digit@lers

En minutos comenzamos…

Lic. Jorge Angel PAEZ

digit@lers

Python para no programadores.

Parte 1

Introducción, conceptos y condicionales. Programación y estructura de un programa. Preparación del entorno. Concepto de IDE, editor de código y otras herramientas. Fundamentos del lenguaje. Tipos de datos. Operaciones básicas. Condicionales.

telecom

EDUCACIÓN IT

INICIEMOS LA GRABACIÓN

CONSIDERACIONES INICIALES

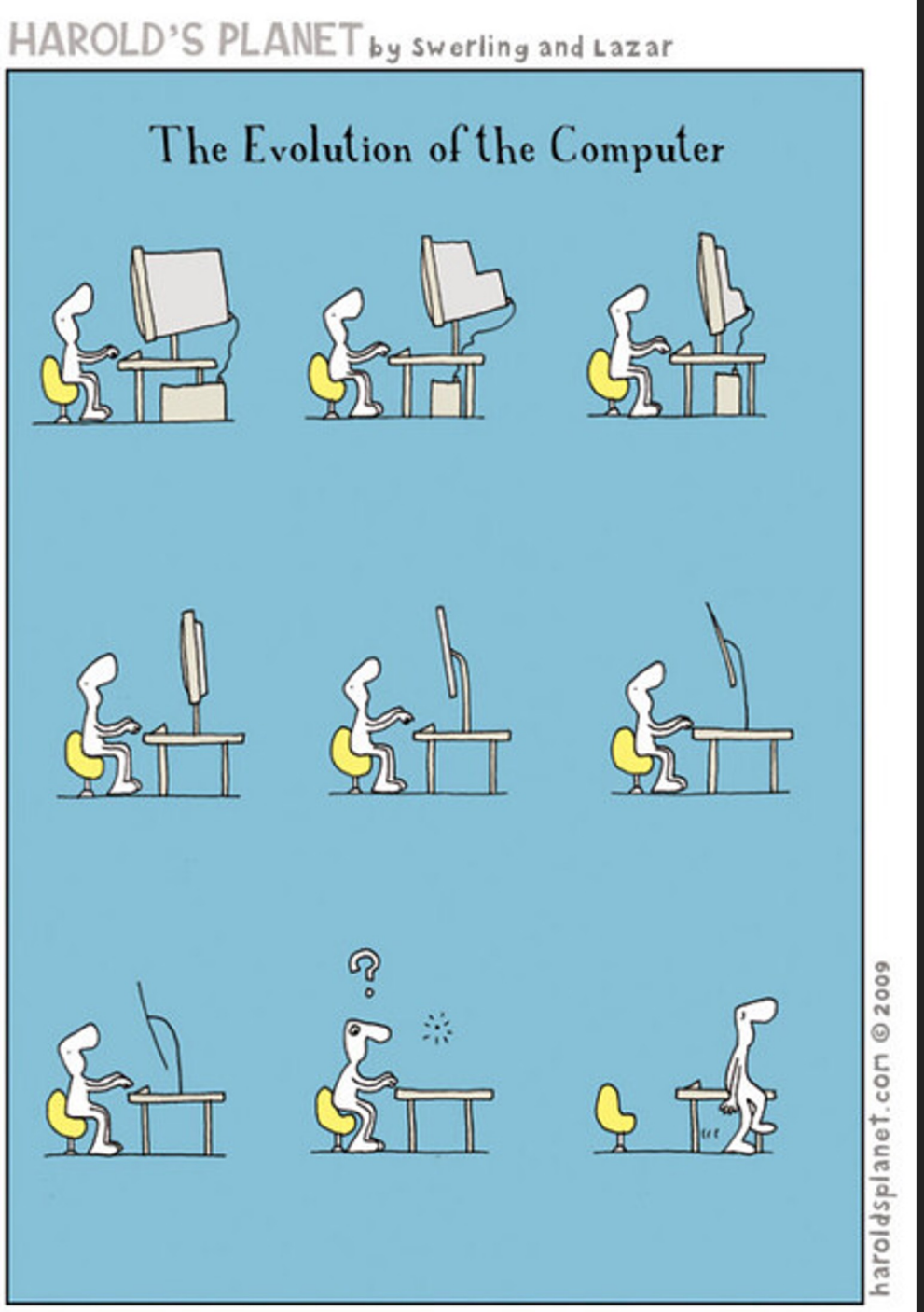
- Las clases grabadas pueden demorar hasta 48 horas
- Utilizaremos el chat de ALUMNI para comunicarnos.
- Accede al contenido (todo el contenido de la cursada se encuentra disponible en ALUMNI - Digitalers Python Developer)
- Clases de 3 horas, LUNES Y MIERCOLES de 19:00 A 22:00 (break de 15 minutos)
- Mas o menos cada clase cubrirá 1 modulo de ALUMNI.
- Inicio: 5/6/2023 - Fin: 13/11/2023
- Durante el mes de Agosto habrá dos días que no dicten clases ya que se impartirán jornadas de Habilidades Blandas.
- Trabajo Practico integrador (Obligatorio)
- Enviar siempre los desafíos del ALUMNI como tarea.
- Los exámenes del ALUMNI son obligatorios.

CONSIDERACIONES INICIALES

- Siempre van a poder adelantar contenidos (queda todo habilitado)
- Se van a encontrar con laboratorios y desafíos. Son ejercicios propuestos que deberán realizar como practica.
- Los exámenes, están preparados con el material de ALUMNI, por lo cual, es NECESARIO antes de rendirlo, haber visto todo el contenido del modulo.
- Cualquier duda administrativa por favor contactar a digitalers@educacionit.com

- Mi mail: jorge.paez@educacionit.com

Introducción a la programación con Python



Un poco de historia

La historia de la programación está fuertemente vinculada a la aparición y crecimiento exponencial de los computadores.

La evolución de los lenguajes

En sus inicios programar implicaba el uso del código binario con cadenas de 0s y 1s, el lenguaje que entiende directamente el computador. ¡Algo sumamente complejo!.
Más tarde se crea el lenguaje ensamblador, que en vez de números utilizaba letras, algo más fácil de recordar.

Un poco más de historia...

Finalmente, surgen los lenguajes de alto nivel, que suelen utilizar términos en inglés para dictar las órdenes a seguir. Este sería un **proceso intermedio** entre el lenguaje de la máquina (binario) y el "código fuente" (proceso por compilador o intérprete).

¿Cuáles son los más conocidos?

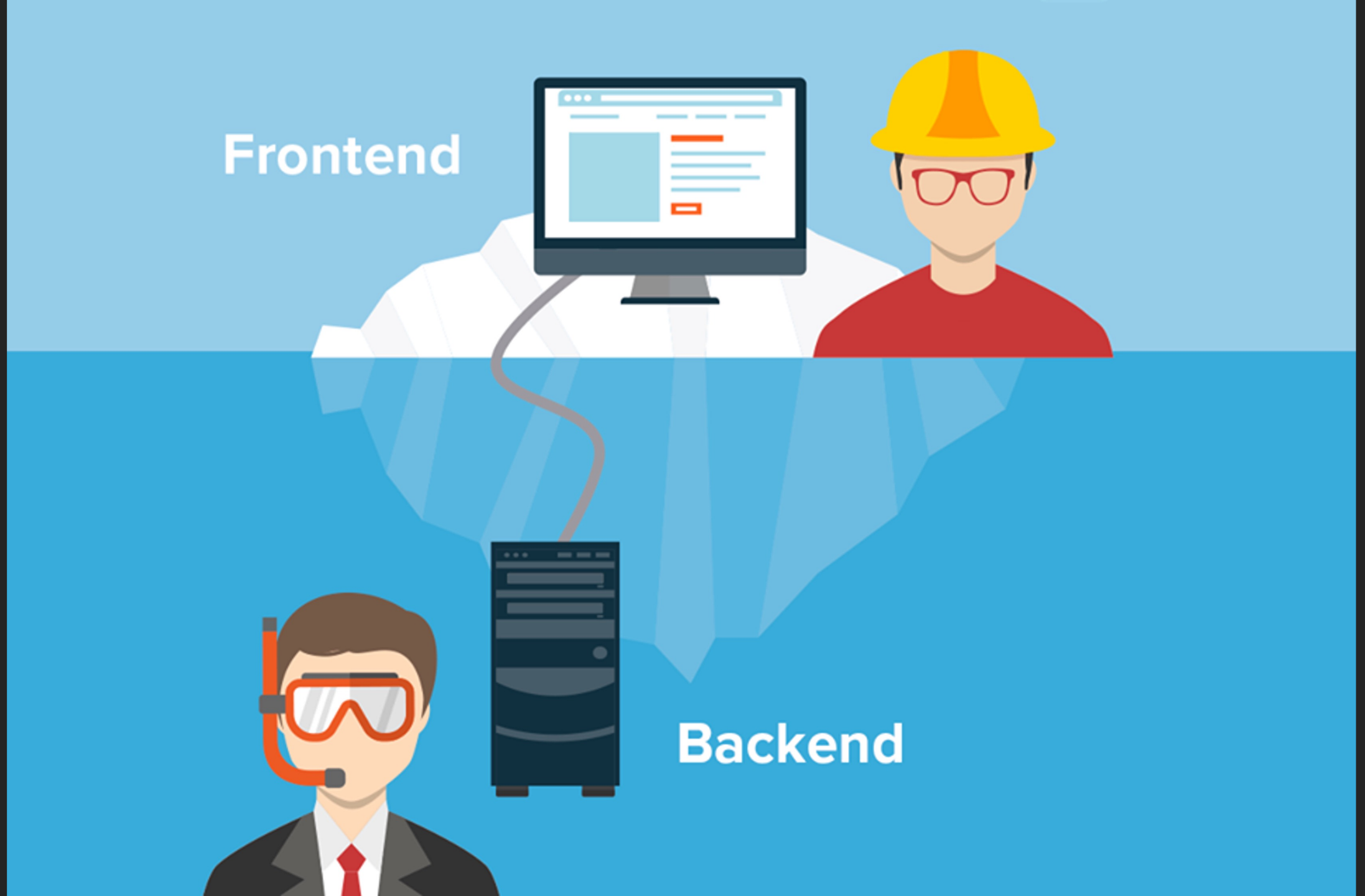



Programación

Es el proceso de escribir instrucciones para la computadora, en el formato y sintaxis que el lenguaje de programación requiere, a fin de que se realice determinada acción.

Programar es comunicarnos con un ordenador

Frontend vs. Backend


Frontend
Backend

Frameworks

Frameworks

Un framework, o marco de trabajo, es un conjunto de herramientas y clases que nos permiten solucionar un problema o funcionalidad. Están escritos en uno o más lenguajes de programación.

Frameworks

Los frameworks establecen, además, una estructura determinada para el código y todos los archivos, como así también una metodología armada del proyecto.

Programadores

¿Cuál es el trabajo del programador?

Los programadores desarrollan aplicaciones y programas informáticos, sirviéndose de las bases de un software existente para crear una interfaz para los usuarios con fines comerciales, profesionales o recreativos.

Python

¿Qué es?

Es un lenguaje de programación de código abierto, orientado a objetos, simple y fácil de entender.
Tiene una sintaxis sencilla que cuenta con una vasta biblioteca de herramientas, que hacen de Python un lenguaje de programación único.
Es un **lenguaje interpretado con tipado dinámico, multiparadigma y multiplataforma**.

¿Para qué se utiliza Python?

Inteligencia Artificial
Big Data
Data Science
Frameworks de Pruebas
Desarrollo Web

Nosotros en este curso nos centraremos en la parte web del desarrollo.

PYTHON

Ventajas	Desventajas
Fácil de aprender	Problemas con hosting
Fácil de usar	Lentitud en multithread
Orientado a objetos	Lentitud de ejecución
Software libre	
Portable	
Multiplataforma	
Poderoso	

Entorno de programación

Google Colab

Para programar con comodidad aplicaciones complejas es necesario instalar un IDE o editor de texto. En este curso les propongo instalar Visual Studio Code;

Pero en por ahora trabajaremos con un entorno de **Google** para enfocarnos solo en la programación.

https://colab.research.google.com

¿Qué es Colab?

Colab, o también conocido como “**Google Colaboratory**", nos permite escribir y ejecutar código de Python en nuestro navegador, con algunas características particulares como:

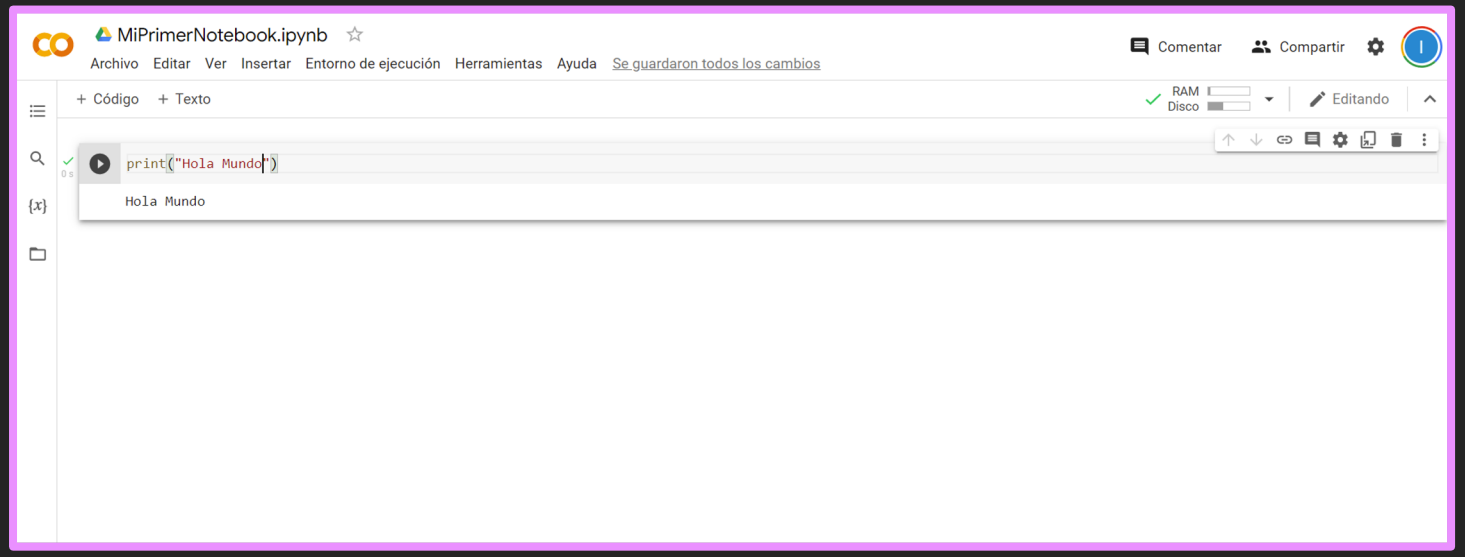
No es requerido realizar ninguna instalación o configuración local del entorno de trabajo.
Acceso gratuito a GPU que nos ofrece Google
Facilidad para compartir y realizar programación “colaborativa”.

ACTIVIDAD EN CLASE

Mi primer Colab

Crearemos un nuevo notebook en Google Colab, titulado: **MiPrimerNotebook** y deberás mostrar por pantalla la frase: "**Hola Mundo**", obteniendo el siguiente resultado:

Si lo has intentado anteriormente, pero no lo has logrado, es momento de consultar tus dudas.



MANOS A LA OBRA!

Números y cadenas de caracteres

Números

Tipos de número

Números en Python

Los números de Python están relacionados con los números matemáticos, pero están sujetos a las limitaciones de la representación numérica en las computadoras. Python distingue entre enteros, números de punto flotante y números complejos.

Enteros

Los números enteros son aquellos que no tienen decimales, tanto positivos como negativos (además del cero). En Python se pueden representar mediante el **tipo int** (de integer, entero).

A diferencia de otros lenguajes de programación, los números de tipo int en Python 3 pueden ser pequeños o grandes, no tienen límite alguno.

Ejemplo: 1, 2, 525, 0, -817

Long

Los números enteros largos o long en Python son iguales a los enteros, no tienen decimales, y **pueden ser positivos, negativos o cero**. Se tratan de números de cualquier tamaño. Se puede definir con una **L** al costado de nuestro número.

Ejemplo:

```
9812893712387912379123L
897538475389475198237891249823L
123873495873 73L
```

Float/Decimal

Los **números reales**, son los que tienen decimales, en Python se expresan mediante el tipo **float**. Desde Python 2.4 cuenta con un nuevo tipo Decimal, para el caso de que se necesite representar fracciones de forma más precisa.

Ejemplo:

0,270
-12,1233
989,87439124387
-74,9349834

Complejos

Los números complejos son los que tienen parte imaginaria, es muy probable que no lo vayas a necesitar nunca. Este tipo se llama **complex**, se almacena usando reales, ya que es una extensión de dichos números.

Ejemplo:

2,1j
-41,832i
88,23 254j

Operaciones numéricas

Operaciones numéricas en Python

En programación y en matemáticas, los operadores aritméticos son aquellos que manipulan los datos de tipo numérico, es decir, permiten la realización de operaciones matemáticas (sumas, restas, multiplicaciones, etc.).

El resultado de una operación aritmética es un dato aritmético, es decir, si ambos valores son números enteros el resultado será de tipo entero; si alguno de ellos o ambos son números con decimales, el resultado también lo será.

OPERADORES ARITMÉTICOS EN PYTHON

Operación	Operador	Ejemplo
Suma	+	3+5 = 8
Resta	-	4 - 1 = 3
Multiplicación	*	3 * 6 = 18
Potencia	**	3 ** 2 = 9
División (cociente)	/	15.0 / 2.0 = 7.5
División (parte entera)	//	15.0 // 2.0 = 7
División (resto)	%	7 % 5 = 1

Precedencia o jerarquías

Precedencia de los operadores

Al igual que ocurre en matemáticas, en programación también tenemos una **prioridad en los operadores**. Esto significa que si una expresión matemática es precedida por un operador y seguido de otro, el operador más alto en la lista debe ser aplicado por primera vez.

Las expresiones con paréntesis se evalúan de dentro a fuera, el paréntesis más interno se evalúa primero.

Precedencia

El orden normal de las operaciones es de izquierda a derecha, evaluando en orden los siguientes operadores:

1. Términos entre paréntesis.
2. Potenciación y raíces.
3. Multiplicación y división.
4. Suma y resta.

Precedencia

En el lenguaje de programación de Python se representan los operadores con el siguiente orden:

1. ( )
2. **
3. X, /, %, //
4. +, -

Cadenas de Caracteres (texto)

Cadenas de caracteres

Las cadenas (o strings) son un **tipo de datos compuestos por secuencias de caracteres que representan texto**. Estas cadenas de texto son de tipo **str** y se delimitan mediante el uso de comillas simples o dobles.

Ejemplo:

“Esto es una cadena de texto”
‘Esto también es una cadena de texto’

Cadenas de caracteres

En el caso de que queramos usar comillas (o un apóstrofe) dentro de una cadena tenemos distintas opciones. La más simple es encerrar nuestra cadena mediante un tipo de comillas (simples o dobles) y usar el otro tipo dentro de la cadena. Otra opción es usar en todo momento el mismo tipo de comillas, pero usando la barra invertida (****) como carácter de escape en las comillas del interior de la cadena para indicar que esos caracteres forman parte de la cadena.

Ejemplos:

“Esto es un ‘texto’ entre comillas dobles”
‘Esto es otro “texto” entre comillas simples’
“Esto es otro \“texto\” todo en comillas dobles”
‘Esto otro \’texto\’ todo en comillas simples’

Print

Print

```
print("Esto es una cadena de texto")
print("Esto también es una cadena de texto")
print(4)
```

¿Para qué sirve?

La forma correcta de mostrar cadenas de texto (u otros objetos) por pantalla en Python es utilizando una función llamada **print** (imprimir). Se indica lo que se desea mostrar por pantalla entre paréntesis.

Print

Otra funcionalidad que tiene es permitir mostrar una cadena en distintas líneas, de forma que con un solo print se muestran varias líneas de cadenas.
Para lograrlo tenemos que pasarlo entre tres comillas dobles, o tres comillas simples.

Ejemplo:

```
print("""una cadena
otra cadena
otra cadena más
""")
```

MANOS A LA OBRA!

Variables

Variables en matemáticas

Dependiendo del contexto, las variables tienen distintos significados. En el caso del Álgebra, una variable representa una cantidad desconocida que se relaciona con otras. Consideremos por ejemplo la ecuación:

Ejemplo:

$x + 3 = 5$

Variables en programación

Variables en programación

En algunos lenguajes de programación, las variables se pueden entender como "cajas" en las que se guardan los datos, pero en Python las variables son "**etiquetas**" que permiten hacer referencia a los datos (que se guardan en unas "cajas" llamadas objetos).

Python es un lenguaje de programación orientado a objetos y su modelo de datos también está basado en objetos.

Variables en programación

Para cada dato que aparece en un programa, Python crea un objeto que lo contiene. Cada objeto tiene:

1. Un **identificador único** (un número entero, distinto para cada objeto). El identificador permite a Python referirse al objeto sin ambigüedades.
2. Un **tipo de datos** (entero, decimal, cadena de caracteres, etc.). El tipo de datos permite saber a Python qué operaciones pueden hacerse con el dato.
3. Un **valor** (el propio dato).

IMPORTANTE

Las variables en Python no guardan los datos, sino que son simples nombres para **poder hacer referencia al lugar en memoria en donde se encuentra guardado el dato.**

Variables en programación

En Python, si escribimos la instrucción: a = 2.
Se crea el objeto "2". Ese objeto tendrá un identificador único que se asigna en el momento de la creación y se conserva a lo largo del programa. En este caso, el objeto creado será de tipo número entero y guardará el valor 2.
Se asocia el nombre al objeto número entero 2 creado.

Variables en programación

Al describir la instrucción anterior no habría que decir 'la variable **a** almacena el número entero **2**', sino que habría que decir 'podemos llamar **a** al objeto número entero **2**'. La variable **a** es como una etiqueta que nos permite hacer referencia al objeto "2", más cómoda de recordar y utilizar que el identificador del objeto.

Definir variables

MANOS A LA OBRA!

Definir una variable

Siempre se escribe a la izquierda de la igualdad, de lo contrario, Python generará un mensaje de error:

```
>>> 2 = mi_variable
SyntaxError: can't assign to literal
```

Para mostrar el valor de la variable hay que escribir su nombre, o "printearlo".

```
>>> mi_variable = 2
>>> mi_variable
2
>>>print(mi_variable)
2
```

Definir una variable

Si una variable no se ha definido previamente, al escribir su nombre o printear la variable generará un error:

```
>>> x = -10
>>> y
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

Definir una variable

Una variable puede almacenar números, texto o estructuras más complicadas (que se verán más adelante). Si se va a almacenar texto, debe escribirse entre comillas simples (') o dobles ("), que son equivalentes. A las variables que almacenan texto se les suele llamar cadenas (de texto).

```
>>> nombre = "Pepito Conejo"
>>> nombre
'Pepito Conejo'
>>> print(nombre)
'Pepito Conejo'
```

Definir una variable

Si no se escriben comillas, Python supone que estamos haciendo referencia a otra variable (que, si no está definida, genera un mensaje de error):

```
>>> nombre = Pepe
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    nombre = Pepe
NameError: name 'Pepe' is not defined

>>> nombre = Pepito Conejo
SyntaxError: invalid syntax
```

Buenas prácticas

Aunque no es obligatorio, se recomienda que el nombre de la variable esté relacionado con la **información que se almacena en ella para que sea más fácil entender el programa.**

Buenas prácticas

Si el programa es trivial o mientras se está escribiendo un programa, esto no parece muy importante, pero si se consulta un programa escrito por otra persona o escrito por uno mismo hace tiempo, **resultará mucho más fácil entender el programa si los nombres están bien elegidos**.

Veamos un ejemplo en vivo

El nombre de una variable debe empezar por una letra o por un guión bajo (_) y puede seguir con más letras, números o guiones bajos (esto en inglés se llama **snake case**).

```
>>> fecha_de_nacimiento = "27 de octubre de 1997"
>>> fecha_de_nacimiento
'27 de octubre de 1997'
```

Los nombres de variables no pueden incluir espacios en blanco.

```
>>> fecha de nacimiento = "27 de octubre de 1997"
SyntaxError: invalid syntax
```

Veamos un ejemplo en vivo

Los nombres de las variables pueden contener mayúsculas, pero ten en cuenta que **Python distingue entre mayúsculas y minúsculas** (en inglés se dice que Python es case-sensitive).

```
>>> nombre = "Pepito Conejo"
>>> Nombre = "Numa Nigerio"
>>> nomBre = "Fulanito Mengánez"
>>> nombre
'Pepito Conejo'
>>> Nombre
'Numa Nigerio'
>>> nomBre
'Fulanito Mengánez'
```

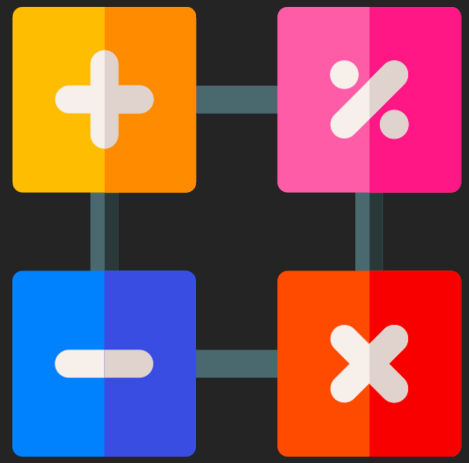
Operadores

¿Qué son?

Formalmente, los operadores son aplicaciones, cálculos que se llevan a cabo sobre dos argumentos conocidos como operandos.

Operando [operador] Operando

- / * +



EXPRESIONES

Se denomina expresión al conjunto que forman los operandos y la operación.
Sumar, restar, dividir o multiplicar, tienen algo en común, y es que sus operadores son operadores aritméticos que sirven para trabajar con números.

Los operadores aritméticos (+, -, /, *) dan lugar a expresiones de distintos tipos:

Aritméticas			**Algebraicas**	
si ambos operandos son valores literales:			si al menos un operando es una variable:	
2 + 5	-1.4 * 54	1/2.5	radio * 3.14	(nota_1 + nota_2)/2

Operaciones aritméticas con variables

Operaciones aritméticas con variables

Podemos utilizar todos los operadores aritméticos antes vistos en las variables numéricas. Algunos ejemplos:

```
>>> a = 2
>>> b = 3
>>>a+b
```

```
>>> a = 5
>>> b = 2
>>> a * b
```

```
>>>a =35
>>>b = 7
>>>a / b
```

```
>>>a = 3
>>>b = 2
>>> a ** b
```

Operaciones aritméticas con variables

Podemos utilizar todos los operadores aritméticos antes vistos en las variables de **string** también. A la suma de cadenas de caracteres la llamaremos **concatenación.** Algunos ejemplos:

```
>>> cadena = “Python”
>>>cadena * 2
“PythonPython”
```

```
>>>cadena = “Python”
>>>otra_cadena = “Hola!”
>>>otra_cadena + cadena
“Hola!Python”
```

Operadores de asignación

Operadores de asignación

Vamos a ver unos tipos de operadores aritméticos que actúan directamente sobre la variable actual modificando su valor. Es decir, no necesitan dos operandos, solamente necesitan una variable numérica. Por eso, se les llama operadores de asignación.

OPERADORES DE ASIGNACIÓN

El operador de asignación más utilizado y el cual hemos utilizado hasta ahora es el signo = .

Este operador asigna un valor a una variable:
número = 15

Además de este operador, existen otros operadores de asignación compuestos, que realizan una operación aritmética básica sobre la variable.

Suma en asignación

Teniendo ya declarada una variable, podemos directamente sumarle un valor,

Por ejemplo 1:

```
>>> a = 0
>>> a += 1
1
```

Ahora, cada vez que yo haga a+=1 se incrementará el valor de a en 1

Para poder aplicar cualquier operador en asignación se debe tener una variable previamente declarada, de lo contrario nos devolverá un error

Resta en asignación

También podemos directamente restarle un valor

Por ejemplo 5:

```
>>> a = 50
>>> a -= 5
45
```

Ahora, cada vez que yo haga a-=5 a se disminuirá el valor de a en 5

Producto en asignación

También podemos directamente hacer un producto a un valor.

Por ejemplo 10:

```
>>> a = 5
>>> a *= 10
50
```

Ahora, cada vez que hagamos a*=10 se multiplicará el valor de a en 10

División en asignación

También podemos directamente hacer una división a un valor.

Por ejemplo 2:

```
>>> a = 10
>>> a /= 2
5
```

Ahora, cada vez que hagamos a/=2 se dividirá el valor de a en 2

Módulo en asignación

También podemos directamente hacer un módulo a un valor.

Por ejemplo 2:

```
>>> a = 10
>>> a %= 2
0
```

Ahora, cada vez que hagamos a%=2 se hará el módulo de a en 2

Potencia en asignación

También podemos directamente hacer una potencia a un valor.

Por ejemplo 2:

```
>>> a = 5
>>> a **= 2
25
```

Ahora, cada vez que hagamos a**=2 se hará una potencia de a en 2

OPERADORES DE ASIGNACIÓN

Operador	Ejemplo	Equivalente
=	a = 2	a = 2
+=	a += 2	a = a + 2
-=	a -= 2	a = a –2
*=	a *= 2	a = a * 2
/=	a /= 2	a = a / 2
%=	a %= 2	a = a % 2
**=	a **= 2	a = a ** 2

Operadores lógicos

El tipo lógico

Tipos de datos

Los números, imágenes, textos, y sonidos, si algo tienen en común es que podemos percibirlos como información, pero hay un tipo de dato distinto, más básico. Es tan básico, que quizás cueste entenderlo como un tipo de dato. Y ese, es el tipo lógico.

Tipo Lógico

El tipo lógico es el tipo de dato más básico de la información racional, y representa únicamente dos posibilidades:

Verdadero
Falso

También denominamos a este tipo como Booleano o Binario.

Tipo lógico

Lingüístico	Matemático
En contexto lingüístico podríamos decir que:	Y en contexto matemático:
“Estoy vivo” es Verdadero (True)	1 +1 = 3??? es Falso (False)

Negación

Si negamos una cosa que es verdad, esta se convierte en mentira. Por lo tanto, si negamos una cosa que es mentira, esta se convierte en verdad.

No Verdadero = Falso
No Falso = Verdadero

¿Y en la programación?

Por ejemplo, a un ordenador podemos preguntarle cosas matemáticas

```
>>> 1 + 1 == 3 False
```

Aquí estamos preguntando si al sumar 1 con 1 el resultado es 3 y Python ya sabe decirnos que esto es falso (false)

Y si le preguntamos si 1 + 1 es igual a 2?

```
>>> 1 + 1 == 2 True
```

Operadores relacionales

Operadores Relacionales

En programación, los operadores relacionales son símbolos que se usan para comparar dos valores.

Si el resultado de la comparación es correcto, la expresión es considerada verdadera (**True**), y en caso contrario será falsa (**False**).

Igualdad

El operador de igualdad sirve para preguntarle a nuestro programa si ambos operandos son iguales.
Devolverá **True** si son iguales, y **False** si son distintos. Este operador se escribe con dos signos igual (==).

Ejemplo:

```
>>> a = 3
>>> a == 3
True
```

No confundir el operador de asignación (=) con el operador de igualdad (==)

Desigualdad

El operador de Desigualdad sirve para preguntarle a nuestro programa si ambos operandos son distintos.
Devolverá **True** si son distintos, y **False** si son iguales.

Ejemplo:

```
>>> a = 3
>>> a != 3
False
```

Este operador se escribe como un signo de exclamación y un signo igual (**!=**) como tachando al operador de igualdad.

Menor que

El operador Menor que sirve para preguntarle a nuestro programa si el primer operando es menor que el segundo operando.

Ejemplo:

```
>>> 7 < 3
False
>>> 1 < 15
True
```

Devolverá **True** si el primero es menor al segundo, y **False** si el primero es mayor que el segundo. Este operador se escribe con un signo de menor que (**<**).

Menor Igual que

El operador **Menor igual que** sirve para preguntarle a nuestro programa si el primer operando es menor que el segundo operando o si ambos son iguales.

Ejemplo:
>>> 7 <= 3
False
>>> 15 <= 15
True

Devolverá **True** si el primero es menor o igual al segundo, y **False** si el primero es mayor que el segundo.
Este operador se escribe con un signo de menor que y un igual (<=).

Mayor que

El operador **Mayor que** sirve para preguntarle a nuestro programa si el primer operando es mayor que el segundo operando.

Ejemplo:
>>> 7 > 3
True
>>> 1 > 15
False

Devolverá **True** si el primero es mayor al segundo, y **False** si el primero es menor que el segundo. Este operador se escribe con un signo de mayor que (**>**).

Mayor igual que

El operador **Mayor igual que** sirve para preguntarle a nuestro programa si el primer operando es mayor que el segundo operando, o si ambos son iguales.

Ejemplo:

```
>>> 7 >= 3
True
>>> 15 >= 15
True
```

Devolverá **True** si el primero es mayor o igual al segundo, y **False** si el primero es menor que el segundo.
Este operador se escribe con un signo de mayor que y un igual (**>=**).

¿Operadores en Strings?

No sólo podemos hacer operaciones relacionales en números, también podemos hacerlas en strings.

Ejemplo:

```
>>> 'Hola' == 'Hola’
True
>>> a = 'Hola’
>>> a[0] != 'H'
False
```

Tipo Lógico

Los Booleanos tienen un valor aritmético por defecto. **True** tiene un valor de 1 y mientras tanto **False** tiene un valor de 0. Es decir, tienen un valor binario que se utiliza para poder operar entre sí.

Ejemplo:

```
>>> True > False
True
>>> True * 3
3
>>> False / 5
0.0
```

MANOS A LA OBRA!

Operadores lógicos

Operadores Lógicos

Existen varios tipos de operadores lógicos en Python. Pero nos estaremos enfocando en los tres más básicos y utilizados:

Not (no - negación)

Or (o de esto o aquello)

And (Y de esto y eso).

Not

El not es la negación o también conocida como el NO. Es un poco especial, ya que solo afecta a los tipos lógicos **True** y **False**; solo requiere un operando en una expresión.

Ejemplo:

```
>>> not True
False
>>> not True == False
False
```

- Negación Lógica (NO)
- Sólo afecta a los lógicos

Más operadores

Los operadores lógicos nos permiten crear grandes expresiones, estos operadores se presentan en dos formas:

Conjunción	Disyunción
Viene de conjunto	Viene de disyunto
Sinónimo de unido, contiguo	Sinónimo de separado
Agrupa uniendo	Agrupa separando

And

El operador de conjunción, es decir, el que agrupa a través de la unión, es el operador lógico **AND**, en castellano conocido como **Y**.

Pero, ¿qué es lo que une? Este operador une una o varias sentencias lógicas:

Estoy vivo **y** estoy dando un curso.

Ambas sentencias están unidas por un **Y** y ambas son afirmaciones verdaderas. Y, ¿visto en conjunto?

VERDADERO y VERDADERO

And

Si tenemos dos afirmaciones que son verdaderas, evidentemente estaremos diciendo la verdad. Python también puede comprender esto, es decir, si preguntamos sobre dos afirmaciones unidas por un **Y**, sabrá decir si es verdadero o falso.

Ejemplo:

```
>>> 2 > 1 and 5 > 2
True
>>> 5 > 20 and 20 < 1
False
```

Para Python, una unión **and** es solamente verdadera **(True)** cuando, y solo cuando, toda la sentencia o conjunto de afirmaciones es verdadera. Es decir, cuando las dos afirmaciones son verdaderas. Si yo tengo una afirmación verdadera y otra falsa, Python siempre va a tomar como que esto es falso **(False).**

And

Expresión	Resultado
True and True	True
True and False	False
False and True	False
False and False	False

Tabla de verdad del And

La cantidad de combinaciones entre dos proposiciones lógicas unidas por un and son cuatro y ya las hemos analizado en la tabla anterior. Eso se llama tabla de verdad

p	q	p ∧ q
V	V	
V	F	
F	V	
F	F	

p	q	p ∨ q
V	V	
V	F	
F	V	
F	F	

Or

Ahora, veamos el operador de disyunción denominado **Or** en castellano **O**.
Si el AND unía, el OR separa. Es decir, si a Python le pregunto por dos afirmaciones, y al menos una es (verdadera)True, Python me dirá que esta afirmación es **True.**

Ejemplo:

```
>>> 2 > 1 or 5 > 2
True
>>> 5 < 20 or 20 < 1
True
```

Para Python, una separación **OR** es solamente verdadera **(True)** cuando, y solo cuando, una de las sentencias o conjuntos de afirmaciones es **True**, es decir, cuando yo tengo una afirmación verdadera. Si yo tengo una afirmación verdadera y otra falsa, Python siempre va a tomar como que esto es **True**.

Or

Expresión	Resultado
True o True	True
True o False	True
False o True	True
False o False	False

Tabla de verdad del Or

Su tabla de verdad quedaría así:

p	q	p ∨ q
V	V	
V	F	
F	V	
F	F	

Expresiones anidadas

Expresiones anidadas

Hemos visto que existen un montón de expresiones distintas y como pueden suponer, es posible crear combinaciones entre estas.

A esto, se lo denomina **expresiones anidadas**.

El problema es que se pueden definir grandes expresiones con multitud de operadores y operandos, y si no sabemos como Python las interpreta a la hora de resolverlas, podríamos causar algunos errores sin querer.

Normas de precedencia

Ya que equivocarse es el pan de cada día, usaremos esta sección para poder aprender las normas de precedencia y aprenderemos como Python resuelve las expresiones complejas con los distintos tipos de operadores.

Si recuerdan, en la clase 1 vimos las procedencias de operadores numéricos:

Términos entre paréntesis.
Potenciación y raíces.
Multiplicación y división.
Suma y resta.

Normas de precedencia

Nos sirven para cuando tengamos que trabajar con expresiones anidadas que sean demasiado grandes.

Ejemplo:

```
>>> a = 15
>>> b = 12
>>> a ** b / 3**a / a * b >= 15 and not (a%b**2)
!= 0
False
```

Nota: En la práctica nunca, o casi nunca, trabajaríamos con una expresión de este estilo, es por mero ejemplo.

¿Por qué nos dio False?

- Expresiones de cualquier tipo entre paréntesis.
- Expresiones aritméticas por sus propias reglas.
- Expresiones relacionales de izquierda a derecha.
- Operadores lógicos (not tiene prioridad, ya que afecta al operando).

DESCANSO

Condicional

Condicional

En la vida diaria, actuamos de acuerdo a la evaluación de condiciones, de manera mucho más frecuente de lo que en realidad creemos: si el semáforo está en verde, cruzar la calle. Si no, esperar a que el semáforo se ponga en verde.

A veces, también evaluamos más de una condición para ejecutar una determinada acción: si llega la factura de la luz y tengo dinero, pagar la factura.

Condicional

Las sentencias de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. La evaluación de condiciones, solo puede arrojar 1 de 2 resultados: **True** o **False** (verdadero o falso).

Condicional

Para describir la evaluación a realizar sobre una condición, se utilizan los operadores relacionales (==, !=, >, <, etc.). Y, para evaluar más de una condición simultáneamente se utilizan los operadores lógicos (not, and, or).

Las sentencias de control de flujo condicionales se definen mediante el uso de tres palabras claves reservadas:

if (si)
elif (si no, si)
else (si no)

Sentencia If

Dentro de las sentencias condicionales el if (si) posiblemente sea la más famosa y utilizada en la programación, esto debido a que nos permite controlar el flujo del programa y dividir la ejecución en diferentes caminos.

Al utilizar esta palabra reservada **if** le estamos indicando a Python que queremos ejecutar una porción de código, o bloque de código, solo si se se cumple una determinada condición, es decir, si el resultado es **True**.

Sentencia If

Primero definimos una variable **edad** y le asignamos un valor entero **30**. Después, a través del condicional, le decimos que queremos imprimir "**Es un adulto**" en pantalla, solo si se cumple la condición de que **edad** sea mayor o igual a 18.

Veamos el siguiente ejemplo:

```
edad =  30
if edad >= 18:
    print('Es un adulto')
if True:
    print('Se cumple la condición')
```

Indentación

Python se basa en la sangría (espacio en blanco al comienzo de una línea) para definir el alcance en el código. Otros lenguajes de programación a menudo usan corchetes para este propósito.

El siguiente código nos arrojará un error:

```
a = 25
b = 50
if b > a:
print("b es más grande que a")
```

Puedes probarlo para verificarlo.

Recordemos que Python admite las condiciones lógicas habituales de las matemáticas:

Es igual a : a == b

No es igual a: a != b

Menos que: a < b

Menor o igual que: a <= b

Mayor que: a > b

Mayor o igual que: a >= b

También podemos apoyarnos del uso de operadores lógicos como ser: AND, OR, NOT.

Ejemplo con AND:

```
a = 195
b = 30
c = 400
if a > b and c > a:
  print(“Ambas condiciones son verdaderas")
```

Ejemplo con OR:

```
a = 195
b = 50
c = 500
if a > b or a > c:
  print(“Al menos una de las condiciones es verdadera")
```

Ejemplo con NOT:

```
x = 10
if not x > 15:
    print("False")
```

If en una sola linea – Ejemplo 1:

```
a = 150
b = 35
if a > b: print(“a es mayor que b")
```

If en una sola linea – Ejemplo 2:

```
a = 5
b = 150
print("A") if a > b else print("B")
```

If en una sola linea – Ejemplo 3:

```
a = 150
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

Else

Sentencia Else

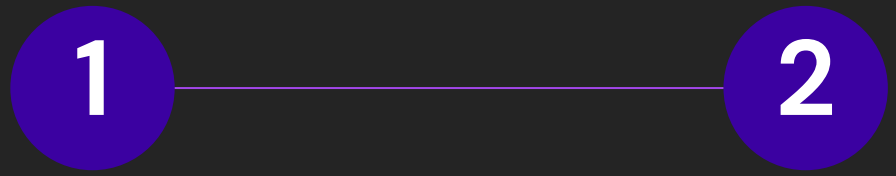
Dentro de las sentencias condicionales el **else** (sino) es una especie de “hermano” de **if** el cual se puede encadenar al final de un bloque de código **if** para comprobar los casos contrarios, es decir los **False.**

Al utilizar esta palabra reservada **else** le estamos indicando a Python que queremos ejecutar una porción de código, o bloque de código, sólo si no se cumple ninguna de las condiciones antes dichas, es decir, si el resultado es **False** siempre.

SENTENCIA ELSE

Veamos el siguiente ejemplo:

```
numero =  24
if numero > 36:
    print("El número es grande")
else:
    print("El número es chico")
```



Primero definimos una variable número y le asignamos un valor entero 24.

Después, a través del condicional, le preguntamos si el número es mayor a 36, si es así, queremos imprimir “El número es más grande” en pantalla, de lo contrario queremos que imprima “El número es más chico”.

Múltiples If

Veamos un ejemplo de cómo podemos trabajar con múltiples ifs anidados:

Ejemplo 1:

```
x = 25
if x > 10:
  print("por encima de diez,")
  if x > 20:
    print("y también por encima de 20!")
  else:
    print("pero no por encima de 20")
```

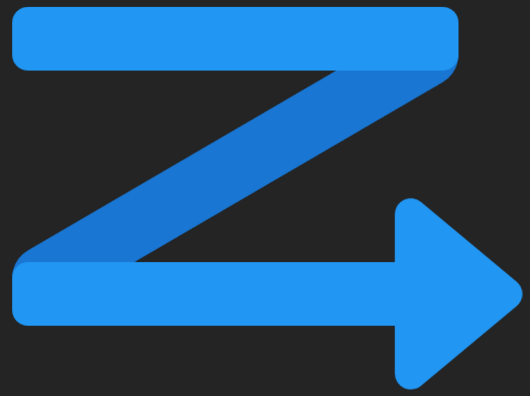
Ejemplo 2:

```
x = 15
if x > 10:
  print("por encima de diez,")
  if x > 20:
    print("y también por encima de 20!")
  else:
    print("pero no por encima de 20")
```

Elif

Sentencia Elif

La última sentencia condicional que podemos encontrar es el **elif** (si no, si), también podríamos decir que es un hermano de **if,** ya que se utiliza en continuación al **if** para poder encadenar muchísimas más comprobaciones.



Al utilizar esta palabra reservada elif le estamos indicando a Python que queremos ejecutar una porción de código, o bloque de código, solo si la condición anterior no se cumple, es decir, si el resultado del **if** o algún **elif** fue **False**.

SENTENCIA ELIF

```
a = 2 + 3
if a == 4:
   print ("A es igual a cuatro")
elif a == 5:
   print ("A es igual a cinco")
elif a == 6:
   print ("A es igual a seis")
else:
   print ("No se cumple la condición")
```

Como podemos observar, la primera condición valida si A es igual a 4, como esto no es verdadero, se evalúa la siguiente condición, si A es igual a 5, si no A es igual a 6? .

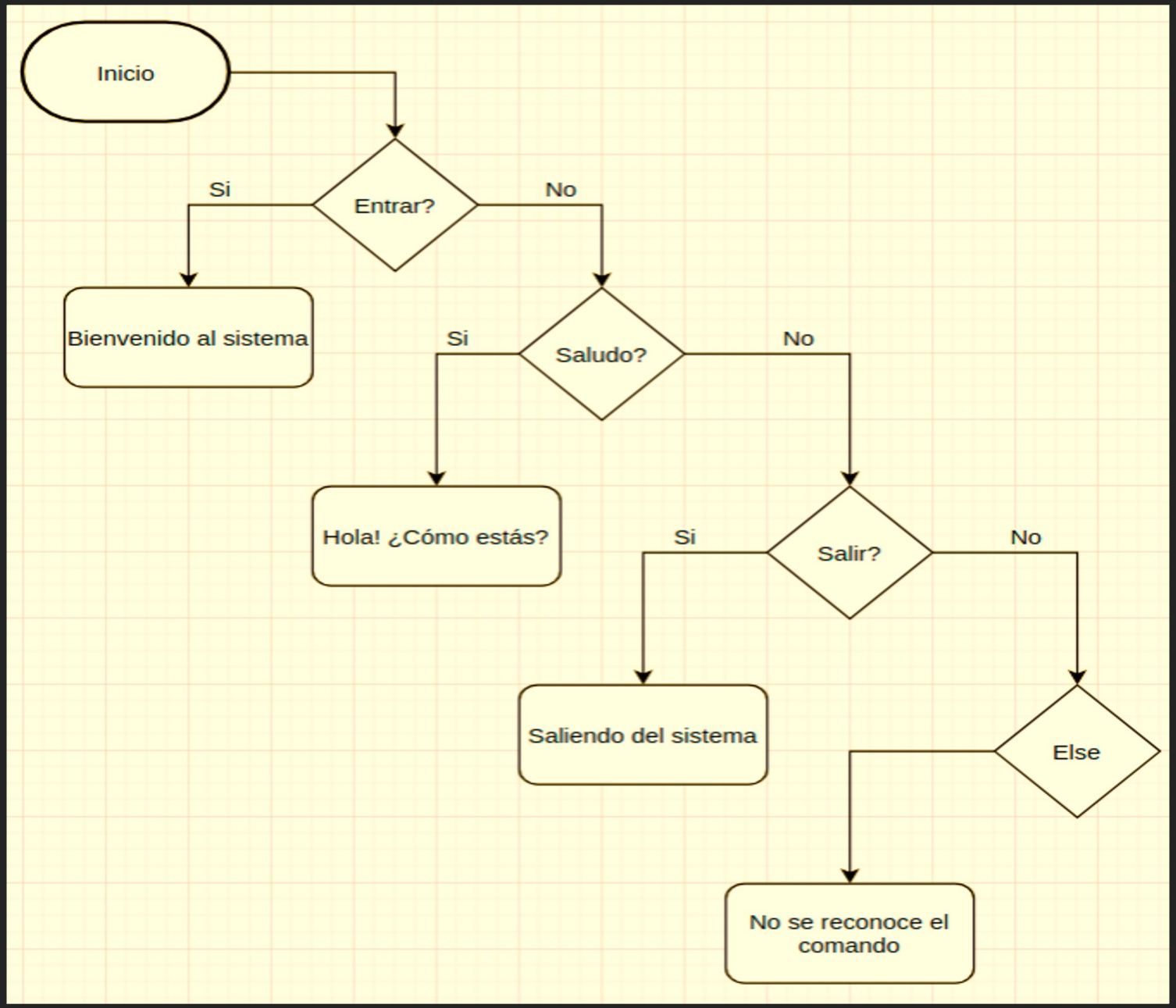
Finalmente, se define un bloque else por default que se ejecutará cuando ninguna de las condiciones anteriores sea verdadera.

Pregunta! ¿Cuál sería el resultado de este ejemplo?

¿Para qué sirve la sentencia Elif?

```
comando =  "SALIR"
if comando == "ENTRAR":
    print("Bienvenido al sistema.")
elif comando == "SALUDO":
    print("Hola! ¿Cómo estás?")
elif comando == "SALIR":
    print("Saliendo del sistema.")
else:
    print("No se reconoce el comando.")
```

Básicamente nos sirve para poder darle múltiples opciones al programa.


Inicio
Entrar?
Si
No
Bienvenido al sistema
Saludo?
Si
No
Hola! ¿Cómo estás?
Salir?
Si
No
Saliendo del sistema
Else
No se reconoce el comando

Cuando se tiene varios **if's** se ven las múltiples condiciones y si todo está bien, nos mostrará el resultado de cada **if**.
Sin embargo, en el caso de múltiples **elif**, comprueba las condiciones de arriba a abajo hasta que se cumpla una de ellas, y de ser así, las demás no se comprueban.

MANOS A LA OBRA!

PREGUNTAS?

MUCHAS GRACIAS