



Introducción a la Programación

Python desde cero

Rubén Gómez Olivencia

2025-09-02



Copyright © Rubén Gómez Olivencia (r.gomezolivencia@irakasle.eus)

- Github: <https://github.com/yuki>

Licencia: [Creative Commons BY-SA 4.0](#)

Este libro se ha realizado teniendo en cuenta la cultura libre. Puedes utilizarlo, modificarlo y compartirlo teniendo en cuenta la licencia [Attribution-ShareAlike](#) de **Creative Commons**. Es por eso que:

- **Atribución:** Debes darme crédito de manera adecuada e incluir un enlace a la licencia e indicar si se han realizado cambios.
- **CompartirIgual:** Si reutilizas, modificas o creas a partir de este material, debes distribuir el trabajo bajo la misma licencia.

Puedes encontrar la última versión de este libro en formato **HTML** en el siguiente [link](#), así como otros libros que he creado. Para descargar el código fuente en formato **Markdown** visita el repositorio en [GitHub](#).

Información



Por favor, ponte en contacto conmigo si encuentras algún fallo, falta de ortografía o quieres mejorar de alguna manera este libro. Gracias.

I Introducción a la programación

1	¿Qué es programar?	7
1.1	¿Para qué sirve programar?	7
2	Lenguajes de programación	7
2.1	Componentes de un lenguaje de programación	8
2.2	Tipos de lenguajes	8
2.2.1	Lenguajes interpretados	9
2.2.2	Lenguajes compilados	10
2.2.3	Lenguajes híbridos	10
3	Pensamiento lógico y resolución de problemas	11
4	Errores al programar	11

II Entorno de desarrollo: Edublocks

1	Introducción a EduBlocks	14
1.1	¿Por qué usar bloques para empezar?	14
1.2	Crear proyectos en Edublocks	14
1.3	Grupos de bloques	15
1.4	Empezamos a programar	16
1.4.1	¡Hola Mundo!	16

III Introducción a Python

1	¿Qué es Python?	19
1.1	Quién lo usa	19
1.2	Dónde lo podemos usar	19
1.3	Versiones de Python	20
2	Variables	20
2.1	Tipos de datos	21
2.1.1	Números enteros (int)	22
2.1.2	Números decimales (float)	22
2.1.3	Cadenas de texto (str)	23
2.1.4	Booleanos (bool)	23

IV Entrada y salida de datos

1 Salida de datos con print()	25
1.1 Concatenar datos con print()	25
1.2 Parámetros especiales: “sep”, “end”	26
2 Entrada de datos con input()	27

V Conversión de tipos

1 Por qué es necesario convertir datos	30
1.1 Funciones de conversión más comunes	30
1.2 Conocer el tipo de dato	30

VI Operadores básicos

1 Qué son los operadores	33
2 Operadores aritméticos	33
3 Operadores lógicos	34
3.1 Operadores especiales: “is” y “is not”	35
4 Operadores lógicos combinados: “and”, “or”, “not”	35

VII Estructuras de control

1 Estructuras de control	38
1.1 Condicionales	38
1.1.1 Condicional “if”	38
1.1.2 Condicional “if else”	41
1.1.3 Condicional “elif”	43
1.1.4 Ejercicio	44
1.2 Bucles	44
1.2.1 Bucle “while”	44
1.2.2 Bucle “for”	46
1.2.3 Instrucciones útiles en los bucles	47

VIII Ejercicios resueltos

1 Programa discoteca	49
2 Programa con inputs	49

3	Ejercicio aritméticos	49
3.1	Ejercicios de operadores lógicos	50
3.2	Ejercicios de operadores lógicos combinados	51
4	Ejercicios de condicionales	51
4.1	Ejercicio de notas	51
4.2	Ejercicio “else anidado”	53
5	Ejercicios de bucles	54
5.1	Bucle inverso	54
5.2	Contador	54
5.3	Bucle for	54
5.4	Tabla de multiplicar	55
5.5	Instrucciones break y continue	56



Introducción a la programación

1. ¿Qué es programar?

Programar es **dar instrucciones a un ordenador** para que realice una tarea. Esto se realiza escribiendo el **código fuente** con las instrucciones que queremos dar. Esto termina en la creación de una aplicación informática (programa, página web, videojuego, ...).

Programar se suele comparar con crear una receta de cocina en la que se deben indicar todos los pasos a realizar (programa) que posteriormente se interpreta por otro cocinero (el ordenador). Cualquier instrucción que no se especifique de manera correcta, hará que la receta final (el programa realizado) no funcione.



Exact Instructions Challenge es un ejemplo de la comparativa realizada.

Información



Programar es como escribir una receta de cocina. Se deben describir todos los pasos e ingredientes a usar.

1.1. ¿Para qué sirve programar?

La programación sirve para automatizar tareas, crear juegos, hacer aplicaciones, controlar robots y mucho más. Casi todo lo que usamos hoy en día tiene programación por dentro: los móviles, los coches, las lavadoras, las páginas web, etc.

Dependiendo de la aplicación que vayamos a realizar, deberemos usar un lenguaje de programación u otro.

2. Lenguajes de programación

Los ordenadores no entienden directamente nuestro idioma. Para hablar con ellos, usamos **lenguajes de programación**, que son formas especiales de escribir instrucciones que el ordenador puede entender y ejecutar.

Hay muchos lenguajes, y cada uno tiene sus características. Algunos son más fáciles de aprender, otros son más rápidos o más potentes para ciertas tareas. Algunos ejemplos de lenguajes de programación conocidos:

- **Python** : muy sencillo, ideal para empezar. Se usa en ciencia, robótica, inteligencia artificial...
- **C o C++**: muy rápido, se usa en videojuegos, sistemas operativos, robots.
- **Java** : muy usado en empresas, aplicaciones móviles (Android), sistemas grandes.
- **JavaScript** : se usa para hacer páginas web interactivas. Se ejecuta en el navegador web (como Chrome o Firefox).
- **PHP**: se usa para crear páginas webs.
- **Scratch o Edublocks**: se programa con bloques, muy visual y fácil para empezar.

2.1. Componentes de un lenguaje de programación

Al igual que los lenguajes que usamos en nuestro día a día, los lenguajes de programación también tienen reglas:

- **Alfabeto:** Conjunto de símbolos para formar las palabras utilizadas por el lenguaje. Normalmente son letras, dígitos, símbolos (+,-,:,(),[],...)
 - if, else, for, while, ...
- **Léxico:** Las palabras que el lenguaje tiene reservadas para programar. Si alguna de las palabras del léxico se escribe mal, el programa no se podrá compilar/ejecutar.
 - if, else, for, while, ...
- **Sintaxis:** Son las reglas que tiene el lenguaje para asegurar que una cadena de palabras forman una oración válida. En lenguaje natural la diferencia sería:
 - “Yo conduzco un coche”
 - “Coche conduzco yo”
- **Semántica:** Determina que la frase tiene sentido (puede estar bien escrita, pero no tiene sentido). Las siguientes frases son un ejemplo en lenguaje natural:
 - “Yo conduzco un coche”
 - “Un coche me conduce”

2.2. Tipos de lenguajes

El ordenador solo entiende **código máquina** (unos y ceros, el código binario). Por eso, los programas escritos en lenguajes como Python o Java necesitan **traducirse** a un lenguaje que el ordenador pueda ejecutar.

Según **cómo se hace esa traducción**, los lenguajes se dividen en:

- Lenguajes **interpretados**
- Lenguajes **compilados**
- Lenguajes **híbridos**

Ejercicio



Busca al menos 2 lenguajes de programación para cada tipo.

Hay otro muchos criterios para diferenciar los lenguajes de programación. Estas diferenciaciones son más profundas, por lo que no vamos a entrar en ellas, pero en la siguiente tabla se pueden ver algunas de ellas.

Criterios de diferenciación	Clasificación
-----------------------------	---------------

Continúa en la siguiente página

Según el nivel de abstracción	Lenguajes de bajo nivel. Lenguajes de alto nivel.
Según el paradigma de programación	Imperativos. Declarativos. Orientados a objetos. Orientados a objetos. Funcionales. Lógicos.
Según su propósito	De propósito general. De propósito específico (por ejemplo, para bases de datos o web).
Según la sintaxis	De sintaxis simple (como Python). De sintaxis estricta (como Java o C++)
Según su tipado	Estático. Dinámico. Fuerte. Débil
Según la gestión de memoria	Con gestión manual (como C). Con recolector de basura (como Java o Python)
Según su orientación al hardware o al sistema	Lenguajes de sistema. Lenguajes de aplicación.

2.2.1. Lenguajes interpretados

Un lenguaje **interpretado** necesita un **intérprete**, que es un programa que lee tu código fuente línea por línea y lo ejecuta directamente. Entre los lenguajes interpretados más conocidos nos encontramos con:

- Python
- JavaScript
- PHP

Los pasos para programar con un lenguaje interpretado son:

1. Escribes el código (por ejemplo, en Python).
2. El intérprete lo lee y lo ejecuta paso a paso.
3. Si hay un error, se detiene en ese momento.

Dependiendo del lenguaje y su intérprete, el paso 2 y 3 pueden variar. En algunos casos el intérprete lee todo el código fuente y si hay algún error (en la escritura), no lo ejecuta y muestra el error antes de ejecutar ninguna instrucción.

Si llevas el código fuente a otro ordenador y no está instalado el intérprete, el programa no se podrá ejecutar.

Ventajas	Desventajas
-----------------	--------------------

Continúa en la siguiente página

Muy fáciles de probar.	Suelen ser más lentos que los compilados.
Ideales para aprender y hacer pruebas rápidas.	

2.2.2. Lenguajes compilados

Un lenguaje **compilado** necesita un **compilador**, que convierte todo el código fuente en un archivo ejecutable. El resultado de la compilación genera código máquina (binario) que es directamente interpretado por el sistema operativo y la CPU. Los lenguajes compilados más conocidos son:

- C
- C++
- Rust

Al programar con lenguajes compilados, los pasos, simplificados, son:

1. Escribes el código (por ejemplo, en C++).
2. Lo **compilas**, lo que hace que genere un archivo ejecutable (por ejemplo, **programa.exe**).
3. Ese archivo ya puede ejecutarse sin necesidad del código fuente ni del compilador.

Los programas compilados se pueden llevar a otro ordenador y se puede ejecutar directamente (siempre que sea la misma arquitectura y sistema operativo). A veces es necesario instalar también dependencias (por ejemplo, DirectX en juegos).

Ventajas	Desventajas
Muy rápido al ejecutarse .	Al cambiar algo, hay que compilar otra vez.
Ideal para programas grandes o que necesitan rendimiento.	Compilar un programa grande puede requerir tiempo.
Los errores aparecen al compilar.	

2.2.3. Lenguajes híbridos

Los lenguajes **híbridos** combinan lo mejor de los dos mundos: primero **compilan** el código a un formato intermedio, y luego lo **interpretan o ejecutan con una máquina virtual**. Ejemplos de lenguajes híbridos:

- Java (usa la JVM)
- C# (usa .NET)
- Kotlin (usado sobre todo en Android)

Al programar en un lenguaje híbrido, los pasos son:

- Escribes el código (por ejemplo, en Java).
- Lo compilas a un formato intermedio (*bytecode* en Java). En el caso de Java, se compila con el llamado **JDK** (Java Development Kit).
- Ese bytecode se ejecuta en una **máquina virtual**, como la JVM (Java Virtual Machine).

En este caso, el intérprete debe estar instalado en el ordenador donde queramos ejecutar el programa

Ventajas	Desventajas
Buena velocidad.	Necesitas tener instalada la máquina virtual.
Portabilidad: el mismo programa funciona en distintos sistemas.	A veces, un poco más complejo de configurar.

3. Pensamiento lógico y resolución de problemas

Antes de escribir el código fuente de un programa, es importante aprender a **pensar como un programador**.

La programación no trata solo de escribir instrucciones, sino de **resolver problemas paso a paso**.

Programar es como resolver un puzzle. Necesitas analizar el problema, pensar cómo dividirlo en partes pequeñas y resolver cada una de las partes con instrucciones claras.

¡Atención!



Antes de programar tenemos que entender qué problema queremos solucionar y pensar cómo lo vamos a solucionar.

Ejercicio



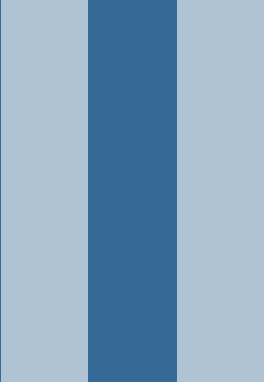
Escribe las instrucciones de un programa para ser un portero de discoteca. Debe tener en cuenta que sólo pueden entrar los mayores de 18 años. **Solución.**

4. Errores al programar

Cuando se programa, es normal que las cosas no funcionen a la primera. Hay tres tipos de errores muy comunes:

- **Errores de sintaxis:** El código tiene una “falta de ortografía”. Por ejemplo, olvidarse de un paréntesis o escribir mal una palabra clave.
- **Errores de lógica:** El código se ejecuta, pero **no hace lo que tú querías**. Por ejemplo, poner el signo **>** en lugar de **<**. Estos errores sólo se pueden apreciar al ejecutar el programa.

- **Errores de ejecución:** El programa se bloquea mientras está funcionando. Se ha escrito una instrucción que hace que el programa “se rompa”: dividir entre cero, leer una variable con valores “extraños”...



Entorno de desarrollo: Edublocks

1. Introducción a EduBlocks

EduBlocks es un entorno de programación por bloques que podemos utilizar para programar en varios lenguajes, entre ellos **Python**. Funciona de forma parecida a [Scratch](#), pero cada bloque **corresponde a una línea de código Python**. De esta manera, se puede aprender cómo funciona Python sin tener que preocuparnos de errores de escritura o de formato.

1.1. ¿Por qué usar bloques para empezar?

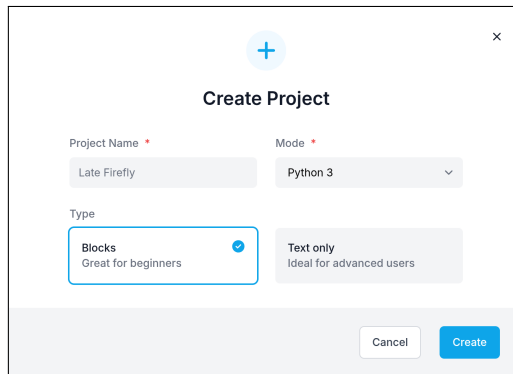
La programación por bloques es una manera **más visual** de programar, ideal para aprender, y también es más fácil de entender. Nos va a ayudar a **evitar errores de escritura**, como olvidar un paréntesis o escribir mal una palabra clave del lenguaje.

También nos va a ayudar a **conocer la estructura del código** de manera sencilla. Además, con EduBlocks, como también nos muestra el código Python, es ideal para ir aprendiendo poco a poco.

1.2. Crear proyectos en Edublocks

Para usar EduBlocks en el navegador hay que acceder a su página web: <https://app.edublocks.org>, donde nos podremos registrar, loguear, o empezar a programar directamente. Si nos registramos nos guardará los proyectos creados.

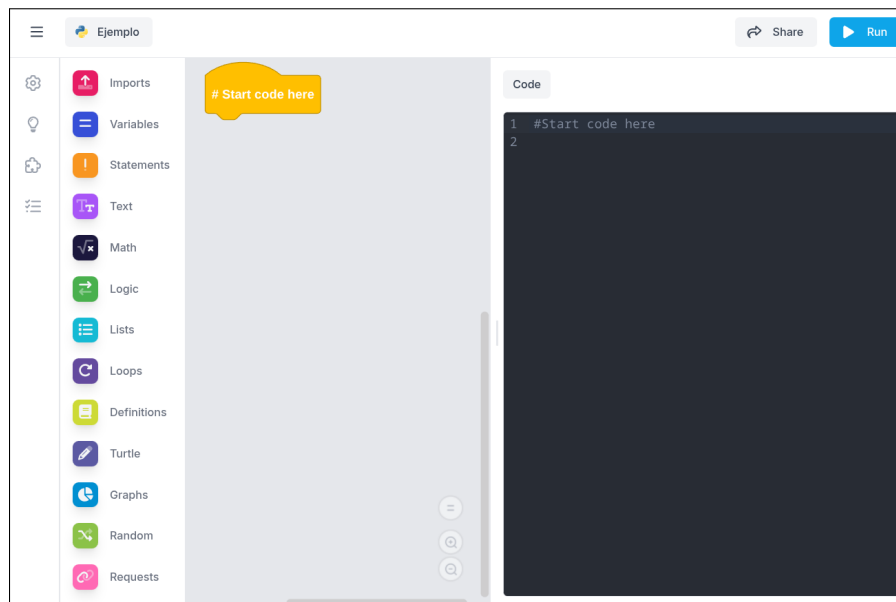
Si seleccionamos **“Python”** nos aparecerá el siguiente pop-up para crear un proyecto:



Ahí podemos elegir:

- **Project name:** El nombre del proyecto. Debemos elegir un nombre significativo para luego poder volver a él.
- **Mode:** Aquí podemos elegir el lenguaje de programación. Tiene que aparecer **Python**.
- **Type:** Podemos elegir si queremos programar por bloques o directamente en texto. Elegiremos **Blocks**.

Al darle al botón **Create**, nos aparecerá el entorno donde podremos empezar a programar:



Se pueden diferenciar tres apartados:

- Grupos de distintos tipos de bloques que vamos a poder usar para programar.
- Lugar donde podemos poner los distintos bloques.
- Código fuente Python de lo que estamos programando.

1.3. Grupos de bloques

Vamos a explicar los distintos grupos de bloques, aunque de momento no hace falta entender qué significan:



Imports

Los lenguajes de programación suelen estar separados en “módulos”, por lo que para usar ciertas características de los lenguajes hay que importar el módulo correspondiente. En EduBlocks hay módulos para fechas, matemáticas...



Variables

Para crear variables debemos usar este grupo, para después poder usarlas. Más adelante hablaremos de las variables.



Statements

Son bloques básicos, para mostrar datos, obtener datos desde el teclado, convertir texto a números...



Text

Bloques para realizar tareas con texto: combinar textos, convertirlo a mayúsculas, minúsculas, contar letras...



Math

En este apartado hay bloques para realizar operaciones matemáticas. Para poder usarlas **hay que importar el módulo de matemáticas**.



Logic

En este grupo **hay bloques muy importantes para la lógica de los programas**. Más adelante veremos cómo usarlos.

Operaciones con listas, que es un tipo de datos.



Lists



Loops

En este grupo **hay bloques muy importantes para realizar tareas repetitivas**. Más adelante los veremos y los usaremos.



Definitions

Bloques para crear funciones.



Turtle

Bloques para dibujar. Es necesario importar el módulo “turtle”.



Graphs

Si queremos crear gráficos (de líneas, de barras horizontales...) podemos usar estos bloques.



Random

Si necesitamos crear números aleatorios podemos usar las funciones de este grupo.



Requests

Los bloques de este grupo nos va a permitir realizar peticiones a páginas web, obtener información de ellas o coger datos de ficheros JSON.

1.4. Empezamos a programar

La mejor manera para aprender es **¡programando!**. Vamos a hacer un programa sencillo para aprender a usar EduBlocks y ver dónde están los bloques más usados y para qué sirven.

1.4.1. ¡Hola Mundo!

Cuando aprendemos a programar en un lenguaje concreto, el primer programa que se suele hacer es conocido como “hola mundo”. Es un programa sencillo que muestra en pantalla un mensaje: “¡Hola mundo!”.

Para programarlo tenemos que:

1. Ir a la categoría **Statements**.
2. Arrastra el bloque que dice: `print("Hello World")`, que sirve para imprimir por pantalla
3. Haz clic en el bloque y cambiar para que ponga: “¡Hola mundo!”.

4. Pulsa el botón



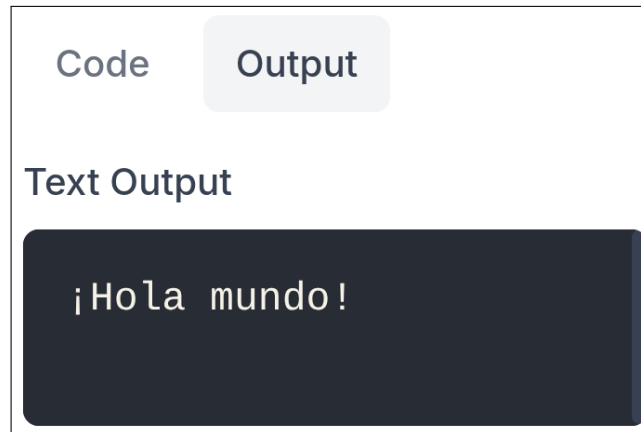
Deberíamos tener en Edublocks lo siguiente:

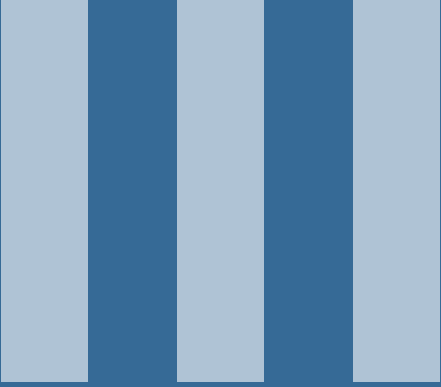
> Código fuente en Python

```
#Start code here
print("¡Hola mundo!")
```



Y el resultado debería ser:





Introducción a Python

1. ¿Qué es Python?

Python es un lenguaje de programación **interpretado** creado por [Guido van Rossum](#) en 1989. Lo creó como un hobby durante las vacaciones y lo llamó Python por la serie [Monty Python's Flying Circus](#).

Los objetivos que Van Rossum tenía para su lenguaje eran:

- Python debería ser **fácil, intuitivo y tan potente** como sus principales competidores.
- El proyecto sería de **código abierto** para que cualquiera pudiera colaborar.
- El código escrito en Python sería tan **comprensible como cualquier texto en inglés**.
- Python debería ser **apto para las actividades diarias** permitiendo la construcción de prototipos en poco tiempo.

Para ver la importancia de Python como lenguaje simple y potente, es importante saber que ha servido como influencia para otros lenguajes posteriores, como [Ruby](#) y [Swift](#).

1.1. Quién lo usa

Python se utiliza en muchos ámbitos dentro de los sistemas de computación:

- **Automatización de tareas:** crear *scripts* para hacer tareas repetitivas en el ordenador. Muy habitual en entornos GNU/Linux.
- **Desarrollo web:** con frameworks como [Flask](#) o [Django](#) se pueden hacer páginas y aplicaciones web.
- **Análisis de datos:** se usa en ciencia de datos y estadísticas con herramientas como Pandas o NumPy.
- **Inteligencia Artificial y Machine Learning:** es uno de los lenguajes más usados a día de hoy para entrenar modelos.
- **Ciencia e investigación:** muy utilizado en universidades y laboratorios para cálculos y simulaciones.
- **Videojuegos:** con bibliotecas como [Pygame](#) se pueden hacer juegos simples. También se puede usar en entornos 3D como [Blender](#) y para hacer scripting en Unity y Unreal.
- **Educación:** ideal para aprender a programar.

1.2. Dónde lo podemos usar

Como ya se ha comentado, Python al ser un lenguaje interpretado, se necesita el intérprete para poder ejecutar los programas realizados, o una plataforma donde poder ejecutar los programas.

Si queremos usar Python en nuestros ordenadores:

- **GNU/Linux:** La gran mayoría de las distribuciones GNU/Linux instalan por defecto Python, ya que se usa como dependencia para muchos programas (sistemas de paquetes, programas de monitorización...).

- **MacOS:** MacOS por defecto tiene instalado el intérprete de Python, pero no suele ser la versión más nueva.
- **Windows:** Es necesario instalar el intérprete. Se puede instalar el [manager](#) desde la Microsoft Store, o alguna de las [versiones disponibles](#), así como desde la [web oficial de Python](#).

También podemos usar Python desde aplicaciones web como:

- [Edublocks](#): Es el sistema que vamos a usar para aprender los conceptos básicos.
- [Jupyter](#): JupyterLab nos permite crear pequeños programas, gráficas y uso para ciencia de datos, junto con documentación, a través de una aplicación web.
- [Google Colab](#): Es un servicio de Jupyter Notebook hosteado por Google con posibilidad de usar recursos de computación como GPUs y [TPUs](#).

En estas plataformas es posible que exista cierta limitación, ya que es posible que no podamos instalar librerías de terceros.

1.3. Versiones de Python

Al igual que sucede con el lenguaje hablado, los lenguajes de programación evolucionan con el tiempo. Los lenguajes de programación suelen tener distintas versiones a lo largo del tiempo, añadiendo mejoras y en algunos casos quitando cosas obsoletas.

Python 2.0 salió en el año 2000 y la última versión de esa rama (2.7.18) en 2020. Python 3 salió en 2008 y fue diseñado para rectificar ciertos errores fundamentales en el diseño del lenguaje. Esto hizo que en ciertos aspectos el lenguaje perdiese retro-compatibilidad (los programas escritos para 2.0 no se pueden ejecutar con el intérprete de 3.x sin hacer cambios).

Durante muchos años python 2.7.X y 3.X han coexistido para dar tiempo a hacer la migración de proyectos grandes, hasta que en 2020 se decidió dejar de dar soporte a las versiones 2.X. Hoy en día todo código que veamos debería ser de la versión 3 en adelante.

Python actualmente está en la versión **3.13** (más concretamente 3.13.5).

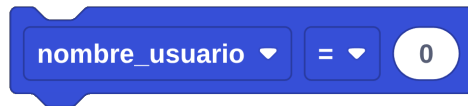
2. Variables

Una **variable** es un nombre que usamos para **guardar un dato** en la memoria RAM del ordenador. Podemos imaginarla como una **caja con una etiqueta**, donde guardamos un valor que luego podemos usar, cambiar o mostrar.

Para crear una variable en EduBlocks tenemos que:

1. Ir a la categoría **“Variables”**.
2. Pulsa el botón  para crear una variable. Nos aparece un *pop-up* para darle un nombre a la variable: `nombre\_usuario`. Verás que ahora en la categoría aparecen 2 bloques nuevos (uno

es para darle valor a la variable y otro para usarla)

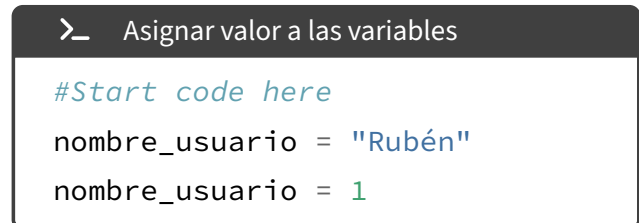
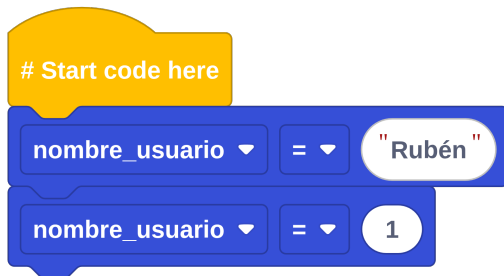


Para asignar valor a la variable



Para usar el valor de la variable

En Python las variables se definen de forma dinámica. Esto quiere decir que no debemos especificar qué tipo de datos va a tener antes de ser declarada.



En la primera sentencia asignamos un valor a la variable, y posteriormente sustituimos el valor por otro. Estos datos son de distinto tipo. En otros lenguajes de programación no se puede realizar la asignación de distintos tipos de datos en la misma variable.

Debido a que los lenguajes de programación tienen un léxico propio, esas palabras no las podremos utilizar como variables, ya que dará error. En la [documentación oficial](#) están las palabras reservadas, como: False, await, else, ...

¡Cuidado!



Hay ciertas [palabras reservadas](#) que no podemos usar como nombres de variables.

2.1. Tipos de datos

En Python, cada dato que usamos tiene un **tipo**, que indica el tipo de información que representa y qué se puede hacer con él. Conocer los tipos de datos es importante para saber cómo usarlos correctamente en un programa.

A continuación una breve descripción de los **tipos más comunes**:

Nombre	Ejemplo en código	Descripción breve
int	<code>edad = 18</code>	Números enteros, sin decimales. Pueden ser positivos o negativos
float	<code>precio = 3.99</code>	Números con decimales. Pueden ser positivos o negativos

Continúa en la siguiente página

str	<code>nombre = "Ruben"</code>	Texto, siempre entre comillas
bool	<code>activo = True</code>	Verdadero (True) o falso (False)
list	<code>frutas = ["pera", "uva"]</code>	Lista ordenada y modificable de elementos
tuple	<code>coordenadas = (3, 5)</code>	Lista ordenada, no modificable (immutable)
dict	<code>persona = {"nombre": "Ruben", "edad": 42}</code>	Datos en forma clave : valor
set	<code>numeros = {1, 2, 3}</code>	Conjunto sin elementos repetidos , sin orden
NoneType	<code>nada = None</code>	Representa la ausencia de valor

Información



Es importante entender los distintos tipos de datos en programación y para qué sirve cada uno.

A continuación vamos a ver ejemplos usando Edublocks y el código en Python real para los tipos de datos más sencillos. Más adelante veremos el resto de los tipos de datos.

Ejercicio



Crea un programa con variables para los primeros cinco tipos de datos e imprime sus valores.

2.1.1. Números enteros (int)

Son números **sin decimales**, positivos o negativos.



```
> Variables con enteros
edad = 20
puntos = -5
```

2.1.2. Números decimales (float)

Son números con decimales (coma flotante). También pueden ser positivos y negativos.

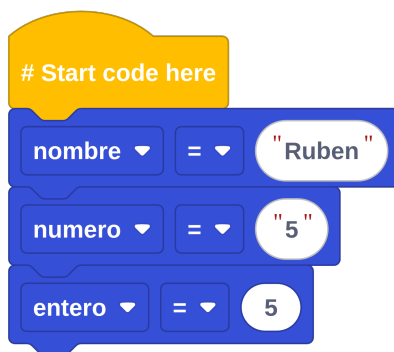


> Variables con floats

```
precio = 14.5
valor = -2.3
```

2.1.3. Cadenas de texto (str)

Son textos entre comillas, que pueden contener letras, números o símbolos. Es importante diferenciar que el contenido de un texto, aunque sea un número, el tipo es distinto.



> Variables con texto

```
nombre = "Ruben"
numero = "5"
entero = 5
```

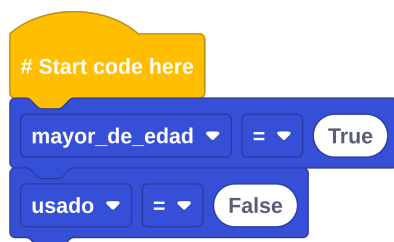
¡Cuidado!



Es importante diferenciar que las variables “numero” y “entero” son dos tipos de datos completamente distintos.

2.1.4. Booleanos (bool)

Solo tienen dos valores posibles: **True** (verdadero) o **False** (falso). Sirve para hacer operaciones lógicas o para guardar datos que sólo pueden ser verdadero o falso.



> Variables con booleans

```
mayor_de_edad = True
usado = False
```

IV

Entrada y salida de datos

1. Salida de datos con print()

Aunque ya hemos visto previamente un ejemplo de cómo mostrar datos, vamos a explicar mejor cómo funciona.



```
> Código fuente en Python

#Start code here
print("Hola")
print(123)
```

En el código vemos que usamos **print**, que es una función interna del lenguaje, al que le pasamos un parámetro. Ese parámetro puede ser un texto u otro tipo de datos.

También podemos pasarle como parámetro el valor de una variable:



```
> Código fuente en Python

#Start code here
nombre_usuario = "Ruben"
print(nombre_usuario)
```

Edublocks nos impide usar variables que no existen, pero si intentamos imprimir una variable que no existe, obtendremos un error como el siguiente:

```
> Código fuente con error

#Start code here
nombre_usuario = "Ruben"
print(nombre)
```

```
> Error al ejecutar

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

¡Cuidado!



Si intentamos usar una variable que no existe, tendremos un error al ejecutar nuestro código.

1.1. Concatenar datos con print()

Print es una función que permite que pasemos más de un parámetro, para poder sacar por pantalla más datos con la misma instrucción.

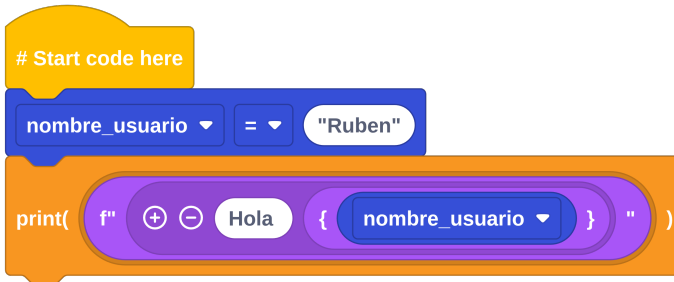
En el grupo de bloques **“Text”** tenemos un bloque para poder añadir más textos al *print*, entre los que también podemos añadir variables.



> Código fuente en Python

```
#Start code here
nombre_usuario = "Ruben"
print("Hola", nombre_usuario)
```

Como alternativa, también podemos **formatear la salida** de una manera más sofisticada, pudiendo hacer que quede mejor. Para ello se hace uso de la función **f**.



> Código fuente en Python

```
#Start code here
nombre_usuario = "Ruben"
print(f"Hola {nombre_usuario}")
```

1.2. Parámetros especiales: “sep”, “end”

Print tiene dos parámetros especiales que tienen un valor por defecto que podemos modificar. Estos parámetros son:

- **sep**: especifica el separador que se usa para concatenar el resto de parámetros. Por defecto es un espacio.
- **end**: especifica cómo se termina el string. Por defecto es un salto de línea, cuyo valor en programación suele ser “\n”.

Un par de ejemplos de cómo se pueden usar estos parámetros:



>_ Código fuente en Python

```
#Start code here
nombre_usuario = "Ruben"
print("Hola", nombre_usuario, sep="_", end="!")
print(1)
print("Te llamas", nombre_usuario, sep=":", end="?")
print(1)
```

La salida debería ser como la imagen de la derecha. Se han añadido los `print(1)` para ver cómo afecta el parámetro `end` al cambiarlo y su valor por defecto. Aparte, si no se ponen, Edublock tiene un fallo y no muestra el resultado.

```
Hola_Ruben!1
Te llamas:Ruben?1
```

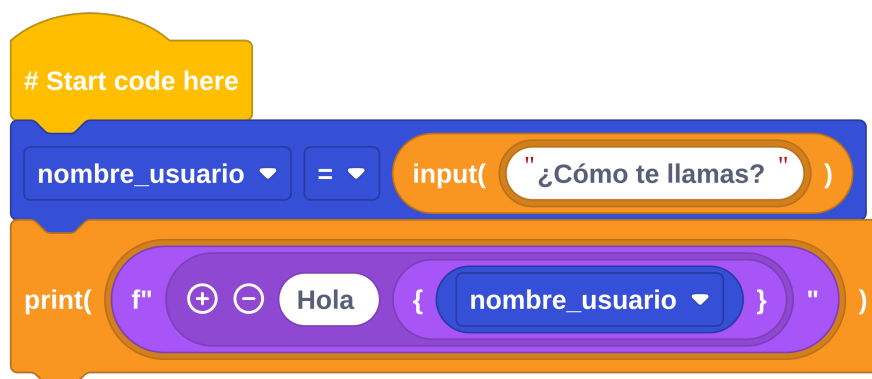
2. Entrada de datos con input()

La gran mayoría de los programas necesitan *input* de los usuarios. En este caso vamos a pedir al usuario que escriba su nombre, y luego lo vamos a saludar. El valor que escriba el usuario **debe guardarse en una variable**.

Los pasos a dar son:

1. Crear una variable.
2. Coge el bloque para asignar valor a la variable
3. Vete a la sección **Statements** y coge el bloque **Input**. Ponlo dentro del bloque anterior y que el texto que sea: “¿Cómo te llamas?”. Esto nos va a pedir el valor al usuario.
4. Imprime por pantalla un texto que diga “Hola” con el valor del dato obtenido.

Deberíamos tener en Edublocks lo siguiente:

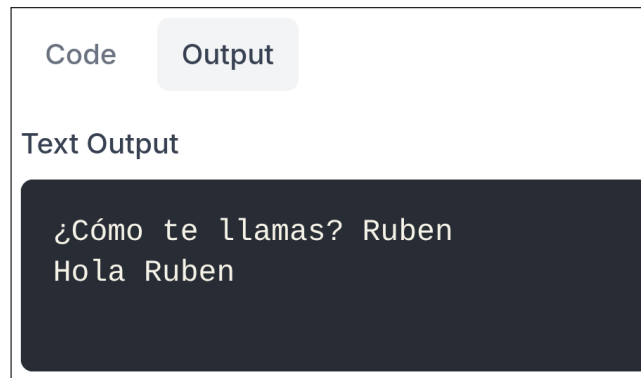


Código de bloques

>_ Código fuente en Python

```
#Start code here
nombre_usuario = input("¿Cómo te llamas? ")
print(f"Hola {nombre_usuario}")
```

Y el resultado, tras usar el programa, es:



Es importante entender que **el input obtenido siempre va a ser de tipo “string”** con lo introducido por el usuario. Da igual que el texto sólo sean números. A continuación veremos cómo convertir datos entre distintos tipos.

Ejercicio

Crea un programa (sin mirar la **solución**) que pregunte al usuario y coja los siguientes datos, para al final imprimir una frase con todos los datos obtenidos:

- Su nombre
- Su edad
- La ciudad donde vive



Conversión de tipos

1. Por qué es necesario convertir datos

A veces es necesario **cambiar el tipo de un dato** para poder realizar ciertas operaciones. Por ejemplo, le pedimos al usuario su edad y queremos ver si es un número primo o no.

Si no realizamos la conversión de datos, el resultado que obtendremos no será el esperado, por lo que nuestro programa no realizará lo que queremos.

Pasar de un tipo de datos a otro se llama **conversión de tipos** o *type casting*, y Python nos permite hacerlo de forma sencilla.

1.1. Funciones de conversión más comunes

A continuación una tabla resumen de cómo se realiza la conversión de datos entre distintos tipos. Algunas de estas conversiones no aparecen en Edublocks.

Función	Qué hace	Ejemplo
<code>int()</code>	Convierte a número entero	<code>int("5") → 5</code>
<code>float()</code>	Convierte a número con decimales	<code>float("3.14") → 3.14</code>
<code>str()</code>	Convierte a cadena de texto	<code>str(10) → "10"</code>
<code>bool()</code>	Convierte a valor lógico (True o False)	<code>bool(0) → False</code>
<code>list()</code>	Convierte a lista	<code>list("hola") → ["h", "o", "l", "a"]</code>
<code>set()</code>	Convierte a conjunto (sin elementos repetidos)	<code>set([1, 2, 2]) → {1, 2}</code>
<code>tuple()</code>	Convierte a tupla	<code>tuple([1, 2]) → (1, 2)</code>

1.2. Conocer el tipo de dato

Si queremos conocer el tipo de datos de una variable, podemos hacer uso de la función `type()`. A continuación un ejemplo de código:

> Conocer el tipo de datos

```
nombre = "Ruben"
print(type(nombre))
print(type(nombre).__name__)
```

Pregunta



¿Qué diferencia hay entre las dos últimas sentencias?

VI

Operadores básicos

1. Qué son los operadores

En programación, los **operadores** son símbolos que se usan para hacer cálculos o comparar valores. En algunos casos sólo se van a poder utilizar con algunos tipos de datos concretos, y otros con cualquier tipo, por lo que hay que asegurar que se usan de manera correcta.

A continuación se van a explicar los distintos tipos de operadores existentes, y aunque se van a utilizar con números, es importante entender que lo **habitual es hacerlo con variables**.

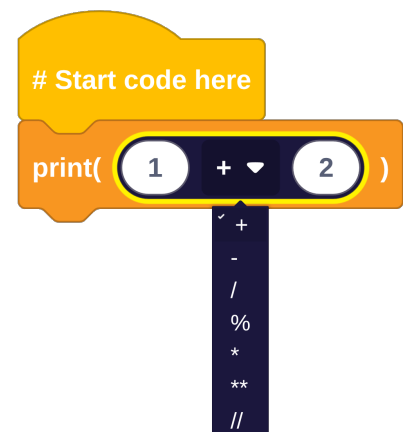
2. Operadores aritméticos

Sirven para hacer **operaciones matemáticas**, por lo que se utilizan con tipos de datos numéricos.

Operador	Nombre	Ejemplo	Resultado
+	Suma	3 + 2	5
-	Resta	7 - 4	3
*	Multiplicación	5 * 2	10
/	División	7 / 2	3.5
//	División entera	7 // 2	3
%	Módulo (resto)	7 % 2	1
**	Potencia	2 ** 3	8

Hay que tener en cuenta que:

- La división `/` siempre da un número con decimales (**float**), aunque sea exacta.
- La división entera `//` dará el resultado sin contar los decimales, y genera un **int**.
- El módulo `%` dará el resto de la división, en tipo **int**.
- Si se combinan dos tipos de datos, el resultado usará el tipo de datos necesario para poder mostrar el resultado.



En Edublocks estas operaciones están en el apartado **Math**, y se puede seleccionar el operador a través de un desplegable.

Ejercicio



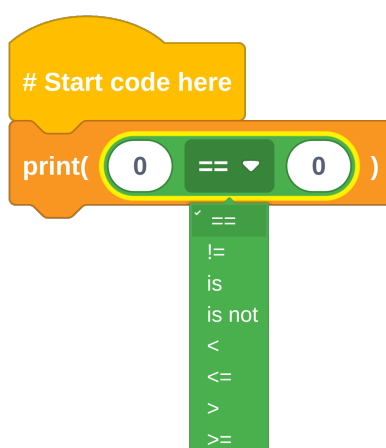
Escribe un programa que calcule el área de un rectángulo (base x altura) usando dos variables (sin mirar la **solución**). Los datos se los tienes que pedir al usuario. Recuerda [la conversión de tipos de datos](#).

3. Operadores lógicos

Se usan para **comparar valores** y obtener un resultado lógico: **verdadero (True)** o **falso (False)**.

Operador	Significado	Ejemplo	Resultado
<code>==</code>	Igual que	<code>3 == 3</code>	<code>True</code>
<code>!=</code>	Distinto de	<code>3 != 5</code>	<code>True</code>
<code><</code>	Menor que	<code>2 < 5</code>	<code>True</code>
<code><=</code>	Menor o igual que	<code>3 <= 3</code>	<code>True</code>
<code>></code>	Mayor que	<code>7 > 10</code>	<code>False</code>
<code>>=</code>	Mayor o igual que	<code>4 >= 5</code>	<code>False</code>
<code>is</code>	Es el mismo objeto	<code>a is b</code>	<code>True</code> si a y b son el mismo objeto
<code>is not</code>	No es el mismo objeto	<code>a is not b</code>	<code>True</code> si son objetos diferentes

Todos los ejemplos utilizados son con números, pero lo habitual es hacer uso de variables. En Edublocks está en el grupo **Logic**.



Ejercicio

¿Qué valor devuelven las siguientes expresiones? (piénsalas primero y luego compruébalo en Python, antes de mirar la **solución**):

- `3 == "3"`
- `4 <= 4.0`
- `"ho!a" != "Ho!a"`

3.1. Operadores especiales: “is” y “is not”

Existen dos operadores especiales que tienen en cuenta la gestión de memoria de los objetos que se usan en el programa. Estos dos operadores se usan cuando el programa a realizar es avanzado y queremos gestionar la memoria que se usa, por lo que no los vamos a usar.

>_ Uso de “is” y “is not”

```
a = [1, 2, 3]
b = a          # b apunta al mismo objeto que a, en memoria
c = [1, 2, 3]  # c tiene el mismo valor, pero es otro objeto

print(a is b) # True
print(a is c) # False
print(a == c) # True, porque los valores son iguales
```

[Aquí](#) se puede ver más ejemplos de su uso y una explicación avanzada en algunos casos especiales.

¡Atención!

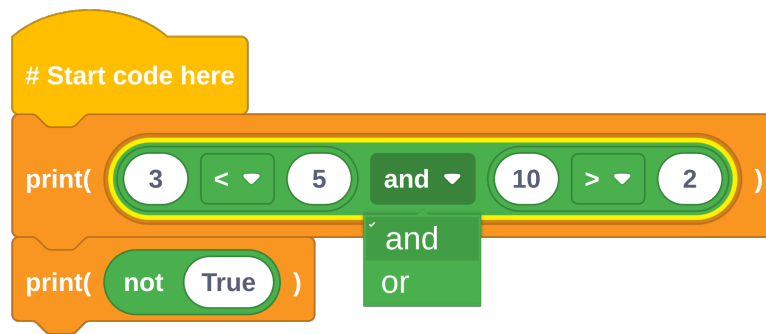


No vamos a hacer uso de estos dos operadores especiales.

4. Operadores lógicos combinados: “and”, “or”, “not”

Sirven para **combinar condiciones**. En el siguiente capítulo los entenderemos mejor.

Operador	Significado	Ejemplo	Resultado
<code>and</code>	Y (ambas verdaderas)	<code>3 < 5 and 10 > 2</code>	<code>True</code>
<code>or</code>	O (al menos una verdadera)	<code>3 > 5 or 10 > 2</code>	<code>True</code>
<code>not</code>	No (niega el valor)	<code>not True</code>	<code>False</code>



Ejercicio

¿Qué valor devuelven las siguientes expresiones? (piénsalas primero y luego compruébalo en Python, antes de mirar la **solución**):



- `True == True and False == True`
- `5 > 3 and 4 < 2`
- `5 > 3 or 4 < 2`
- `not (7 <= 7)`
- `not (0)`

VII

Estructuras de control

1. Estructuras de control

Las **estructuras de control** son fundamentales en programación, por lo que todos los lenguajes de programación las tienen.

Estas estructuras nos permiten que el programa **tome decisiones** o **repita acciones** de forma automática, según ciertas condiciones. Gracias a ellas, podemos crear programas más inteligentes y útiles.

Las estructuras de control se pueden diferenciar por:

- **Condicionales**
- **Bucles**

Es importante entender que las estructuras de control van a formar bloques de código que pueden ejecutarse, o no. Estos “bloques” se van a diferenciar porque va a usar **sangría** dentro de la estructura. Más adelante vamos a verlo mejor en los ejemplos.

¡Cuidado!



La sangría en el código Python es muy importante.

Otros lenguajes de programación pueden tener más estructuras aparte de las que vamos a ver, o con otras palabras registradas. Las que vamos a ver suelen ser común en la gran mayoría de lenguajes, **por lo que es muy importante entender cómo funcionan**.

¡Atención!



Las estructuras de control son fundamentales en programación. Es importante aprender a usarlas.

1.1. Condicionales

Las estructuras condicionales permiten que el programa **elija entre distintas opciones** según si una condición es verdadera o falsa. Teniendo en cuenta la condición, se ejecutará una parte de código u otra.

1.1.1. Condicional “if”

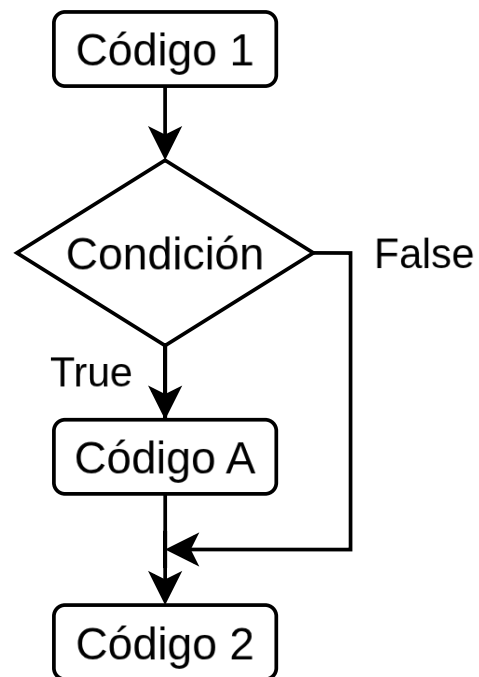
El condicional **if** se usa para indicar que **si se cumple una condición se debe ejecutar un bloque de código**. El diagrama de flujo sería el siguiente:

Vamos a entender qué sucede:

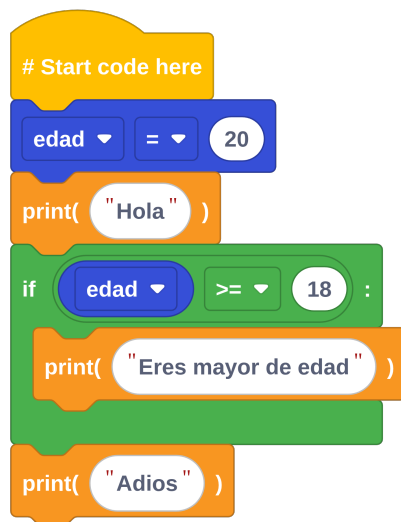
- El “código 1” se va a ejecutar siempre en nuestro programa.
- Se llega al condicional **if**, que tiene forma de rombo, y se analiza la

condición:

- Si la condición se cumple: se ejecuta el bloque “código A”.
 - Si la condición no se cumple, se salta el bloque y se sigue el flujo normal.
- El “código 2” siempre se va ejecutar.



Para verlo en código con Edublocks, y su correspondencia en Python, tenemos el siguiente ejemplo:



```
> Código fuente en Python

#Start code here
edad = 18
print("Hola")
if edad >= 18:
    print("Eres mayor de edad")
print("Adios")
```

El bloque **if** en Edublocks está dentro del grupo **Logic**. Vamos a entender qué hace el código:

1. Se crea una variable, con el tipo de datos **int**, con un valor de 20.
2. Se imprime por pantalla “Hola”.
3. Se analiza la condición `edad >= 18`. ¿Es esta condición **True**? Sí, porque 20 es mayor a 18, por lo tanto se imprime el código que está dentro del bloque **if**: “Eres mayor de edad”.

¡Atención!



Fíjate en la sangría del código después de la sentencia **if**.

4. Se imprime por pantalla “Adios”.

Vamos a analizar el código `if edad >= 18:`, **porque es muy importante escribirlo bien**, ya que si no tendremos errores de léxico.

- **if**: es la palabra clave de la estructura condicional.
- **edad >= 18**: es la condición que se analiza.
- **:**: la condición siempre termina con los dos puntos :

Es muy importante ver que el bloque de código dentro del `if` está **sangrado a la derecha** con espacios. Esto significa que todo el código que esté a esa altura pertenece al bloque dentro del `if`.

Todas las líneas dentro del bloque **deben tener la misma sangría**. De no ser así, al ejecutar el código tendremos un error `"IndentationError: unexpected indent"` o el código se ejecutará donde no queremos.

Ejercicio



Analiza los siguiente ejemplos y comprueba cómo es la **sangría** del código. ¿Cuándo se ejecuta cada sentencia `print`?

>_ Código

correcto

```
#Start code here
edad = 18
print("Hola")
if edad >= 18:
    print("1")
    print("2")
    print("3")
print("Adios")
```

>_ Mala sangría

```
#Start code here
edad = 18
print("Hola")
if edad >= 18:
    print("1")
    print("2")
    print("3")
print("Adios")
```

>_ Código ¿correcto?

```
#Start code here
edad = 18
print("Hola")
if edad >= 18:
    print("1")
    print("2")
print("3")
print("Adios")
```

¡Cuidado!



La sangría en el código Python es muy importante.

1.1.1.1. Estructuras de control anidadas

Las estructuras de control se pueden anidar. Esto quiere decir que dentro del bloque de una estructura se pueden añadir otras estructuras nuevas (en este caso condicionales).

Ejercicio



Analiza el siguiente ejemplo, comprueba cómo es la **sangría** del código. ¿Qué se imprime en pantalla? Y si...

- edad = 17?
- edad = 67?



> Código fuente en Python

```
#Start code here
edad = 20
print("Hola")
if edad >= 18:
    print("Eres mayor de edad")
    if edad >= 65:
        print("¿Estás jubilado?")
print("Adios")
```

Cuando queremos anidar una nueva estructura de control dentro de otra hay que continuar la sangría, ya que eso determina dónde se va a ejecutar.

¡Atención!



¡RECUERDA! La sangría en el código Python es muy importante.

1.1.2. Condicional “if else”

La palabra reservada **else** se utiliza para indicar el bloque de código que se va a ejecutar **cuando la condición del if no se cumple**.

Información



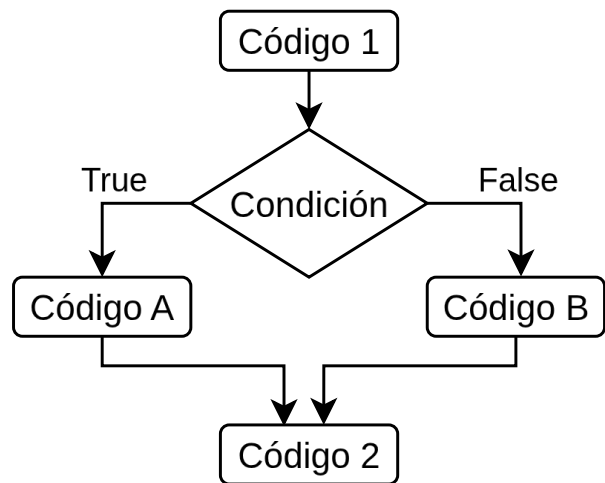
En un condicional “if-else”, uno de los dos bloques se va a ejecutar.

El diagrama de flujo sería el siguiente::

- El “código 1” se va a ejecutar siempre en

nuestro programa.

- Se llega al condicional **if**, y se analiza la condición:
 - Si la condición se cumple: se ejecuta el bloque “código A”.
 - Si la condición no se cumple: se ejecuta el bloque del **else** “código B”.
- El “código 2” siempre se va ejecutar.



Tal como se puede ver, al utilizar una infraestructura “**if-else**” siempre va a haber un bloque de código que se va a ejecutar. Vamos a usar el siguiente código como ejemplo:



> Código fuente en Python

```

#Start code here
edad = 17
print("Hola")
if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
print("Adios")
  
```

El bloque **else** en Edublocks también está dentro del grupo **Logic**. Vamos a entender qué hace el código:

1. Se crea una variable con un valor de 17, es de tipo de datos **int**.
2. Se imprime por pantalla “Hola”.
3. **edad >= 18**. ¿Es esta condición **True**? No, **por lo tanto se salta al bloque else y se ejecuta su contenido**.
4. Se imprime por pantalla “Adios”.

Ejercicio

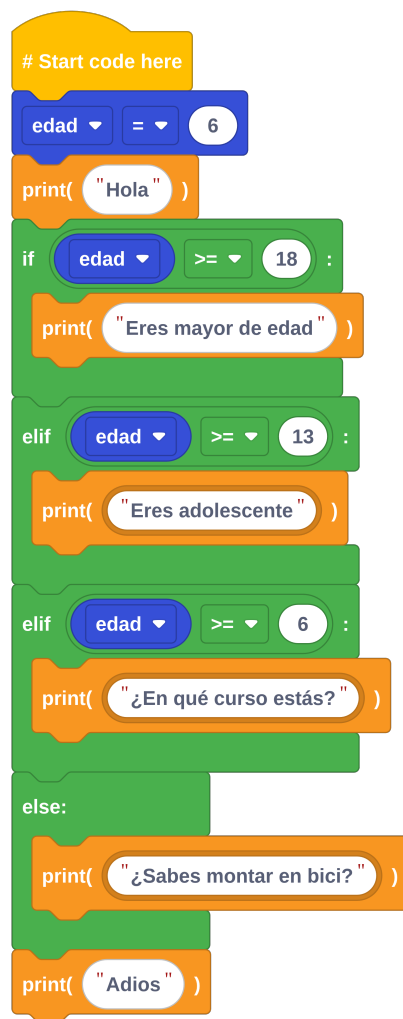


Añade en el **else**, sin ver la **solución**:

- si es mayor de 13: que escriba: "Eres adolescente"
- si no, pero es mayor a 6: escriba: "¿En qué curso estás?"
- si no: escriba: "¿Sabes montar en bici?"

1.1.3. Condicional "elif"

elif se puede resumir en que es la unión de un **else** y un **if**, sin tener que hacer un bloque dentro del **else**. Sirve para evaluar cuando hay más de una condición, y dejará de evaluar las opciones cuando se encuentre con la primera condición que es verdadera.



> Código fuente en Python

```
#Start code here
edad = 6
print("Hola")
if edad >= 18:
    print("Eres mayor de edad")
elif edad >= 13:
    print("Eres adolescente")
elif edad >= 6:
    print("¿En qué curso estás?")
else:
    print("¿Sabes montar en bici?")
print("Adios")
```

En el ejemplo anterior se puede ver cómo se han añadido varias condiciones. Vamos a ver qué sucede cuando **edad = 6**:

1. **edad >= 18**: No, por lo tanto salta a la alternativa. Anteriormente sería un **else**, pero en este código existe una nueva sentencia condicional.
2. **edad >= 13**: No, por lo tanto saltamos.
3. **edad >= 6**: ¡Sí! por lo tanto ejecutamos el bloque de código dentro de esta sentencia, y no se sigue

analizando el resto de la “cascada” (el `else`).

Ejercicio



Compara este código con la **solución del ejercicio anterior**.

1.1.4. Ejercicio

Realiza el siguiente ejercicio. Existen varias **soluciones posibles**, así que intenta hacer dos: utilizando estructuras anidadas, y por otra parte usando `elif`.

Ejercicio



Escribe un programa (sin mirar la **solución**) que pida una nota del 0 al 10 y diga si es:

- “Suspenso” (< 5)
- “Aprobado” (5 a 6.9)
- “Notable” (7 a 8.9)
- “Sobresaliente” (9 a 10)

1.2. Bucles

Un **bucle** es una estructura que permite **repetir un bloque de instrucciones varias veces**, de forma automática **mientras se cumpla una condición**. Python tiene dos tipos principales de bucles, llamados `while` y `for`.

Sirven para hacer tareas repetitivas sin tener que escribir el mismo código una y otra vez, ya que eso sería muy ineficiente. Por ejemplo, si queremos mostrar los números del 1 al 10, podemos usar un bucle en lugar de escribir 10 veces `print()`.

Al escribir un bucle hay que tener en cuenta que la condición en algún momento no se cumpla, porque si no estaríamos en lo que se llama **bucle infinito**.

¡Cuidado!



Hay que tener cuidado al escribir bucles para no hacer **bucles infinitos**.

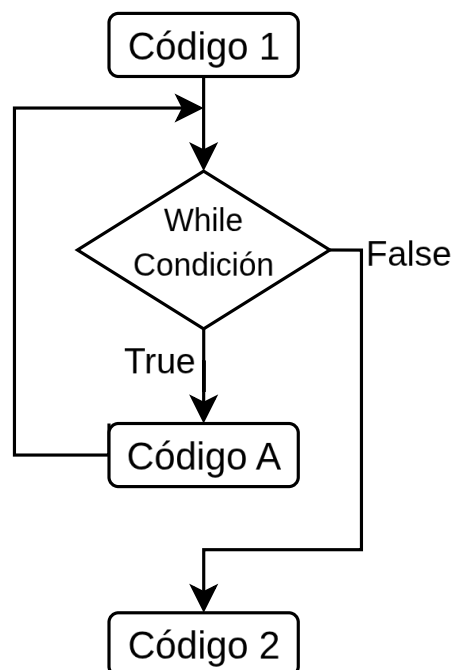
1.2.1. Bucle “while”

Sirve para repetir un bloque de instrucciones mientras se cumpla una condición. Cuando la condición deja de cumplirse, el bucle se detiene y se continúa con el flujo normal del programa.

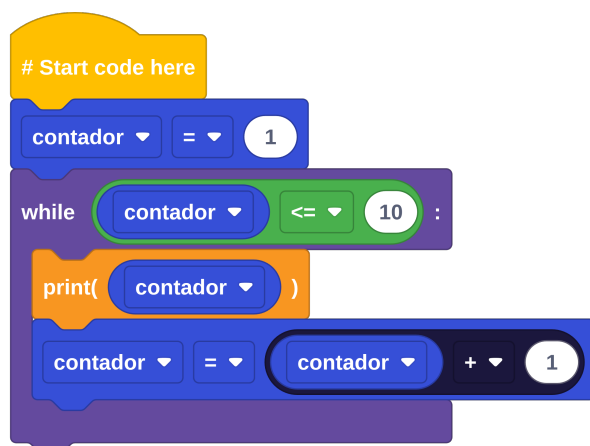
Teniendo en cuenta el diagrama de flujo:

1. El “código 1” se va a ejecutar siempre en nuestro programa.

2. Se llega a la sentencia “while condición” y se analiza:
 - Si la condición se cumple: se ejecuta el “código A” y se vuelve al paso 2.
 - Si no se cumple: se salta el bloque del bucle.
3. Se ejecuta “código 2”.



Vamos a hacer un ejemplo que imprima los primeros 10 números. En Edublocks los bucles están en el grupo **Loops**.



```
>_ Código fuente en Python

#Start code here
contador = 1
while contador <= 10:
    print(contador)
    contador = contador + 1
```

Del código anterior podemos destacar:

- `contador = 1`: inicializamos una variable con el valor **1**.
- `while`: es la palabra clave para indicar que vamos a entrar en el bucle.
 - `contador <= 10`: es la condición para que entre en el bloque del bucle.
 - `contador = contador + 1`: esta sentencia es la más importante, ya que hacemos que la variable se incremente en cada iteración. De no añadirla, **estaríamos en un bucle infinito porque contador siempre valdría 1**.

Ejercicio



Haz un programa (sin mirar la **solución**) que pida un número al usuario, e imprima todos los números en orden inverso hasta llegar a 0.

Ejercicio

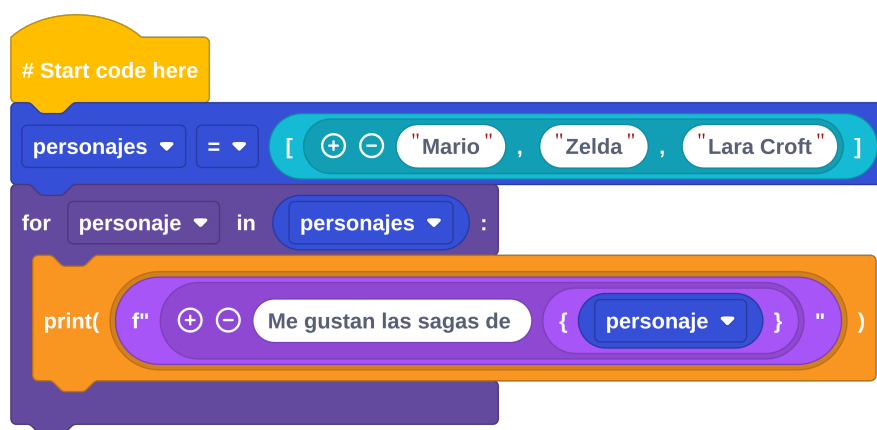


Haz un programa (sin mirar la **solución**) que sume todos los números del 1 al 100.

1.2.2. Bucle “for”

El bucle **for** se utiliza para recorrer elementos de una secuencia: una lista (**list**), un conjunto **set**, una cadena de texto, un rango de números con **range()**...

A continuación un ejemplo usando una lista:



> Código fuente en Python

```
#Start code here
personajes = ["Mario", "Zelda", "Lara Croft"]
for personaje in personajes:
    print(f"Me gustan las sagas de {personaje}")
```

A continuación la explicación más importante del código anterior:

- **personajes**: es una variable de tipo lista que contiene distintos elementos.
- **for**: es la palabra clave para indicar que vamos a entrar en un bucle
 - **personaje**: sirve para crear una nueva variable a la que se le va a asignar cada elemento de la lista a medida que el bucle recorre la lista.
 - **in personajes**: indica que se va a recorrer el contenido de la lista **personajes**.

¡Atención!



Es importante elegir un buen nombre de la variable con la que iteramos, tiene que tener sentido.

Otro ejemplo con la función **range()**. Esta función acepta distinta cantidad de parámetros:

- **range(10)**: genera una lista **desde 0 hasta el número indicado menos uno** (no se incluye el número indicado).
- **range(start, stop, step)**: genera una lista empezando por **start** hasta el **stop** menos uno, aumentando en pasos de **step** (si no se pone, por defecto es 1).



> Código fuente en Python

```
#Start code here
for i in range(10):
    print(f"El {i}")
print("Nuevo contador")
for i in range(-10, 10):
    print(f"El {i}")
```

Ejercicio



Crea un bucle **for** desde 0 hasta 400 usando pasos de 5. **Solución.**

Ejercicio



Haz un programa (sin mirar la **solución**) que pida un número al usuario, e imprima su tabla de multiplicar hasta el 10 (si el número introducido es mayor a 10, la tabla será hasta el mismo número).

1.2.3. Instrucciones útiles en los bucles

Existen unas instrucciones que podemos utilizar dentro de los bucles que pueden ser útiles:

- **break**: Sirve para salir del bucle antes de que termine.
- **continue**: Sirve para saltar a la siguiente iteración del bucle (sin ejecutar nada más en la iteración actual).

Ejercicio



Haz un programa (sin mirar la **solución**) que imprima del 1 al 30, **pero**:

- Salte los números 7, 15 y 23 sin imprimirlos.
- Pare el bucle si llega al número 27.

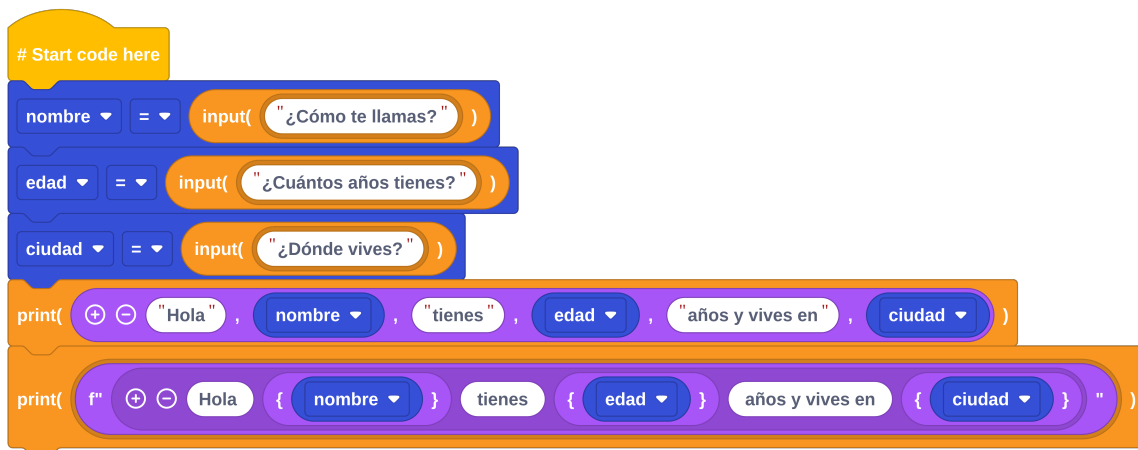


Ejercicios resueltos

1. Programa discoteca

1. Pedir DNI a la persona.
2. Comprobar si la edad es mayor o igual que 18 ($edad \geq 18$).
3. Si es mayor o igual $\rightarrow$ dejar entrar.
4. Si es menor $\rightarrow$ no dejar entrar y apartar de la cola.
5. Dejar que avance la siguiente persona.
6. Ir a la instrucción 1.

2. Programa con inputs



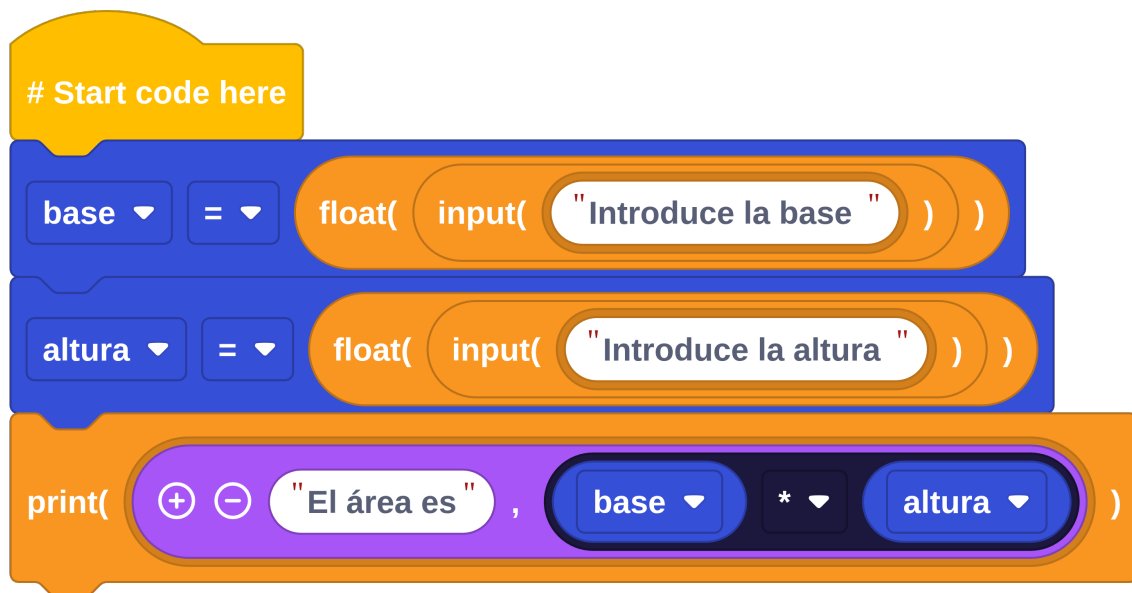
> Código fuente en Python

```
#Start code here
nombre = input("¿Cómo te llamas?")
edad = input("¿Cuántos años tienes?")
ciudad = input("¿Dónde vives?")
print("Hola", nombre, "tienes", edad, "años y vives en", ciudad)
print(f"Hola {nombre} tienes {edad} años y vives en {ciudad}")
```

3. Ejercicio aritméticos

Escribe un programa que calcule el área de un rectángulo (base x altura) usando dos variables. Los datos se los tienes que pedir al usuario.

Recuerda usar [la conversión de tipos de datos](#).



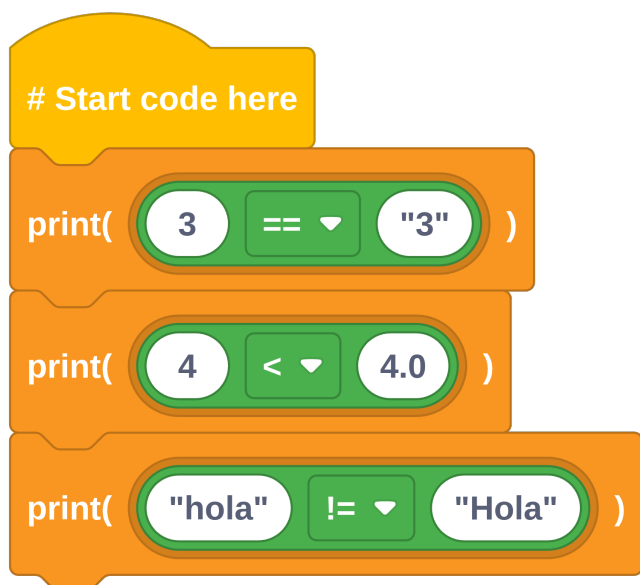
> Código fuente en Python

```
#Start code here
base = float(input("Introduce la base "))
altura = float(input("Introduce la altura "))
print("El área es", base * altura)
```

3.1. Ejercicios de operadores lógicos

¿Qué valor devuelven las siguientes expresiones? (piénsalas primero y luego compruébalo en Python):

- `3 == "3"`
- `4 <= 4.0`
- `"hola" != "Hola"`



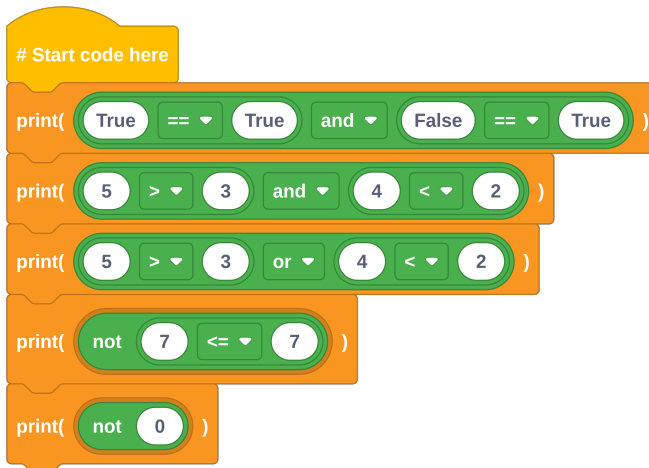
> Código fuente en Python

```
#Start code here
print(3 == "3")
print(4 < 4.0)
print("hola" != "Hola")
```

3.2. Ejercicios de operadores lógicos combinados

¿Qué valor devuelven las siguientes expresiones? (piénsalas primero y luego compruébalo en Python):

- `True == True and False == True`
- `5 > 3 and 4 < 2`
- `5 > 3 or 4 < 2`
- `not (7 <= 7)`
- `not (0)`



> Código fuente en Python

```
#Start code here
print(True == True and False == True)
print(5 > 3 and 4 < 2)
print(5 > 3 or 4 < 2)
print(not 7 <= 7)
print(not 0)
```

4. Ejercicios de condicionales

4.1. Ejercicio de notas

Escribe un programa que pida una nota del 0 al 10 y diga si es:

- “Suspenso” (< 5)
- “Aprobado” (5 a 6.9)
- “Notable” (7 a 8.9)
- “Sobresaliente” (9 a 10)

Solución con “else anidados” y “elif”:



> Código fuente en Python

```

#Start code here
print("¿Qué nota has sacado?")
nota = float(input(""))
if nota <= 5:
    print("Suspense")
else:
    if nota <= 6.9:
        print("Aprobado")
    else:
        if nota <= 8.9:
            print("Notable")
  
```

> Código fuente en Python

```

#Start code here
print("¿Qué nota has sacado?")
nota = float(input(""))
if nota <= 5:
    print("Suspense")
elif nota <= 6.9:
    print("Aprobado")
elif nota <= 8.9:
    print("Notable")
else:
    print("Sobresaliente")
  
```

```
else:
    print("Sobresaliente")
```

4.2. Ejercicio “else anidado”

Amplía el código del ejemplo **if-else** para que cuando entre en el **else** se compruebe:

- si es mayor de 13: que escriba: “Eres adolescente”
- si no, pero es mayor a 6: escriba: “¿En qué curso estás?”
- si no: escriba: “¿Sabes montar en bici?”



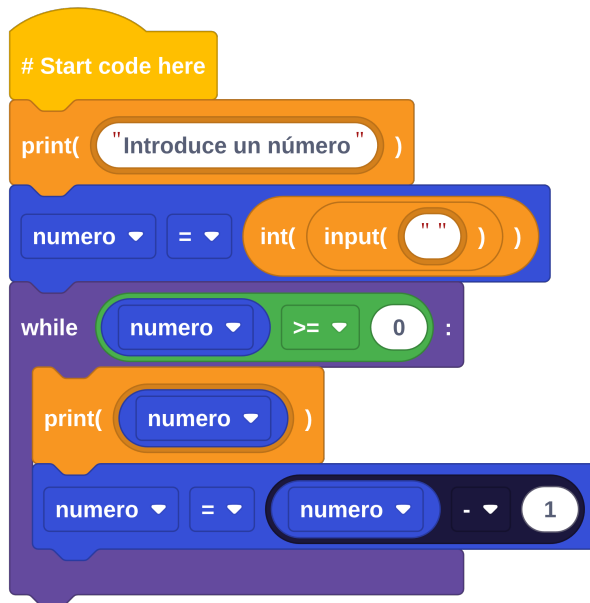
> Código fuente en Python

```
#Start code here
edad = 6
print("Hola")
if edad >= 18:
    print("Eres mayor de edad")
else:
    if edad >= 13:
        print("Eres adolescente")
    else:
        if edad >= 6:
            print("¿En qué curso estás?")
        else:
            print("¿Sabes montar en bici?")
print("Adios")
```

5. Ejercicios de bucles

5.1. Bucle inverso

Haz un programa que pida un número al usuario, e imprima todos los números en orden inverso hasta llegar a 0.

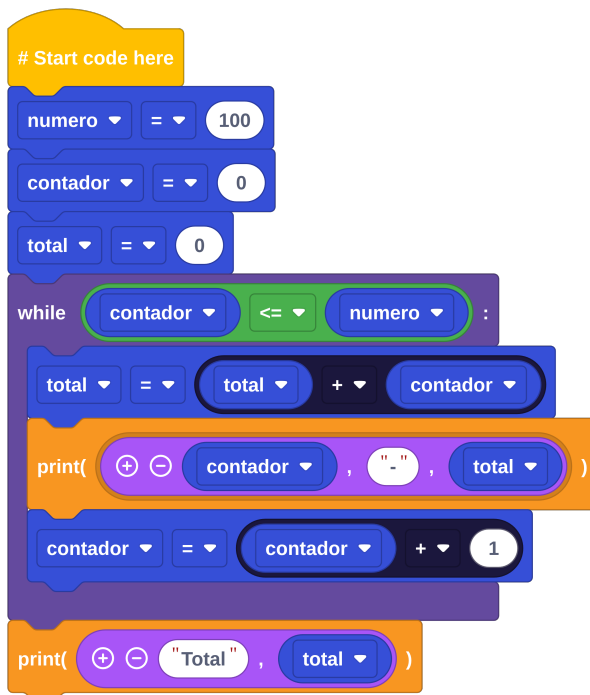


> Código fuente en Python

```
#Start code here
print("Introduce un número")
numero = int(input(""))
while numero >= 0:
    print(numero)
    numero = numero - 1
```

5.2. Contador

Haz un programa que sume todos los números del 1 al 100.

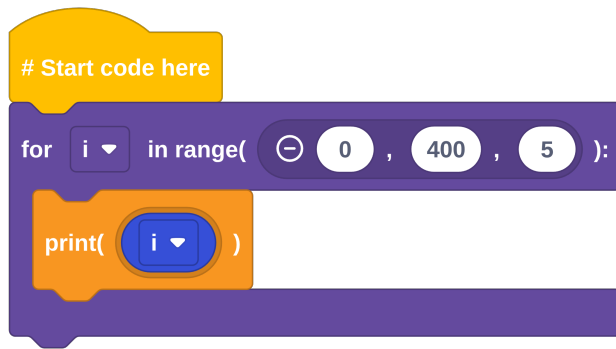


> Código fuente en Python

```
#Start code here
numero = 100
contador = 0
total = 0
while contador <= numero:
    total = total + contador
    print(contador, "-", total)
    contador = contador + 1
print("Total", total)
```

5.3. Bucle for

Crea un bucle **for** desde 0 hasta 400 usando pasos de 5.

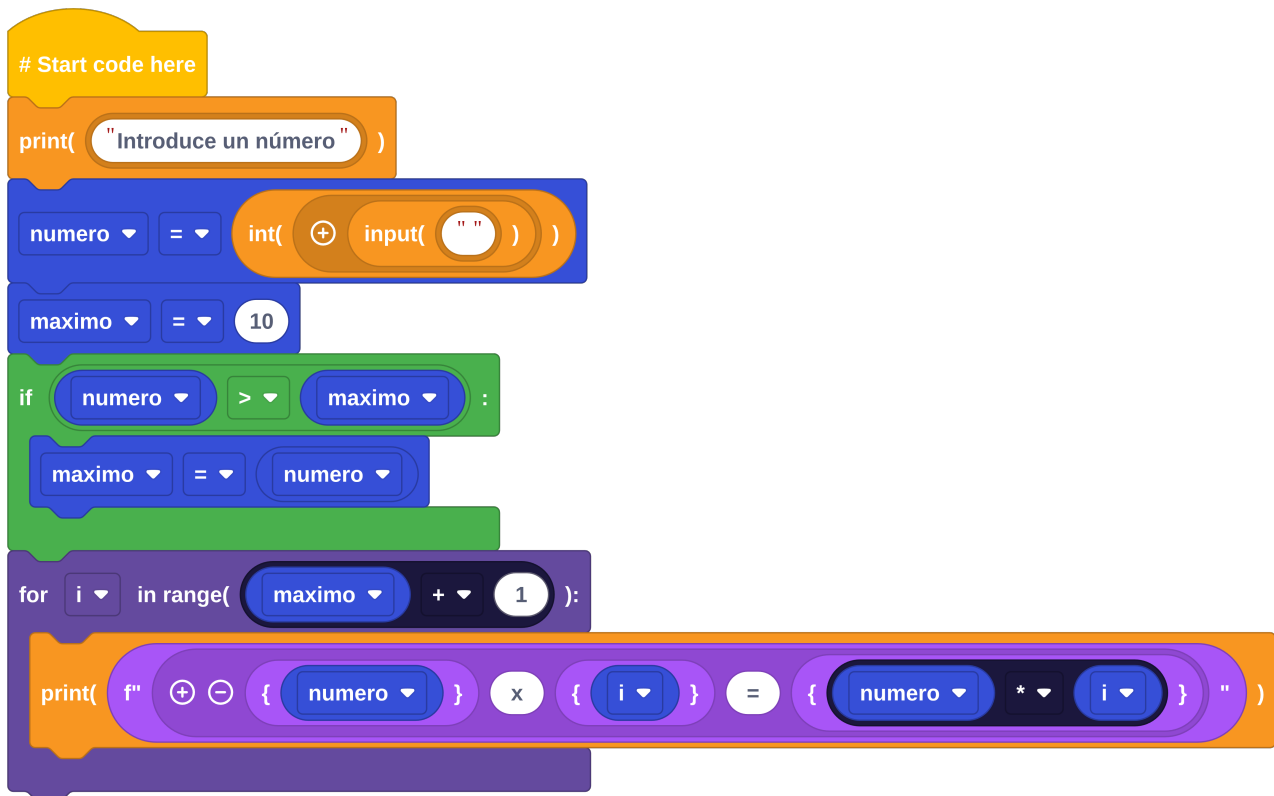


> Código fuente en Python

```
#Start code here
for i in range(0, 400, 5):
    print(i)
```

5.4. Tabla de multiplicar

Haz un programa que pida un número al usuario, e imprima su tabla de multiplicar hasta el 10 (si el número introducido es mayor a 10, la tabla será hasta el mismo número).



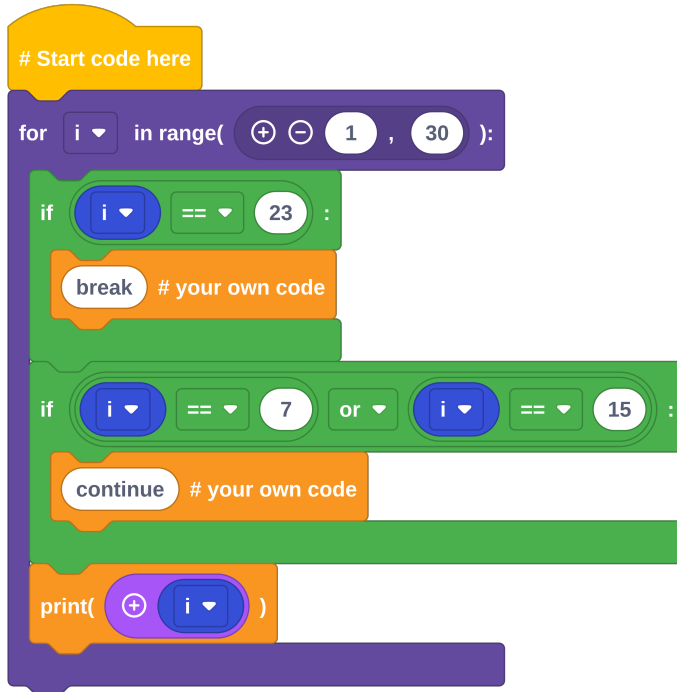
> Código fuente en Python

```
#Start code here
print("Introduce un número")
numero = int(input(""))
maximo = 10
if numero > maximo:
    maximo = numero
for i in range(maximo + 1):
    print(f"{numero}x{i}={numero * i}")
```

5.5. Instrucciones break y continue

Haz un programa que imprima del 1 al 30, **pero**:

- Salte los números 7 y 15 sin imprimirlos.
- Pare el bucle si llega al número 23.



> Código fuente en Python

```
#Start code here
for i in range(1, 30):
    if i == 23:
        break # your code
    if i == 7 or i == 15:
        continue # your code
    print(i)
```