Informática 1º Bto

Bloque C: Programación

Programando en Python

IES Fco. Grande Covián
Informática I - version 3.0.1

1. Introducción

Un ordenador es un sistema capaz de realizar cálculos y tomar decisiones lógicas con una velocidad y precisión extraordinarias.

Desde un punto de vista funcional, un ordenador se compone de dos elementos fundamentales:

- **Hardware**: la parte física del sistema, formada por los componentes electrónicos y mecánicos que permiten su funcionamiento.
- **Software**: el conjunto de datos e instrucciones que el hardware procesa y almacena para ejecutar diversas tareas.

El funcionamiento de un ordenador se basa en la ejecución de instrucciones organizadas en secuencias llamadas **programas informáticos**.

> **Programa informático**: Conjunto de instrucciones codificadas que determinan el comportamiento del sistema para resolver un problema concreto.

Del mismo modo que un texto puede redactarse en distintos idiomas (español, inglés, etc.), los programas informáticos pueden desarrollarse en distintos **lenguajes de programación**, cada uno con su propia sintaxis y sus propias reglas.

En esta unidad, estudiaremos el lenguaje **Python**, una herramienta versátil y ampliamente utilizada, con el objetivo de aprender a desarrollar programas capaces de resolver problemas sencillos de forma estructurada y eficiente.

2. Algoritmos

> **Algoritmo**: Secuencia ordenada de pasos que permite resolver un problema o realizar una tarea de forma sistemática.

Un algoritmo se parece a una **receta de cocina**. Una receta dice cómo preparar un plato siguiendo una serie de pasos.

Por ejemplo, para preparar un pastel, los pasos a seguir podrían ser: Precalentar el horno, mezclar la harina con el azúcar y los huevos, poner la mezcla en un molde... y así sucesivamente.

De forma análoga, un algoritmo indica al ordenador qué pasos debe seguir para obtener un resultado.

Características de un algoritmo

Todo **algoritmo** debe cumplir ciertas características esenciales:

1. **Claridad y precisión**:

   Cada paso debe estar definido sin ambigüedades. Por ejemplo, ***"cocinar hasta que el queso burbujee"*** es más preciso que ***"cocinar hasta que esté hecho"***.

2. **Definición de entradas (inputs)**:

   Debe especificar el tipo de datos que recibirá, como números positivos, palabras o listas de valores.

3. **Generación de salidas (outputs)**:

   Debe producir uno o varios resultados bien definidos, como el número mayor entre dos valores o una lista ordenada.

4. **Finitud**:

   Debe concluir en un número finito de pasos y proporcionar un resultado en un tiempo razonable.

5. **Corrección**:

   Debe garantizar resultados correctos en todas las situaciones previstas.

6. **Precondiciones**:

   Si el algoritmo requiere ciertas condiciones sobre los datos de entrada (por ejemplo, aceptar solo números positivos), estas deben cumplirse para que el algoritmo funcione correctamente.

El diseño de un algoritmo correcto y eficiente es clave en programación, ya que define la lógica para resolver un problema de manera estructurada.

> **Ejemplo**: Dada una lista de números positivos, devuelve el número mayor de la lista.

- **Entradas (Inputs)**: Una lista L de números positivos, debe contener al menos un número.
- **Salidas (outputs):** Una 'caja' (variable) que llamaremos max, almacenará el número mayor de la lista.
- **Algoritmo:**
  - Asignar a max el valor 0.
  - Para cada número x en la lista L, comparar su valor con max. Si x es mayor que max, asignar a max el valor x.
  - Mostrar el valor almacenado en max que contiene el valor máximo en la lista.

¿Cumple esta "receta" los requisitos para ser un algoritmo?

1. **Claridad y precisión:** ✅ Sí. Cada paso está definido de manera inequívoca y puede traducirse fácilmente a código (en nuestro caso, Python).
2. **Definición de entradas:** ✅ Sí. Se especifican los datos que recibe.
3. **Definición de salidas:** ✅ Sí. Se establece claramente el resultado esperado.
4. **Finitud:** ✅ Sí. La lista ***L*** tiene un tamaño finito, por lo que, tras recorrer todos sus elementos, el algoritmo finalizará.

5. **Corrección:** ✅ Sí. Aunque en un entorno más riguroso se requeriría una demostración formal para garantizar su validez en todos los casos.
6. **Precondiciones**:✅ Sí. Fijamos que los números introducidos como valores de entrada serán positivos.

3. Diagramas de flujo

La representación de algoritmos únicamente mediante bloques de texto puede resultar tediosa y difícil de interpretar, especialmente para personas distintas de quien los ha diseñado. Por este motivo, se utilizan representaciones gráficas conocidas como

> **Diagrama de flujo**: Representación gráfica de un algoritmo mediante símbolos.

Dentro de cada símbolo se escriben las acciones que deben realizarse en cada paso del algoritmo.

Los diagramas de flujo son herramientas muy importantes porque permiten **diseñar la estructura de un programa sin necesidad de dominar un lenguaje de programación concreto**. Por ello, la elaboración del diagrama suele considerarse la **primera fase del desarrollo de un programa**.

Además, un diagrama de flujo facilita que otras personas puedan comprender el funcionamiento del algoritmo de forma rápida y clara.

La principal desventaja de los diagramas de flujo es el **espacio necesario** para su representación. No obstante, actualmente existen numerosas aplicaciones y herramientas online que permiten crearlos de manera sencilla y eficiente.

3.1 Elementos en un Diagrama de Flujo

Un diagrama de flujo se construye a partir de una serie de símbolos, cada uno con un significado específico:

Terminal (Inicio/Fin)

Indica el comienzo y el final del diagrama de flujo.	 **Inicio / Fin**

Datos (Entrada/Salida)

Se utiliza para introducir valores en el algoritmo (entrada) o para mostrar resultados (salida). Habitualmente se emplean las palabras leer (entrada) y escribir (salida).	 **Entrada / Salida de Datos**

Proceso

Representa una operación o acción a ejecutar, como una asignación de valores o un cálculo.	 **Proceso**

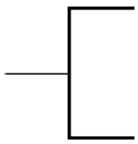
Decisión (Condicional)

Permite evaluar una expresión lógica y tomar uno de dos caminos posibles, normalmente ***Sí/No*** o ***Verdadero/Falso***.	 **Condicional**

Líneas de flujo (Flechas)

Indican la dirección y el orden en el que se ejecutan los pasos del algoritmo.	 **Flecha**

Anotación

Se utiliza para añadir aclaraciones o explicaciones sobre pasos complejos del algoritmo.	 **Anotación**

Entrada Manual (Input)

Representa la introducción de datos mediante el teclado. Su forma recuerda a la de un teclado.	 **Entrada de Datos Manual**

Salida por pantalla

Indica la generación o de un documento resultado del proceso.	 **Salida por pantalla**

Salida de documento

Indica la generación o de un documento resultado del proceso.	 **Documento**

El Conector de Página (Símbolo de Salto)

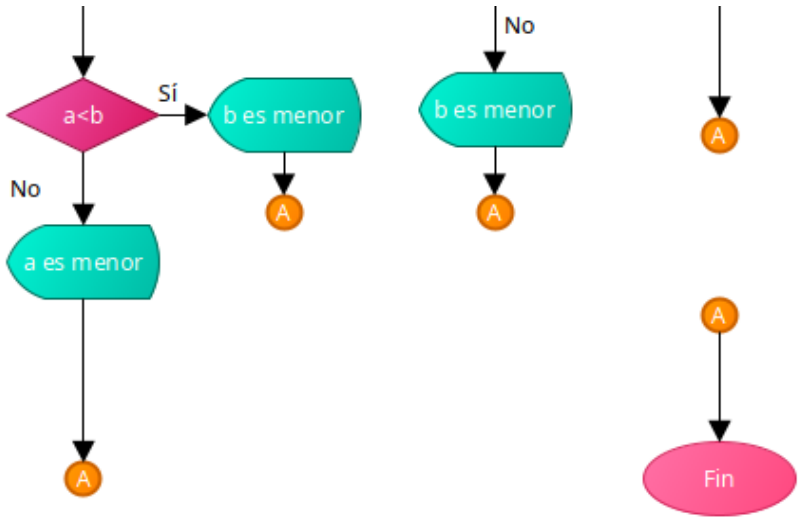
Para mantener la claridad visual en diagramas de flujo complejos, se utilizan **conectores circulares** etiquetados con una letra (comúnmente la "**A**").

Estos actúan como "nodos de transferencia" que permiten unificar múltiples rutas de salida sin necesidad de trazar líneas largas o cruces caóticos que ensucien el diseño.

En lugar de dirigir cada resultado individualmente hacia el terminal de **Fin**, se coloca un conector de salida en cada rama lógica; todos estos "teletransportadores" convergen virtualmente en un único conector de entrada situado justo antes del cierre del algoritmo.

Este recurso no solo mejora la estética profesional y el orden del diagrama, sino que facilita su escalabilidad y lectura técnica al evitar la acumulación de ángulos rectos innecesarios.

A modo de ejemplo:



3.2 Reglas para su elaboración

Para que un diagrama de flujo sea claro y correcto, debe cumplir las siguientes reglas:

1. El diagrama debe tener un **único inicio y fin**.
2. Todos los símbolos deben estar **conectados** de manera clara.
3. Un **proceso** puede tener varias entradas.
4. Un **decisión** debe tener dos salidas claramentes identificadas (***Sí/No*** o ***Verdadero/Falso***).
5. El **inicio** no recibe entradas.
6. El **fin** no tiene salidas.
7. Las **líneas de conexión** deben ser rectas y evitar cruces innecesarios
8. El flujo debe organizarse **de arriba hacia abajo y de izquierda a derecha**.
9. Se debe evitar el uso de **terminos propios de lenguajes de programación**.
10. Es recomendable incluir **anotaciones** para facilitar la comprensión.
11. Cada bloque debe recibir las entradas por **arriba** o por la **izquierda** y producir las salidas por **abajo** o a la **derecha**.
12. Si el diagrama ocupa varias hojas, estas deben **numerarse** para mantener el orden.

Ventajas del uso de diagramas de flujo

- Facilitan la comprensión de los algoritmos, ya que el cerebro humano interpreta fácilmente la información visual.
- Permiten detectar errores, redundancias, conflictos o cuellos de botella.
- Son una herramienta muy útil para el aprendizaje de la programación.

Desventajas del uso de diagramas de flujo.

- A medida que aumenta la complejidad del algoritmo, el diagrama puede volverse extenso y difícil de seguir.

3.3 Cómo elaborar un diagrama de flujo

Para crear un diagrama de flujo, el primer paso consiste en ***definir correctamente el algoritmo***, es decir, establecer de forma clara y ordenada todos los pasos necesarios para resolver el problema.

Una vez definido el algoritmo, se representará gráficamente mediante los símbolos propios del diagrama de flujo.

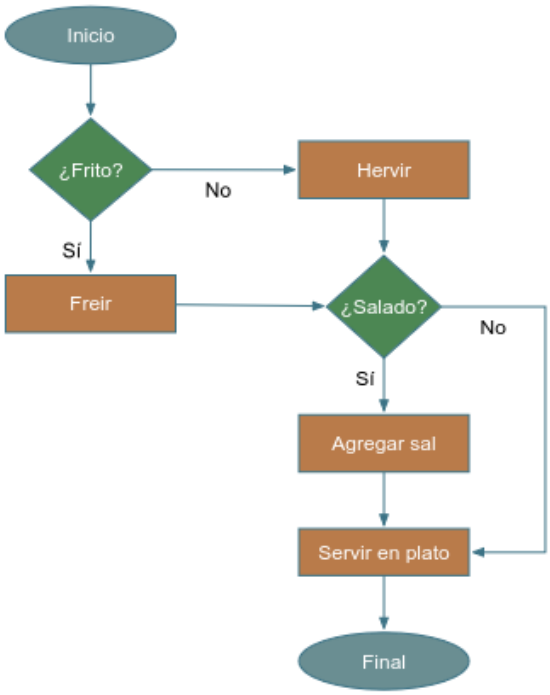
> **Ejemplo:** Elaboración del algoritmo y el diagrama de flujo para cocinar un huevo según las preferencias otra persona:

Algoritmo (descripción en lenguaje natural):

1. Preguntar si la persona quiere el huevo frito.
2. Si la respuesta es ***sí***, freír el huevo.
   Si la respuesta es ***no***, hervir el huevo.
3. Una vez cocinado, preguntar si desea añadir sal.
4. Si la respuesta es ***sí***, añadir sal al huevo.
5. Servir el huevo en el plato.

Diagrama de flujo:

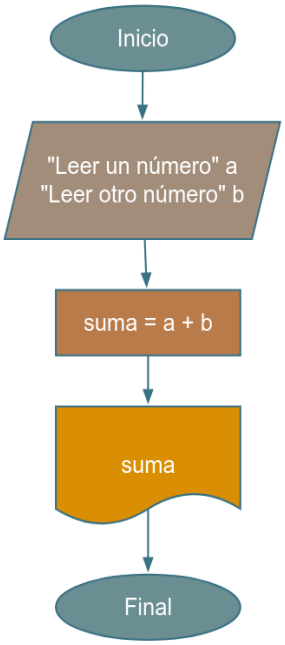
Una vez definido el algoritmo con todos los pasos y decisiones, se elabora un **esquema gráfico** que represente visualmente el flujo del proceso. Este esquema constituye el **diagrama de flujo**.



3.4 Ejemplos de Diagramas de Flujo

Diagrama de flujo que solicita dos números, suma sus valores y muestra el resultado por pantalla.

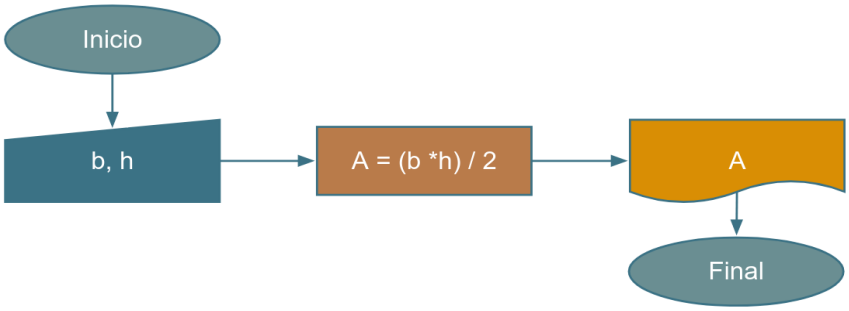
Para resolver este problema, el flujo lógico se divide en cuatro etapas fundamentales:



1. **Inicio y Fin:** Representados por óvalos (terminales), marcan el punto de partida y la conclusión del algoritmo.
2. **Entrada de Datos:** Se utiliza el símbolo de entrada (paralelogramo) para capturar los dos números proporcionados por el usuario.
3. **Procesamiento:** Un rectángulo indica la operación aritmética donde se ejecuta la suma: Resultado=A+B.
4. **Salida de Resultados:** Se utiliza el símbolo de impresión o pantalla para mostrar el valor final obtenido.

Calcular el área de un triángulo y muestra el resultado por pantalla.

1. **Inicio y Fin**: Los terminales que delimitan el proceso.
2. **Entrada de Datos**: Símbolo de entrada para capturar los valores de la Base (b) y la Altura (h).
3. **Procesamiento**: Un rectángulo donde se realiza el cálculo matemático: Area=(b∗h)/2.
4. **Salida de Resultados**: Símbolo de pantalla que muestra el valor de la variable Area.
5. En este caso ni siquiera hemos puesto las operaciones dentro de los símbolos de entrada y de salida, la forma del símbolo las define.



Determinar si un número es par o impar

Este flujo introduce una bifurcación lógica basada en una condición:

1. **Inicio y Fin**: Delimitan el inicio y el cierre del análisis.
2. **Entrada de Datos**: Se solicita al usuario que ingrese un Número (N).
3. **Decisión**: Un rombo evalúa la operación N(mod2)=0. Esta operación (módulo) verifica si el resto de dividir el número entre 2 es cero.
4. **Salida de Resultados (Dos Caminos)**: * Si la respuesta es SÍ, se muestra en pantalla "Es par".

   Si la respuesta es NO, se muestra en pantalla "Es impar".

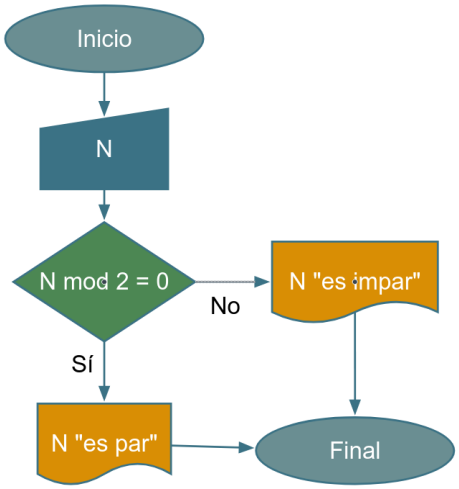
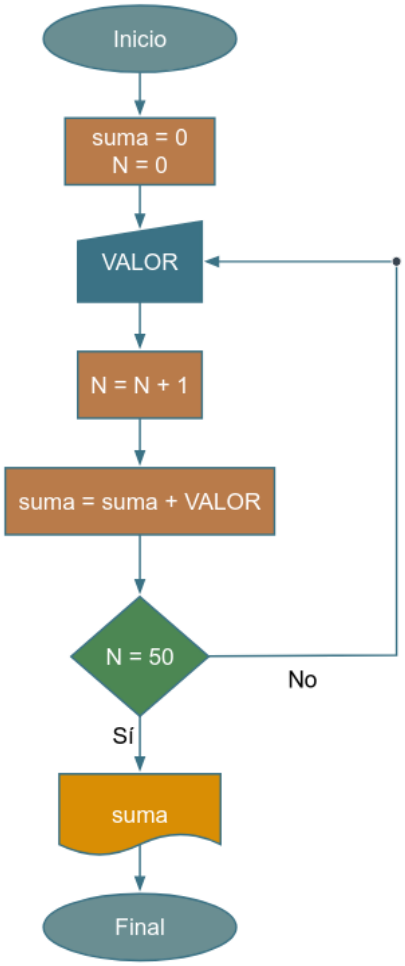


Diagrama de flujo para mostrar la suma de los 50 primeros números.

Este es un ejemplo de bucle o ciclo, donde el programa repite acciones hasta cumplir un objetivo:

1. **Inicio y Fin**: Terminales del programa.
2. **Inicialización** (Procesamiento): Antes de empezar, definimos dos variables: la Suma (S=0) y el Contador (N=0).
3. **Procesamiento** (Incremento): Se le suma 1 al contador (N=N+1) y se acumula en la suma (S=S+N).
4. **Decisión** (Bucle): Un rombo pregunta: ¿Es N=50?
   1. Si NO es 50, el flujo regresa al paso anterior para seguir sumando.
   2. Si SÍ es 50, el flujo continúa hacia la salida.
5. **Salida de Resultados**: Se muestra el valor acumulado en S.



4. Tipos de lenguajes programación

Aunque existen cientos de lenguajes de programación, todos se pueden clasificar en tres grandes categorías según su cercanía al hardware o al lenguaje humano:

1. Código máquina.
2. Lenguaje ensamblador
3. Lenguajes de alto nivel.

Código máquina

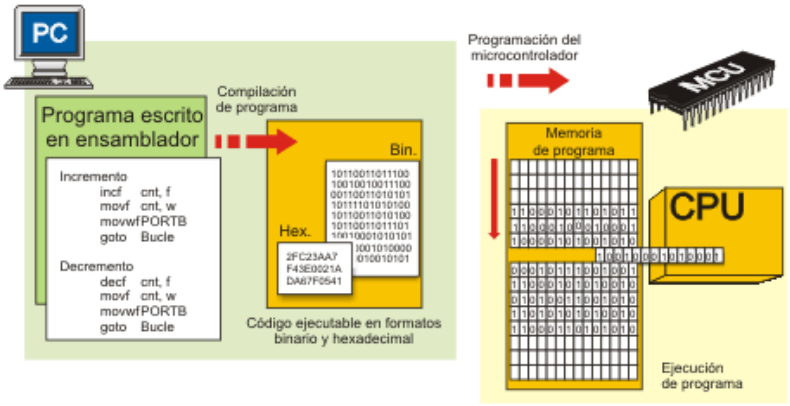
Es el único lenguaje que el ordenador puede interpretar de forma directa.

1. **Composición**: Está formado exclusivamente por secuencias de 0s y 1s (sistema binario) que indican al procesador exactamente qué operaciones ejecutar.
2. **Dependencia**: Cada procesador tiene su propio código máquina, lo que lo hace totalmente dependiente del hardware.
3. **Dificultad**: Para un ser humano es prácticamente incomprensible y su uso es extremadamente lento, ya que requiere detallar cada instrucción minúscula de forma manual.

Lenguaje ensamblador

Surgió para simplificar la programación en los primeros ordenadores, que antes se hacía directamente en código máquina.

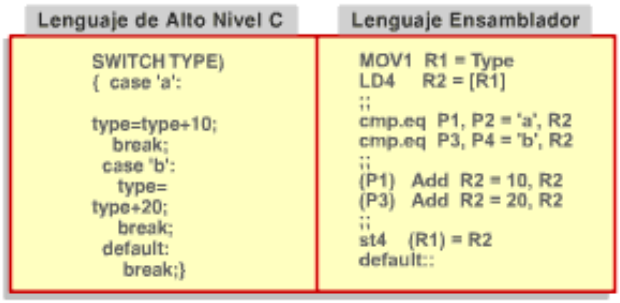
1. **Funcionamiento**: Sustituye los 0s y 1s por abreviaturas y palabras clave (generalmente en inglés), como ADD para sumar o MOV para mover datos.
2. **El proceso de traducción**: Aunque es más fácil de leer para nosotros, el ordenador no puede entenderlo directamente. Por ello, se necesita un programa llamado **Ensamblador**, que traduce estas instrucciones a código máquina.
3. **Limitación**: Al igual que el anterior, sigue dependiendo del tipo de hardware específico para el que se escribe.



Lenguajes de alto nivel y compiladores

Son los lenguajes modernos que permiten escribir código de forma intuitiva, similar al lenguaje humano y a la notación matemática.

1. **Eficiencia**: Una sola instrucción en un lenguaje de alto nivel puede equivaler a decenas de líneas de código ensamblador, lo que ahorra mucho tiempo.
2. **Universalidad**: A diferencia de los anteriores, no suelen depender de un hardware específico; el mismo programa puede funcionar en diferentes ordenadores.



Intérpretes y compiladores

Para que un programa de alto nivel pueda funcionar, debe ser traducido a código máquina. Existen dos métodos principales para realizar esta tarea:

- **Intérpretes**: Traducen y ejecutan el código línea por línea en tiempo real. Su ventaja es que permiten probar el código rápidamente mientras se escribe, pero la ejecución es más lenta porque la traducción se repite cada vez que iniciamos el programa.
- **Compiladores**: Traducen el código completo de una sola vez y generan un archivo independiente (ejecutable). Esto hace que el programa final

sea mucho más rápido, pero obliga al programador a volver a traducir (recompilar) todo el proyecto cada vez que realiza un cambio.

En este curso nos enfocaremos en **Python**, un lenguaje de alto nivel muy versátil que utiliza un sistema **interpretado**, lo que lo hace ideal para aprender a programar de forma ágil.

5. Ejecutar un programa Python

¿Qué es Python?

Python es un **lenguaje de alto nivel** diseñado para ser fácil de leer y escribir. Su sintaxis clara y sencilla lo hace ideal tanto para principiantes como para programadores experimentados.

Es un lenguaje **multiparadigma**, lo que significa que admite diferentes estilos de programación, como la programación estructurada, la programación orientada a objetos y la programación funcional.

Python es un lenguaje interpretado

A diferencia de los lenguajes compilados, Python no traduce el código fuente directamente a código máquina antes de ejecutarlo. En su lugar, utiliza un **intérprete**, que lee y ejecuta el código línea por línea.

> **Ventaja**: Si el programa falla, se detiene exactamente en la línea donde está el error, lo que facilita mucho la corrección (depuración).

5.1 Ejecución desde el punto de vista del programador (lo que tú haces)

5.1.1 Crear el código

La forma más sencilla de programar en Python es escribir un archivo de texto con las órdenes a ejecutar, lo que se conoce como **código fuente**.

> **Código fuente**: Conjunto de instrucciones que le indican al ordenador qué debe hacer.

- En Python, los archivos de código fuente deben tener la extensión **.py**.
- Ejemplo de código fuente:

```
print("Hola")
print(2 ** 100)
```

- Para escribir código en Python, se puede utilizar cualquier editor de texto. Una vez guardado el archivo con extensión **.py**, es necesario indicar al intérprete de Python que lo ejecute.

5.1.2 Ejecutar el módulo .py

El archivo .py que acabamos de crear puede ejecutarse de varias formas diferentes:

Desde la línea de comandos

En Windows:

1. Abre una ventana de comandos presionando Window+R, escribe cmd y pulsa Enter.
2. Usa el comando cd para navegar hasta la carpeta donde se guardó el archivo.
3. Escribir el siguiente comando y presionar enter:

```
python nombreArchivo.py
```

4. El resultado se mostrará por pantalla.

En Linux Mint:

1. Abre una terminal con Ctrl+Alt+T o desde el menú de aplicaciones.

2. Usa cd para moverte a la carpeta donde está el archivo.
3. Ejecutar el programa con:

```
python3 nombreArchivo.py
```

4. Verás la salida directamente en la termianl.

Ejecutar con doble clic en el archivo

- En **Windows**, hacer doble clic sobre el archivo .py lo ejecutará, pero si el programa termina rápidamente, la ventana se cerrará antes de ver el resultado.
- En **Linux**, es posible hacer doble clic si el archivo tiene permisos de ejecución (chmod +x nombreArchivo.py), aunque en muchos casos se abrirá en un editor en lugar de ejecutarse.

Desde un entorno de desarrollo (IDE)

Python puede ejecutarse desde entornos como: **IDLE, SublimeText, PyCharm, VS Code**

5.2 Ejecución desde el punto de vista del intérprete (lo que hace Python)

Cuando se ejecuta un programa en Python, el intérprete realiza un trabajo interno invisible en dos pasos:

5.2.1 Compilación a "bytecode".

Python traduce tu código fuente a un lenguaje intermedio llamado **Bytecode**.

¿Qué es el bytecode?

- Es un conjunto de instrucciones más simples y cercanas al lenguaje máquina.
- Es **multiplataforma**, lo que significa que puede ejecutarse en cualquier ordenador con la versión de Python correspondiente.

Archivos .pyc y la caché de compilación

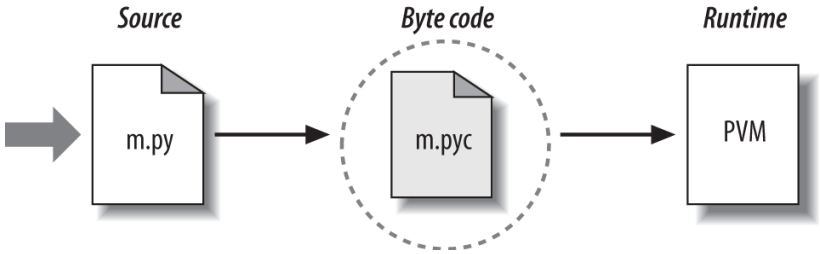
- Si Python tiene permisos de escritura, el bytecode se guarda en un archivo con extensión. pyc dentro de un subdirectorio llamado _pycache_.
- Esto mejora el rendimiento al evitar recompilar el código en cada ejecución.
- Si el código fuente cambia o la versión de Python es diferente, se generará un nuevo bytecode y en su caso archivo .pyc.

> **Nota:** Si ejecutamos Python desde el terminal sin permisos de escritura, no se crea el archivo .pyc, el código se compilará en memoria en cada ejecución, lo que ralentiza el proceso.

5.2.2 Ejecución en la Máquina Virtual de Python (PVM)

Una vez generado y cargado en memoria, el bytecode se envía a la **Máquina Virtual de Python (PVM)**, que se encarga de interpretar y ejecutar estas instrucciones.

PVM: Parte del intérprete que simula el funcionamiento de un ordenador, permitiendo que el bytecode se ejecute de forma independiente de las características específicas del sistema operativo o del hardware.



6. Utilizar Python desde la línea de comandos

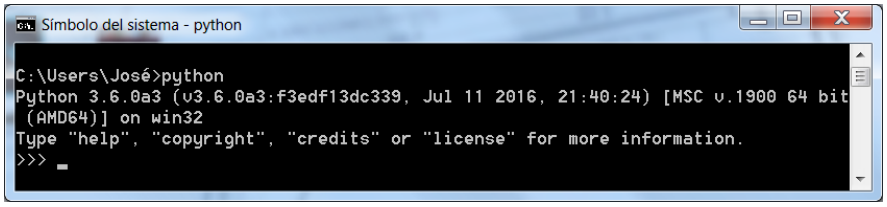
Comenzaremos a trabajar con Python de la forma más sencilla: a través de la **línea de comandos**. En este modo, utilizaremos el intérprete de manera similar a una calculadora, lo que nos permitirá introducir instrucciones y evaluar sus resultados de forma inmediata.

Cómo acceder al intérprete

Para abrir nuestra sesión de trabajo, seguiremos estos pasos según nuestro sistema operativo:

En Windows:

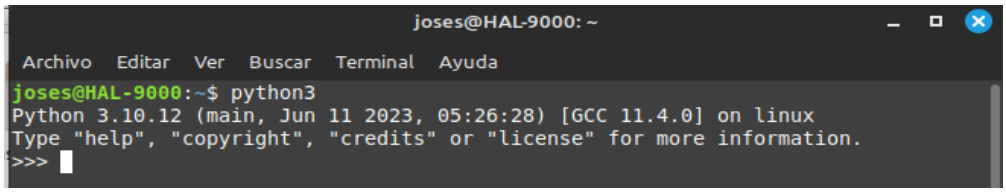
Abrimos la consola (CMD) y escribimos la palabra python. También podemos acceder directamente seleccionando "Python" desde el menú de



inicio.

En Linux:

Abrimos una ventana de terminal y escribimos el comando python3.



La sesión interactiva y el "prompt"

En cuanto veamos los iconos >>>, sabremos que estamos en una **sesión interactiva** del intérprete. Este símbolo nos indica que Python está esperando nuestras órdenes.

A esta forma de trabajar la denominamos **crear código interactivamente**. Su funcionamiento es muy directo: cada vez que pulsamos **Enter**, Python

ejecuta la orden y nos muestra el resultado automáticamente (en este modo, no necesitamos usar la función print para ver el valor de una operación).

A esta forma de trabajar, escribiendo algo de código en la ventana y ejecutarlo directamente se le llama ***crear código interactivamente***. Cada vez que se pulsa enter, se muestra el resultado de la orden ejecutada (ni siquiera haría falta poner print).

```
>>>pera = 'okay'
>>>pera
'okay'
>>>2 ** 8
256
```

> El hecho de que al teclear python (o python3) en la venanta de comandos, el ordenador sea capaz de ejecutarlo se debe a que el sistema operativo es capaz de encontrar donde está instalado Python por haberlo incluido esta ruta en su path de búsqueda de programas cuando hemos realizado la instalación.

Limitaciones

Esta forma de trabajo **no permite guardar código**, normalmente no la utilizaremos, pero es útil para experimentar con el lenguaje sin riesgo a estropear cosas en el programa real.

> Cuando utilizamos una orden que requiere más de una línea de código (ordenes compuestas) es necesario hacer dos veces clic en Enter para ejecutarla.

Vamos a comenzar a utilizar algunos sencillos comandos de Python. Para comprender su funcionamiento utilizaremos una sesión interactiva a través de la consola.

> Salir de la sesión interactiva: Ejecuta la orden exit()

7. Tipos de datos

Python puede manejar los siguientes tipos de datos:

Numéricos: Pueden ser:

- Enteros: tipo **int**
- Coma flotante: tipo **float**, números decimales
- Complejos: no los veremos

Textos: Será dato tipo texto, **string**, cualquier información encerrada entre comillas (simples, dobles o triples (permiten utilizar saltos de línea dentro del dato tipo texto)).

Booleanos: Solo puede tomar dos valores True y False.

8. Operadores. Trabajando con números

Podemos utilizar la consola para realizar cálculos. Los operadores matemáticos en Python son:

Operador	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
División entera	//
Resto de la división	%
Potencia	**
Paréntesis	()
Redondear *	round(a,b)

*round(a,b) redondearía el número a con b decimales (ej. round(3.14, 1) generaría 3.1.

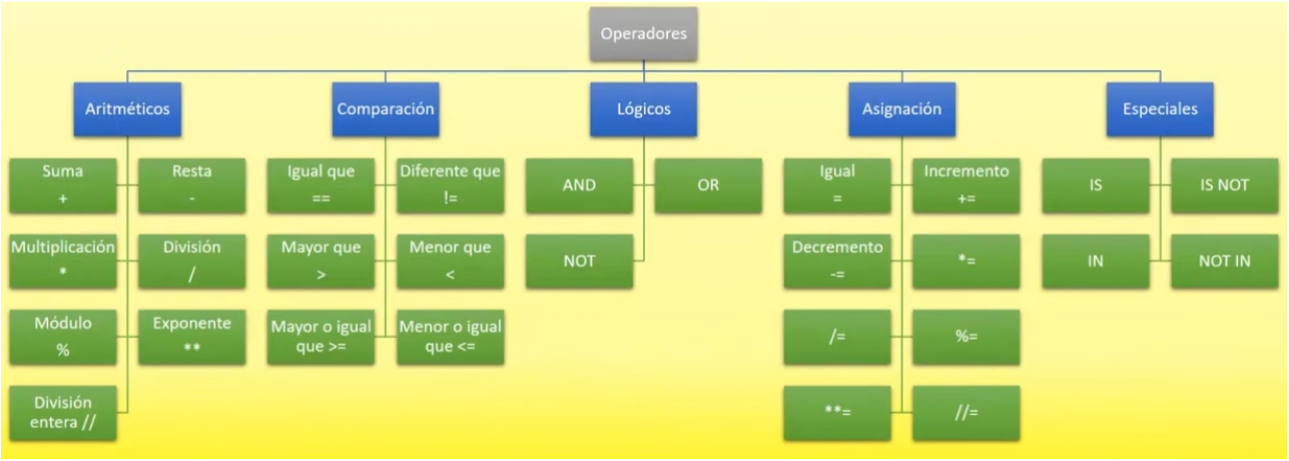
> **Ojo**: El punto decimal se indica por medio del símbolo punto (.) y no con la coma (,).

> **Ojo**: Trabajando con datos tipo string, el operador + actua como concatenador.

Una vez que pulsemos Enter, tras introducir la expresión matemática, Python mostrará dos nuevas líneas, la primera (sin el prompt >>>) contendrá el resultado y la siguiente(que comenzará con el prompt >>>) quedará a la espera de que introduzcamos una nueva orden.

```
# Abre una sesión interactiva y realiza los siguientes cálculos:
>>> 2 + 2
>>> 50 - 5*6
>>> (50 - 5*6) / 4
>>> 8 / 5 # la división siempre retorna un número de punto flotante
>>> 17 // 3 # la división entera descarta la parte fraccional
>>> 17 % 3 # el operador % retorna el resto de la división
>>> 2 ** 7 # 2 a la potencia de 7
```

La siguiente imagen muestra los operadores aritméticos vistos en la tabla anterior más otra serie de operadores que veremos más adelante.



9. Comenzando a utilizar variables

Variable: Nombre que se utiliza para hacer referencia a un valor almacenado en la memoria del ordenador y que es utilizado por el programa.

Al utilizar el nombre de una variable dentro del código, accederemos al valor al que la variable hace referencia.

Normas para dar nombres a las variables

- El nombre debe empezar por una letra o por un guion bajo (_).
- El nombre no puede tener espacios en blanco.
- Es recomendable evitar utilizar caracteres no ingleses (ñ, tildes...)
- Python distingue entre mayúsculas y minúsculas (case sensitive).

Interesa utilizar nombres de variables que tengan relación con el dato almacenado.

En general, los nombres de variables deben estar en minúsculas.

Si el nombre de la variable consta de varias palabras, se recomienda separarlas con guiones bajos (_). Por ejemplo: nombre_variable, edad_persona.

Asignar un dato a una variable

El signo igual (=) se utiliza dentro del código para asignar un dato a una variable.

Así la expresión n = 5 haría que se creara una variable (por ser la primera vez que se utilizaría en el código) con el nombre n y haría referencia al valor 5.

```
>>>ancho = 20
>>>largo = 5 * 9
>>>ancho * largo
900
```

En el código anterior:

- la primera línea crea una variable con el nombre ancho que almacenará el valor 20. Esta línea no genera un resultado por pantalla.

- La siguiente línea crea la variable de nombre largo y se almacena en ella el resultado de multiplicar cinco por nueve.

- La tercera línea de código realiza una multiplicación, pero no utiliza valores numéricos sino los nombres de las dos variables. La orden se leería "multiplica el valor almacenado en la variable ancho por el valor almacenado en la variable largo y muestra el resultado en una nueva línea".

Tipos de variable

Igual que los datos en Python pueden ser de muchos tipos, una variable será de un tipo determinado (int, float, string, boolean y muchos más.).

En otros lenguajes de programación antes de utilizar una variable por primera vez hay que definir su nombre y su tipo (generalmente a este proceso se le llama "declarar una variable").

En Python, el tipo de una variable será el mismo que el tipo del dato al que la asociemos. Observa estos dos ejemplos:

```
>>> ancho = 20
>>> ancho * 2
40
```

En la primera línea asignamos a la variable ancho el valor 20. Por lo tanto, la variable acho se crea como variable de tipo entero.

En la siguiente línea el operador * actúa sobre la variable ancho multiplicando su valor. Así el resultado de la operación es 40.

Sin embargo, si hacemos:

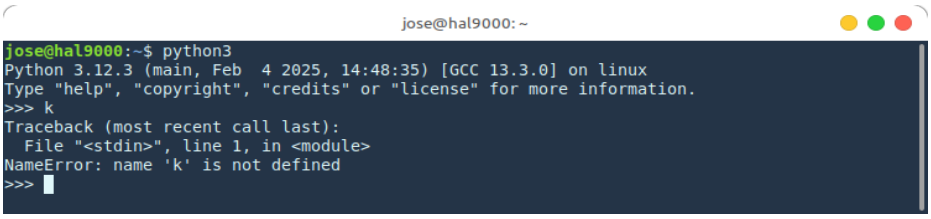
```
>>> ancho = "hola"
>>> ancho * 2
'holahola'
```

El resultado es diferente. Ancho pasa a ser una variable tipo texto. Con este tipo de variables (y de datos) el operador * no actúa como multiplicador, si no que se utiliza para ver cuantas veces es necesario representar la cadena de texto.

Las variables deben existir antes de ser utilizadas

En caso de que queramos mostrar el dato asociado a una variable que todavía no ha sido definida obtendremos un mensaje de error.

La siguiente pantalla muestra el resultado de hacer una llamada a la variable k que todavía no ha sido definida):



Las variables son objetos

En Python todos los elementos que utilizamos son "Objetos". Como veremos más adelante Python es un lenguaje orientado a objetos. Una variable Python es un objeto.

A modo de curiosidad (de momento) utiliza la función type() tal y como indica el ejemplo. Esta función mostrará el tipo (la clase) a la que pertenece ese objeto.

```
>>> type(ancho)
<class 'str'>
```

En este caso la variable "ancho" es un objeto tipo str (cadena de texto).

De momento no entenderás que quiere decir esto, pero más adelante será muy importante.

Asignaciones múltiples

Es posible asignar valores a varias variables en una única línea siguiendo sintaxis del tipo:

```
x, y, z = 7, "hola", 9.6
```

Constantes

Una constante es una variable cuyo valor no va a cambiar a lo largo del programa. Aunque las constantes no existen como tal en Python y en realidad funcionan como una variable más, se sugiere que su nombre esté formado por letras mayúsculas para diferenciarlas de las variables normales:

```
DIAS_SEMANA = 7
```

10. Cadenas de caracteres – función print()

Cadena de caracteres: Cualquier información que Python interprete como un texto.

Las cadenas de caracteres en Python serán ***objetos*** de la ***clase string***.

Python interpra como cadena de caracteres cualquier elemento contenido entre comillas simples ('...') o comillas dobles ("...").

Esto permite incluir dentro de nuestra cadena de caracteres cualquier de los dos iconos. Por ejemplo, para mostrar la palabra don't, no es posible crear una cadena de caracteres en la forma 'don't', en su lugar se utilizará la forma "don't".

Si utilizamos el intérprete interactivo para definir una cadena de caracteres obtendremos:

```
>>> 'hola que tal'
'hola que tal'
```

Es decir, se mostrará la cadena junto con los iconos que la limitan. Normalmente interesará mostrar por pantalla solamente el contenido de la cadena. Para ello utilizaremos la función ***print()***

print() Función que muestra por consola la información contenida entre los paréntesis, argumentos.

- La orden añade al final un salto de línea.
- Los argumentos podrán ser cadenas de caracteres, valores numéricos y/o variables.

Volviendo al ejemplo anterior, el resultado sería:

```
>>>print('hola que tal')
hola que tal
```

En el caso de trabajar con variables:

```
>>>c = 'hola que tal'
>>>print(c)
hola que tal
```

10.1 print() con más de un argumento

La función print admite varios argumentos seguidos.

- Si los argumentos están separados por comas, los separa con un espacio en blanco:

```
>>>print('Hola', 'adiós')
Hola adiós
```

- Si los argumentos no se separan con comas, no añade un espacio en blanco entre ellos:

```
>>>print('Hola''adiós')
Holaadiós
```

Si alguno de los argumentos a mostrar es una variable, es necesario utilizar la coma o el signo + (concatenación) tal y como veremos al estudiar la función input().

10.2 Opciones cadenas de caracteres y print()

Esta serie de opciones permiten dar un cierto formato a la información mostrada por pantalla:

\n – Retorno de carro

El modificador \ seguido de la letra n, dentro de una cadena de caracteres se interpreta como un retorno de carro.

```
>>>print('C:\algun\nombre')  # aquí \n significa nueva
línea!
C:\algun
ombre
```

r - Crear cadenas crudas

Si añadimos una r antes de la primera comilla del print(), los caracteres \ dejan de ser especiales y se muestran como un símbolo más:

```
>>>print(r'C:\algun\nombre')  # nota la r antes de la
comilla
C:\algun\nombre
```

\t - Tabulador

El modificador \t, dentro de una cadena de caracteres se interpreta como un tabulador.

```
print("1\t2\t3")
1       2       3
```

end='' - Evitar el salto de línea al final de la función print()

Al final añadimos el argumento end=''.

```
print('Hola', end='')
print('adiós')
Holaadiós
```

Concatenar cadenas de caracteres

El símbolo + permite unir cadenas de caracteres. Especialmente útil cuando se quieren separar cadenas largas en partes.

No se pueden concatenar cadenas y números con el operador suma. En ese caso será necesario convertir los números a cadenas con la función str().

Repetir cadenas de caracteres

El símbolo * permite repetir cadenas de caracteres:

```
>>>print(2 * 'ho' + 'la')
hohola
```

Ojo: Dos cadenas de caracteres yuxtapuestas son automáticamente consideradas como una sola cadena:

```
>>>print(2 * 'ho''la')
holahola
```

len() – Longitud cadena:

Función que devuelve la longitud de una cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>>len(s)
33
```

Líneas múltiples

Si dentro del print() definimos la cadena por medio de tres pares de comillas, los finales de línea incluidos en su interior son tenidos en cuenta, generando cadenas de más de una línea:

```
print("""
Uso: algo [OPTIONS]
-h Muestra el mensaje de uso
-H nombrehost Nombre del host al cual conectarse
""")
```

El código anterior genera la salida:

```
Uso: algo [OPTIONS]
-h Muestra el mensaje de uso
-H nombrehost Nombre del host al cual conectarse
```

10.3 Funciones de tipo f (f-strings)

Función asociada a print que permite dar un formato específico a la forma en que se presentará la información por pantalla.

Por ejemplo, para concatenar un valor de texto con una variable la expresión sería:

```
>>> variable=5
>>> print(f"El número es {variable}")
El número es 5
```

La f indica que se aplica una función de tipo f (formato). El mensaje ahora estará entre comillas como si fuera un único texto y el valor de la variable se mostrará al escribir entre llaves su nombre.

10.4 Índices en las cadenas de texto

Cada carácter de una cadena de texto queda referenciado por medio de un índice asociado a su posición en la cadena. Al primer carácter se le asigna el índice 0, al segundo el 1, al tercero el 2....

> **Ojo:** Esta opción es válida para cadenas de texto. Si trabajamos con números es necesario transformarlos previamente en una cadena por medio de la función str().

Característica muy útil cuando la cadena de caracteres está guardada en una variable. Al añadir el índice entre corchetes al nombre de la variable, se mostrará el carácter asociado al índice:

```
>>>variable = 'melocotón'
>>>print(variable[0]) # caracter en la posición 0
m
>>>print(variable[5]) # caracter en la posición 5
c
```

También es posible empezar a contar los caracteres desde la derecha. Para ello el carácter final queda referido con el índice −1, el anterior con el −2 y así sucesivamente.

```
>>>variable = 'melocotón'
>>>print(variable[-1]) # caracter en la posición -1
n
>>>print(variable[-9]) # caracter en la posición -9
m
```

"porción de listas de listas de texto"

Una porción de lista de texto es una porción de una cadena de caracteres con más de un carácter.

Para definir una porción de lista de texto utilizaremos la siguiente notación:

```
variable[a:b]
```

Donde variable es el nombre de la variable que contiene la cadena de caracteres, a es el índice del primer carácter incluido en la porción de lista y b el índice el primer carácter excluido de la porción de lista. Sería algo parecido a un intervalo [a,b) en matemáticas.

```
>>>variable = 'melocotón'
>>>print(variable[0:3]) #desde la posición 0 (incluida)
hasta la 3 (excluida)
mel
```

Esta forma de definir los intervalos permite dividir cadenas en dos partes de manera sencilla:

```
>>>variable = 'melocotón'
>>>print(variable[:3])
mel
>>>print(variable[3:])
ocotón
```

Observa que si no definimos el índice inicial de la porción de lista se toma por defecto el valor cero y si no definimos el índice final se toma la longitud total de la cadena.

11. Listas

Lista: Colección de datos. Cada uno de los datos indivuales queda definido dentro de la lista por un índice.

Definimos una lista agrupando los elementos entre corchetes separándolos entre sí por comas:

```
>>> cuadrados = [1, 4, 9, 16, 'palabra']
>>> cuadrados
[1, 4, 9, 16, 'palabra']
```

En el ejemplo creamos una lista y la asignamos a una variable a la que llamamos cuadarados.

Cada elemento de la lista pasa a estar identificado por su índice. El valor de cada índice será un número entero, asignado de forma correlativa y comenzando por el 0. En ejemplo anterior:

```
>>>cuadrados[0] # añadiendo un índice se muestra su valor
asociado
1
>>>cuadrados[-1] # índices negativos comienzan a contar
por la derecha (-1)
'palabra'
>>>print(cuadrados[-1])
palabra
>>>cuadrados[-3:] # porción de listas retornan una nueva
lista
[9, 16, 'palabra']
```

Modificar el valor almacenado en un elemento de la lista:

```
>>>cubos = [1, 8, 27, 65, 125] # hay algo mal aquí
>>> 4 ** 3 # el cubo de 4 es 64, no 65!
64
>>>cubos[3] = 64 # reemplazar el valor incorrecto
>>>cubos
[1, 8, 27, 64, 125]
```

En el ejemplo hemos modificado el valor almacenado en el cuarto elemento de la lista.

Modificar los valores almacenados en una porción de lista

Podemos modificar, borrar parcialmente, borrar completamente...

> **Ojo:** Recuerda que el intervalo está cerrado por la izquierda y abierto por la derecha.

```
>>>letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>>letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # reemplazar algunos valores
>>>letras[2:5] = ['C', 'D', 'E']
>>>letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # ahora borrarlas
>>>letras[2:5] = []
>>>letras
['a', 'b', 'f', 'g']
>>> # borrar la lista reemplazando todos los elementos por
una lista vacía
>>>letras[:] = []
>>>letras
[]
```

11.1 Opciones trabajando con listas

Añadir un nuevo elemento a la lista

El método append() permite añadir un nuevo elemento a la lista:

```
>>>cubos.append(216) # agregar el cubo de 6
>>>cubos.append(7 ** 3) # y el cubo de 7
>>>cubos
[1, 8, 27, 64, 125, 216, 343]
```

Aunque todavía no lo hemos estudiado, al ejecutar el método append() estamos utilizando programación orientada a objetos. Estamos ejecutando un método del objeto cubos que pertenece a la clase list.

Insertar un nuevo elemento a la lista

El método insert() permite insertar un nuevo elemento a la lista **en una posición específica**. Este método tendrá dos parámetros, el primero indica el índice asignado al nuevo elemento y el segundo el valor de ese elemento. Ten en cuenta que los índices de todos los elementos que tuvieran originalmente un valor mayor o igual al del nuevo elemento, se incrementan una unidad.

```
>>> cubos=[1, 8, 27, 64, 125, 216, 343]
>>> cubos[2]
27
>>> cubos.insert(2,"insertado")
>>> cubos
[1, 8, 'insertado', 27, 64, 125, 216, 343]
>>> cubos[2]
'insertado'
>>> cubos[3]
27
```

Eliminar un elemento de la lista por su índice

Utilizaremos el método pop. Indicaremos como parámetro el **valor del índice** del elemento a eliminar. El resto de la lista se actualizará.

```
>>> cubos=[1, 8, 27, 64, 125, 216, 343]
>>> cubos.pop(1)
8
>>> cubos
```

```
[1, 27, 64, 125, 216]
```

Eliminar un elemento de la lista por su valor

Método remove. Habrá que indicar como parámetro el **valor del elemento** a eliminar (no su índice). El resto de la lista se actualizará:

```
>>> cubos=[1, 8, 27, 64, 125, 216, 343]
>>> cubos.remove(125)
>>> cubos
[1, 8, 27, 64, 216, 343]
```

len():

En este caso la función len nos devuelve un valor indicando el número de elementos en la lista:

```
>>>letras = ['a', 'b', 'c', 'd']
>>>len(letras)
4
```

Índice de un elemento de la lista

Utilizaremos el método index:

```
>>> cubos=[1, 8, 27, 64, 125, 216, 343]
>>> cubos.index(64)
3
```

> Si un elemento está repetido, index indica la posición del que tenga un índice menor.

Comprobar si un valor está en una lista

La operación **in** permite comprobar si un valor está incluido en una lista:

- Si el elemento está incluido en la lista in devuelve el valor lógico True.
- Si el elemento no está incluido en la lista in devuelve el valor lógico False.

La sintaxis de la operación es:

```
elemento in lista
```

Donde elemento es el elemento que queremos comprobar y lista el nombre de la lista en la que lo queremos buscar. Así por ejemplo:

```
>>> xs=[78455, 79211, 54988, 66540, 47890]
>>> 78 in xs
False
>>> 66540 in xs
True
```

Contar cuantas veces está un elemento en una lista

Utilizaremos el método count:

```
milista=["José",30,4,4,1992]
print(milista.count(4))
```

El método count indicará cuantas veces aparece el valor 4 en la lista, en este caso 2.

Crear listas a partir de otras listas

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>>x
[['a', 'b', 'c'], [1, 2, 3]]
>>>x[0]
['a', 'b', 'c']
>>>x[0][1]
'b'
```

Asignar cada valor de una lista a una variable diferente

Podríamos hacer la asignación utilizando índices en la forma:

```
milista=["José",30,4,1992]
nombre=milista[0]
dia=milista[1]
mes=milista[2]
agno=milista[3]
print(nombre,dia,mes,agno)
```

Daría como resultado:

```
José 30 4 1992
```

Python permite realizar la asignación en una única línea de código con la sintaxis:

```
milista=["José",30,4,1992]
nombre,dia,mes,agno=milista
```

```
print(nombre,dia,mes,agno)
```

Conseguimos el mismo resultado, pero reducimos el número de líneas de código y evitamos el uso de índices.

Estadísticas simples con una lista de números

Algunas funciones de Python son útiles cuando se trabaja con listas numéricas. Por ejemplo, puedes encontrar el mínimo, máximo y la suma de una lista de números:

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
print(min(digits))
0
print(max(dgits))
9
print(sum(digits))
45
```

12. Ejecutar un archivo .py

> Ejecutar un archivo .py consiste en iniciar el intérprete de Python para que lea el código del archivo, lo interprete y ejecute sus instrucciones en el orden en que aparecen.

Dicho de forma sencilla:

Ejecutar un archivo es decirle a Python que lea el archivo y haga exactamente lo que está escrito en él.

Existen varias formas diferentes de ejecutar un archivo .py. Veamos las más habituales:

12.1 Ejecutar el archivo desde la línea de comandos

Normalmente no ejecutaremos el código Python desde la línea de comandos. Es mucho más habitual escribir el código en archivos de texto (.py) que luego ejecutaremos. Utilizaremos la palabra módulo para referirnos a estos archivos.

> **Módulo**: Cualquier archivo de texto que contiene ordenes en Python.

Una vez guardado el módulo podemos hacer que el intérprete de Python ejecute las órdenes contenidas en él, de arriba hacia abajo y de una en una, de muchas formas distintas (línea de comandos, doble clic en su icono, IDE...)

> Para cerrar la sesión interactiva del interprete podemos utilizar la orden exit(), Ctrl+Z (Windows), Crtl+D (Linux) o cerrar la ventana de comandos.

Veamos un primer ejemplo:

12.1.1 Nuestro primer programa

Abrimos el bloc de notas (valdría cualquier editor de texto) y escribimos el siguiente código:

```
# Un primer script en Python
import sys          # Carga un módulo de librería
print(sys.platform)
print(2 ** 100)     # Elevamos 2 a la potencia 100
x = 'Spam!'
print(x * 8)        # Repetimos una cadena de texto ocho
veces
```

Guardamos el archivo como script1.py en nuestro directorio de trabajo (D:\Code).

En resumen, este código:

- Importa un módulo (otro programa incluido en las librerías de herramientas de Python) que muestra el nombre del tipo de plataforma que estamos utilizando.

- Ejecuta la función print para mostrar el resultado del módulo anterior.
- Muestra el resultado de elevar 2 a la 100
- Almacena la palabra Spam! en una variable a la que llamaremos x
- Utiliza la función print para escribir ocho veces el valor de la variable x.
- Se han añadido comentarios. Todo aquello precedido por #.

Comentario: Línea/s de código que el compilador va a ignorar. Su única función es mejorar la legibilidad del programa.

El archivo podría llamarse script1 sin añadir la extensión .py. El compilador lo entendería igual, pero es muy aconsejable añadir la extensión .py por dos razones:

1: Si queremos importar este módulo desde otro, sólo será posible si está guardado con esa extensión.

2: El editor reconocerá que es un archivo Python y podrá utilizar propiedades como el coloreado sintáctico o la indentación automática.

12.1.2 Ejecutar desde la línea de comandos

Una vez que hemos escrito nuestro código y lo hemos guardado como un archivo (por ejemplo, script1.py), llega el momento de ejecutarlo. Para ello, primero debemos abrir nuestra terminal y desplazarnos hasta la carpeta donde se encuentra el archivo.

Trabajando en Windows

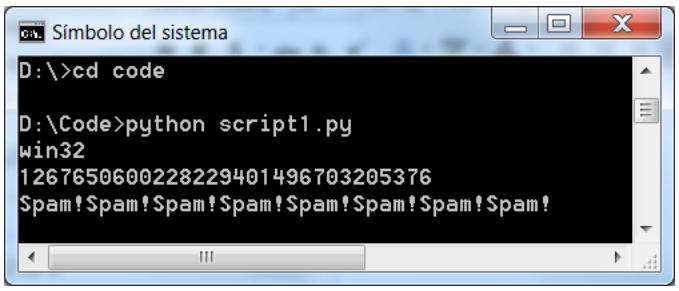
En el entorno Windows, utilizaremos la consola de comandos (CMD).

Navegación: Usaremos el comando cd (change directory) para movernos y dir para listar los archivos. Por ejemplo, si nuestro archivo está en la unidad D:, dentro de una carpeta llamada Code, escribiremos:

- d: (para cambiar de unidad).
- cd Code (para entrar en la carpeta).

Ejecución: Una vez situados en el directorio correcto, tecleamos:

```
python script1.py
```



Trabajando en Linux (Mint / Ubuntu)

En Linux, utilizaremos la Terminal. Los comandos varían ligeramente:

Navegación: Usaremos cd para movernos y ls (list) para ver el contenido de la carpeta. Si nuestro archivo está en /home/usuario/Code, escribiremos:

– cd /home/usuario/Code

Ejecución: En la mayoría de las distribuciones de Linux, Python 3 se invoca de forma específica:

```
python3 script1.py
```

Una variación. Podemos utilizar las propiedades de la ventana de comandos, así si ejecutamos el programa según:

```
python script1.py > saveit.txt
```

El resultado se guardará en un archivo de nombre saveit.txt en el directorio de trabajo.

12.2 Ejecutar haciendo clic en el icono del módulo

Windows

Al hacer doble clic sobre el icono de un archivo, el sistema operativo lee la extensión del y lo abre con el programa apropiado para ese tipo de extensiones según se indique en el archivo de asociación de nombres de archivo.

Los archivos con extensión .py se ejecutarán por la aplicación py.exe a través de una consola.

Si ejecutamos el código anterior haciendo doble clic en Windows, veremos que el programa se ejecuta, pero inmediatamente se cierra la pantalla sin dejar a la vista el resultado.

Un truco sería llamar desde el código la función predefinida input(), la veremos más adelante:

```
# Un primer script en Python
import sys           # Carga un módulo de librería
print(sys.platform)
print(2 ** 100)           # Elevamos 2 a la potencia 100
x = 'Spam!'
print(x * 8)         # Repetimos una cadena de texto ocho
veces
input()        # <== ADDED
```

Input queda esperando la siguiente información que se introduzca por teclado. En este contexto lo utilizaremos como una pausa. Al pulsar la tecla Enter el programa terminará.

Ejecutar los programas por clic tiene muchas más limitaciones (por ejemplo detección de errores).

13. Sublime Text

Sublime Text: Qué es y sus ventajas

Sublime Text es un editor de texto avanzado y ligero, especialmente diseñado para programadores. Se caracteriza por su interfaz intuitiva, rapidez y capacidad de personalización, lo que lo convierte en una herramienta ideal tanto para principiantes como para usuarios experimentados.

Ventajas de utilizar Sublime Text:

- **Interfaz sencilla y limpia:** Facilita la concentración en el código sin distracciones.
- **Rendimiento elevado:** Su rapidez permite trabajar de forma fluida incluso con archivos de gran tamaño.
- **Personalización y extensibilidad:** Admite una amplia variedad de plugins y temas, lo que permite adaptar el entorno a las necesidades del programador.
- **Soporte para múltiples lenguajes:** Ofrece realce de sintaxis y autocompletado para diversos lenguajes de programación.
- **Multiplataforma:** Está disponible para Windows, Linux y macOS, lo que garantiza una experiencia consistente en distintos sistemas operativos.

Estas características hacen de Sublime Text una opción muy recomendable para escribir y gestionar tu código, permitiéndote concentrarte en aprender a programar sin complicaciones.

13.1 Primer programa con Sublime Text

Vamos a crear nuestro primer proyecto con Sublime Text. Será algo sencillo, un módulo que muestre un mensaje por pantalla. Pero nos servirá para organizar nuestros trabajos

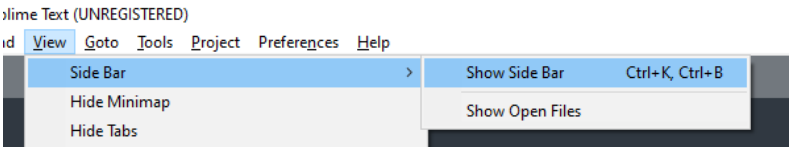
Pasos previos 1: Crear una carpeta de trabajo

> Aunque no es necesario, es buena idea organizar nuestras aplicaciones en proyectos. Un proyecto contendrá todos los archivos que vayamos a crear en nuestra aplicación.

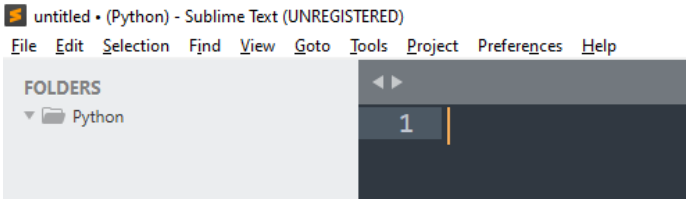
Consideraremos como proyecto, todos los módulos que estén incluidos en una misma carpeta.

En primer lugar, crea una carpeta en tu disco duro donde guardar todos tus archivos .py. Te recomiendo que se llame Python.

Abre Sublime Text y desde el menú File, open folder... selecciona la carpeta que has creado en el punto anterior (en este tutorial se llamará Python). Selecciona en el menú View la opción Side Bar, se abrirá un submenú. En el selecciona la opción Show Side Bar:

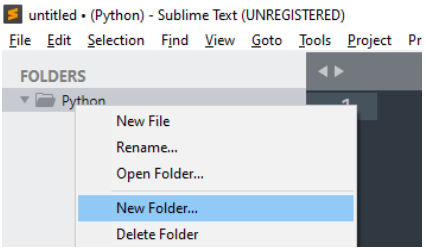


A partir de ese momento cuando abras Sublime Text siempre se abrirá ese directorio y se mostrará la estructura de carpetas de tu directorio Python:



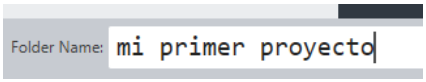
Pasos previos 2: Crear un nuevo proyecto

Aunque en rigor Sublime Text no trabaja con proyectos, vamos a agrupar nuestros trabajos en carpeta. Llamaremos proyecto a cualquier carpeta que tenga su raíz en el directorio que hemos creado en el punto anterior (Python).

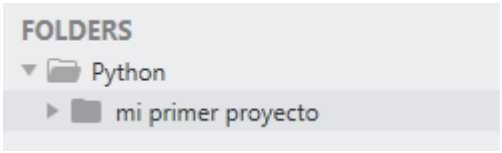


Para crear un nuevo proyecto, puedes hacer clic con el botón derecho del ratón sobre el icono de la carpeta Python y seleccionar la opción New Folder...

Se abrirá una caja de texto en la parte inferior de la pantalla. Asigna un nombre a la carpeta (por ejemplo: mi primer proyecto).

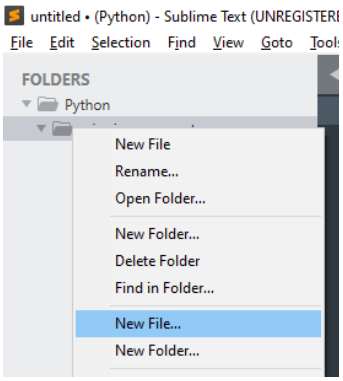


Al aceptar se mostrará la nueva carpeta como subcarpeta de Python en la ventana de la izquierda.



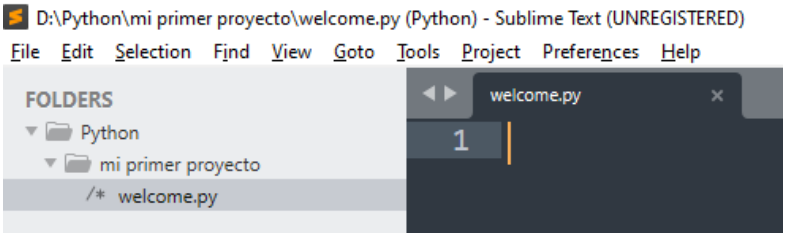
Pasos previos 3: Crear un nuevo archivo de código fuente

Hacemos clic con el botón derecho sobre el icono de la carpeta mi primer proyecto. Seleccionamos la opción New File...



Se mostrará la misma caja de texto que en el apartado anterior. Escribimos en ella el nombre para nuestro archivo, sin olvidarnos de la extensión .py. (por ejemplo welcome.py).

Se mostrará el nombre del archivo, dentro de la carpeta "mi primer proyecto" y en la parte de la derecha veremos la ventana de edición con una etiqueta con el nombre welcome.py.



Comentando tus programas

> **Comentario**: Línea/s de código que el compilador va a ignorar. Su única función es mejorar la legibilidad del programa.

La forma de indicar al interprete el comienzo de un comentario es a través del símbolo #. A partir de ese instante, y hasta que comience una nueva línea de código, el compilador ignorará la información y a efectos prácticos no tendrá ningún valor.

> **Consejo:** Iniciar el programa con un comentario que indique la finalidad del programa, el autor, fecha y hora de la última modificación.

Código que genera el texto por pantalla

Utilizamos la función print para escribir por pantalla una cadena de caracteres

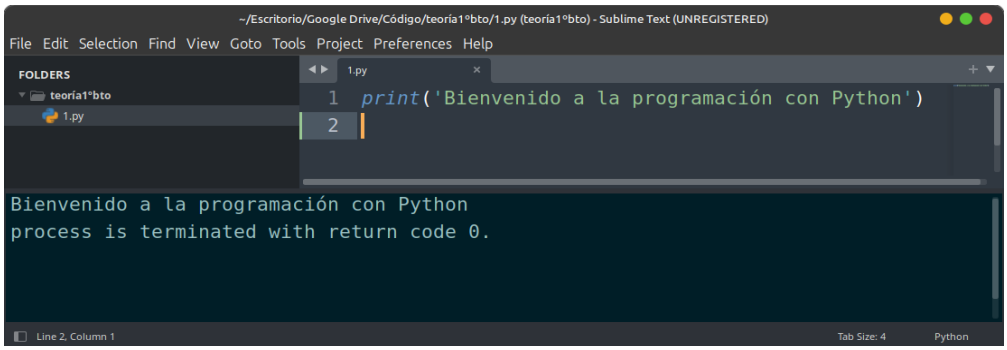
```
print('Bienvenido a la programación con Python')
```

Compilar y ejecutar la aplicación

Antes de ejecutar el archivo debes guardarlo (puedes usar la combinación de teclas Ctrl+S).

Si has configurado SubText como se indica en el capítulo de instalación, teclea las teclas Ctrl+Alt+B.

En la parte inferior se mostrará la salida del programa:



14. Pasando valores al programa – input()

input() Función que permite introducir información a través del teclado

Cuando es necesario transmitir un valor al programa, Python ofrece la función input(). La forma más habitual de utilizar esta función será:

```
nombreVariable = input('mensaje')
```

Cuando el intérprete de Python llegue a una instrucción como la anterior, mostrará por pantalla el mensaje 'mensaje' y quedará esperando a que introduzcamos a través del teclado una cadena de caracteres o un valor. En el momento que pulsemos la tecla Enter, el valor introducido por teclado quedará almacenado en la variable nombreVariable y continuará con la ejecución de la siguiente orden.

Así, por ejemplo:

```
valor=input('Introduce un número: ')
print('El número introducido es '+valor+'!')
```

El significado de los símbolos + en la orden print se explica más adelante

Se mostrará un texto solicitando un número. Tras introducirlo se generará un mensaje a través de la función print uniendo dos cadenas de caracteres y el número introducido a través de la función input() que ha sido almacenado en la variable valor.

De forma similar se podría introducir una cadena de caracteres:

```
nombre = input ("¿Cuál es tu nombre? ")
print("Hola, " + nombre + "!")
```

El resultado obtenido con los dos bloques de código anterior sería:

```
Introduce un número: 1
El número introducido es 1!
¿Cuál es tu nombre? Pedro
Hola, Pedro!
```

14.1 Números e input()

Por defecto input() convierte la respuesta en una cadena de caracteres, así si intentamos hacer un cálculo obtendríamos un mensaje de error. Si se quiere que Python interprete la entrada como un número habrá que utilizar una función para realizar la transformación:

Si queremos que la respuesta se tome como número entero: int()

```
valor=int(input('Introduce tu edad en años: '))
```

Si queremos que la respuesta se tome como un número decimal: float()

```
valor=float (input('Introduce el precio en euros: '))
```

Si intentamos guardar un número con decimales en una variable int se generará un mensaje de error. En cambio, sí almacenamos un número entero en una variable float no habrá problema. (ej: float(8)=8.0)

14.2 Argumento de la función input()

Esta función sólo admite un único argumento. Por tanto en el código:

```
nombre = input("Dime tu nombre: ")
apellido = input("Dime tu apellido, ", nombre, ": ")
print("Me alegro de conocerte,", nombre, apellido)
```

La primera línea es correcta, sin embargo, la segunda generará un error.

Estamos intentando generar un mensaje complejo formado por dos cadenas de texto y el contenido de una variable. La función input() sólo puede utilizar un argumento. Podemos resolver el problema dos formas:

Escribo la pregunta utilizando la función print() y en la orden siguiente utilizo input().

```
nombre = input("Dime tu nombre: ")
print("Dime tu apellido", nombre+": ")
apellido=input()
print("Me alegro de conocerte,", nombre, apellido)
```

Utilizar en la segunda orden input la concatenación:

> **Concatenación** unir dos elementos (variables o cadenas de texto) en uno único.

Para concatenar dos elementos es escribirán uno a continuación de otro, separados por medio el símbolo de suma (+).

```
nombre = input("Dime tu nombre: ")
apellido = input("Dime tu apellido, "+ nombre+ ": ")
print("Me alegro de conocerte,", nombre, apellido)
```

Ejercicio: Crea un proyecto que sea capaz de leer dos números enteros que introduciremos a través del teclado, realizará la suma de los dos y mostrará el resultado por pantalla. Añade tantos comentarios como sea necesario para conseguir que el código sea fácilmente entendible.

La salida debe ser similar a:

```
Introduce el primer entero: 45
Introduce el Segundo entero: 72
La suma es: 117
```

15. Sentencias condicionales: if... elif...else...

15.1 Sentencia condicional if... (Una opción)

La **orden if** evalúa una condición. Si la condición se cumple se ejecutan las ordenes agrupadas dentro del bloque if. Si no se cumple no se ejecutarán las ordenes agrupadas en el bloque if.

La sintaxis de un bloque if es:

```
if condición:
  bloque de órdenes que se ejecutarán si la condición es
cierta.
Estas órdenes pueden ocupar más de una línea.
```

Condición	es una expresión lógica que el programa evaluará y que generará un valor cierto o falso.
:	Símbolo que marca el final de la expresión lógica. Es obligatoria.
Bloque de órdenes	Conjunto de órdenes que se ejecutan solamente si la condición lógica ha sido evaluada como verdadera.

El bloque de órdenes ha de ir indentado (sangrado). Python utiliza el sangrado para identificar las órdenes que están asociadas al bloque if.

Todas las órdenes asociadas al if han de tener el mismo sangrado.

Normalmente el sangrado en Python es de cuatro espacios. Al escribir el símbolo (:), Eclipse automáticamente realizará el sangrado.

15.1.1 Comparadores lógicos

Realizan una comparación entre dos expresiones. Cuando la expresión es cierta generan un valor booleano True, cuando no lo es generan un valor booleano False.

Símbolo	Comparador	Genera True	Genera False
>	Mayor que	3>2	2>3
<	Menor que	2<3	3<2
>=	Mayor o igual que	2>=1+1	2>=1+2
<=	Menor o igual que	4-2<=1	4-2<=2
==	Igual que	2==1+1	2==1
!=	Distinto de	6/2!=2	6/2!=3

15.1.2 Operadores lógicos

Son tres y permiten construir condiciones más complejas.

Símbolo	Operador	Genera True	Genera False
and	Y lógico	true and true	true and false false and false false and true
or	O lógico	true or true true or false false or true	false or false
not	Negación	not false	not true

15.1.3 Expresiones compuestas

Están formadas por varios componentes. Se recomienda utilizar paréntesis para asegurar el orden en que las expresiones son evaluadas.

Rn Python no es obligatorio el uso de paréntesis. El orden de prioridad por defecto es: primero se evalúan los not, luego los and y por último los or.

> Utiliza paréntesis!!!!!!!

Crea un módulo que pida al usuario un número positivo. La respuesta ha de ser almacenada en un variable a la que llamaremos numero. Comprueba si el número es negativo. Si lo es, el programa debe avisar del error. Finalmente, el programa muestra siempre el número por pantalla.

15.2 if ... else ... (dos opciones)

En este caso el if permite que el programa ejecute un bloque de instrucciones cuando se cumple la condición incluida en la línea del if y otras cuando no se cumple.

La sintaxis de un bloque if...else... es:

```
if condición:
  bloque de órdenes que se ejecutarán si la condición es
cierta.
else:
  Bloque de órdenes que se ejecutarán si la condición es
falsa.
```

Los bloques de órdenes pueden tener más de una línea e irán indentados.

Todas las órdenes de un bloque han de tener el mismo sangrado, cada bloque puede tener un sangrado distinto... aunque eso reduce la claridad del código.

La línea con la orden else solo ha de incluir el else y los dos puntos.

Aunque no es muy habitual, si deseamos dejar uno de los bloques vacío, el bloque de instrucciones deberá contener la orden pass. Esta orden le dice a Python que no haga nada y salga del bloque de órdenes.

Diseña un programa que pregunte la edad al usuario y la almacene en la variable 'edad'. A continuación, comprobad si la edad es inferior a 18 años. Si la condición es cierta el programa debe avisar por pantalla de que el usuario es menor de edad y si es falsa de que es mayor de edad. Finalmente, el programa se despide en cualquier caso.

15.3 if... elif... else... (más de dos opciones)

Esta sentencia condicional permite encadenar varias condiciones. elif es la abreviatura de else if (si no).

La sintaxis de un bloque if...elif...else... es:

```
if condición1:
  bloque de órdenes a ejecutarar si la condición1 es
cierta.
elif condificón2:
  bloque de órdenes a ejecutar si la condición2 es cierta.
else:
  bloque de órdenes a ejecutarar si todas las condiciones
son   falsas.
```

Se pueden incluir tantos bloques elif como sean necesarios.

Observa que en este tipo de sentencia las opciones son siempre mutuamente excluyentes. En cualquier caso solamente se ejecutará uno de los bloques.

En este tipo de estructuras hay que ser cuidadoso con el orden con que escribimos las condiciones. Con ello:

- Evitaremos olvidar alguna de las situaciones posibles.
- Evitaremos que el programa ejecute un bloque de instrucciones no deseado.

Veamos el siguiente ejemplo:

Modifica el programa anterior de tal forma que distinga tres situaciones. Si introducimos un valor negativo el programa mostrará un mensaje de error. Si el valor está comprendido entre 0 y 17, mostrará un mensaje indicando que el usuario es menor de edad y si es mayor o igual a 18 el mensaje informará de la mayoría de edad del usuario.

Existen múltiples soluciones sin embargo alguna de ellas nos puede dar resultados extraños como informar de que una persona que tenga –14 años sea menor de edad por ser el valor menor de 18. Investiga la mejor solución.

Compliquemos ahora las cosas un poco. Utilizando un único bloque if..elif..else diseña un programa que indique si un número es múltiplo de dos, de dos y de cuatro o si no es múltiplo de dos. El problema radica en

que todos los múltiplos de cuatro son múltiplos de dos pero no todos los múltiplos de dos son múltiplos de cuatro.

15.4 Sentencias condicionales anidadas

Una sentencia anidada condicional puede contener a su vez otra sentencia anidada. Por ejemplo:

```
if condición1:
  if condición2:
    bloque de órdenes que se ejecutarán si la condición1 y
la 2    son ciertas.
  else:
    Bloque de órdenes que se ejecutarán si la condición 1
es      cierta y 2 es falsa.
else:
  Bloque de órdenes que se ejecutarán si la condición 1 es
  falsa.
```

En este caso se considera la primera condición. Si es true se considera la segunda condición y si también es cierta se ejecutan las ordenes contenidas en el primer bloque. Si la segunda no lo es se ejecuta el segundo bloque de órdenes. Solo en el caso de que la primera condición sea falsa e independientemente de cómo sea la segunda se ejecuta el tercer bloque de instrucciones.

Las estructuras anidadas pueden ser tan complicadas como se necesite y pueden contener cualquiera de los tres tipos de sentencias if que hemos estudiado.

Diseña un programa que haciendo sólo con dos preguntas al usuario sea capaz de adivinar el número entre 1 y 4 que este está pensando.

15.5 Sintaxis alternativa con valores lógicos

Observa el siguiente código:

```
color = True
talla = False
if (color == True):
    print("Ejemplo sencillo")
```

Hemos definido dos variables tipo booleano. Color toma el valor True y talla el valor False.

La condición incluida dentro del bucle if toma el valor True ya que se cumple la condición (el valor de la variable color es True).

La respuesta del programa es pues:

```
Ejemplo sencillo
```

Es decir, lo que hace que se ejecute el contenido incluido dentro del bloque if es el hecho de que el resultado que genera la comparación incluida en él es el valor True.

Eso se puede tener en cuenta de cara a definir un código similar pero más sencillo:

```
color = True
talla = False
if color:
    print("Ejemplo sencillo")
```

En este caso el bloque if analiza el valor de la variable color, como este es True el resultado será el mismo que en el caso anterior.

No ocurriría lo mismo con la variable talla, el código "completo" sería:

```
color = True
talla = False
if (talla == False):
    print("Ejemplo sencillo")
```

Que daría el mismo resultado que en los casos anteriores.

Sin embargo, si intentamos simplificar el código en la forma anterior:

```
color = True
talla = False
```

```
if talla:
    print("Ejemplo sencillo")
```

El programa no dará resultado, la línea if esta, recibe el valor False y no se cumple la condición.

Complicando un poco las cosas, supón que necesitamos escribir el código final si la variable color toma el valor True y la variable talla el valor False. El código "completo" sería:

```
color = True
talla = False
if ((color == True) and (talla == False)):
    print("Ejemplo sencillo")
```

Si queremos simplificarlo podríamos hacer:

```
color = True
talla = False
if (color and talla == False):
    print("Ejemplo sencillo")
```

15.6 Concatenación de comparadores lógicos

Sintaxis en la que utilizamos más de un comparador en la misma expresión. Por ejemplo, si queremos determinar si número es mayor que 0 pero menor que 100:

```
valor = int(input("introduce tu edad:"))
if 0 < valor < 100:
  print("Edad no válida")
else:
  print("Edad válida")
```

La comparación del if evaluará de izquierda a derecha todas las comparaciones y si todas ellas son True, generará un valor True.

Es una forma de ahorrar código. Podremos añadir todos los comparadores que deseemos.

15.7 If con el operador in

El operador in genera un valor True o False en función de que un elemento esté en una lista o tupla. Eso permite:

```
print("Asignaturas optativas: Informática, Latín, Griego")
asignatura=input("Escoge una asignatura:")
if asignatura in ('Informática', 'Latín', 'Griego'):
  print("Asignatura elegida:", asignatura)
else:
  print("La asignatura", asignatura, "no está
contemplada")
```

16. El bucle for

En general un bucle en programación se definirá como:

> **Bucle**: Estructura de control que repite un bloque de instrucciones.

En Python existen varios tipos de bucles. Un bucle for es:

> **Bucle for**: Bucle que repite un bloque de instrucciones un *número determinado de veces*.

Llamaremos:

Iteración: Cada una de las veces que se ejecuta el cuerpo del bucle.

La sintaxis básica del bucle for es:

```
for variableControl in elemento de control
  Cuerpo del bucle
```

Veamos la orden un poco más a fondo:

variableControl	Variable que tomará en cada iteración un valor definido por 'elemento de control'. La iteración se repetirá tantas veces como valores pueda tomar 'variableControl' y en cada iteración el valor almacenado en la variable es definido por el elemento de control.
Elemento de control	Elemento que define los distintos valores que va a ir tomando la variable 'variableControl'. Puede ser una lista, una cadena de caracteres, un rango.
Cuerpo del bucle	Indentado y contiene las órdenes a ejecutar en cada iteración.

> Por costumbre se utilizar la letra i como nombre de la variable de control

Veamos varios ejemplos:

Bucle for controlado por una lista

En este caso:

- El bucle realizará tantas iteraciones como elementos tenga la lista.
- En cada iteración la variable irá tomando sucesivamente en cada iteración los valores contenidos en la lista.

```
print("Comienzo")
for i in [1, 1, 1]:
  print("Hola ", end="")
print()
print("Final")
```

El for del código anterior se ejecutará tres veces, la variable i tomará el valor 1 en las tres iteraciones y como resultados se mostrará la palabra hola tres veces en la misma línea.

En este caso el valor que toma la variable de control no es importante, lo único que nos interesa es que la iteración se realice tres veces.

En otros casos el valor que toma la variable de control en cada iteración es fundamental.

Modifica el código del ejemplo anterior para, utilizando un bucle for con tres iteraciones, calcular los cuadrados de 3, 4 y 5.

Python considerará que pertenece al bucle todo el código indentado que tenga la misma indentación que la línea siguiente a la línea con la orden for.

La lista puede contener cualquier tipo de elementos, no sólo números.

Modifica el código del ejemplo anterior para, utilizando un bucle for con tres iteraciones, crear tres saludos personalizados a tres amigos que se llaman Luis, Mireia y Andrés.

Bucle controlado por una cadena de caracteres

En este caso:

El bucle realizará tantas iteraciones como caracteres tenga la cadena de caracteres.

En cada iteración la variable irá tomando sucesivamente en cada iteración cada uno de los caracteres.

```
for i in "AMIGO":
print("Dame una ", i)
print("¡AMIGO!")
```

Mostrará como resultado:

```
Dame una A
Dame una M
Dame una I
Dame una G
Dame una O
¡AMIGO!
```

Escribe un programa que, utilizando un bucle for, muestre la tabla de multiplicar de un número que el usuario introduzca a través del teclado.

Bucle controlado por la función range()

La función range genera series de números y se puede definir de tres maneras:

range(a)	El bucle deberá realizar a iteraciones. La variable de control (i) toma el valor 0 en la primera iteración y va aumentando una unidad en cada pasada hasta alcanzar el valor a-1.
range(a,b)	El bucle deberá realizar b-a iteraciones. La variable de control (i) toma el valor a en la primera iteración y va aumentando una unidad en cada pasada hasta llegar al valor b-1.
range(a,b,c)	Similar al anterior pero ahora: - El valor inicial de la variable de control es a. - El incremento del valor de la variable de control en cada iteración es c. - El bucle termina cuando la variable de control llega al valor b.

Ejemplo: Código para mostrar 10 veces la palabra Hola con un bucle for y un tipo range(a).

```
print("Comienzo")
for i in range(10):
  print("Hola ", end="")
print()
print("Final")
```

Ejemplo: Código muestra los 100 primeros números naturales utilizando un bucle for y un tipo range(a,b).

```
for i in range(1,101):
  print(i)
```

Ejemplo: Múltiplos del 3 contenidos entre 1 y 100.

```
for i in range(3,101,3):
  print(i)
```

16.1 Contadores y acumuladores

En muchas ocasiones se necesitan variables que cuenten cuántas veces ha ocurrido algo o que acumulen valores.

En ambos casos es necesario dar un valor inicial a la variable.

Contador: Variable que lleva la cuenta del número de veces que se ha cumplido una condición.

El código asociado a un contador sería:

```
i = i + 1
```

Crea un programa que determine cuantos números naturales menores de 100 son múltiplos de tres.

Acumulador: Variable que en cada iteración va incrementando su valor en una cantidad (constante o variable).

El código asociado a un acumulador sería:

```
i = i + a
```

Donde a podrá ser un número o una variable.

Crea un programa que, utilizando un bucle for, calcule la suma de los 100 primeros números naturales.

16.2 Bucles anidados

Bucle anidado: Bucle incluido dentro del bloque de instrucciones de otro bucle principal.

Hablaremos de:

Bucle interior: El que se encuentra dentro del otro.

Bucle exterior: Al bucle principal.

Existen dos tipos de bucles anidados. De variables independientes y de variables dependientes.

Bucle anidado con variables independientes

Las dos variables son independientes cuando los valores que toma la variable de control del bucle interior no dependen del valor de la variable de control del bucle exterior. Por ejemplo:

```
for i in [0, 1, 2]:
  for j in [0,1]:
    print('i vale', i, 'y j vale', j)
```

Es más recomendable utilizar tipos range() en lugar de listas. El código anterior quedaría:

```
for i in range(3):
  for j in range(2):
    print('i vale', i, 'y j vale', j)
```

En ambos casos el resultado sería:

```
i vale 0 y j vale 0
i vale 0 y j vale 1
i vale 1 y j vale 0
i vale 1 y j vale 1
i vale 2 y j vale 0
i vale 2 y j vale 1
```

Habitualmente se utiliza la letra i como nombre de la variable de control del bucle exterior y la j como nombre de la variable de control del bucle interior (k si hay un tercer nivel de anidamiento.

Bucle anidado con variables dependientes

Los valores que toma la variable de control del bucle interior dependen del valor de la variable de control del bucle exterior. Por ejemplo:

```
for i in [1, 2, 3]:
for j in range(i):
     print("i vale", i, "y j vale", j)
```

Daría por resultado:

```
i vale 1 y j vale 0
i vale 2 y j vale 0
i vale 2 y j vale 1
i vale 3 y j vale 0
i vale 3 y j vale 1
i vale 3 y j vale 2
```

17. Bucle while

Bucle while: Bloque de código repite un grupo de instrucciones *mientras se cumpla una condición*.

La sintaxis básica del bucle while es:

```
while condición:
  Cuerpo del bucle
```

Veamos la orden un poco más a fondo:

Condición	Expresión lógica que se evalúa antes de cada iteración. Si se evalúa como true se ejecuta el cuerpo del bucle. Una vez ejecutado el cuerpo del bucle vuelve a ser evaluada. Si se evalúa como false se termina la ejecución del bucle sin ejecutar el cuerpo del mismo.
Cuerpo del bucle	Está indentado y contiene el conjunto de órdenes que se ejecutarán en cada iteración.
Variable(s) de control	Variables que aparecen dentro de la condición. Estas variables pueden definirse antes del bucle while pero también puede modificarse su valor en el interior.

Veamos un ejemplo sencillo:

```
i = 1
while i<= 3:
  print(i)
  i += 1
print("Programa terminado")
```

Definimos un valor inicial para la variable de control. La condición del bucle comprueba si el valor de la variable de control es menor o igual que 3. En caso de serlo escribirá el valor de i e incrementa su valor en una unidad. A continaución vuelve a evaluar la condición. En este caso el bloque del código se ejecutará tres veces mostrando los valores 1, 2 y 3. Tras ello el Python saldrá del bucle y continuará con la siguiente orden, en este caso el print.

Un bucle for se ejecutará un número fijo de veces. Sin embargo un bucle while se ejecutará un número indefinido de ocasiones, hasta que deje de cumplirse una condición. Por ejemplo:

Interpreta el siguiente código:

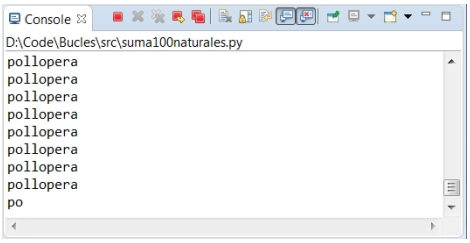
```
numero = int(input("Escriba un número positivo: "))
while numero < 0:
  print("¡Ha escrito un número negativo! Inténtelo de nuevo")
  numero = int(input("Escriba un número positivo: "))
print("Gracias por su colaboración")
```

17.1 Bucles infinitos

> Bucle infinito: Bucle en el que la condición se cumple siempre, el bucle no terminará nunca de ejecutarse.

Aunque a veces es necesario utilizar bucles infinitos en un programa, normalmente se deben a errores que se deben corregir.

Los bucles infinitos no intencionados deben evitarse pues significan perder el control del programa. Para interrumpir un bucle infinito cuando estamos ejecutando un programa desde pyDev es suficiente con hacer clic en la tecla stop (cuadrado rojo) en la ventana de consola:



Cuando se está ejecutando el programa desde otras instancias, la ejecución del mismo se interrumpirá pulsando la combinación de teclas Ctrl+C. En ese caso se mostrará un mensaje tipo:

```
Traceback (most recent call last):
  File "ejemplo.py", line 3, in <module>
print(i)
Keyboard Interrupt
```

> Encuentra la razón por la que se genera un bucle infinito en cada uno de los siguientes ejemplos:

```
i = 1
while i<= 10:
print(i, "", end="")
```

```
i = 1
while i> 0:
print(i, "", end="")
i += 1
```

```
i = 1
while i!= 100:
print(i, "", end="")
```

```
i += 2
```

17.2 Ordenes break, continue, pass y else en bucles

Estas órdenes permiten modificar la forma en la que se ejecuta la secuencia de comandos en un bucle for y while. Su función es:

break	Termina la ejecución del bucle más inmediato que la contiene.
continue	Termina la ejecución de la iteración actual pasando a la siguiente.
else	Solamente se ejecuta si el bucle termina normalmente (no por acción del break.
pass	No hace nada. Se utiliza para dar contenido a una orden que no va a hacer nada pero necesita por sintaxis tener algo escrito en ella.

17.2.1 break

Partimos del programa:

```
i = 1
while i <= 10:
  print(i,'- ',end='')
  i += 1
print("Programa terminado")
```

El bucle se ejecutará normalmente, realizado diez iteraciones, mostrando el resultado:

```
1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10 – Programa terminado
```

Si simplemente introducimos la orden break:

```
i = 1
while i<= 10:
     print(i,'- ',end='')
     i += 1
     break
print("Programa terminado")
```

El resultado tendrá la forma:

```
1 - Programa terminado
```

La orden break actúa en la primera iteración y por lo tanto termina la ejecución del bucle mñas próximo en la que está incluida (en este caso es el único).

Utilizar la orden break de esta forma no tiene mucho sentido ya que siempre se va a ejecutar, deteniendo el bucle. Normalmente la orden estará incluida dentro de un condicional. Con ello el bucle se saltará cuando se cumpla dicha condición. Un ejemplo sencillo:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  i += 1
  if i==7:
    break
print("Programa terminado")
```

En este caso la salida sería:

```
1 – 2 – 3 – 4 – 5 – 6 – Programa terminado
```

17.2.2 continue

Si en lugar de break utilizamos continue, el resultado es diferente:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  i += 1
  if i==7:
    continue
print("Programa terminado")
```

El resultado será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - Programa terminado
```

Parece que continue no haga nada, el resultado no ha cambiado. Pero no es así, el problema es que el resultado no es apreciable. Piensa en ello, continue termina la ejecución de la vuelta actual del bucle. Al estar colocado al final de la iteración el que i sea igual a 7 o no lo sea no afecta al resultado del bucle. Debemos hacer:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==7:
```

```
    continue
    i += 1
print("Programa terminado")
```

En este caso el programa creará un bucle infinito que no terminará nunca ya que al llegar i al valor 7, la condidicón del if es cierta, se ejecuta el continue, terminando la vuelta del bucle actual y por lo tanto desde ese momento nunca se vuelve a ejecutar la orden i += 1 con lo que i continúa valiendo 7.

Si queremos evitar este problema podemos utilizar el código:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==7:
    i += 2
    continue
  i += 1
print("Programa terminado")
```

En este caso la salida será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 9 - 10 - Programa terminado
```

Sin mostrar el valor 8 ya que cuando i vale 7 modificamos su valor sumando dos unidades en lugar de una, la orden continue hace que i+=1 no se ejecute cuando i vale 7.

17.2.3 else

El else asociado a un bucle while o for delimita un conjunto de órdenes que solamente se ejecutan si hemos salido del bucle sin que se haya ejecutado una orden break.

Según el ejemplo anterior:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==7:
    break
  i += 1
else:
     print("Programa terminado")
```

La salida será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 -
```

Cuando i=7 el bucle se interrumpe, por lo tanto el bloque de ordenes asociado a else no se ejecutará.

En cambio:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==77:
    break
  i += 1
else:
  print("Programa terminado")
```

La salida en este caso será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - Programa terminado
```

La orden break no llega a ejecutarse ya que i nunca puede llegar a tomar el valor 77. Por lo tanto el bucle termina normalmente y a continuación se ejecuta el bloque de instrucciones asociados al else.

17.2.4 pass

La mejor forma de entender la funcionalidad de esta orden es ver un ejemplo. Siguiendo con el caso anterior:

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==77:
    break
  i += 1
else:
```

Hemos eliminado las ordenes asociadas al bloque else:. El resultado es:

```
File "D:\Code\8. Ejercicios while 2\src\while201.py", line 9
^
SyntaxError: unexpected EOF while parsing
```

Se genera un error de sintáctico ya que else requiere que en su interior exista al menos una orden. La solución es incluir la orden pass que no tiene ninguna utilidad más que la de dar contenido (hacer montón).

```
i = 1
while i<= 10:
  print(i,'- ',end='')
  if i==77:
    break
  i += 1
else:
  pass
```

La salida es:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
```

18. Funciones

Listado de las funciones más habituales.

Función	Descripción
str()	Transforma un número en cadena de caracteres.
int()	Transforma una cadena de caracteres en un número entero.
float()	Transforma una cadena de caracteres en un número decimal.
round(a,b)	Redondea el número a con b decimales.
chr(num)	Devuelve el carácter correspondiente al valor “num” en código ASCII.
ord('x’)	Devuelve el código ASCII del carácter 'x’.

18.1 Formateando la presentación de números

Podemos utilizar la función format para definir varios aspectos de la presentación de un valor numérico por pantalla. Aquí tenéis varios ejemplos:

```
numero = 3141.59265

# Dos decimales de precisión:
print(format(numero, '0.2f'))

# Jusitificación a la derecha con diez caracteres, un decimal de precisión:
print(format(numero, '>10.1f'))

# Jusitificación a la izquierda con diez caracteres, un decimal de precisión:
print(format(numero, '<10.1f'), "hola")

# Separador de miles:
print(format(numero, ','))
print(format(numero, '0,.1f'))

# Notación científica:
print(format(numero, 'e'))
print(format(numero, '0.2E'))
```

19. Métodos

Listado de métodos de interés.

Función	Descripción
cadena.lower()	Devuelve la cadena de caracteres o variable en letras minúsculas.
cadena.upper()	Devuelve la cadena de caracteres o variable en letras mayúsculas.

Sumario