciona bastante flexibilidad a la hora de invocar a una función.

En el siguiente ejemplo tenemos 2 parámetros “fijos” y uno que permite agregar tantos como necesitemos.

```
def print_record(nombre, apellido, **rec):
    print("Nombre: ", nombre)
    print("Apellidos:", apellido)
    for k in rec:
        print("{0}: {1}".format(k, rec[k]))

print_record("Juan", "Coll", edad=43, localidad="Madrid")
```

Salidas:

```
Nombre:  Juan
Apellidos: Coll
edad: 43
localidad: Madrid
```

Funciones - KWARGS - Ejemplos

En resumen:

- Utiliza ***args** para pasar de forma opcional a una función un número variable de argumentos posicionales.
- El parámetro ***args** recibe los argumentos como una **tupla**.
- Emplea ****kwargs** para pasar de forma opcional a una función un número variable de argumentos con nombre.
- El parámetro ****kwargs** recibe los argumentos como un **diccionario**.
- Por último, utilizar ***args** y ****kwargs** ahorra tiempo y esfuerzo en cuanto a análisis e implementación dado que flexibiliza el diseño de funciones, pero su uso indiscriminado puede originar resultados inesperados cuya revisión y arreglo nos podría llevar más tiempo aún que el diseño estratégico original.

Funciones - De librería o biblioteca

En programación una librería se define como un conjunto de implementaciones funcionales.

Así podemos reconocer una librería como un conjunto de módulos cuya agrupación tiene una finalidad específica y que también puede ser invocada, tal como un módulo. Pero no es un módulo, sino un conjunto de ellos con una estructura determinada para lograr una finalidad. Entonces se trata de organización y capacidad a la hora de desarrollar una aplicación. Esta puede hacer con miles de líneas en un solo archivo, o con 5 archivos de cientos de líneas; así surgen dos opciones. Hacerlo como un módulo aparte, o dentro del código del mismo programa; pero aquí surge un problema. Al incluir todo el código de lo que podrían ser diferentes módulos en el mismo archivo obtendremos como resultado un enredo de millones de líneas de código o quizás más en un solo file. Y donde obtengas una serie de errores te encontrarás más perdido en el espacio que esta nave.

Funciones - De librería o biblioteca

Y también si no dispones de las librerías, es como si no tuvieses la capacidad de lograr lo que quieres sin un esfuerzo mayor. Por ejemplo suponiendo que tu quieras hacer una calculadora en python, y la misma ya está programada lógicamente. Pero tu quieres brindarle al usuario un entorno gráfico. Sin librería, estarías perdido.

Salvo que tu programes desde 0 todo el entorno gráfico de tu programa!. Pero para eso existen las librerías (conjuntos de módulos) y módulos “prefabricados del lenguaje o incorporados por tí” por decir así que facilitaran el trabajo.

Bien si un módulo es como una caja de herramientas y nosotros vamos a hacer cálculos, necesitamos un módulo que posea estos accesorios y debemos incorporarlo a nuestro programa, vamos a importar el módulo Math usando la orden import. De esta manera:

Funciones - De librería o biblioteca

```
import math #Importamos el módulo math
x = int(input("Ingresa un numero \n"))
raiz = math.sqrt(x) #Utilizamos la función sqrt del módulo math
print (raiz)
```

Salida:

```
Ingresa un numero
4
2.0
```

Funciones - De librería o biblioteca

La sentencia **import** se utiliza para importar un módulo. Se puede usar cualquier archivo de código Python como un módulo ejecutando esta sentencia en otro archivo de código Python. La sentencia import tiene la siguiente sintaxis:

```
import os
import datetime
```

Cuando el interprete encuentra una sentencia import, este importa el módulo si el mismo esta presente en la ruta de búsqueda. Una ruta de búsqueda es una lista de directorios que el interprete busca antes de importar un módulo.

Librería datetime

Esta libreria permite crear objetos para manejar fechas y horas por ejemplo:

```python
from datetime import datetime

dt = datetime.now()    # Fecha y hora actual

print(dt)
print(dt.year)         # año
print(dt.month)        # mes
print(dt.day)          # día

print(dt.hour)         # hora
print(dt.minute)       # minutos
print(dt.second)       # segundos
print(dt.microsecond)  # microsegundos

print("{}:{}:{}".format(dt.hour, dt.minute, dt.second))
print("{}/{}/{}".format(dt.day, dt.month, dt.year))
```

```
2023-01-17 21:14:11.792970
2023
1
17
21
14
11
792970
21:14:11
17/1/2023
```

Librería datetime

Es posible crear un datetime manualmente pasando los parámetros (year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None). Sólo son obligatorios el año, el mes y el día.

```
1  from datetime import datetime
2
3  dt = datetime(2000,1,1)
4  print(dt)
```

```
2000-01-01 00:00:00
```

Librería Random

Este módulo contiene funciones para generar números aleatorios:

```python
import random

# Flotante aleatorio >= 0 y < 1.0
print(random.random())

# Flotante aleatorio >= 1 y <10.0
print(random.uniform(1,10))

# Entero aleatorio de 0 a 9, 10 excluído
print(random.randrange(10))

# Entero aleatorio de 0 a 100
print(random.randrange(0,101))

# Entero aleatorio de 0 a 100 cada 2 números, múltiples de 2
print(random.randrange(0,101,2))

# Entero aleatorio de 0 a 100 cada 5 números, múltiples de 5
print(random.randrange(0,101,5))
```

```
0.1270397042725383
9.913503207259332
4
5
48
70
```

Librerias TKinter

Tk es una herramienta para desarrollar aplicaciones de escritorio multiplataforma, esto es, aplicaciones nativas con una interfaz gráfica para sistemas operativos Windows, Linux, Mac y otros. Técnicamente, Tk es una biblioteca de código abierto escrita en C y desarrollada en sus orígenes para el lenguaje de programación Tcl; de ahí que usualmente nos refiramos a ella como Tcl/Tk. Desde sus primeras versiones Python incluye en su biblioteca o librería estándar el módulo tkinter, que permite interactuar con Tk para desarrollar aplicaciones de escritorio en Python. La curva de aprendizaje de Tk es relativamente pequeña si la comparamos con otras bibliotecas del rubro (como Qt), de modo que cualquier programador con una mínima base de Python puede comenzar rápidamente a crear aplicaciones gráficas

Funciones - Biblioteca - Otras

De acuerdo, con los objetivos de las librerías de Python existen diferentes clasificaciones. Estos son algunos tipos de librerías de Python:

- **Deep learning**: Están enfocadas, de cara a la predicción de datos; a través del Big Data.
- **Machine learning**: Estas librerías son útiles para el machine learning, ya que mejoran el proceso de información y la resolución de problemas de clasificación y el análisis de regresión de datos.
- **Visualización**: Sirven para entender y comprender los datos, de una forma más legible.

Funciones - Biblioteca - Otras

Las librerías más conocidas son:

- random
- math
- pandas
- numpy
- tk
- pyqt5

Pero hay muchas más que nos pueden ayudar a resolver de manera mas eficiente nuestro trabajo

Funciones - Biblioteca - Ejemplo

```
import random
print(random.choice(["rojo","azul","verde"]))
```

Salida: "rojo"

```
lista = [1,2,3,5]
random.shuffle(lista)
print(lista)
```

salida: [5,3,1,2] #cambia de forma aleatoria
#los elementos de la lista

```
import math
print(math.sqrt(4))
```

salida: 2

Unidad 07 – Introduccion a la Programacion Orientada a Objetos

Parte 1

- Introduccion
- Definiciones.
- Abstracción.
- Clase y objetos.
- Atributos y métodos.
- Herencia
- Polimorfismo

Introducción

- Los problemas suelen tener varias soluciones posibles.
- En programación existen diversas metodologías que nos ayudan a enfrentar un problema.
- Cada metodología tiene diversos lenguajes que las soportan.
- Algunos lenguajes soportan varias metodologías.

Metodología	Lenguaje
Estructurada	Fortran, C, Pascal, Basic
Orientada a objetos (OOP)	**C++**, Java, Smalltalk , Python
Orientada a eventos	VisualBasic

Programación Orientada a Objetos

Definición:
Se puede definir **POO** como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción. La **POO** es un **método** de programación en el cual los programas se organizan en colecciones cooperativas de **objetos,** cada uno de los cuales representa una ***instancia*** de alguna ***clase,*** y cuyas clases son, todas ellas, miembros de una **jerarquía de clases** unidas mediante **relaciones de Herencia.**

Comentarios:

- Usamos objetos en lugar de algoritmos como bloque fundamental
- Cada objeto es una instancia de una clase
- Las clases están relacionadas entre sí por relaciones tan complejas como la herencia

Programación Orientada a Objetos

Programa orientado a objetos:
Un programa orientado a objetos es una colección de clases. Necesitará una función principal que cree objetos y comience la ejecución mediante la invocación de sus funciones miembro. Esta organización conduce a separar partes diferentes de una aplicación en distintos archivos. La idea consiste en poner la descripción de la clase para cada una de ellas en un archivo separado. La función principal también se pone en un archivo independiente. El compilador ensamblará el programa completo a partir de los archivos independientes en una única unidad.

Cuando se ejecuta un programa orientado a objetos, ocurren tres acciones:

1) Se crean los objetos cuando se necesitan
2) Los mensajes se envían desde uno objetos y se reciben en otros
3) Se borran los objetos cuando ya no son necesarios y se recupera la memoria ocupada por ellos.

Ventajas de la POO

- Proximidad de los conceptos modelados respecto a objetos del mundo real
- Facilita la reutilización de código
  - Y por tanto el mantenimiento del mismo
- Se pueden usar conceptos comunes durante las fases de análisis, diseño e implementación
- Disipa las barreras entre el *qué* y el *cómo*

Muy adecuado para el desarrollo de software en **grandes colaboraciones**

Desventajas de la POO

- Mayor complejidad a la hora de entender el flujo de datos
  - Pérdida de linealidad
- Requiere de un lenguaje de modelización de problemas más elaborado:
  - *Unified Modelling Language* (UML)
  - Representaciones gráficas más complicadas
- Otra debilidad de la POO aparece cuando programadores diferentes trabajan en una aplicación como un equipo.
- Dado que programadores diferentes manipulan funciones separadas que pueden referirse a tipos de datos compartidos, los cambios de un programador se deben reflejar en el trabajo del resto del equipo.
- Otro problema de la programación estructurada es que raramente es posible anticipar el diseño de un sistema completo antes de que se implemente realmente.

Conceptos de la POO

Conceptos básicos

- Objeto
- Clase

Características de la POO

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Otros conceptos POO

- Tipos
- Persistencia

Tipos de relaciones

- Asociacion
- Herencia
- Agregación
- Instanciación

Representaciones gráficas

- Diagramas estáticos (de Clases, de objetos...)
- Diagramas dinámicos (de interacción...)

Objeto y Clase

Definición de objeto

Un ***objeto*** es una unidad que *contiene los* ***datos*** *y las* ***funciones*** *que operan* sobre ellos.
Los datos se denominan ***miembros dato*** y las funciones ***métodos o funciones miembro***.
Los datos y las funciones se ***encapsulan*** en una única entidad.
Los **datos están ocultos** y sólo mediante las funciones miembro es posible acceder a ellos.
Los objetos representan una entidad concreta o abstracta del mundo real, básicamente se le conoce como ***la instancia de una clase*** y en si es lo que le da el sentido a estas.

Al igual que las clases se componen de tres partes fundamentales:

- **Estado:** Representa los atributos o características con valores concretos del objeto.
- **Comportamiento:** Se define por los métodos u operaciones que se pueden realizar con el.
- **Identidad:** Es la propiedad única que representa al objeto y lo diferencia del resto.

Objeto y Clase

Definición de Clase

Una ***clase*** es un ***tipo*** definido por el usuario que determina las e***structuras de datos y las operaciones asociadas con ese tipo***.
Cada vez que se construye un objeto de una clase, se crea una ***instancia*** de esa clase.
En general, los términos ***objetos e instancias de una clase*** se pueden utilizar indistintamente.
Una clase es una colección de objetos similares y un objeto es una instancia de una definición de una clase.
La comunicación con el objeto se realiza a través del paso de ***mensajes***.
El envío de un mensaje a una instancia de una clase produce la ejecución de un método o función miembro.
El paso de mensajes es el término utilizado para referirnos a la ***invocación o llamada de una función miembro de un objeto***.

Objeto y Clase

En la imagen, los moldes representan las clases, mientras que las galletas obtenidas de estos moldes representan los objetos instancias de estas clases, por ejemplo atributos del objeto galleta podría ser sabor, color, tamaño etc.....

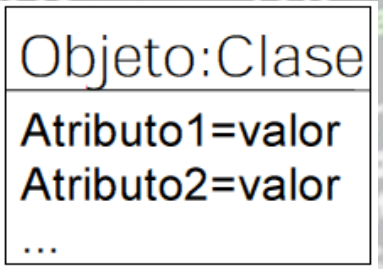
Objeto y Clase

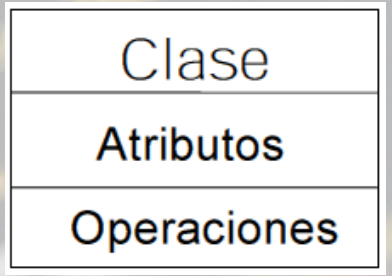
Un objeto es algo de lo que hablamos y que podemos manipular

- Existen en el mundo real (o en nuestro entendimiento del mismo)

Una clase describe los objetos del mismo tipo

- Todos los objetos son instancias de una clase
- Describe las propiedades y el comportamiento de un tipo de objetos





Mensajes: activación de objetos

Los objetos pueden ser activados mediante la recepción de mensajes o métodos. Un método es simplemente una petición para que un objeto se comporte de una determinada manera, ejecutando una de sus funciones miembro.
La técnica de enviar mensajes se conoce como paso de mensajes.

Estructuralmente un mensaje consta de tres partes:

- la ***identidad del objeto receptor***
- la ***función miembro*** *del receptor* cuya ejecución se ha solicitado
- cualquier otra ***información adicional*** *que el receptor pueda necesitar* para ejecutar el método requerido.

La notación utilizada es:

nombre_del_objeto.función_miembro

Conceptos OOP: Abstracción

La abstracción se puede definir como la capacidad de examinar algo sin preocuparse de los detalles internos.
Las estructuras de datos y los tipos de datos son un ejemplo de abstracción.
Los procedimientos y funciones son otro ejemplo.

- Nos permite trabajar con la complejidad del mundo real
  - Resaltando los aspectos relevantes de los objetos de una clase
  - Ocultando los detalles particulares de cada objeto
- Separaremos el comportamiento de la implementación
- Es más importante saber qué se hace en lugar de cómo se hace:

Un sensor de temperatura
- Se define porque...
  - mide la temperatura
  - nos muestra su valor
  - se puede calibrar...

Un sensor de temperatura
- No sabemos... (no nos importa)
  - cómo mide la temperatura
  - de qué está hecho
  - cómo se calibra

Conceptos OOP: Abstracción

- La abstracción **no es única:**

Un coche puede ser...

- **Una cosa con ruedas, motor, volante y pedales (conductor)**

- **Algo capaz de transportar personas (taxista)**
- **Una caja que se mueve (simulador de tráfico)**

- **Conjunto de piezas (fabricante)**

Conceptos OOP: Encapsulamiento

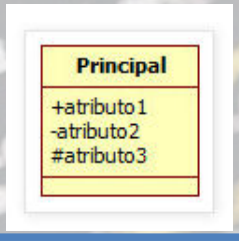
Este concepto es uno de los mas importantes en términos de seguridad dentro de nuestra aplicación, la **encapsulación** es la forma de **proteger** nuestros datos dentro del sistema, estableciendo básicamente los **permisos o niveles de visibilidad o acceso de nuestros datos**.

Se representa por 3 niveles:

Público: Se puede acceder a todos los atributos o métodos de la clase.

Protegido: Se puede acceder a los atributos o métodos solo en la misma jerarquía de herencia.

Privado: Solo se puede acceder a los atributos o métodos de la clase en la que se encuentran.



Conceptos OOP: Encapsulamiento

Con la Encapsulación mantenemos nuestros datos seguros, ya que podemos evitar que por ejemplo se hagan modificaciones al estado o comportamiento de un objeto desde una clase externa, una buena practica es trabajar con métodos **setter** y **getter** que permiten manipular nuestros datos de forma segura.

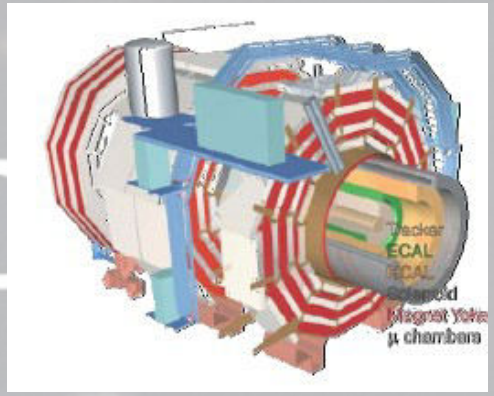
- Ninguna parte de un sistema complejo debe depender de los detalles internos de otra.
- Complementa a la abstracción
- Se consigue:
  - Separando la interfaz de su implementación
  - Ocultando la información interna de un objeto
  - Escondiendo la estructura e implementación de los métodos (algoritmos).
  - Exponiendo solo la forma de interactuar con el objeto

Conceptos OOP: Modularidad

- Consiste en separar el sistema en bloques poco ligados entre sí: módulos.
  - Organización del código
- Es una especie de encapsulamiento de más alto nivel.
  - El C++ no lo impone aunque lo soporta (namespace)
  - El Java es más formal (packages)
- Difícil pero muy importante en sistemas grandes.
  - Suele aplicarse refinando el sistema en sucesivas iteraciones
  - Cada módulo debe definir una interfaz clara

Conceptos OOP: Modularidad

Ejemplo: Simulación detector de particulas
Puede dividirse en los siguientes módulos...
1. Geometría: Describe el detector físicamente (forma, materiales, tamaño)
2. Partículas: Las partículas cuyas interacciones nos interesan
3. Procesos: Aquí enlazamos la información del detector (materia) con las propiedades de las partículas.
4. ...



☐Podríamos dividir el módulo de procesos en *procesos electromagnéticos, procesos hadrónicos*, ...
☐Lo mismo podríamos hacerlo con las partículas: *leptones, hadrones*, ...

Relaciones

- Están presentes en cualquier sistema
- Definen como se producen los intercambios de información (esencialmente datos).
- También ayudan a comprender las propiedades de unas clases a partir de las propiedades de otras.
- Existen 4 tipos de relaciones:
  - Asociación
  - Herencia ← La más característica
  - Agregación
  - Instanciación

Relaciones de Herencia

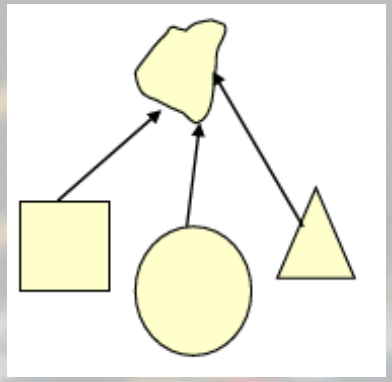
La herencia

Es la propiedad que permite a los objetos construirse a partir de otros objetos.

- Una ***clase*** se puede dividir en ***subclases***.
- La clase original se denomina ***clase base.***
- Las clases que se definen a partir de la clase base, compartiendo sus características y añadiendo otras nuevas, se denominan ***clases derivadas***.
- Las clases derivadas pueden heredar código y datos de su clase base añadiendo su propio código y datos a la misma.
- La herencia impone una relación jerárquica entre clases en la cual una ***clase hija*** hereda de su ***clase padre***.
- Si una clase sólo puede recibir características de otra clase base, la herencia se denomina ***herencia simple***.
- Si una clase recibe propiedades de más de una clase base, la herencia se denomina ***herencia múltiple***.

Relaciones de Herencia

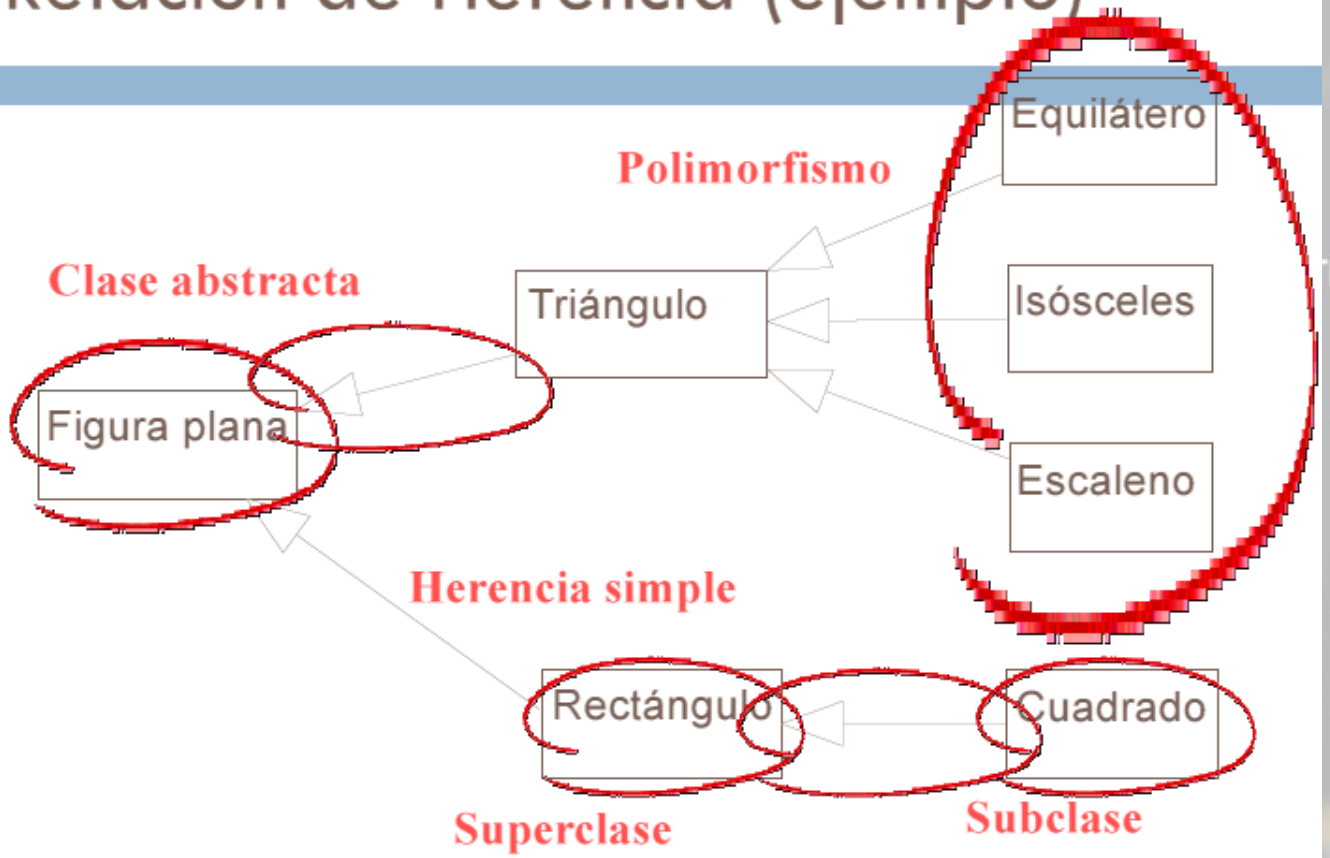
- ¡Relación característica de la POO!
- Puede expresar tanto especialización como generalización
- Evita definir repetidas veces las características comunes a varias clases
- Una de las clases comparte la estructura y/o el comportamiento de otra(s) clase(s).
- También se denomina relación “es *un/a*” (*is a)*



Relaciones de Herencia (vocabulario)

- **Clase base o superclase:** clase de la cual se hereda (***clase padre***)
- **Clase derivada o subclase:** clase que hereda (***clase hija***)
- **Herencia simple:** Hereda de una sola clase
- **Herencia múltiple:** Hereda de varias clases
  - Java solo la soporta parcialmente
  - Presenta diversos problemas (¿qué hacer cuando se hereda más de una vez de la misma clase?)
- **Clase abstracta:** La que no lleva, ni puede llevar, ningún objeto asociado
- **Polimorfismo:** Posibilidad de usar indistintamente todos los objetos de un clase y derivadas.

Relación de Herencia (ejemplo)



Polimorfismo

En un sentido literal, significa la cualidad de tener más de una forma, básicamente mediante el polimorfismo programamos de forma general en lugar de hacerlo de forma especifica, se usa cuando se trabaja con la herencia y objetos de características comunes los cuales comparten la misma superClase y árbol jerárquico, al trabajar con este concepto optimizamos y simplificamos en gran medida nuestro trabajo..

En **POO**, el ***polimorfismo*** se refiere al hecho de que una misma operación puede tener diferente comportamiento en diferentes objetos.

Por ejemplo, consideremos la operación sumar:

- el operador + realiza la suma de dos números de diferente tipo.
- Además se puede definir la operación de sumar dos cadenas mediante el operador suma.

Polimorfismo

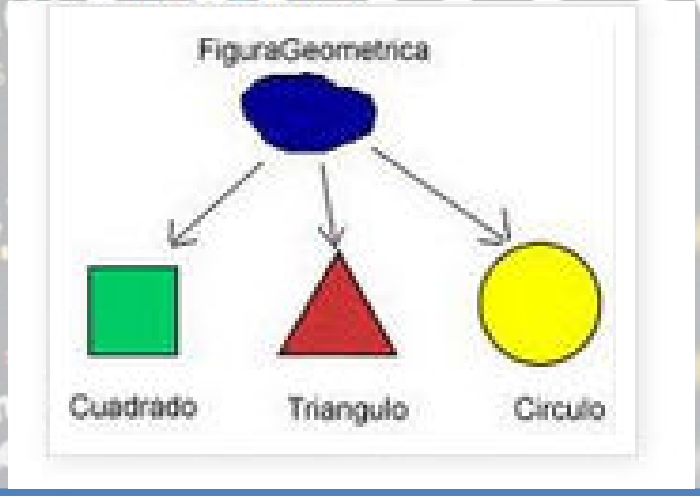
Básicamente podemos definirlo como la capacidad que tienen los objetos de comportarse de múltiples formas sin olvidar que para esto se requiere de la herencia.

En si consiste en hacer referencia a objetos de una clase que puedan tomar comportamientos de objetos descendientes de esta.

Con el polimorfismo usamos la generalización olvidando los detalles concretos de los objetos para centrarnos en un punto en común mediante una clase padre.

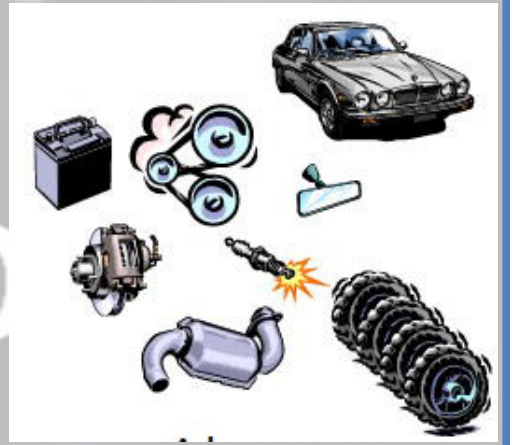
Polimorfismo

Tomando como ejemplo la imagen, podemos decir que un objeto de la clase FiguraGeometrica puede usarse para referirse a cualquier objeto de cualquier subClase de FiguraGeometrica, en otras palabras una figura geométrica puede ser un cuadro, un triángulo, un cuadrado o cualquier figura que en términos generales sea geométrica.....



Relaciones de Agregacion

- Una clase contiene a otra clase
  - Ésta “es parte de” aquélla.
- También se denomina relación “es *parte de*” (has a)
- Una clase puede contener a otra:
  - Por valor: Cuando los objetos de la clase contenida se crean y destruyen al mismo tiempo que los de la clase continente
  - Por referencia: Cuando no necesariamente ocurre lo anterior



Relaciones de Agregacion

Un **coche** está hecho de

- Volante
- Palanca de cambio
- Motor
- Ruedas

Coche

Volante

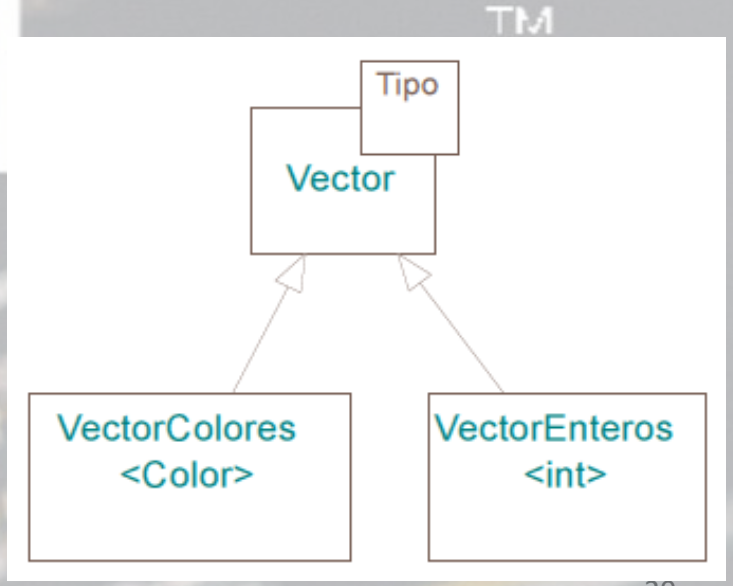
Marchas

Motor

Ruedas

Relaciones de Instanciación

- En determinados casos una clase (p.ej. un vector) puede implementarse independientemente del tipo (real, complejo, color...) de alguno de sus atributos:
  - Definimos una **clase parametrizada** o *template* (plantilla)
  - Para cada uno de los tipos que necesitemos definimos una nueva Clase ⇨ Instanciación



Representaciones gráficas

- Nos sirven para comunicarnos con otros usuarios o desarrolladores.
- Documentan nuestro sistema
- Hay múltiples vistas y tipos de diagramas:

Estáticos:

- Diagramas de clases ⇨Los de los ejemplos
- Diagramas de objetos
- ...

Dinámicos:

- Diagramas de estado: Muestra el ciclo de vida de los objetos, sistemas, etc.
- Diagramas secuenciales: Muestra como los objetos interaccionan entre ellos
- ...

Conceptos POO: Jerarquía

- Es una clasificación u ordenamiento de las abstracciones
- Hay dos jerarquías fundamentales:

Estructura de clases:
- Jerarquía "es *un/a*"
- Relaciones de herencia

Estructura de objetos:
- Jerarquía "parte *de*"
- Relaciones de agregación
- Está implementada de manera genérica en la estructura de clases

Conceptos POO: Jerarquía

Ejemplo: Figuras planas y diagramas

- Una **figura plana** es:
  - Algo con una posición en el plano
  - Escalable
  - Rotable

Herencia simple

- Un cuadrado *es una* figura
- Un círculo *es una* figura

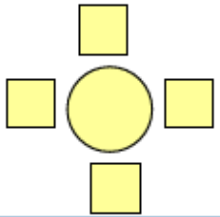
- Un **gráfico** es algo que se puede dibujar en 2D

Herencia múltiple

es una figura
es un gráfico

- Un **diagrama** es un conjunto de cuadrados y círculos

Agregación



Conceptos POO: Tipo

- Es el reforzamiento del concepto de clase
- Objetos de tipo diferente no pueden ser intercambiados
- El C++ y el Java son lenguajes fuertemente “tipeados”
- Ayuda a corregir errores en tiempo de compilación
  - Mejor que en tiempo de ejecución

Conceptos POO: Persistencia

- Propiedad de un objeto de trascender en el tiempo y en el espacio a su creador (programa que lo generó)
- No se trata de almacenar sólo el estado de un objeto sino toda la clase (incluido su comportamiento)
- No está directamente soportado por el C++
  - Existen librerías y sistemas completos (OODBMS) que facilitan la tarea
  - Frameworks (entornos) como ROOT lo soportan parcialmente (reflex)
- El concepto de serialización del Java está directamente relacionado con la persistencia

Relación de Asociación

- Relación más general
- Denota una dependencia semántica
- Es bidireccional
- Primer paso para determinar una relación más compleja

Ejemplo: Relación entre un producto y una venta. Cualquier venta está asociada a un producto, pero no es, ni forma parte de, ni posee ningún producto... al menos en una primera aproximación.

- **Cardinalidad:** multiplicidad a cada lado
  - Uno a uno: Venta-Transacción
  - Uno a muchos: Producto-Venta
  - Muchos a muchos: Comprador-Vendedor

Apendice:
Paradigmas de la Programacion:

- Paradigma imperativo
- Paradigma declarativo
- Programación Orientada a Objetos (POO)
- Programación reactiva

¿Qué son los paradigmas de programación?

- Un paradigma de programación es una manera o estilo de programación de software.
- Existen diferentes formas de diseñar un lenguaje de programación y varios modos de trabajar para obtener los resultados que necesitan los programadores.
- Se trata de un conjunto de métodos sistemáticos aplicables en todos los niveles del diseño de programas para resolver problemas computacionales.

Los lenguajes de programación adoptan uno o varios paradigmas en función del tipo de órdenes que permiten implementar como, por ejemplo, Python o JavaScript, que son multiparadigmas.

Paradigma imperativo

Los programas consisten en una sucesión de instrucciones o conjunto de sentencias, como si el programador diera órdenes concretas. El desarrollador describe en el código paso por paso todo lo que hará su programa.
Algunos lenguajes: Pascal, COBOL, FORTRAN, C, C++, etc.

Otros enfoques subordinados al paradigma de programación imperativa son:

- **Programación estructurada:** La programación estructurada es un tipo de programación imperativa donde el flujo de control se define mediante bucles anidados, condicionales y subrutinas, en lugar de a través de GOTO.
- **Programación procedimental:** Este paradigma de programación consiste en basarse en un número muy bajo de expresiones repetidas, englobarlas todas en un procedimiento o función y llamarlo cada vez que tenga que ejecutarse.
- **Programación modular:** consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más manejable y legible. Se trata de una evolución de la programación estructurada para resolver problemas de programación más complejos.

Paradigma declarativo

Este paradigma no necesita definir algoritmos puesto que describe el problema en lugar de encontrar una solución al mismo. Este paradigma utiliza el principio del razonamiento lógico para responder a las preguntas o cuestiones consultadas.

Este paradigma a su vez se divide en dos:

- **Programación Lógica:** Este paradigma se basa en la fórmula "algoritmos = lógica + control" (la llamada Ecuación Informal de Kowalski), lo que significa que un algoritmo se crea especificando conocimiento mediante ***axiomas (lógica)*** y el problema se resuelve mediante un mecanismo de inferencia que actúa sobre el mismo ***(control)***.
- Entre los lenguajes de programación lógica podemos destacar: Prolog, Lisp o Erlang.

- **Programación funcional:** la programación funcional o functional programming (FP) se centra en las funciones. En un programa funcional, todos los elementos pueden entenderse como funciones y el código puede ejecutarse mediante llamadas de función secuenciales. Por el contrario, no se asignan valores de forma independiente. Una función se imagina mejor como una variante especial de un subprograma. Esta es reutilizable y, a diferencia de un procedimiento, devuelve directamente un resultado. Ejemplos: Lisp, Scala, Java, Kotlin.

Programación Orientada a Objetos (POO)

La POO es un paradigma de programación, esto es, un modelo o un estilo de programación que proporciona unas guías acerca de cómo trabajar con él y que está basado en el concepto de ***clases y objetos***. Este tipo de programación se emplea para estructurar un programa de software en piezas simples y reutilizables de planos de código (***clases***) para crear ***instancias individuales de objetos***.

- En este modelo de paradigma se construyen modelos de ***objetos*** que representan elementos del problema a resolver, que tienen ***características y funciones***.
- Permite separar los diferentes componentes de un programa, simplificando así su creación, depuración y posteriores mejoras.
- La POO disminuye los errores y promociona la reutilización del código.
- Es una manera especial de programar, que se acerca de alguna manera a cómo expresaríamos las cosas en la vida real.
- Podemos definir un objeto como una estructura abstracta que, de manera más fiable, describe un posible objeto del mundo real y su relación con el resto del mundo que lo rodea a través de interfaces.
- Ejemplos de lenguajes de programación orientados a objetos serían Java, Python o C#.

Programación Orientada a Objetos (POO)

La programación orientada a objetos se sirve de diferentes conceptos como:

- **Abstracción de datos**
- **Encapsulación**
- **Eventos**
- **Modularidad**
- **Herencia**
- **Polimorfismo**

- Con el paradigma de POO lo que se busca es dejar de centrarse en la lógica pura de los programas, para comenzar a pensar en objetos, lo que forma la base de dicho paradigma. Esto ayuda bastante en sistemas grandes, pues en lugar de pensar en funciones, se piensa en las relaciones o interacciones de los distintos elementos del sistema.
- Un programador diseña un programa de software organizando piezas de información y comportamientos relacionados en una plantilla denominada clase. Después, se crean objetos individuales a partir de la plantilla de clase. Todo el programa de software se ejecuta haciendo que diversos objetos interactúen entre sí para crear un programa mayor.

Programación reactiva

Este paradigma se basa en escuchar lo que emite un evento o cambios en el flujo de datos, en donde los objetos reaccionan a los valores que reciben de dicho cambio. Las librerías más conocidas son Project Reactor, y RxJava. React/Angular usan RxJs para hacer uso de la programación reactiva.

La programación reactiva es un paradigma enfocado en el trabajo con flujos de datos finitos o infinitos de manera asíncrona. Su concepción y evolución ha ido ligada a la publicación del Reactive Manifesto, que establecía las bases de los sistemas reactivos, los cuales deben ser:

- Responsivos: aseguran la calidad del servicio cumpliendo unos tiempos de respuesta establecidos.
- Resilientes: se mantienen responsivos incluso cuando se enfrentan a situaciones de error.
- Elásticos: se mantienen responsivos incluso ante aumentos en la carga de trabajo.
- Orientados a mensajes: minimizan el acoplamiento entre componentes al establecer interacciones basadas en el intercambio de mensajes de manera asíncrona.

Responsividad o capacidad de respuesta es un concepto de informática que hace referencia a la capacidad específica de un sistema o unidad funcional para completar las tareas asignadas en un tiempo determinado.

Unidad 07 – Programacion Orientada a Objetos con Python

Parte 2

- Definiciones.
- Abstracción.
- Clase y objetos.
- Atributos y métodos.
- Constructores.
- Métodos de Acceso Setter y Getter.

Python - Multiparadigma

admite

- Programación Estructurada
- Programación Orientada a Objetos
- Programación Funcional

Programación Estructurada

La programación estructurada está basada en módulos funcionales bien marcados, generalmente jerarquizados de acuerdo al tipo específico del problema.

Características

- Los programas son más fáciles de entender.
- Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica.
- La estructura del programa es más clara, las instrucciones están más relacionadas entre sí, y es más fácil comprender lo que hace cada función.
- Favorece la reducción del esfuerzo en las pruebas.
- Aumenta la productividad del programador.

Programación Estructurada

Enfoque del Paradigma

La programación estructurada se encuentra ubicada dentro de los paradigmas de programación imperativos. A grandes rasgos, busca imponer restricciones a la transferencia directa de control, con el propósito de establecer una estructura más flexible a las diferentes estructuras que trabajan con el GOTO. Para ello, implementa la modularización para organizar el programa de forma que cada parte de este tenga una función especifica.

Es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y a tres estructuras de control básicas: secuencia, selección (if y switch) e iteración (bucles for y while); asimismo, se considera innecesario y contraproducente el uso de la transferencia incondicional (GOTO); esta instrucción suele acabar generando el llamado código espagueti, mucho más difícil de seguir y de mantener, además de originar numerosos errores de programación.

Programación Estructurada

Estructuras de Control

Todos los programas se ven como compuestos por ***estructuras de control***:

- ***Sequence:*** declaraciones ordenadas o subrutinas ejecutadas en secuencia.
- ***Selection:*** una o varias instrucciones se ejecutan dependiendo del estado del programa. Esto generalmente se expresa con la palabra clave como ***if..then..else..endif***. La declaración condicional debe tener al menos una condición verdadera y cada condición debe tener un punto de salida como máximo.
- ***Iteration:*** una instrucción o bloque se ejecuta hasta que el programa alcanza un cierto estado, o se han aplicado operaciones a cada elemento de una colección. Esto generalmente se expresa con palabras clave como ***while, repeat, for o do..until***. A menudo, se recomienda que cada bucle solo tenga un punto de entrada (y en la programación estructural original, también solo un punto de salida, y algunos lenguajes lo imponen).
- ***Recursion:*** una declaración se ejecuta llamándose repetidamente a sí misma hasta que se cumplen las condiciones de terminación. Si bien en la práctica son similares a los bucles iterativos, los bucles recursivos pueden ser más eficientes desde el punto de vista computacional y se implementan de manera diferente como una pila en cascada.

Programación Orientada a Objetos - POO

Características

- Es una forma especial de programar, más cercana a la forma de expresar las cosas en la vida real que otros tipos de programación.
- Hay que pensar de una manera distinta, para escribir programas en términos de objetos, propiedades, métodos y otros conceptos nuevos.
- El adecuado diseño de clases favorece la reusabilidad.
- La facilidad de añadir, suprimir o modificar nuevos objetos nos permite hacer modificaciones de una forma muy sencilla.

Enfoque del Paradigma

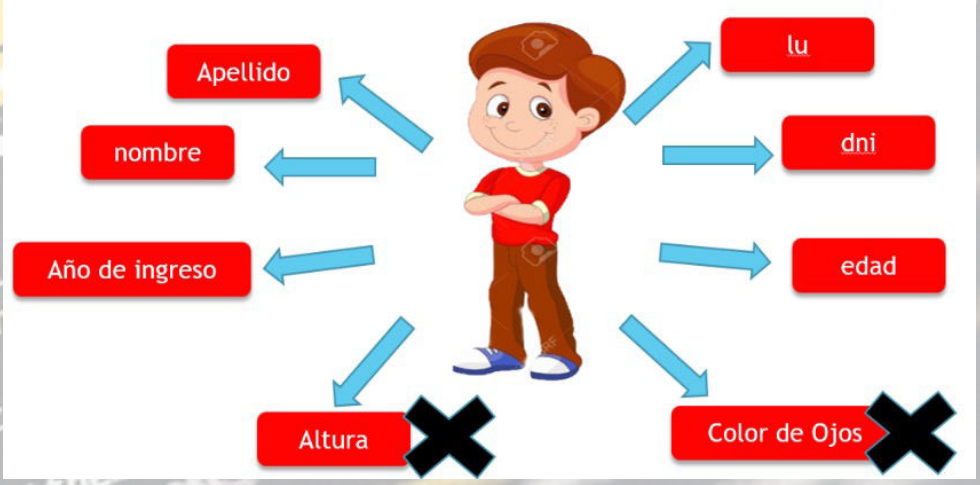
- POO propone resolver un problema computacional a partir de la colaboración entre objetos.

Abstracción

En un problema computacional, la abstracción brinda a los programas orientados a objetos sencillez de leer y comprender y mantener, permiten ocultar detalles de implementación dejando visibles sólo los detalles más relevantes.

Haremos una abstracción de la realidad para diseñar un objeto. Es decir, expresamos las características esenciales de un objeto, las cuales permiten distinguirlo de los demás.

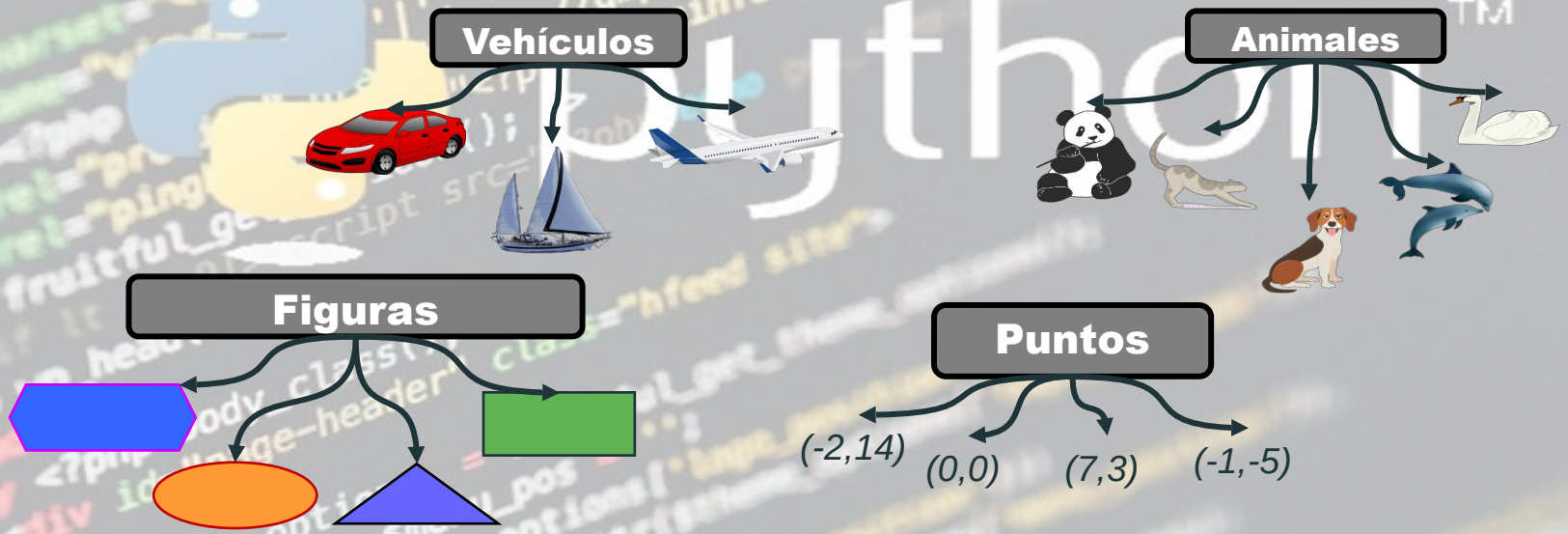
Por ejemplo, podemos abstraer a una Persona:



¿A qué nos referimos con Objetos?

Tratamos de establecer una equivalencia entre un objeto del mundo real con un componente software.

Por ejemplo:



¿A qué nos referimos con Objetos?

Todos los objetos presentan dos componentes principales:

- Un conjunto de características: ***Atributos***.
- Un comportamiento determinado: ***Métodos***.

Como vemos en estos ejemplos, un auto, una moto, un velero y un avión tienen características en común como color, marca y modelo.

Su comportamiento puede ser descrito con operaciones como frenar, acelerar o girar.

Componentes de un Objeto

TODO OBJETO PRESENTA

Características = Atributos

Comportamiento = Métodos

Atributo

Contenedor de un tipo de dato asociado a un objeto, cuyo valor puede ser alterado por la ejecución de algún método.

Método

Algoritmo asociado a un objeto cuya ejecución se desencadena tras la recepción de un "mensaje".

Clase

- Las clases son plantillas para la creación de objetos. Como tal, la clase forma la base para la programación orientada a objetos, la cual es uno de los principales paradigmas de desarrollo de software en la actualidad.

- Una clase es una plantilla o molde para crear objetos. También a una clase se le llama modelo.

- Como convención en el paradigma orientado a objetos, la primera letra del nombre de una clase empieza con mayúscula.

- ¿Cómo crearemos una clase en Python?
- La estructura de clase más simple en Python luciría de la siguiente manera:

```
class ClassName:
    pass
```

Clase

Como podemos ver, la definición de una clase comienza con la palabra clave **class**, y **ClassName** sería el nombre de la clase (identificador).

Ahora vamos a definir una clase Persona (persona), que por el momento no contendrá nada, excepto la declaración de **pass**.

Según la documentación de Python:
La sentencia **pass** no hace nada. Puede ser utilizada cuando se requiere una sentencia sintácticamente pero el programa no requiere acción alguna.

```
class ClassName:
    pass
```

Instancia de una Clase - Objeto

Teniendo una **clase** podremos crear a partir de ella **objetos** con características específicas. Es decir, emplearemos esa ***"plantilla"*** o ***"molde"*** para crear un objeto.

A este proceso se lo conoce como ***"instanciar"***, y nos permite generar una **instancia** u **objeto**.

En **POO** decimos que las **instancias tienen vida**.

Para nuestra clase **Persona**, podemos generar tantas instancias como sean necesarias para resolver un problema.

Instancia de una Clase - Objeto

```
persona1 = Persona()
```

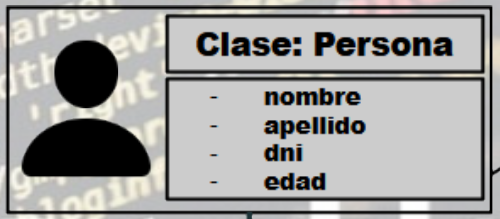
Para **instanciar un objeto** lo haremos como si se tratase de la asignación de una variable normal.

nombre_variable = Clase_con_( )

Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
1  # Creamos una clase
2  class Persona:
3      pass
4
5  # Creamos un objeto de esta clase
6  miPersona = Persona()
```

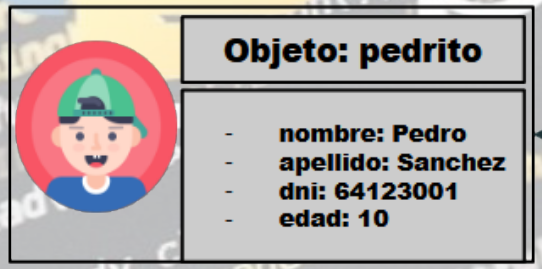
Instancia de una Clase - Objeto

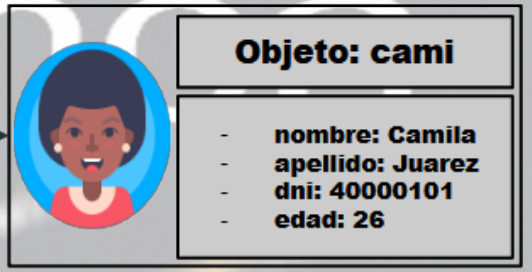


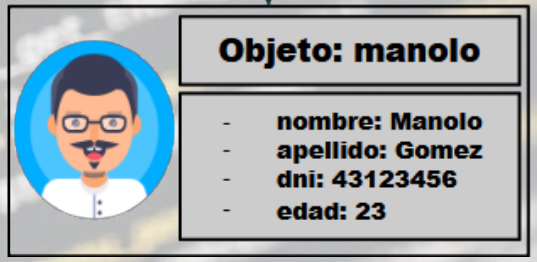
Este será nuestro *"molde" o "plantilla"*

pedrito es una instancia de la clase **Persona**

instancias







manolo es otra instancia de la clase **Persona**

cami es otra instancia de la clase **Persona**

Atributos

Como vimos previamente, un **objeto** cuenta con **atributos**, que son las características que puede tener.

Si el objeto es **Persona**, **los atributos podrían ser: dni, nombre, apellido, edad, etc...**

Los atributos describen el estado de un objeto. Los atributos pueden ser de cualquier tipo de dato.

```
class Persona:
    pass
```

```
#Dentro de __init__() definiremos. Los atributos
#de Persona y le daremos un estado inicial
#realizando asignaciones sobre estos
def __init__(self):
    self.nombre = ''
    self.apellido = ''
    self.dni = 0
    self.edad = 0
```

Métodos

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un parámetro self es una variable que representa la instancia de la clase.

Dentro de una clase definirán métodos como ser el constructor, métodos de acceso (***setter*** y ***getter***),y más adelante métodos personalizados creados por nosotros mismos.

Los métodos representan la funcionalidad de un objeto.
Hay métodos como el constructor, setter, getter, y los métodos que podemos definir nosotros mismos dentro de una clase para darle funcionalidad

Constructor

El ***constructor*** es un método que se utiliza para inicializar todos los atributos de las instancias.

Particularmente en **Python**, un constructor se define mediante el método de nombre **__init__()** para todas las clases.

Al instanciar objetos de esta clase **Persona**, todos ellos tendrán que realizar el comportamiento que vemos en **__init__().** Es decir, cualquier objeto de esta clase, al momento de ser instanciado tendrá nombre y apellido nulos o vacíos, dni y edad serán 0.

```
class Persona:

    def __init__(self):
        self.nombre = ''
        self.apellido = ''
        self.dni = 0
        self.edad = 0
```

Por supuesto podemos definir un comportamiento de inicialización distinto si así lo necesitamos

Constructor

Al momento de ***inicializar*** los atributos de un objeto podemos asignar ciertos datos mediante los parámetros del método **__init__()**.

Para la clase **Persona**, podemos definir el constructor indicando cuales serán su **nombre, apellido, dni** y **edad** de la siguiente manera:

```
class Persona:

    def __init__(self,nombre,apellido,dni,edad):
        self.nombre = nombre
        self.apellido = apellido
        self.dni = dni
        self.edad = edad
```

Constructor

Desafortunadamente en **Python**, no podemos definir varios constructores a la vez. Es decir, no podemos definir más de un método **__init__()** en la misma clase. Y si lo intentamos, solo se ejecutará el último en ser definido.

En su lugar podemos emplear los diferentes tipos de parámetros, como vimos en la unidad anterior (parámetros **posicionales**, **por default** u **opcionales**). Así podemos definir un **constructor** que permitirá establecer un estado predeterminado a los atributos de un objeto.

Constructor

El siguiente fragmento de código muestra cómo eliminar la necesidad de varios constructores con el constructor con parámetros por default.

```
class Persona:

    def __init__(self,nombre="Carlos",apellido="Santana",dni=10321444,edad=75):
        self.nombre = nombre
        self.apellido = apellido
        self.dni = dni
        self.edad = edad
```

Encapsulamiento

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas desarrolladoras de otros lenguajes. En **Python** existe un cierto ***«sentido de responsabilidad»*** a la hora de programar y manejar este tipo de situaciones.

Una posible solución «pythónica» para la privacidad de los atributos es el uso de propiedades. La forma más común de aplicar propiedades es mediante el uso de decoradores

@property

y

@nombreAtributo.setter

Encapsulamiento

- Es el ***ocultamiento*** del estado de un objeto.
- Para lograr el ***encapsulamiento*** de los objetos, los atributos de los mismos deben ser únicamente modificados mediante ***metodos***.
- Para que los atributos de una clase no sean accedidos de manera directa debemos ***protegerlos***.

- Para indicar el nivel de ***visibilidad*** o ***acceso*** de un atributo en Python empleamos la siguiente notación:
  - **Anteponer (“_”) un guión bajo**, nos indica que el atributo es ***protegido***, es decir, sólo deberíamos acceder a él dentro de la definición de la clase o desde una clase hija.
  - **Anteponer (“__”) doble guión bajo**, nos indica que el atributo es ***privado,*** es decir, sólo deberíamos acceder a él dentro de la definición de la clase.

Encapsulamiento

Aquí podemos ver que los atributos se encuentran encapsulados

```
class Persona():
    def __init__(self, nombre, apellido, dni, edad):
        self.__nombre=nombre
        self.__apellido=apellido
        self.__dni=dni
        self.__edad=edad
```

Métodos de Acceso - Setter y Getter

Como dijimos anteriormente el ***encapsulamiento*** solo permite el acceso a un atributo mediante un método, estos métodos se conocen como ***métodos de acceso*** o bien ***getter*** y ***setter***.

Los ***getter*** y ***setter*** se utilizan en **POO** para garantizar el principio de la encapsulación de datos.

- ***getter:*** se emplea para obtener los datos para ver su valor.
- ***setter:*** se emplea para la asignación o modificacion de los datos.

Métodos de Acceso - Setter y Getter

Para definir estos métodos en **Python** emplearemos ***decoradores*** y se identifican por tener un **@** (como lo veremos en el ejemplo).

El propósito principal de cualquier decorador es cambiar los métodos o atributos de la clase de tal manera que el usuario de la clase no necesite hacer ningún cambio en el código.

Una vez que los atributos están encapsulados para poder acceder a ellos emplearemos los siguientes decoradores:

Ya sea para obtener o modificar los datos de un objeto en particular, usaremos el decorador:

@property para un **getter**, que nos permitirá obtener un dato asociado a un atributo en concreto.

@nombreAtributo.setter para un **setter**, que nos permitirá modificar un atributo en particular.

Métodos de Acceso - Setter y Getter

```
@property
def nombre(self):
  return self.__nombre
```

Así definimos un ***getter*** para el atributo nombre, este método ***retorna*** dicho atributo que, como vemos se encuentra encapsulado.

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

Así definimos un ***setter*** para el atributo nombre, este método ***asigna*** un nuevo nombre al atributo que se encuentra encapsulado.

Método __str__

Como hemos visto con anterioridad es una tarea común el mostrar ciertos datos por consola, y para esto recurrimos a **print().**

En **POO** suele ocurrir algo similar, necesitaremos conocer una representación de nuestros objetos, pero si queremos mostrarlos a través de **print()** nos encontraremos con un resultado como este:

```
print(pedrito)
print(cami)
print(manolo)
```

```
<__main__.Persona object at 0x00000214E9BFBDC0>
<__main__.Persona object at 0x00000214E9BFBD60>
<__main__.Persona object at 0x00000214E9BFBD00>
```

Como podemos apreciar al ***mostrar nuestros objetos, el resultado no es muy descriptivo.*** Esto ocurre porque no hemos definido como representaremos a los objetos de la clase **Persona**, y para ello deberemos implementar un nuevo método llamado ***__str__***

Método __str__

Este método retorna la representación de nuestro objeto como una cadena de texto:

```
def __str__(self):
    cadena= "\nNombre: "+self.__nombre
    cadena+= "\nApellido: "+self.__apellido
    cadena+= "\nDNI: "+str(self.__dni)
    cadena+= "\nEdad: "+str(self.__edad)
    return cadena
```

Método __str__

Entonces si quisiéramos mostrar las mismas instancias del ejemplo anterior tendríamos este resultado:

```
print(pedrito)
print(cami)
print(manolo)
```

```
Nombre: Pedro
Apellido:
Sanchez
DNI: 46412301
Edad: 10

Nombre: Camila
Apellido:
Juarez
DNI: 382376168
Edad: 26

Nombre: Manolo
Apellido: Gomez
DNI: 43762129
Edad: 23
```

Objetos

Si definieramos un constructor con parámetros por default, como vimos anteriormente, podemos tener la siguiente instancia:

```
def __init__(self, nombre= "Carlos",
apellido="Santana", dni=12372364, edad=75):
        self.__nombre=nombre
        self.__apellido=apellido
        self.__dni=dni
        self.__edad=edad
```

instanciamos 1 objeto

```
unaPersona=Persona()
```

Si mostramos el objeto **unaPersona**, veremos lo siguiente**:**

```
print(unaPersona)
```

```
Nombre: Carlos
Apellido: Santana
DNI: 12372364
Edad: 75
```

Objetos

Por supuesto podemos acceder a un atributo en particular si contamos con un ***método de acceso.***

Por ejemplo, podemos mostrar el atributo ***nombre*** de las instancias creada anteriormente:

```
@property
def nombre(self):
    return self.__nombre

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

```
print(pedrito.nombre)
print(cami.nombre)
print(manolo.nombre)
print(unaPersona.nombre)
```

```
Pedro
Camila
Manolo
Carlos
```

Objetos con Atributo datetime (fecha)

Para crear un objeto **datetime** usaremos lo siguiente:

1.Importar **datetime**

```
from datetime import datetime
```

2.Crear un objeto fecha pasando el año, mes y día a elección

```
fecha = datetime(anio,mes,dia)
```

3.En el constructor de la clase **Persona** asignaremos la fecha con el atributo correspondiente.

Objetos con Atributo datetime (fecha)

```
class Persona():
    def __init__(self, nombre, apellido, dni, edad,fecha):
        self.__nombre=nombre
        self.__apellido=apellido
        self.__dni=dni
        self.__edad=edad
        self.__fecha=fecha
    @property
    def fecha(self):
        return self.__fecha

    @fecha.setter
    def fecha(self,fecha):
        self.__fecha = fecha
```

Objetos con Atributo datetime (fecha)

4. Crearemos un método para aplicar un formato personalizado a la fecha:

```
def fechaString(self):
    date_string= datetime.strftime(self.__fecha,"%d/%m/%y")
    return date_string
```

Objetos con Atributo datetime (fecha)

5. Creamos el objeto **datetime** e instanciamos nuestra **Persona.** Podemos probar el método para formatear el atributo de tipo fecha:

```
from datetime import datetime
fecha= datetime(2022,10,12)
print("*** Fecha creada con datatime ***")
print(fecha)
p1= Persona("Ballesteros","Cristian",12345632,30,fecha)
informacion= p1
print(informacion)
print("*** Fecha del objeto llamando al metodo fechaString de la clase persona
***")
fechaCadena= p1.fechaString()
print(fechaCadena)
```

Unidad 08 – Parte 3

Objetos dentro de Objetos.
Herencia.
Polimorfismo.

Objetos dentro de Objetos

Ahora que hemos implementado **clases,** y hemos instanciado **objetos** a partir de ellas. Podemos notar que no hay una diferencia sustancial al momento de emplear uno de estos objetos o una variable "***simple***"**.** Esto se debe a que en **Python, TODO ES UN OBJETO**.

En retrospectiva, cuando utilizamos variables de los tipos de datos **int, float, str** o **bool**, estábamos trabajando también con objetos. Estos tipos de datos están predefinidos y disponibles para el usuario, de manera que no tengamos que implementarlos por nuestra cuenta.

Objetos dentro de Objetos

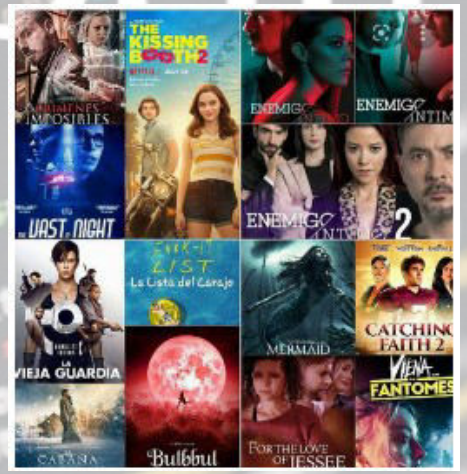
Así que, cuando necesitemos resolver un problema que requiera de un tipo de dato que no se encuentre predefinido, o bien necesite de alguna **característica** o **comportamiento** que no se encuentre implementado previamente**, siempre podremos crear una nueva clase**.

Tomando en cuenta estas afirmaciones, ahora resulta natural que, estas **clases**, se puedan poner en **colecciones** o utilizarlas dentro de otras clases (por ejemplo, como un atributo de otra clase).

Objetos dentro de Objetos

En base a estos conceptos proponemos el siguiente ejercicio.

Describiremos un **catálogo de películas** para analizar cómo se diseña e implementa un anidamiento de objetos:



Objetos dentro de Objetos

Comenzaremos definiendo la clase **Película** para poder instanciar todas las películas que deseamos en variables independientes.

```
class Pelicula:
   def __init__(self, titulo, duracion, lanzamiento):
       self.titulo = titulo
       self.duracion = duracion
       self.lanzamiento = lanzamiento
      def __str__(self):
       return self.titulo+\
       ', '+str(self.duracion)+', '+str(self.lanzamiento)
```


```
peli = Pelicula("El Padrino", 175, 1972)
```

Objetos dentro de Objetos

```
class Catalogo:
    def __init__(self, peliculas=None):
        if peliculas != None:
            self.peliculas = peliculas
        else:
            self.peliculas = []
    def agregar(self, p):
        self.peliculas.append(p)
    def __str__(self):
        cadena = '\nCatálogo:'
        for pelicula in self.peliculas:
            cadena += '\n'+str(pelicula)
        return cadena
```

A continuación definimos la clase **Catálogo**, que contará con una lista de **Películas** como atributo.

Objetos dentro de Objetos

```
peli1 = Pelicula("El Padrino", 175, 1972)
peli2 = Pelicula("El Padrino II", 202, 1974)
peli3 = Pelicula("El Padrino III",162, 1991)
catalogo = Catalogo([peli1,peli2,peli3])
print(catalogo)
```

Así podemos *anidar una clase dentro de otra*. Siendo que al menos un atributo pertenece a otra clase.

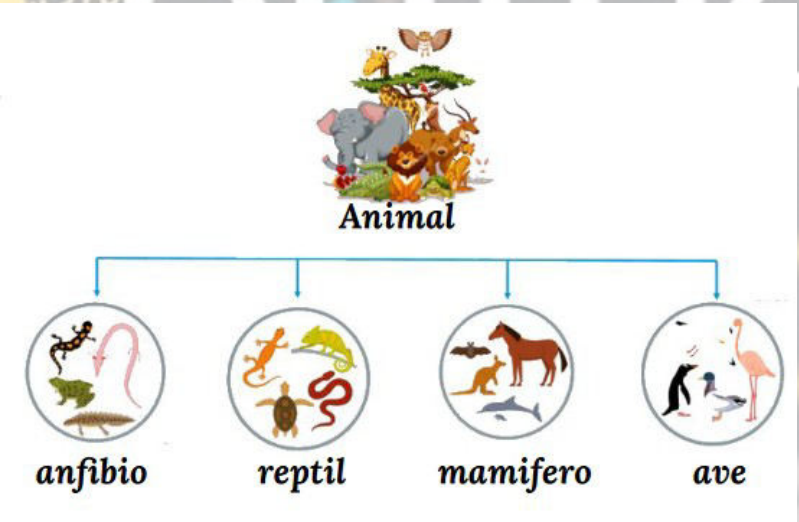
Herencia

Características

- La herencia es una poderosa característica que otorga la capacidad de crear una nueva clase extendiendo una clase ya existente.
- Esto permite que una clase herede los atributos y métodos de la clase principal.
- Nos permite crear clases más especializadas que reutilicen código de una clase general.
- La clase de la que hereda una clase se llama padre o superclase. Una clase que hereda de una superclase se llama clase hija o subclase.
- Existe una relación jerárquica entre clases similar a las relaciones o categorizaciones que conocemos de la vida real.

Herencia

Para heredar primero necesitaremos una clase padre, a modo de ejemplo emplearemos la clase **Animal.** No realizaremos **encapsulamiento** ya que es la primera implementación de **herencia** que haremos, pero puede realizarse si lo necesitamos:



Herencia

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass
    def moverse(self):
        pass
    def __str__(self):
        return "Soy un Animal del tipo "+str(type(self).__name__)
```

Por supuesto, como los métodos **hablar()** y **moverse()** están vacíos, las instancias de la clase **Animal** no realizarán nada al invocarlos:

```
animal1 = Animal("Canis lupus familiaris",5)
```

Herencia

En **Python**, al momento de heredar debemos emplear la siguiente sintaxis:

```
class ClaseHija(ClasePadre):
```

Como vemos deberemos crear una nueva clase y especificar entre paréntesis, a continuación del nombre de la clase que heredará, el nombre de la clase padre. Entonces teniendo en cuenta la clase **Animal** antes implementada, heredamos a una clase **Perro:**

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando en 4 patas")
```

Herencia

Podemos notar que, además de realizar la herencia implementamos los métodos **hablar()** y **moverse()** que encontramos en la clase padre.
Entonces ¿qué ocurrirá si un objeto ***firulais*** de la clase perro realiza el comportamiento **hablar()**?

```
firulais = Perro("Pastor Aleman",4)
firulais.hablar()
```

→ `Guau!`

Lógicamente ***firulais*** *ladra,* puesto que es un **Perro** y hemos ***especializado*** el comportamiento **hablar()** para todo **Perro**

Herencia

Podemos decir también que los atributos y métodos de la clase padre fueron heredados a la clase hija. Es decir, que cualquier objeto de la clase **Perro** cuenta con los atributos **especie** y **edad**, además del método **__str__**. Que a pesar de no haber sido implementado en la clase **Perro** puede ser invocado de igual manera.

```
print(firulais.especie)
print(firulais.edad)
print(firulais)
```

```
Pastor Aleman
4
Soy un Animal del tipo Perro
```

Vemos que podemos mostrar por consola a los atributos de la instancia ***firulais***, y por supuesto también podemos mostrar al mismo **firulais** dado que en la clase padre de **Perro,** es decir en **Animal,** fue definido el método **__str__**.
De esta manera podemos emplear la herencia para reducir sustancialmente el código repetido, y emplear la herencia para dedicarnos exclusivamente a la **especialización** de las clases.

Herencia

De la misma manera podemos heredar a otras clases. Es decir, podemos especializar el comportamiento siempre que lo necesitemos mediante la herencia.

```
class Vaca(Animal):

    def hablar(self):
        print("Muuu!")

    def moverse(self):
        print("Caminando en 4 patas")
```

Podemos notar que el comportamiento **moverse()** es idéntico para **Perro** y **Vaca**, ambos se mueven *"Caminando con 4 patas"*, lo cual perfectamente puede suceder en la realidad.

Herencia

Objetos de diferentes clases realizan un mismo comportamiento, pero el hecho de realizar **herencia** radica en aquellos comportamientos que son **específicos** de una clase hija. Es decir, el método **hablar()** es la especialización que poseen ambas clases, puesto que ambas especies realizan sonidos diferentes (perros ladran y vacas mugen).

```
lola = Vaca("Bos taurus",2)
lola.hablar()
```

→

```
Muuu!
```

Herencia

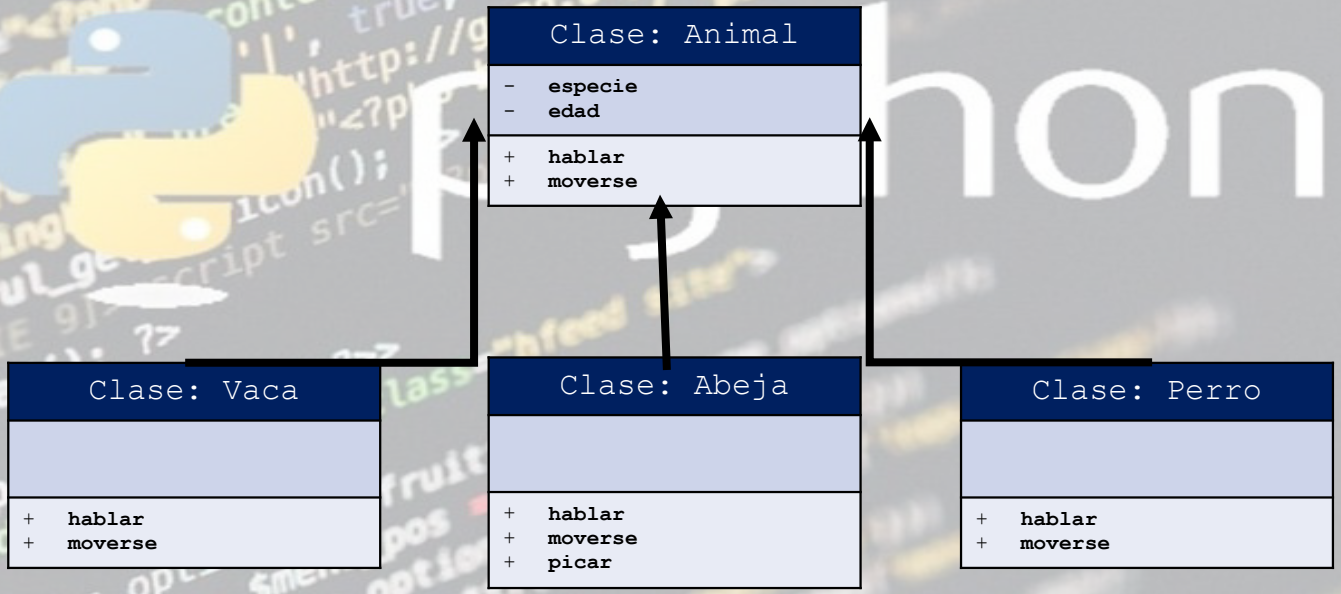
Cabe destacar que no nos encontramos limitados a implementar únicamente los métodos **hablar()** y **moverse()**, sino que podemos agregar nuevos comportamientos propios de la clase hija. Por ejemplo:

```
class Abeja(Animal):

    def hablar(self):
        print("Bzzzz!")

    def moverse(self):
        print("Volando")

    def picar(self):
        print("Picar!")
```

Como vemos el método **picar()** es propio de la clase **Abeja.**

Herencia

A partir de estas clases podemos establecer la siguiente jerarquía de clases, donde **Vaca, Perro** y **Abeja** heredan de **Animal.**



Duck Typing en Python

El término ***polimorfismo*** visto desde el punto de vista de Python es complicado de explicar sin hablar del **duck typing**.

El **duck typing** **o tipado de pato** es un concepto relacionado con la programación que aplica a ciertos lenguajes orientados a objetos, y que tiene origen en la siguiente frase:

"*if it walks like a duck and it quacks like a duck, then it must be a duck*".

Lo que se podría traducir al español como: *Si camina como un pato y habla como un pato, entonces tiene que ser un pato*.

Duck Typing en Python

¿Y qué relación tienen los patos con la programación?

Pues bien, se trata de un símil en el que los patos son objetos y hablar/andar métodos. Es decir, que si un determinado objeto tiene los métodos que nos interesan, nos basta, siendo su tipo irrelevante.

En pocas palabras, **en Python nos dan igual los tipos de objetos, siempre y cuando se implementen los métodos necesarios.**

Duck Typing en Python

Para ejemplificar este concepto definiremos una **clase Pato** y otra **clase Gato**. Ambas contarán con el comportamiento **hablar()**.

```
class Pato:
    def hablar(self):
        print("Cuac")
```

```
class Gato:
    def hablar(self):
        print("Miau")
```

A continuación definiremos una función que necesita del **método hablar()**.

```
def llamar_amigos(x):
    x.hablar()
    x.hablar()
```

Duck Typing en Python

Veremos la practicidad de **duck typing**, al momento de *invocar* la función **llamar_amigos**. Donde nos es indiferente conocer el tipo del objeto que usaremos como argumento de dicha función. Lo único importante, es que el objeto cuente con el método **hablar().**

```
plucky = Pato()
llamar_amigos(plucky)
```

```
michi = Gato()
llamar_amigos(michi)
```

A pesar de que obtengamos un resultado diferente en cada caso, lo cual es lógico considerando la naturaleza de cada objeto. Es importante resaltar que siempre tendremos una salida para la función **llamar_amigos,** siempre que hayamos implementado el método **hablar()** en la clase sobre la cual instanciamos nuestros objetos.
Como aclaración final, debemos mencionar que **duck typing** es una característica válida en **Python** sólo porque este se trata de un lenguaje de tipado dinámico, **y no todos los lenguajes de programación soportan este mecanismo**.

Polimorfismo

En el ámbito de POO el polimorfismo hace referencia a la habilidad que tienen objetos de diferentes clases para responder a métodos con el mismo nombre pero con implementaciones diferentes.
Por ejemplo, para las clases **Perro**, **Vaca** y **Abeja** podemos decir que tanto el método hablar como el método moverse emplean el **polimorfismo**, puesto que cada clase responde de una manera en particular.

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando")
```

```
class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando")
```

```
class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")
```

Polimorfismo

El polimorfismo nos brindará cierta independencia al momento de realizar un comportamiento, puesto que nuestros objetos son abstracciones de la realidad y es importante que puedan resolver una misma tarea acorde a la naturaleza del mismo objeto. A final de cuentas, existen múltiples interpretaciones para resolver una misma tarea.

Unidad 09.1 – Manejo de excepciones

Múltiples excepciones, invocación de excepciones. Creación de excepciones propias.

Manejo de Excepciones

Llamamos excepciones en Python a los errores generados por nuestro código fuente. Si alguna función de nuestro programa genera un error y este no lo maneja, el mismo se propaga hasta llegar a la función principal que la invocó y genera que nuestro programa se detenga.

Manejar los errores nos va a permitir evitar que nuestro programa deje de funcionar de golpe y nos va a dar la posibilidad de mostrar un error personalizado al usuario en vez de los clásicos errores del intérprete Python.

Para ello recurrimos a ciertas palabras reservadas que nos van a permitir realizar algunas acciones antes de detener nuestro programa por completo.

Manejo de Excepciones

- Al programar podemos anticipar errores de ejecución, incluso en un programa sintáctica y lógicamente correcto, pueden llegar a haber errores causados por entrada de datos inválidos o inconsistencias predecibles.

- En Python, puedes usar los bloques **try** y **except** para manejar estos errores como excepciones.

Manejo de Excepciones

Los errores pueden provenir de un error de cálculos o del ingreso de un dato del usuario que nuestro código no es capaz de procesar.

“Supongamos división por cero”

```
a = 5
b = 0
print("a/b = ",a/b)
```

```
---------------------------------------------------------
ZeroDivisionError Traceback (most recent call last)
<ipython-input-2-90888ba7a852> in <module>()
      1 a = 5
      2 b = 0
----> 3 print("a/b = ",a/b)

ZeroDivisionError: division by zero
```

Manejo de Excepciones

Un ejemplo más. Si intentamos acceder a la lista utilizando el valor de índice 2, ocurrirá un error:

```
li = [0, 1]
print(li[2])
```

```
---------------------------------------------------------------
IndexError Traceback (most recent call last)
<ipython-input-1-fcd9bda3ce0e> in <module>()
      1 li = [0, 1]
----> 2 print(li[2])

IndexError: list index out of range
```

Manejo de Excepciones

Suponiendo que las líneas anteriores de código forman parte de un programa, lo que ocurriría al ejecutar la última de ellas, es que el intérprete detendría automáticamente la ejecución del programa.

Para evitar esto, podemos capturar la excepción **IndexError**, la cual ha sido lanzada al detectar el error:

```
li = [0, 1]
try:
    print(li[2])
except IndexError:
    print("Error: índice no válido")
```

Manejo de Excepciones

Empleando el código anterior la ejecución no será detenida y el intérprete continuará ejecutando el programa.

El **bloque try/except** es el utilizado para capturar excepciones.

Justo después de la palabra clave **except** debemos indicar el **tipo de excepción** que deseamos detectar.

Por defecto, si no indicamos ninguna, cualquier excepción será capturada.

try: es el bloque con las sentencias que queremos ejecutar. Sin embargo, podría llegar a haber errores de ejecución y el bloque dejará de ejecutarse.

except: se ejecutará cuando el bloque try falle debido a un error.

```
li = [0, 1]
try:
    print(li[2])
except IndexError:
    print("Error: índice no válido")
```

Manejo de Excepciones

A partir de la sentencia **try**, se tendrá en cuenta cualquier línea de código y si, como consecuencia de la ejecución de una de ellas, se produce una excepción, serán ejecutadas las sentencias de código que aparecen dentro de la sentencia **except**.
En concreto, la sintaxis de la cláusula **try/except** es como se muestra a continuacion.

Estructura completa para el manejo de excepciones:

(obligatorio)
```
try:
    # Código propenso a fallas
```
(obligatorio)
```
except <tipo de error (opcional)> :
    # Código que se ejecutará si el bloque try lanza un error.
```
(opcional)
```
else:
    # Esto se ejecutará si el bloque try se ejecuta sin errores.
```
(opcional)
```
finally:
    # Este bloque se ejecutará siempre.
```

Manejo de Excepciones

Entonces, lo que hay dentro del **try** es la sección del código que podría lanzar la excepción que se está capturando en el **except**. Por lo tanto cuando ocurra una excepción, se entra en el **except** pero el programa no se detiene.
También se puede capturar diferentes excepciones como se ve en el siguiente ejemplo:

```
try:
    #c = 5/0       # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except ZeroDivisionError:
    print("No se puede dividir entre cero!")

except TypeError:
    print("Problema de tipos!")
```

Manejo de Excepciones

Puedemos también hacer que un determinado número de excepciones se traten de la misma manera con el mismo bloque de código. Sin embargo suele ser más interesante tratar a diferentes excepciones de diference manera.

```
try:
    #c = 5/0      # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except (ZeroDivisionError, TypeError):
    print("Excepcion ZeroDivisionError/TypeError")
```

Manejo de Excepciones

Otra forma si no sabemos que excepción puede saltar, puedemos usar la clase genérica **Exception**. En este caso se controla cualquier tipo de excepción. De hecho todas las excepciones heredan de **Exception**.

```
try:
    #c = 5/0       # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except Exception:
    print("Ha habido una excepción")
```

Manejo de Excepciones

No obstante hay una forma de saber que excepción ha sido la que ha ocurrido.

```
try:
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except Exception as ex:
    print("Ha habido una excepción", type(ex))

# Ha habido una excepción <class 'TypeError'>
```

Manejo de Excepciones

Uso de else

Al ya explicado **try** y **except** le podemos añadir un bloque más, el **else**. Dicho bloque se ejecutará si no ha ocurrido ninguna excepción. Fíjate en la diferencia entre los siguientes códigos.

```
try:
    # Forzamos una excepción al dividir entre 0
    x = 2/0

except:
    print("Entra en except, ha ocurrido una excepción")

else:
    print("Entra en else, no ha ocurrido ninguna excepción")

#Entra en except, ha ocurrido una excepción
```

Manejo de Excepciones

Uso de finally

A los ya vistos bloques **try**, **except** y **else** podemos añadir un bloque más, el **finally**. Dicho bloque se ejecuta siempre, haya o no haya habido excepción.

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

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

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

Manejo de Excepciones

Lanzando excepciones.

La declaración **raise** permite al programador forzar a que ocurra una excepción específica.

Por ejemplo:

```
>>> raise NameError('Holaaa!!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Holaaa!!
```

Manejo de Excepciones

Lanzando excepciones.
El único argumento de raise indica la excepción a generarse. Tiene que ser o una instancia de excepción, o una clase de excepción (una clase que hereda de **Exception**). Si se pasa una clase de excepción, la misma será instanciada implícitamente llamando a su constructor sin argumentos.
Si es necesario determinar si una excepción fue lanzada pero sin intención de gestionarla, una versión simplificada de la instrucción raise te permite relanzarla:

```
>>> try:
...     raise NameError('Holaaa!!')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Holaaa!!
```

Manejo de Excepciones

Excepción definida por el usuario.

Entre las excepciones estándar que son más frecuentes están **IndexError**, **ImportError**, **IOError**, **ZeroDivisionError**, **TypeError** y **FileNotFoundError**.

Los programadores pueden nombrar sus propias excepciones creando una nueva clase de excepción. Las excepciones deben derivarse de la clase **Exception**, ya sea directa o indirectamente.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return(repr(self.value))
try:
    raise(MyError(3*2))
except MyError as error:
    print('A New Exception occured: ',error.value)
```

Manejo de Excepciones

Excepción definida por el usuario.

Muchos módulos estándar utilizan esto para informar de errores que pueden ocurrir dentro de las funciones que definen. Para saber más sobre la clase Exception, ejecute el siguiente código:

help(Exception)

Unidad 09.2 – Manejo de ficheros

Ficheros de texto, ficheros y objetos.
Ficheros CSV y ficheros JSON.
Uso de JSON.

Ficheros / Archivos

Imaginemos que debemos trabajar con un gran caudal de datos, sería muy engorroso:

- Ingresar por teclado en cada ejecución
- Manipularla con gran cantidad de variables

Necesitamos información ***persistente***

Llevar registro de asistencia y notas de alumnos para determinar su condición de "regular" o "libre"

Ficheros

Un fichero es un conjunto de bits almacenados en un dispositivo de memoria persistente.

Este conjunto de información se identifica con:

- ➔ Un nombre
- ➔ Una extensión
- ➔ Una dirección o ruta de directorios

Se los conoce como archivos porque son equivalentes digitales a los archivos escritos en expedientes o libretas de una oficina tradicional.

Ficheros

Ficheros

Ficheros

Las operaciones que realizaremos son:

- ***Creación***: *para crearnos un fichero en el disco.*
- ***Apertura***: *para abrir un fichero y comenzar a trabajar.*
- ***Cierre***: *para cerrar un fichero y dejar de trabajar con él.*
- ***Extensión***: *para añadir información al fichero*.

Un puntero es la manera en cómo el ordenador accede y escribe en el fichero correctamente.

Es muy importante evitar sobreescribir información.

Ficheros

Creación y escritura: La función **open()** requiere dos argumentos de entrada, el nombre del fichero y el modo de apertura del fichero.

- *'r': Para leer el fichero (el fichero debe ser existente).*
- *'r+': Para leer y escribir el fichero (el fichero debe ser existente).*
- *'w': Para escribir en el fichero (si no existe lo crea si existe lo sobreescribe.*
- *'a': Para añadir contenido a un fichero existente o que aun no exista.*
- *'a+': Para añadir y leercontenido a un fichero existente o que aún no exista.*

Ficheros

Cierre de un archivo: La función **close()** se utiliza para cerrar una archivo una vez que ya lo hemos utilizado. No requiere argumentos de entrada. Aunque es verdad que el fichero se cerrara automáticamente, es importante especificarlo para evitar tener comportamientos inesperados.

Por lo tanto tenemos tres pasos:

- ➔ *Abrir el fichero que quieramos indicando el modo de hacerlo.*
- ➔ *Usar el fichero para recopilar o procesar los datos que necesitábamos.*
- ➔ *Cuando hayamos terminado de usarlo, cerramos el fichero.*

Ficheros txt

Creación y escritura

```
from io import open

texto = "Una línea con texto\nOtra línea con texto"

# Ruta donde se crea el fichero, w indica escritura (puntero al principio)
fichero = open('fichero.txt','w')

# Escribimos el texto
fichero.write(texto)

# Cerramos el fichero
fichero.close()
```

este script no tiene salida en pantalla

Ficheros

Lectura y visualización por pantalla

```
from io import open

# Ruta donde se lee el fichero, r indica lectura (por defecto es r)
fichero = open('fichero.txt','r')

# Lectura completa
texto = fichero.read()

# Cerramos el fichero
fichero.close()

print(texto)
```

Ficheros

Método **readlines()** para generar una lista

```
from io import open

fichero = open('fichero.txt','r')

# Leamos creando una lista de linea
texto = fichero.readlines()

fichero.close()
print(texto)
```

Ficheros

Agregar datos al final de un fichero

este script si tiene salida en pantalla

```
from io import open

# Ruta donde leeremos el fichero, a indica extensión (puntero al final)
fichero = open('fichero.txt','a')
fichero.write('\nOtra línea más al final de todo')
fichero.close()

# Ahora leemos e imprimimos el fichero con agregado de texto al final
fichero = open('fichero.txt','r')
texto = fichero.read()
print(texto)
```

Ficheros

También se puede leer un fichero utilizando la instrucción estándar ***with*** de la siguiente forma:

```
from io import open

# Ruta donde leeremos el fichero, a indica extensión (puntero al final)
with open('fichero.txt','r') as fichero
    for linea in fichero
      print(linea)
```

este script si tiene salida en pantalla

Ficheros

Métodos ***seek()*** y ***read()***.

Es posible posicionar el puntero en el fichero manualmente usando el método **seek(inicio)** indicando un número que seria la posicion desde donde arrancaria la lectura, y el metodo **read(nro)** indicando el numero de caracteres que seran leidos con ese metodo.

```
fichero = open('fichero.txt','r')
fichero.seek(0)  #Puntero al principio
fichero.read(10) #Leemos 10 caracteres
```

Ficheros

Métodos ***seek()*** y ***read()***.

Para posicionar el puntero justo al inicio de la segunda línea, podríamos ponerlo justo en la longitud de la primera:

```
fichero = open('fichero.txt','r')

fichero.seek(0)
fichero.seek(len(fichero.redline()))
fichero.read()
```

Ficheros

Métodos ***seek()*** y ***read()***.

Se puede abrir un fichero en modo lectura con escritura, pero éste debe existir previamente. Por defecto el puntero estará al principio y si escribimos algo se sobrescribirá el contenido.

```
# Creamos un fichero de prueba con 4 líneas
fichero = open('fichero2.txt','w')
texto = "Línea 1\nLínea 2\nLínea 3\nLínea 4"
fichero.write(texto)
fichero.close()
# Lo abrimos en lectura con escritura y escribimos algo
fichero = open('fichero2.txt','r+')
fichero.write("0123456")
fichero.close()
```

Ficheros

Modificar una línea

Se sugiere leer todas las líneas en una lista, modificar la línea en la lista, posicionar el puntero al principio y reescribir de nuevo todas las líneas:

```
fichero = open('fichero2.txt','r+')
texto = fichero.readlines()
# Modificamos la línea que queramos a partir del índice
texto[2] = "Esta es la línea 3 modificada\n"
# Volvemos a poner el puntero al inicio y reescribimos
fichero.seek(0)
fichero.writelines(texto)
fichero.close()
# Leemos el fichero de nuevo
fichero = open("fichero2.txt", "r")
print(fichero.read())
fichero.close()
```

Ficheros

Resumen

La función **open()** requiere dos argumentos de entrada, el **nombre del fichero** y el **modo de apertura del fichero**.

‘r’: Para leer el fichero.
‘r+’: Para leer y escribir el fichero.
‘w’: Para escribir en el fichero.
‘a’: Para añadir contenido a un fichero existente.
‘a+’: Para añadir contenido a un fichero existente o que aún no exista.

Ficheros

Resumen

Con la función **close()** cerramos el fichero. El objeto de tipo **“file”** la puede invocar y no tiene ningún parámetro asignado

- El fichero normalmente terminará siendo cerrado automáticamente.
- Pero para evitar inconvenientes es aconsejable cerrarlo una vez finalizada su función.

Ficheros

La sentencia **with** nos permite trabajar con archivos de manera más prolija. Ya que simplifica la apertura y cierre del archivo en una sola sentencia y sintaxis.

```
with open("fichero2.txt", "r") as fichero:
    print(fichero.read())
```

De esta manera la apertura del archivo se hace en la primera línea. Luego, en el bloque de código, trabajamos con el archivo (usando la variable creada con la **keyword as**). Finalmente, cuando se sale del bloque de código (al salir del nivel de indentación) el archivo se cierra automáticamente.

Ficheros

La sentencia **with**

```
contenido = "Este es el contenido de mi archivo.\nCreado con Python."
with open("fichero2.txt", "w") as fichero:
    fichero.write(contenido)
```

La sentencia **with** no es exclusiva para manejo de archivos. Es la herramienta de Python para trabajar con Gestores de Contexto (context managers) y es común su uso con archivos, bases de datos, y otros elementos que necesiten operaciones de inicialización y finalización.

Ficheros CSV

- El formato **CSV (Comma Separated Values)** es uno de los más comunes y sencillos para almacenar una serie de valores como tabla.
- Cada fila se representa por una línea diferente, mientras que los valores que forman una columna aparecen separados por un carácter concreto. El más común de los caracteres empleados para esta separación es la coma.
- Software para trabajar con hojas de cálculo, como Microsoft Excel o Calc LibreOffice permiten importar y exportar datos en este formato.
- Python incorpora un módulo específico para trabajar con ficheros CSV, que permiten, tanto leer datos, como escribirlos: **reader()** y **writer()**.

Ficheros CSV

Modulo **CSV (Comma Separated Values).**

Tiene varias funciones y clases disponibles para leer y escribir CSVs, y estas incluyen:

- Función csv.reader
- Función csv.writer
- Clase csv.Dictwriter
- Clase csv.DictReader

Ficheros CSV

Funcion csv.writer()

Para escribir a un archivo **CSV** vamos a usar un objeto **writer** y su método **writerow()**. El método **writerow(datos)** escribe en el archivo los datos pasados como parámetro como una fila completa.

La lista **“contactos”** contiene tuplas de tres elementos, donde el primer elemento es un nombre, el segundo un puesto de trabajo, y el tercero un e-mail. Podemos ver cada tupla como una fila, y cada posición en la tupla como una columna.

Al abrir/crear el archivo tenemos que usar el parámetro **newline = " "** para que se interpreten bien los caracteres contenidos en los datos, sobre todo si contienen saltos de línea.

Ficheros CSV

Funcion csv.writer()

Una vez abierto el archivo creamos un objeto **writer**, al que llamamos **“escritor”**, pasando como parámetros el archivo al que queremos escribir y el delimitador de columnas.
Podemos ir escribiendo filas de datos al archivo con el método **writerow()** de nuestro objeto “escritor”. Para esto iteramos sobre nuestra lista de contactos. En cada iteración se escribe una fila al archivo.
Al finalizar vamos a tener un archivo llamado **contactos.csv** con el contenido.

Ficheros CSV

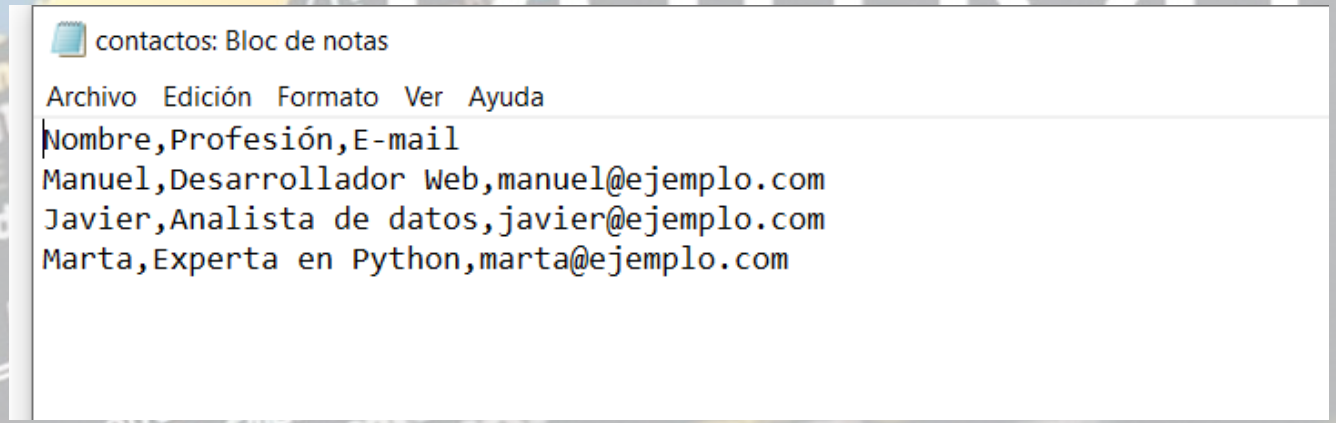
Escritura **CSV (Comma Separated Values).**

```
import csv
contactos = [
  ("Nombre", "Profesión", "E-mail"), # <- La primera tupla es CABECERA del csv
  ("Manuel", "Desarrollador Web", "manuel@ejemplo.com"),
  ("Javier", "Analista de datos", "javier@ejemplo.com"),
  ("Marta", "Experta en Python", "marta@ejemplo.com")
]
with open("contactos.csv", "w", newline="\n") as csvfile:
  writer = csv.writer(csvfile, delimiter=",")
  for contacto in contactos:
      writer.writerow(contacto)
```

Ficheros CSV

Funcion **csv.writer()**

contactos.csv abierto desde un editor de texto plano.



Ficheros CSV

Función **csv.reader()**

Para leer un archivo CSV vamos a usar un objeto reader el cual puede iterar sobre las filas del archivo. Si omitimos ‘r’ por defecto sera interpretado como un open de lectura.

```
with open("contactos.csv", "r", newline="\n") as csvfile:
  lector = csv.reader(csvfile, delimiter=",")
  for nombre, profesion, correo in lector:
      print(nombre, profesion, correo)
```

Ficheros CSV

Funcion **csv.reader()**

La función **reader()** recibe como parámetros un archivo abierto en modo de lectura, y el delimitador de columnas y devuelve un objeto **lector**, el cual es iterable.

Una vez creado el objeto reader, al que llamamos **lector**, podemos iterar sobre él. En cada iteración, el lector va a devolver una lista que representa cada fila.

Ficheros JSON

JSON son las siglas en inglés de **JavaScript Object Notation** y fue definido dentro de uno de los estándares (ECMA- 262) en los que está basado el lenguaje JavaScript.

Para estructurar la información que contiene el formato, se utilizan las llaves **{ }** y se definen pares de nombres y valor separados entre ellos por dos puntos (:).

{"apellido": "Fernández", "nombre": "José Luis", "area": "Finanzas", "ciudad": "Madrid"}

JSON puede trabajar con varios tipos de datos como valores; admite cadenas de caracteres, números, booleanos, listas y null.

Python dispone de un módulo llamado **json** que cuenta con dos funciones principales, una para codificar y otra para decodificar.

Ficheros JSON

Serialización.

Llamamos serialización o codificación al proceso de convertir un objeto de Python (str, list, int, dict, etc) en texto en formato JSON.

Existen dos funciones del módulo json que se usan para la serialización:

- **dumps(obj)**
- **dump(obj, fo)**

Ficheros JSON

Serialización.

dumps(obj): devuelve una ***cadena*** en formato JSON que representa el objeto Python **obj** pasado como parámetro.

```
import json
# Dato en formato Python pasado como parametro
datos = {"nombre": ”Lisa”, “apellido”: “Simpson”, “domicilio”: “Av. Siempreviva
742”, “edad”: 8}
datos_json = json.dumps(datos)
print(datos_json)
```

Ficheros JSON

Serialización.

Mostrará el texto en formato JSON.

```
{
"nombre": ”Lisa”,
“apellido”: “Simpson”,
“domicilio”: “Av. Siempreviva 742”,
“edad”: 8
}
```

Ficheros JSON

Serialización.

dump(obj, fo): serializa un objeto Python **obj** a una secuencia JSON y lo guarda en el archivo **fo** pasado como parámetro.

El archivo “simpson.json” tendrá formato **JSON**.

```
import json
datos = {"nombre": ”Lisa”, “apellido”: “Simpson”, “domicilio”: “Av. Siempreviva
742”, “edad”: 8}
with open("simpson.json", "w") as jsonfile:
    json.dump(datos, jsonfile)
```

Ficheros JSON

Deserialización.

La deserialización o decodificación es el proceso de convertir texto en formato JSON en objetos Python.

Existen dos funciones del módulo json que se usan para la deserialización:

- loads(s)

- load(fo)

loads(s): deserializa una cadena **s** que contiene un documento JSON en un objeto Python. Si la cadena contiene datos que no están en formato JSON se lanzará una excepción **JSONDecodeError**.

Ficheros JSON

Deserialización.

```
import json
s = r'{"nombre": "Bart", "apellido": "Simpson", "domicilio": "Av. Siempreviva
742", "edad": 10, "escolarizado": true}'
bart = json.loads(s)
print(type(bart)) # -> dict
print(bart["escolarizado"]) # -> True
```

Ficheros JSON

Deserialización.

load(fo): deserializa el archivo de texto **fo** con contenido en JSON y devuelve un objeto Python.
Digamos que tenemos el archivo **departamento.json** con el siguiente contenido:

```
{
  "departamento": 8,
  "nombredepto": "Ventas",
  "director": "Juan Rodríguez",
  "empleados":
  [{
      "nombre": "Pedro",
      "apellido": "Fernández"
    },
    {
      "nombre": "Emiliano",
      "apellido": "Castro"
    }]
}
```

Ficheros JSON

Deserialización.

Si abrimos el archivo y utilizamos la función **load()** de la siguiente forma.

```
import json
with open("departamentos.json") as jsonfile:
        departamento = json.load(jsonfile)
print(type(departamentos)) # -> dict
print(departamento.empleados) # -> Lista de empleados
```

Ficheros JSON

Ejemplo: El sueldo de un vendedor es la suma de un monto fijo pagado por el gerente más un porcentaje de sus ventas mensuales.

Si sus ventas fueron menores a $20000, no recibe porcentaje, si estuvieron entre $20000 y $50000 recibe el 20% de ellas y si superan los $50000 recibe el 25% de ellas. Teniendo como dato el sueldo fijo y el monto de la venta mensual cada vendedor, calcule, muestre y almacene el salario final que recibirá cada uno.

Los datos se almacenan en un archivo.txt

Los datos se almacenan en un archivo.csv

Los datos se almacenan en un archivo.json

¿Es posible manipular la información en cualquiera de los 3 formatos?

Unidad 10 – Base de Datos Relacional

Base de Datos: Concepto.

Base de Datos Relacional.

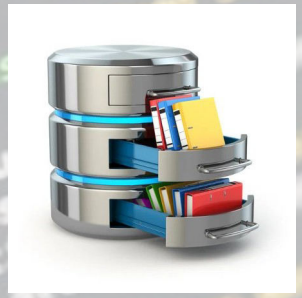
Lenguaje SQL parte 1.

¿Qué es una base de datos?

Se llama base de datos, o también banco de datos, a un conjunto de información perteneciente a un mismo contexto, ordenada de modo sistemático para su posterior recuperación, análisis y/o transmisión.
Existen actualmente muchas formas de bases de datos, que van desde una biblioteca hasta los vastos conjuntos de datos de usuarios de una empresa de telecomunicaciones.
Las bases de datos son el producto de la necesidad humana de **almacenar la información**, es decir, de preservarla contra el tiempo y el deterioro, para poder acudir a ella posteriormente.

Sistemas de Gestión de BD

El manejo de las bases de datos se lleva a cabo mediante sistemas de gestión (llamados DBMS por sus siglas en inglés: Database Management Systems o Sistemas de Gestión de Bases de Datos), actualmente digitales y automatizados, que permiten el almacenamiento ordenado y la rápida recuperación de la información. En esta tecnología se halla el principio mismo de la informática.



Tipos de bases de datos

Existen muchos tipos diferentes de bases de datos. La mejor base de datos para una organización específica depende de cómo pretenda la organización utilizar los datos.

- **Bases de datos relacionales:** Es, en esencia, un conjunto de tablas formadas por filas (registros) y columnas (campos); así, cada registro (cada fila) tiene una ID única, denominada clave y las columnas de la tabla contienen los atributos de los datos.
- **Bases de datos orientadas a objetos**
- **Bases de datos distribuidas**
- **Bases de datos NoSQL.:** No relacionales Entre otras.

Tipos de bases de datos

- Bases de datos OLTP
- Base de datos multimodelo
- Bases de datos de autogestión.

Que es una tabla en base de datos

Cuando estudiamos el tema Base de Datos nos encontramos con el término **“tablas”**, en primer lugar debemos tener claro que una Base de Datos es una especie de almacén en el cual podemos organizar y guardar gran cantidad de información para su posterior uso. Las Bases de Datos están compuestas por una o más tablas.

¿Qué es una tabla en base de datos?

Una **tabla** en base de datos, se refiere a los objetos o estructuras que contienen todos los datos organizados a través de **filas** y **columnas**, las tablas se pueden comparar con una hoja de cálculo en Excel.

Que es una tabla en base de datos

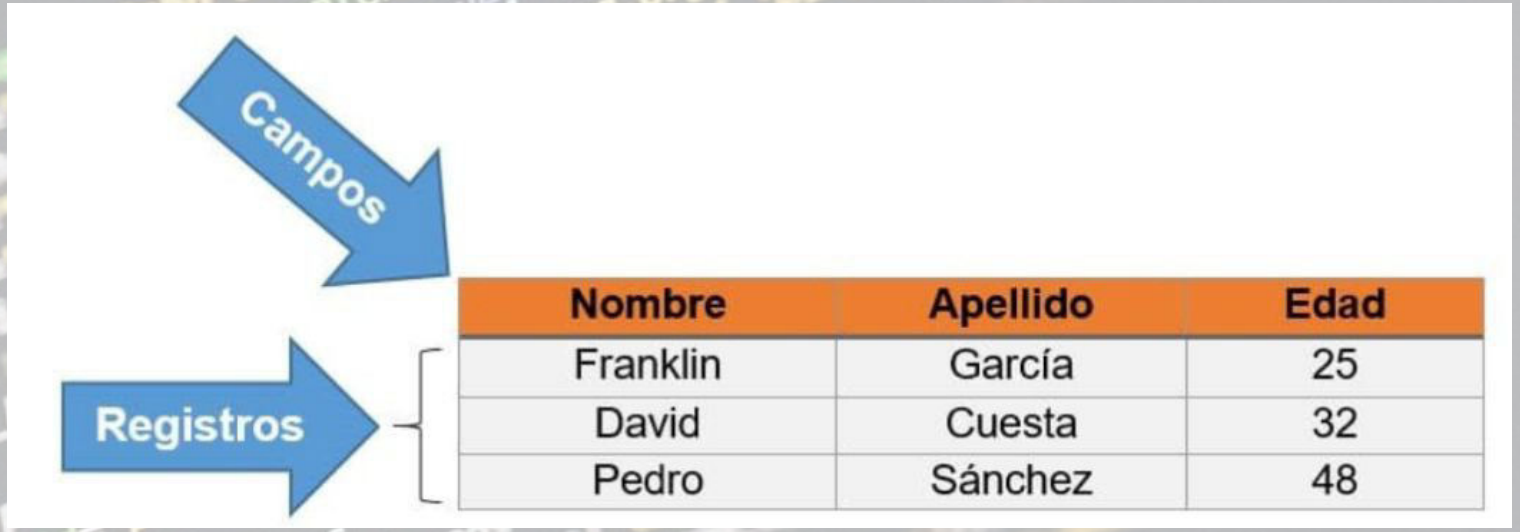
Estructura de las tablas:

Las tablas están compuestas por campos y registros, en donde:

- **Campo:** Se refiere al nombre de la columna. Es un dato único y además se le asigna obligatoriamente un tipo de dato.
- **Registro:** Se refiere a cada fila que conforma la tabla, dicho de otra manera son los datos y registros que almacenamos.
- Cabe aclarar que en ocasiones pueden quedar datos nulos.

Que es una tabla en base de datos

A continuación vemos un ejemplo de una tabla:



Nombre	Apellido	Edad
Franklin	García	25
David	Cuesta	32
Pedro	Sánchez	48

Que es una tabla en base de datos

- Cada **fila** es un **registro único**
- Cada **columna** es un **campo dentro del registro**.

Evidentemente las tablas nos permiten organizar la información de manera clara y nos facilitan la obtención de datos. A este proceso se lo conoce como lectura de datos.

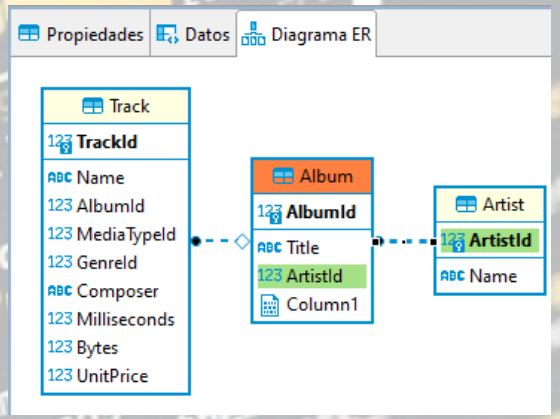
Customer | Enter a SQL expression to filter results (use Ctrl+Space)

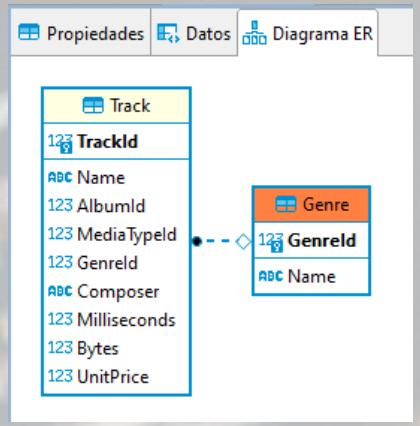
Grilla | Texto

	CustomerId	FirstName	LastName	Company	Address
1	1	Luís	Gonçalves	Embraer - Empresa Brasileira de Aeronáutic	Av. Brigadeiro Faria Lima, 2170
2	2	Leonie	Köhler	[NULL]	Theodor-Heuss-Straße 34
3	3	François	Tremblay	[NULL]	1498 rue Bélanger
4	4	Bjørn	Hansen	[NULL]	Ullevålsveien 14
5	5	František	Wichterlová	JetBrains s.r.o.	Klanova 9/506
6	6	Helena	Holý	[NULL]	Rilská 3174/6
7	7	Astrid	Gruber	[NULL]	Rotenturmstraße 4, 1010 Innere S
8	8	Daan	Peeters	[NULL]	Grétrystraat 63

¿Que es una base de datos relacional?

Las bases de datos **Relacionales** son muy utilizadas actualmente.
Se basan en la idea de crear relaciones entre conjuntos de datos.
Cada **relación** puede ser también una **tabla**.
Cada **tabla** consta de **registros** formados por **filas**, y **columnas**, también conocidos como **tuplas y campos**.





¿Que es una base de datos relacional?

Dentro de las bases de datos relacionales, existen muchos **SGBD** (Sistema Gestor de Base de Datos o motores de bases de datos). La mayoría son compatibles con Python. Algunos son pagos, otros gratuitos, los hay sencillos y otros muy avanzados. Hagamos un repaso:

- SQL Server
- Oracle
- MySQL:
- PostgreSQL
- SQLite

En Python, cada uno de ellos cuenta con módulos libres y programas conectores para comunicar las bases de datos y el lenguaje de programación. Sin embargo, pese a que son sistemas distintos, el lenguaje de las consultas no varía mucho, sino sería muy difícil pasar de un sistema a otro y los SGBD no podrían competir entre ellos.

El lenguaje SQL

Los **SGBD** implementan su propia sintaxis o lenguaje propio para realizar consultas y modificaciones en sus registros.

El lenguaje más utilizado en las bases de datos relacionales es el lenguaje **SQL** (Structured Query Language - Lenguaje de Consulta Estructurada)

Este lenguaje abarca muchísimo contenido, por lo que en este módulo sólo veremos algunas consultas básicas para utilizar el SQLite en nuestros scripts de Python.

El lenguaje SQL

Ejemplos de consultas SQL

Crear una tabla:
CREATE TABLE articulo(
codigo **integer primary key autoincrement**, descripcion **text**, precio **real**);

Nombre de la Tabla: articulo.

codigo	descripcion	precio

El lenguaje SQL

Ejemplos de consultas SQL

Insertar elementos en una tabla:
insert into articulos(descripcion,precio) **values** ("naranjas", 23.50);
insert into articulos(descripcion,precio) **values** ("peras", 34);

Nombre de la Tabla: articulo.

codigo	descripcion	precio
1	naranjas	23.50
2	peras	34

El lenguaje SQL

Ejemplos de consultas SQL

Consultar datos de una tabla:
SELECT codigo, descripcion, precio
FROM articulo;

Muestra todos los registro de artículo:

codigo	descripcion	precio
1	naranjas	23.50
2	peras	34

El lenguaje SQL

Ejemplos de consultas SQL

Consultar datos de una tabla:
SELECT * **FROM** articulo;

Muestra todos los registro de artículo:

codigo	descripcion	precio
1	naranjas	23.50
2	peras	34

El lenguaje SQL

Ejemplos de consultas SQL

Consultar datos de una tabla:
SELECT codigo, descripcion, precio
FROM articulo **WHERE** precio > 23.50;

Muestra todos los registros de articulo que cumplan con la condición de que el precio sea mayor a 23.50.

codigo	descripcion	precio
2	peras	34

Unidad 11 – Base de Datos Relacional

CRUD.

Clave primaria.

SQL segunda parte.

Clave foránea.

CRUD

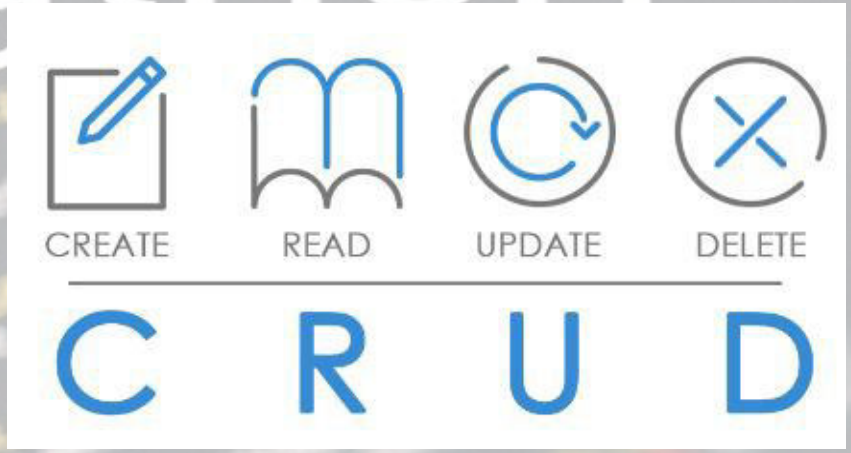
El concepto **CRUD** está estrechamente vinculado a la gestión de datos digitales.
CRUD hace referencia a un acrónimo en el que se reúnen las primeras letras de las cuatro operaciones fundamentales de aplicaciones persistentes en sistemas de bases de datos:

Create (Crear registros)

Read (Leer registros)

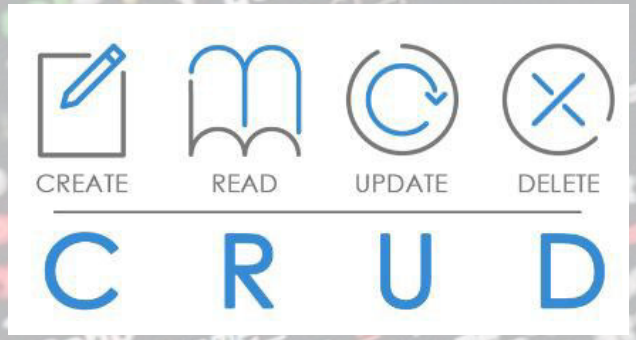
Update (Actualizar registros)

Delete (Borrar registros)



CRUD

En pocas palabras, **CRUD** resume las funciones requeridas por un usuario para crear y gestionar datos. Varios procesos de gestión de datos están basados en **CRUD**, en los que dichas operaciones están específicamente adaptadas a los requisitos del sistema y de usuario, ya sea para la gestión de bases de datos o para el uso de aplicaciones.



CRUD-Operation	SQL	RESTful HTTP
Create	INSERT	POST, PUT
Read	SELECT	GET, HEAD
Update	UPDATE	PUT, PATCH
Delete	DELETE	DELETE

Crear una tabla con campos

La sintaxis de como crear una tabla es la siguiente:
CREATE TABLE *Nombre_Tabla* (
'Nombre_Columna1' **Tipo_de_Dato** (longitud),
'Nombre_Columna2' **Tipo_de_Dato** (longitud),
'Nombre_Columna3' **Tipo_de_Dato** (longitud),
....);

Los parámetros **«Nombre_Columna»** especifican los nombres de las columnas que integran las tablas.
Los parámetros **«Tipo_de_Dato»** especifican que tipo de datos admitirá esa columna (ej. varchar, integer, decimal, etc.).
El parámetro **«Longitud»** especifica la longitud máxima de caracteres que admitirá la columna de la tabla.

Crear una tabla con campos

Cabe aclarar que hay varias propiedades que pueden ser aplicadas a las columnas de la tabla, por ejemplo una de las más comunes es **PRIMARY KEY**, la cual nos permite indicar qué columna será clave primaria, también podemos establecer que alguna columna no acepte valores nulos, a través de la propiedad **NOT NULL**.

Ejemplo: Crear una tabla con campos

```
CREATE TABLE articulos (
        codigo INT PRIMARY KEY NOT NULL,
        descripcion TEXT(100),
        precio REAL
        );
```

Crear una tabla con campos

En el ejemplo del código anterior estamos creando la tabla **“articulos”** que tiene 3 columnas, podemos darnos cuenta de que una de las novedades es en la columna **código**, ya que tenemos la propiedad **PRIMARY KEY**, la cual indica que es la clave primaria que identifica de manera única cada **registro/fila** de la tabla, así mismo con la propiedad **not null** estamos indicando que esa columna es indispensable, es decir que siempre debe ser ingresado un **código**.

Crear una tabla con campos

Cabe aclarar que la propiedad **not null** puede ser implementada en otras columnas que consideremos que son indispensables, por ejemplo **descripcion** es fundamental que sea ingresado, en ese caso deberíamos agregarle la propiedad **not null** para garantizar que el nombre siempre será ingresado.
Para efectos de este ejemplo no ha sido aplicado; sin embargo, cuando estamos creando unas Base de Datos con una buena estructura, debemos tomar en cuenta todos esos aspectos.

Clave Primaria (PK)

La clave primaria, **PRIMARY KEY**, identifica de manera única cada fila de una tabla.

La columna definida como clave primaria (**PRIMARY KEY**) debe ser **UNIQUE** (valor único) y **NOT NULL** (no puede contener valores nulos).

Cada tabla solo puede tener una clave primaria (**PRIMARY KEY**). Una clave primaria o **PRIMARY KEY** es una columna o un grupo de columnas que identifica de forma exclusiva cada fila de una tabla.

Se puede crear una llave primaria para una tabla utilizando la definicion de **PRIMARY KEY**.

Clave Primaria (PK)

Cada tabla puede contener **solo una clave primaria**.
Todas las columnas que participan en la clave primaria **deben** definirse como **NOT NULL.**

Clave Primaria (PK)

Sintaxis para crear la clave al momento de definir el campo que sera **Clave Primaria**:

```
CREATE TABLE table_name (
        pk_columna tipodeDato PRIMARY KEY NOT NULL,
        ...
        ...
        );
```

Clave Primaria (PK)

Ejemplo **PRIMARY KEY** o clave primaria en SQL.

La clave primaria se puede agregar al momento de crear el campo que será la clave primaria.

```
CREATE TABLE articulos (
        codigo INT PRIMARY KEY NOT NULL,
        descripcion TEXT,
        precio REAL
        );
```

Clave primaria autoincremental

El mismo sitio de **SQLite3** dice que no recomienda el auto incremento o las columnas auto incrementales. Aunque algunas veces es usado para ir incrementando un valor entero y tomarlo como **id**. Es como la columna autoincremental de SQL.

Una columna numérica de una tabla que es **clave primaria** le podemos asignar que sea **autoincrementable** lo que significa que su valor se incrementa automáticamente al momento de insertar un nuevo registro, por lo cual el valor del id o clave primaria no debe ser asignado al momento se inserta un nuevo registro.

Clave primaria autoincremental

Podemos observar aquí como los IDs van tomando valores en orden creciente.

ID	Nombre	Apellido	Calle	Ciudad
1	Juan	Uno	Calle 274	Veraguas
2	Pedro	Dos	Calle 004	Chiriqui
3	Miguel	Tres	Calle 174	Darien
4	Daniel	Cuatro	Calle 284	Colon
5	Ramiro	Cinco	Calle 277	Panama

Ejemplo - PK autoincremental

Ejemplo de como hacer que mi campo de clave primaria sea auto Incremental:

```
CREATE TABLE articulos(
        codigo INT(4) PRIMARY KEY NOT NULL
        AUTOINCREMENT, descripcion TEXT, precio REAL
        );
```

Consultas a una tabla SELECT

SELECT columnas **FROM** nombre_Tabla;

En donde:
SELECT: Es el comando que se utiliza para obtener registros de las tablas.

columnas: Hace referencia a los nombres de las columnas de las cuales queremos consultar registros.

FROM: Es el comando para especificar la tabla de la cual vamos a consultar los datos.

Nombre_Tabla: Hace referencia al nombre de la tabla de la cual queremos consultar los registros.

Consultas a una tabla SELECT

Todas las columnas. Ejemplo:

- **SELECT** * **FROM** nombreTabla;
- **SELECT** * **FROM** articulos;

***:** Se usa para decir que seleccionamos **todas las columnas** de la tabla.

Algunas columnas:

- **SELECT** columna1, column2 **FROM** articulos;
- **SELECT** descripcion, precio **FROM** articulos;

columna1,columna2: Podemos elegir las columnas que deseemos obtener, se puede poner tantas columnas como deseemos.

Cláusula Where

La cláusula **WHERE** nos permite condicionar las consultas con relación a los registros de una o varias columnas que se especifiquen en la sentencia, en tal sentido todos los registros que entran en el filtro del **where** se mostrarían en los resultados que obtendremos.

La cláusula **where** es utilizada en los casos que no necesitamos que nos muestre todos los registros de una tabla, sino que únicamente los registros que cumplan ciertas condiciones.

Las condiciones conllevan expresiones lógicas que se comprueban con la cláusula **where**.

El valor que devuelven las comparaciones es un valor **TRUE** o **FALSE**, dependiendo del cumplimiento de la condición especificada..

Cláusula Where

Podemos hacer uso de cualquier **expresión lógica** y en ella implementar algún operador de los siguientes:

> “Mayor”,

>= “Mayor o igual”

< “Menor”

<= “Menor o igual”

= “Igual”

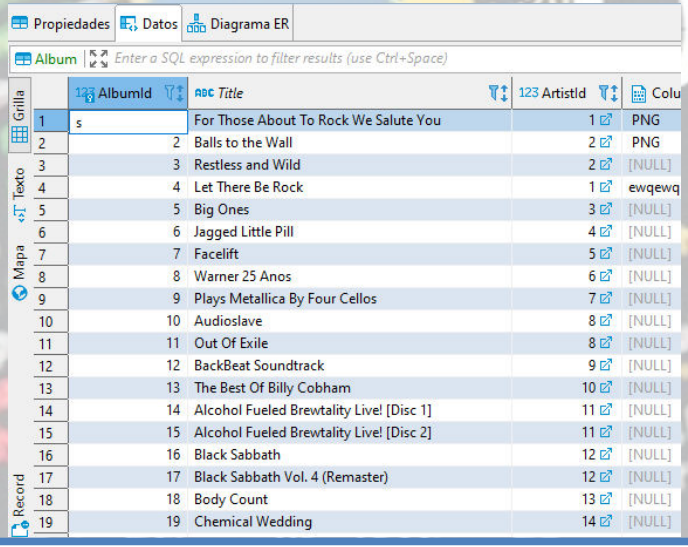
<> o != “Distinto”

IS [NOT] NULL “para validar si el valor de una columna es nulo ó no es nulo, es decir, si contiene o no contiene algún registro”.

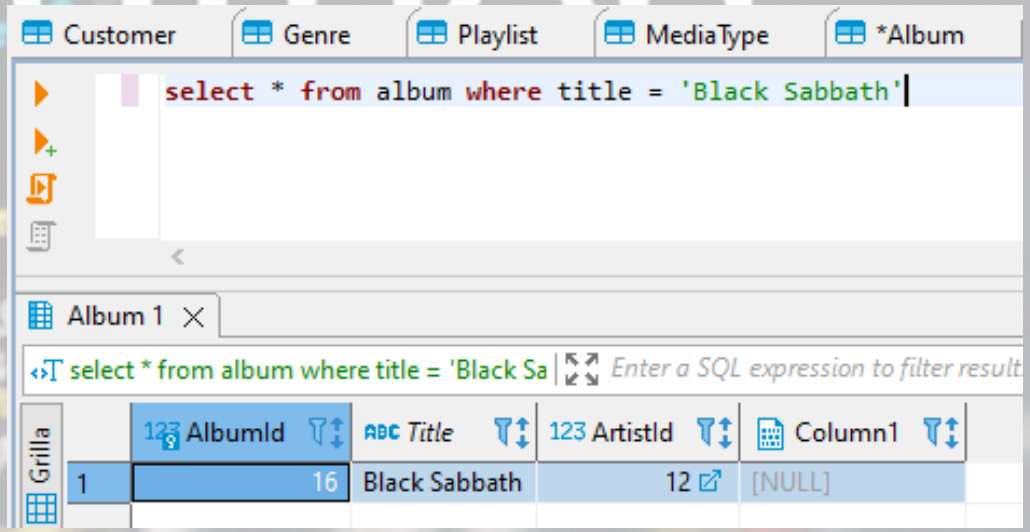
Como así también, los operadores lógicos: **AND , OR y NOT**

Ejemplo: Clausula Where

SELECT * **FROM** articulos **WHERE** precio = 34;
En este caso estaríamos logrando que nos muestre todos los registros que tengan el precio igual a 34.
Otro ejemplo:



	AlbumId	Title	ArtistId	Colu
1	s	For Those About To Rock We Salute You	1	PNG
2	2	Balls to the Wall	2	PNG
3	3	Restless and Wild	2	[NULL]
4	4	Let There Be Rock	1	ewqewq
5	5	Big Ones	3	[NULL]
6	6	Jagged Little Pill	4	[NULL]
7	7	Facelift	5	[NULL]
8	8	Warner 25 Anos	6	[NULL]
9	9	Plays Metallica By Four Cellos	7	[NULL]
10	10	Audioslave	8	[NULL]
11	11	Out Of Exile	8	[NULL]
12	12	BackBeat Soundtrack	9	[NULL]
13	13	The Best Of Billy Cobham	10	[NULL]
14	14	Alcohol Fueled Brewtality Live! [Disc 1]	11	[NULL]
15	15	Alcohol Fueled Brewtality Live! [Disc 2]	11	[NULL]
16	16	Black Sabbath	12	[NULL]
17	17	Black Sabbath Vol. 4 (Remaster)	12	[NULL]
18	18	Body Count	13	[NULL]
19	19	Chemical Wedding	14	[NULL]

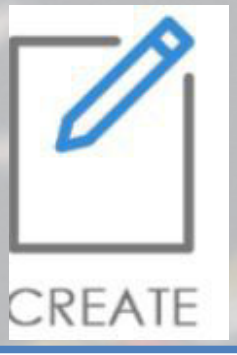


	AlbumId	Title	ArtistId	Column1
1	16	Black Sabbath	12	[NULL]

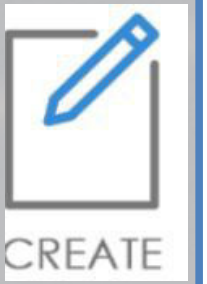
Insertar o crear

Para agregar datos a las tablas se utiliza la instrucción **Insert into**, este comando es uno de los más usados en los diferentes gestores de Base de Datos.
Para insertar los registros se puede hacer de uno en uno, o agregar varios registros a través de una misma instrucción **insert into**.
La sintaxis para usar la instrucción **insert into** en una tabla de MySQL es la siguiente:

INSERT INTO "NombreTabla"
("PrimeraColumna", "SegundaColumna", etc)
VALUES
("Dato1", "Dato2", etc);



Insertar o crear



Explicación del código:

NombreTabla: Es el nombre de la tabla en la que insertamos registros.

PrimeraColumna, SegundaColumna,..: Son las columnas de la tabla en la que vamos a insertar registros.

“Dato1”, “Dato2” , ... : Son los valores que vamos a guardar en cada columna especificada.

Es importante mencionar que la sintaxis anterior se puede reducir en los casos que vamos agregar datos a todas las columnas, ya que podríamos plantearlo de la forma siguiente:

INSERT INTO “NombreTabla” **VALUES** (“Dato1”, “Dato2”, etc);

Insertar o crear

En el ejemplo anterior de insertar:
INSERT INTO “NombreTabla” **VALUES** (“Dato1”, “Dato2”, etc);

Cuando aplicamos esta sintaxis, debemos **respetar la estructura de la tabla** y además después del comando **VALUES** enviar todos los datos para cada columna **en el orden correcto**, ya que dicha sintaxis indica que vamos a insertar registros a todas las columnas, por lo tanto debemos enviar los datos exactamente como los hemos especificado al momento de crearla.

Existe otra opción de insertar registros y es mencionando columnas específicas, para este caso debemos tomar en cuenta que las columnas que omitimos puedan quedar nulas, es decir que NO le hayamos agregado la propiedad **NOT NULL**.

Insertar o crear

Ejemplo de insertar un solo registro:

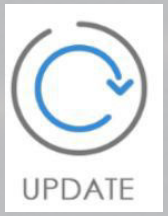
```
INSERT INTO articulos (descripcion, precio)
VALUES ('Manzana',150);
```

Ejemplo de insertar muchos registros:

Para agregar varios registros a con un mismo **insert**, se agrega una coma en los valores que le enviamos en **VALUES**, y especificamos los datos a insertar.

```
INSERT INTO articulo
        (codigo, descripcion, precio)
VALUES
        (30, 'fideos', 200),
        (50, 'jugo', 300),
        (51, 'galletas dulces', 150);
```

Actualizar registros



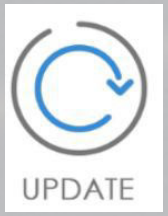
Para modificar los datos que contiene actualmente una tabla, se usa la instrucción **UPDATE**, a la que se suele denominar **"consulta de actualización"**.

La instrucción **UPDATE** puede modificar uno o varios registros y, por lo general, tiene esta forma.

UPDATE nombreTabla
SET nombreColumna = valor

Para actualizar todos los registros de una tabla, especifique el nombre de la tabla y, después, use la cláusula **SET** para especificar el campo o los campos que se deben cambiar.

Ejemplo - Actualizar registros



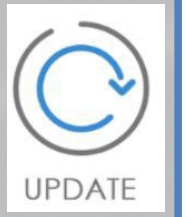
Para actualizar un solo campo:

UPDATE articulo
SET descripcion = 'Galletas dulces'

En la mayoría de los casos, es recomendable que condicionen la instrucción **UPDATE** con una cláusula **WHERE** para limitar la cantidad de registros que se cambiarán.

UPDATE articulos
SET precio = 350
WHERE codigo= 51

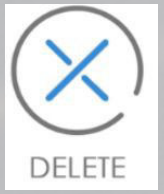
Ejemplo - Actualizar registros



Para actualizar más de un campo sólo debemos poner los campos que vamos a acyualizar y los valores correspondientes, por ejemplo:

UPDATE articulo
SET descripcion = 'galletas saladas', precio=100
WHERE codigo= 51

Eliminar registros de una tabla



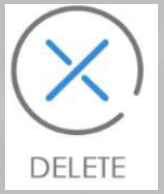
Para eliminar los datos que contiene actualmente una tabla, se usa la instrucción **DELETE**, a la que se suele denominar **"consulta de eliminación"**. También se llama **"truncar una tabla"**.
La instrucción **DELETE** puede quitar uno o varios registros de una tabla y, en general, tiene esta forma de sintaxis:

DELETE FROM nombreTabla

La instrucción **DELETE** no quita la estructura de la tabla, sino solamente los datos que contiene en ese momento la estructura de la tabla. Para quitar todos los registros de una tabla, use la instrucción **DELETE** y especifique de qué tabla o tablas quiere eliminar todos los registros.

DELETE FROM articulo

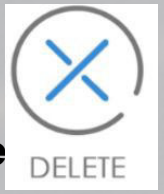
Eliminar registros de una tabla



En la mayoría de los casos, es recomendable que condicionemos la instrucción **DELETE** con una cláusula **WHERE** para limitar la cantidad de registros que se quitarán.

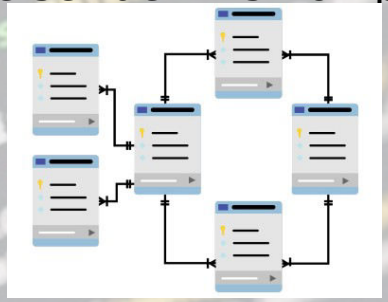
DELETE FROM Alumnos
WHERE id = 3

Relaciones entre Tablas

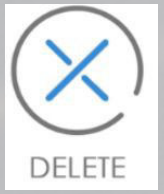


En el contexto de bases de datos relacionales, una **clave foránea** o **llave foránea** o **clave ajena** es una **limitación referencial entre dos tablas**.
La clave foránea identifica una columna o grupo de columnas en una tabla que se refiere a una columna o grupo de columnas en otra tabla en donde estas son claves primarias.
Una clave foránea se indica al final de la creación de una tabla con:

FOREIGN KEY(fk) REFERENCES otra_tabla(campo_de_esa_tabla)
Nota: la clave foránea debe ser del mismo tipo que la clave primaria a la que se refiere.



Relaciones entre Tablas

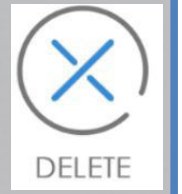


Por ejemplo crear una relación entre la tabla productos y categorías, un producto puede tener una sola categoría pero una categoría puede estar en varios productos, por lo cual la tabla categorías tiene el uno y la tabla de productos tiene la relación de muchos.

```
CREATE TABLE categoria(
        id_cate INTEGER(4) PRIMARY KEY AUTO_INCREMENT,
        descripcion VARCHAR(30)
        );
```

Relaciones entre Tablas

Luego creamos la tabla producto donde crearemos un campo adicional para guardar la clave primaria de la tabla CATEGORIAS.



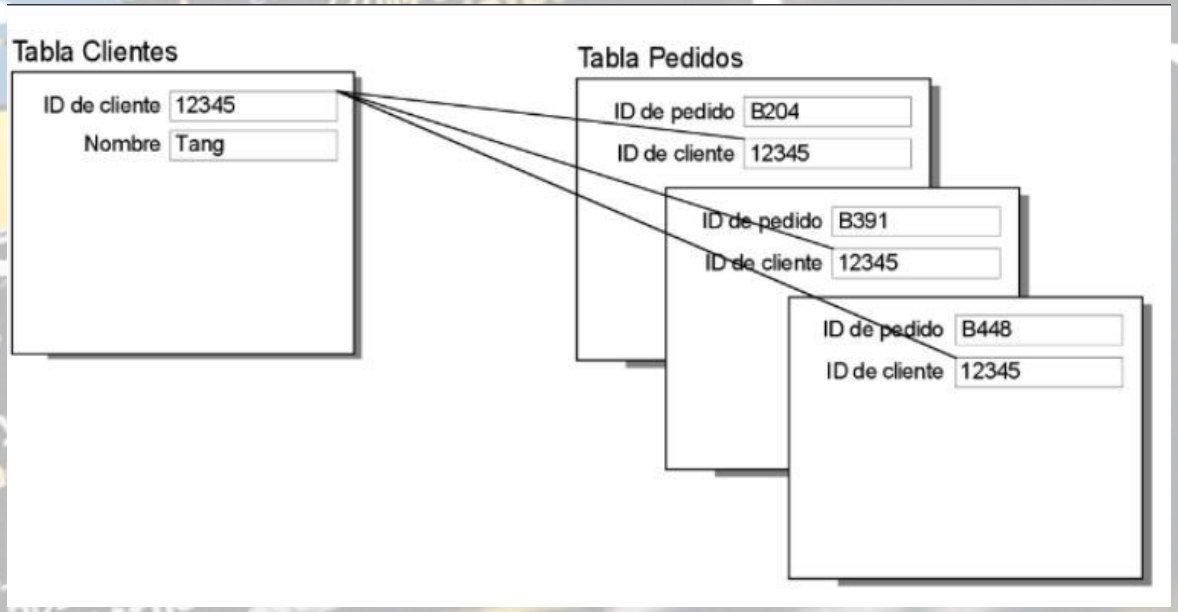

```
CREATE TABLE producto (
        id_prod INTEGER PRIMARY KEY AUTO_INCREMENT,
        descripcion VARCHAR(30),
        precio DOUBLE,
        idcategoria INTEGER(4),
        FOREIGN KEY(idcategoria) REFERENCES categoria(id_cate)
        );
```

Nota importante: la clave primaria de categoría y el campo de clave foránea deben ser del mismo tipo de dato y longitud.

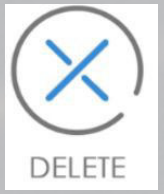
Relaciones de uno a muchos

DELETE

En una relación de uno a muchos, un registro de una tabla se puede asociar a uno o varios registros de otra tabla.. Por ejemplo, cada cliente puede tener varios pedidos de ventas.



Relaciones de uno a muchos



En este ejemplo, el campo de clave principal de la tabla Clientes, ID de cliente, se ha diseñado para contener valores exclusivos.
El campo de clave Foránea de la tabla Pedidos, ID de cliente, se ha diseñado para permitir varias instancias del mismo valor.
Esta relación devuelve registros relacionados cuando el valor del campo ID de cliente de la tabla Pedidos es el mismo que el valor del campo ID de cliente de la tabla Clientes.

Unidad 12 – Base de Datos SQLite con Python

SQLITE

BD + Python

SQLite Python

PySQLite

PySQLite proporciona una interfaz compatible con Python DBI API 2.0 estandarizada para la base de datos SQLite.

Si su aplicación necesita admitir no solo la base de datos SQLite sino también otras bases de datos como MySQL, PostgreSQL y Oracle, PySQLite es una buena opción.

PySQLite es parte de la biblioteca estándar de Python desde la versión 2.5.
En este curso usaremos Sqlite.

Conexión a la base de datos

Al realizar la conexión, **si la base de datos no existe, entonces la crea**. Siempre debe cerrarse la conexión al finalizar el uso, sino luego no se podrá seguir manipulandola.

```
import sqlite3 # Importamos el modulo

# Conexion a la DB
conexion = sqlite3.connect('ejemplo.db')
conexion.close() # Cerramos la conexion
```

Crear una tabla

Antes de ejecutar una consulta (**query**) en código **SQL**, tenemos que crear un cursor.

cursor: es un objeto que ayuda a ejecutar la consulta y obtener los registros de la base de datos. El cursor juega un papel muy importante en la ejecución de la consulta.

```
# Importamos el modulo
import sqlite3
# Conexion a la DB
conexion = sqlite3.connect('ejemplo.db')
# Creamos el cursor
cursor = conexion.cursor()

# Creamos una tabla de usuarios con nombres, edades y emails
cursor.execute("CREATE TABLE IF NOT EXISTS usuarios(nombre
VARCHAR(100),edad INTEGER,email VARCHAR(100))")
# Cerramos la conexion a la DB
conexion.close()
```

Inserción o carga de datos en una tabla

Cargar un registro de datos: **INSERT INTO/VALUES**

```
import sqlite3

conexion = sqlite3.connect('ejemplo.db')
cursor= conexion.cursor()

#Insertamos un registro en la tabla usuarios
cursor.execute("INSERT INTO usuarios VALUES
('Hector',27,'hector@ejemplo.com')")

# Guardamos los cambios haciendo un commit
conexion.commit()
conexion.close()
```

Cargar varios registros

Insertando varios registros con **.executemany()**:

```
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()
usuarios = [('Mario',51,'mario@ejemplo.com' ),
( 'Mercedes' ,38,'mercedes@ejemplo.com' ),
( 'Juan',19,'Juan@ejemplo.com' )]

# Ahora utilizamos el metodo executemany() para insertar varios
cursor.executemany ("INSERT INTO usuarios VALUES(?,?,?)", usuarios)

# Guardamos los cambios haciendo commit
conexion.commit()
conexion.close()
```

Lectura de datos en una tabla

Lectura de una tupla Recuperando el primer registro con **.fetchone()**:

```
# Lectura de una tupla
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

# Recuperamos los registros de la tabla usuario
cursor.execute("select * from usuarios" )

# Recorremos el 1er registro con fetchone que devuelve una tupla
usuario = cursor.fetchone()
conexion.close()
```

Lectura multiple

Recuperando varios registros con **.fetchone()**:

```
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

# Recuperamos los registros de la tabla usuarios
cursor.execute("select * from usuarios" )

# Recorremos los registros con fetchone dentro de un while
usuario = cursor.fetchone()

while usuario:
    print("Un usuario: " , usuario)
    usuario = cursor.fetchone()
conexion.close()
```

Lectura multiple

Recuperando varios registros con **.fetchall()**:

```
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

cursor.execute("select * from usuarios" )

# Recorremos los registros con fetchall y se devuelve en una lista
usuarios = cursor.fetchall()

for usuario in usuarios:
    print("Un usuario: " , usuario)
conexion.close()
```

Actualizar/Modificar un dato en una tabla

Indicamos cuál campo queremos modificar, con qué valor lo haremos y a cuáles filas afectará el cambio: **UPDATE, SET, WHERE**

```
# Actualizamos un registro
import sqlite3
conexion = sqlite3.connect('ejemplo.db')
cursor = conexion.cursor()

# Actualizamos un registro de la tabla usuario
# Sintaxis SQL: UPDATE <tabla> SET <col1=val1>, <col2=val2>, ... WHERE
<condicion>
cursor.execute("UPDATE usuarios SET edad=80 WHERE nombre ='Juan'")
conexión.commit()
conexion.close()
```

Eliminar un registro de una tabla

Indicamos cuál registro/fila queremos eliminar de acuerdo a alguna condición de alguno de los campos:

```
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

# Eliminamos un registro de la tabla usuarios
cursor.execute("DELETE FROM usuarios WHERE nombre='Juan'")

conexion.commit()
conexion.close()
```

Mostrar resultado ordenado

Con **SELECT** obtenemos todos los registros de la tabla y con **ORDER BY** los ordenamos por edad de **menor a mayor**:

```
# Traer registros ordenados por edad ascendente
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

cursor.execute("SELECT * FROM usuarios ORDER BY edad")
personas = cursor.fetchall()
for persona in personas:
    print(persona)
conexion.close()
```

Mostrar resultado ordenado

DESC nos permite ordenar en **orden descendente**:

```
# Traer registros ordenados por edad descendente
import sqlite3
conexion = sqlite3.connect('ejemplo.db' )
cursor = conexion.cursor()

cursor.execute("SELECT * FROM usuarios ORDER BY edad DESC")
Personas = cursor.fetchall()
for persona in personas:
    print(persona)
conexion.close()
```

COUNT

COUNT() nos permite contar cuantas veces se repite una búsqueda. Por ejemplo, en el código siguiente, cuenta la cantidad nombres guardados:

```
# Count
import sqlite3
conexion= sqlite3.connect('ejemplo.db')
cursor= conexion.cursor()

def cuenta_nombre():
    cursor.execute(f"SELECT COUNT (nombre) FROM usuarios")
    Salida = cursor.fetchone() # Devuelve una tupla
    return salida[0] # Retorno sólo la cantidad

print(f"Actualmente existen {cuenta_nombre()} usuarios")
```

WHERE

WHERE nos permite filtrar los resultados según el parámetro que necesitemos usar:

```
def cuenta_nombre (nombre):
    cursor.execute(f"SELECT COUNT (nombre) FROM usuarios WHERE
                           Nombre = '{nombre}'")
    return cursor.fetchone()[0]

nombre = "Juan"
print(f"EL nombre {nombre} se repite {cuenta_nombre (nombre)} veces")
```

LIKE y %

LIKE se puede utilizar con los caracteres **comodín**, como por ejemplo el **%** que reemplaza a cualquier combinación de caracteres:

```
def empieza_con (inicio):
    cursor.execute(f"SELECT nombre, edad,email FROM usuarios
                           WHERE nombre LIKE '{inicio}%' ")
    return cursor.fetchall()

Inicial = "M"
print(f"Los nombres que inicial con {inicial} son:")
Personas = empieza_con(inicial)
for persona in personas:
    print(persona)
```