Python

Python es un lenguaje de programación:

- de alto nivel: más cercano al lenguaje humano que al de la máquina (independiente del hardware subyacente)
- multiplataforma: está implementado en GNU/Linux, MS Windows, Mac OS, etc., usando el mismo código en todas estas plataformas
- multiparadigma: implementa la programación orientada a objetos pero también permite usar los paradigmas de la programación imperativa, estructurada y funcional
- dinámicamente tipado: no es necesaria la declaración explícita del tipo de las variables, sino que se establece al asignarles un valor durante la ejecución del código, pudiendo cambiar durante la misma
- fuertemente tipado: tratar de sumar un número a una cadena o concatenar cadenas y números, genera un error (otros lenguajes permiten resolver estos casos generalmente convirtiendo primero los valores numéricos en cadenas de texto y luego concatenando las cadenas resultantes), aunque es posible pasar de un tipo a otro de manera explícita
- interpretado: el código fuente se ejecuta línea a línea (archivos de texto con extensión .py)
- está concebido para maximizar la legibilidad de su código

El código python en los siguientes párrafos se representa con `tipografía mono espaciada`, y las salidas además en `color azul`.

Editores de textos vs. IDEs

Para escribir nuestros programas necesitamos al menos un editor de textos (algunos disponen de ciertas ayudas como indentación automática o resaltado del código mediante distintos colores), o un entorno de desarrollo integrado o IDE, que en general nos permiten además ejecutar el código sin salir del mismo (con un editor debemos grabar primero y luego desde una terminal del sistema operativo ejecutar el compilador o intérprete para ejecutar el nuevo programa). Muchos editores desdibujan esta separación al incorporar complementos que emulan gran parte de la funcionalidad de un IDE.

Algunos IDEs recomendables son:

- Visual Studio Code
- Sublime Text
- Geany
- Thonny
- IDLE (Integrated Development and Learning Environment, incluido al instalar Python)

Algunos editores recomendables son:

- Xed
- Notepad++
- Nano
- Vim
- Emacs

También es posible utilizar un IDE basado en la web:

- https://replit.com/languages/python3
- https://www.programiz.com/python-programming/online-compiler/
- https://www.w3schools.com/python/python_compiler.asp
- https://trinket.io/
- https://editor.raspberrypi.org/en/

Comentarios

Permiten hacer más legible el código:

Con numeral (#)	Con 3 comillas simples	Con 3 comillas dobles

`#comentario de una linea`	`‘’’`
	`comentario de`
	`múltiples lineas`
	`‘’’`
		`“””`
		`comentario de`
		`múltiples lineas`
		`“””`

A veces necesitamos comentar un bloque de código para evitar su ejecución cuando estamos probando nuevas funcionalidades. Para comentar varias líneas seleccionamos las líneas y se presiona una combinación de teclas simultáneamente. Esto agrega el carácter # más algún otro (un espacio generalmente), para diferenciar este código “ocultado” de los comentarios normales del programa. Esta combinación de teclas suele ser distinta y particular a cada editor o IDE:

- VSCode: CTRL+SHIFT+7 (#), o CTRL+SHIFT+A (‘’ al inicio y al final de la selección)
- Geany: CTRL+E (# ~)
- Thonny: CTRL+3 (#)
- Xed: CTRL+/ o CTRL+SHIFT+7 (#)

Variables

Son espacios de memoria asociados a un nombre que pueden contener un dato. Los nombres de variables deben comenzar con una letra o el guion bajo (_), y no pueden ser palabras reservadas o funciones nativas de Python, ni contener operadores o espacios en blanco.
Se recomienda el uso de minúsculas y palabras separadas por guiones bajos (snake case), o minúsculas con la primer letra de las demás palabras en mayúsculas (camel case), solo si esa es la convención usada en un proyecto en particular.

Palabras reservadas de python

False, None, True, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield.

Funciones nativas de python

abs(), all(), any(), basestring(), bin(), bool(), bytearray(), callable(), chr(), classmethod(), cmp(), compile(), complex(), delattr(), dict(), dir(), divmod(), input(), enumerate(), int(), eval(), isinstance(), execfile(), issubclass(), file(), iter(), filter(), len(), float(), list(), format(), locals(), frozenset(), long(), getattr(), map(), globals(), max(), hasattr(), memoryview(), hash(), min(), help(), next(), hex(), object(), id(), oct(), open(), staticmethod(), ord(), str(), pow(), sum(), print(), super(), property(), tuple(), range(), type(), raw_input(), unichr(), reduce(), unicode(), reload(), vars(), repr(), xrange(), reversed(), zip(), round(), __import__(), set(), apply(), setattr(), buffer(), slice(), coerce(), sorted(), intern().

Tipos de datos

Simples		
`int`	enteros (*)	`a=3`
`float`	reales	`x=1.41, y=3E6, z=5.36e-3`
`str`	cadena de caracteres	`cad="H", txt='qwerty', msg="letra 'q'", com="\""`
`bool`	booleanos (**)	`a=True, b=False`
`NoneType`	none	`nn=None (ausencia de valor)`

(*) para facilitar su lectura es posible usar el guion bajo como separador de miles (1_111_111)

(**) en los valores booleanos o lógicos la primer letra siempre está en mayúsculas (`True` o `False`).

Representación en otras bases numéricas			
Base	**Nombre**	**Prefijo**	**Ejemplo**
2	Binarios	0b (dígito cero-letra b)	0b111, 0b1010, 0b1111000
8	Octales	0o (dígito cero-letra o)	0o123, 0o77, 0o354
16	Hexadecimales	0x (dígito cero-letra x)	0x123, 0x1a, 0x2FE

```
#la función print nos permite mostrar datos por pantalla
print(0b111, 0o123, 0x123)  #print convierte a base 10 los int de base 2, 8 o 16
7 83 291
```

Estructurados o colecciones		
`tuple`	tuplas	`tp=1, 2, 'xy' #o también: tp=(1, 2, 3)`
`list`	listas	`lst=[3.5, 'a', 'xyz', 7]`
`dict`	diccionarios	`dc={'apl': 'García', 'nom': 'Angel'}`
`set`	conjuntos	`a={'Ana', 'Olga', 'Pedro'}`

Tipos de Datos y Casting de Datos

```
#para consultar el tipo de dato de una variable se usa la función type()
n = 4
print(type(n))
<class 'int'>
```

La función `input()` nos permite leer lo que el usuario ingresa por teclado y su salida es siempre una cadena de texto. Normalmente esa cadena se almacena en una variable para su posterior empleo en el código. En este ejemplo se fuerza esa salida de tipo `str` al tipo `float`, esto es lo que se denomina casting de datos:

```
var_real=float(input("Ingrese el precio: "))
```

Lo mismo se aplica a otros tipos compatibles, pero no es posible ejecutar por ejemplo `int('hola')`, porque generaría un error (la cadena `'hola'` no tiene una equivalencia numérica).

```
Pi=3.14
pi_ent=int(pi)
print(pi_ent)
3
```

```
tpl=1, 3, 5, 7
lst=list(tpl)
print(tpl, lst)
(1, 3, 5, 7) [1, 3, 5, 7]
```

Operadores

Asignación		
=	a = b	Asignación simple: a toma el valor de b (*)
+=	a += b	a = a + b (**)
-=	a -= b	a = a – b (***)
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
//=	a //= b	a = a // b
**=	a **= b	a = a ** b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

(*) a = b es una asignación simple, pero las asignaciones también pueden ser múltiples, mediante el uso de tuplas y listas:
nombre, edad = “Pepe”, 35 #equivalente a: nombre, edad = (“Pepe”, 35)
#es equivalente a:
nombre=“Pepe”
edad=35		

(**) Incrementar. Contador: a +=1		
(***) Decrementar. Cuenta regresiva: contador -=1		

Aritméticos	
+	sumar, concatenar (str, list, dict)
-	restar, opuesto
*	producto (int, float), repetir (str) (*)
/	cociente (entero con operandos int)
//	cociente entero (resultado truncado)
%	resto
**	Potencia
(*) cad * n repite la cadena cad n veces (un argumento str y otro int): print(‘x’ * 10, 5 * ‘abc’) xxxxxxxxxx abcabcabcabcabc Si el valor int es cero o negativo la expresión devuelve una cadena vacía.	

Relacionales		
<	menor	Comparan valores o expresiones numéricas, cadenas de caracteres, etc., y devuelven un valor lógico (True o False). Podemos encadenarlas, las siguientes son equivalentes: a<b<c a<b and b<c
<=	menor o igual	
>	mayor	
>=	mayor o igual	
==	igual	
!=	distinto	

Lógicos (operandos y resultado bool)		
and	conjunción	Operan con valores o expresiones lógicas o booleanas: valores True o False. No tienen un operador de asignación asociado como los operadores bit a bit.
or	disyunción	
not	negación	

Tablas de verdad

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

a	B	a or b
False	False	False
False	True	True
True	False	True
True	True	True

a	not a
False	True
True	False

Valores truthy y falsy

Los términos "truthy" y “falsy” en Python se refieren a cualquier valor en una expresión lógica que es evaluado como `True` o `False` aunque no sean iguales a dichos valores booleanos.

Truthy	Falsy
todos los valores numéricos diferentes de cero, por ejemplo: 1, 2.5, -3	el valor numérico cero (0)
cadenas de caracteres no vacías, por ejemplo: "hola", ´123’	cadenas de caracteres vacías ("" o '')
listas, tuplas, conjuntos y diccionarios no vacíos	listas, tuplas, conjuntos y diccionarios vacíos
el valor True	el valor False

Ejemplos:

`if n!=0:` es equivalente a `if n:` `#evalúa cualquier valor no nulo como True`

`if n%2==1:` es equivalente a `if n%2:` `#evalúa 1 como True y 0 como False`

Operadores lógicos bit a bit (operandos y resultado int)			
&	and (bit)	a=b&c	
|	or (bit)	a=b|c	
~	not (bit)	a=~b	
^	xor (bit)	a=b^c	
<<	desplazamiento a la izquierda	a=b<<3	a=b*8 #2**3=8
>>	desplazamiento a la derecha	a=b>>3	a=b//8 #los bits despl. se pierden

Tablas de verdad

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

a	~a
0	1
1	0

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

Diferencia entre operadores lógicos y bit a bit

Cuando operamos con enteros los operadores lógicos lo hacen con el valor final, considerando como verdadero (`True`), cualquier valor distinto de cero (`0`). La operación `and` evalúa el primer operando y devuelve cero si este es igual a cero, sino devuelve el segundo. La operación `or` evalúa el primer operando y lo devuelve si este es distinto de cero, sino devuelve el segundo. La operación not devuelve `True` si el operando es igual a cero, o `False` si es distinto de cero.
Los operadores bit a bit trabajan con la representación interna del `int` (binaria).

```
i, j = 15, 22  #i=0b00000111 j=0b00010110
print('Lóg.:', i and j, i or j, not i, not j, '- Bit:', i&j, i|j, i^j, ~i, ~j)
Lóg.: 22 15 False False - Bit: 6 31 -16 -23 25
#~i y ~ j devuelven el complemento a la base del número
#   0b00001111      0b00001111      0b00001111    ~ 0b00001111
# & 0b00010110    | 0b00010110    ^ 0b00010110                    ~ 0b00010110
#   0b00000110      0b00011111      0b00011001      0b11110000      0b11101001
#            6              31              25             -16             -23
#en una expresión lógica 22 y 15 serán evaluados luego como True
```

Con los operadores lógicos a nivel bit podemos trabajar sobre bits individuales de registros (un registro es una variable que representa en cada uno de sus bits estados que pueden tomar valores `0` o `False`, o `1` o `True`). Para esto se implementa el uso de máscaras donde un único bit es distinto al resto (justamente el bit sobre el cual necesitamos operar), lo que nos permite leer o cambiar bits específicos.

Como trabajar con el bit 2 (el 3° desde la derecha):				
Para	**Dato**	**Máscara**	**Operación**	**Resultado**
Leer	a=0b00000101	m=0b00000100	r=a&m	r=0b00000100 (1)
Apagar	a=0b00000101	m=0b11111011 (2)	r=a&m	r=0b00000001
Encender	a=0b00000001	m=0b00000100	r=a|m	r=0b00000101
Cambiar estado	a=0b00000101	m=0b00000100	r=a^m	r=0b00000000

```
(1) Uso: if r:  #False si r=0
(2) Esta máscara es la negación de la original: 0b11111011 = ~ 0b00000100
    También podemos usar la máscara original: r=a&(~m)
    La máscara para trabajar sobre el bit n será siempre 2**n
```

Pertenencia	
in	Evalúa si valores o expresiones de cualquier tipo pertenecen a un str, tuple, list, dict
not in	

Identidad	
is	x is y (x == y)
is not	x is not y (x != y)

Jerarquía o precedencia de de operadores	
()	Paréntesis, rompen cualquier jerarquía, pueden anidarse
~ + -	Suma, resta y negación (bit a bit), unarios
**	Potenciación
* / // %	Producto, cociente, cociente entero, resto
+ -	Suma y resta binarios
<< >>	Desplazamiento de bits
< <= > >=	Relacionales de comparación
== !=	Relacionales de igualdad y desigualdad
&	And (bit a bit)
|	Or (bit a bit)
= += -= *= /= %= &= ^= |= >>= <<=	Operadores de asignación (simple o compuesta)
Los operadores aritméticos de igual jerarquía se ejecutan de izquierda a derecha, a excepción de la potenciación.	

Cadenas de caracteres

Se pueden usar comillas simples (‘), o dobles (“), para cadenas de una sola línea, y \’ o \” para incluir comillas en una cadena (\n para nueva línea). Para cadenas de varias líneas manteniendo los saltos de línea podemos usar comillas triples.

`Msg=‘cadena’`
`print(msg)`
`cadena`
	`msg=“cadena”`
	`print(msg)`
	`cadena`

`msg=‘’’cadena de`
`varias líneas`
`’’’`
`print(msg)`
`cadena de`
`varias líneas`
	`Msg=”cadena de \nvarias líneas”`
	`print(msg)`
	`cadena de`
	`varias líneas`

```
#para evitar saltos de línea no deseados se usa r o R antes de las comillas
print(r”cadena de \nvarias líneas”)
cadena de \nvarias líneas
```

Indexación de cadenas	
`cadena[0]`	primer carácter
`cadena[-1]`	último carácter
`cadena[1:3]`	subcadena desde el carácter 1 (el 2°, incluido), hasta el 3 (el 4°, no incluido)
`cadena[:2]`	subcadena desde el principio hasta el carácter 2 (no incluido)
`cadena[4:]`	subcadena desde el carácter 4 (incluido), hasta el final
`cadena[-2:]`	subcadena con los últimos dos caracteres

Funciones y métodos para cadenas

Función o método	**Resultado**
`len(cad)`	longitud de la cadena `cad`
`cad.count(subcad)`	número de veces que se repite `subcad` dentro de `cad`
`cad.capitalize()`	cadena con el 1° carácter en mayús. y el resto en minús.
`cad.title()`	1° letra de cada palabra en mayús. y el resto en minús.
`cad.lower() / upper()`	cadena con todas las letras en minús. / mayús.
`cad.rstrip() / lstrip() / strip()`	elimina espacios al final / principio / ambos extremos
`cad.isalpha()`	True si todos sus caracteres son alfabéticos y cad!=’’
`cad.isdigit()`	True si todos sus caracteres son dígitos y cad!=’’
`cad.endswith(subcad)`	True si `cad` finaliza con `subcad`
`cad.startswith(subcad)`	True si `cad` inicia con `subcad`
`lista=cad.split(separador)`	genera un list del str usando separador como delimitador
`cad=cad.replace(busca, reemplaza)`	reemplaza `busca` por `reemplaza` en cad
`dir(cad)`	muestra todos los métodos asociados a `cad`

Ver: `https://docs.python.org/es/3/library/stdtypes.html#string-methods`

Tuplas

Una tupla contiene una colección de datos inmutable (no pueden ser modificados una vez creados), separados por comas, indexados con enteros (el primer elemento es el de índice 0), donde sus elementos pueden ser de distintos tipos. Para su definición se pueden usar o no los paréntesis:

```
miTupla=()  #crear una tupla vacía
miTupla=tuple()  #otra forma de crear una tupla vacía
#tupla de 1 elemento (la coma indica que es de tipo tuple y no str)
miTupla="Pepe",
#creación o "empaquetado" de tuplas
miTupla="Pepe", "Carpintero", 33
#es equivalente a: miTupla = ("Pepe", "Carpintero", 33)
print(miTupla[2])
33
print(miTupla.index(33))
2
print(miTupla[1:2])
('Carpintero',)

#algunas operaciones con tuplas
print('Pepe' in miTupla)  #¿Está el elemento 'Pepe' en miTupla?
True
print(miTupla.count(33))  #cantidad de veces que se encuentra el argumento
1
print(len(miTupla))  #cantidad de elementos
3

#asignación múltiple o "desempaquetado" de tuplas
miTupla="Pepe", "Carpintero", 33  #empaquetado de tupla
nombre, ocupacion, edad=miTupla  #desempaquetado de tupla
print(miTupla)
("Pepe", "Carpintero", 33)
print(nombre, ocupacion, edad)
Pepe Carpintero 33
```

Listas

Similar a la tupla, pero es una estructura de datos mutable (sus elementos se pueden modificar), se define con valores separados por comas y encerrados entre corchetes. Sus elementos al igual que en las tuplas pueden ser de distinto tipo:

```
miLista=[]  #crear una lista vacía
miLista=list()  #otra forma de crear una lista vacía
miLista=[1, 5, 'x', 'abc', 3.14]
print(miLista)
[1, 5, 'x', 'abc', 3.14]

print(miLista[1])  #el indexado devuelve un elemento
5
miLista[1] = 3  #sus elementos se pueden modificar
print(miLista)
[1, 3, 'x', 'abc', 3.14]

#también se pueden recortar y concatenar generando nuevas listas o rebanadas
#lista[inicio:fin:paso] incluye inicio, no incluye fin, salta paso entre items
#por ejemplo:
#    elementos de índice par  : lista[0:len(lista):2] o lista[::2]
#    elementos de índice impar: lista[1:len(lista):2] o lista[1::2]
print(miLista[1:4])  #rebanada con los elementos 1 (incluido), al 4 (excluido)
[3, 'x', 'abc']
print(miLista[:3])  #desde el inicio hasta el 3 (excluido)
[1, 3, 'x']
print(miLista[2:])  #2 (incluido), hasta el final
['x', 'abc', 3.14]
print(miLista[:])  #toda la lista (inicio a final)
[1, 3, 'x', 'abc', 3.14]
print(miLista[::2])  #toda la lista (inicio a final), saltando de 2 en dos
[1, 'x', 3.14]

print(miLista + [77, 88])  #concatenación de dos listas
[1, 3, 'x', 'abc', 3.14, 77, 88]

#el método append(elem) permite agregar un elemento al final de la lista
miLista.append(33)  #equivalente a: miLista=miLista + [33]
print(miLista[:])
[1, 3, 'x', 'abc', 3.14, 33]

#para insertar un nuevo elemento en una posición determinada
miLista.insert(1, 'km')
print(miLista)
[1, 'km', 3, 'x', 'abc', 3.14, 33]

#eliminar elementos cuando conocemos su posición
del(miLista[2])  #también: del miLista[2]
print(miLista[:])
[1, 'km', 'x', 'abc', 3.14, 33]

#eliminar el primer elemento coincidente encontrado (error si no lo encuentra)
miLista.remove('abc')
print(miLista)
[1, 'km', 'x', 3.14, 33]
```

```
#consultar si un elemento está en la lista
print(3.14 in miLista)
True

#guardar el último elemento de la lista y luego eliminarlo
eliminado = miLista.pop()
print(miLista)
[1, ‘km’, ‘x’, 3.14]
print(eliminado)
33

#eliminar el segundo elemento (o cualquier otro)
miLista.pop(1)
print(miLista)
[1, ‘x’, 3.14]

#consultar el índice de un elemento
print(miLista.index(3.14))
2

#las listas pueden contener otras listas como elementos
miLista=[1, 2, [3, 4], 5]
#el elemento 2 es una lista
print(miLista[2])
[3, 4]
#puedo consultar el elemento de una lista dentro de otra lista
print(miLista[2][1])
4

#ordenar los elementos de una lista
miLista = [3, 5, 7, 1]
miLista.sort()
print(miLista)
[1, 3, 5, 7]
```

Funciones y métodos de listas	
`miLista.clear()`	Limpia la lista (`miLista = []`)
`miLista.append(item)`	Agrega un elemento al final de la lista
`miLista.insert(pos, item)`	Inserta un elemento en la posición especificada
`miLista.extend(otraLista)`	Agrega `otraLista` al final de `miLista`
`miLista+=otraLista`	Agrega `otraLista` al final de `miLista`
`largo=len(miLista)`	Retorna la cantidad de elementos de `miLista`
`cant=miLista.count(item)`	Devuelve el número de veces que está item en la lista
`pertenece=item in miLista`	Retorna True si el elemento pertenece a la lista
`pos=miLista.index(item)`	devuelve el índice del item (error si no pertenece a la lista)
`eliminado=miLista.pop()`	Elimina el último item (y lo devuelve como valor)
`eliminado=miLista.pop(n)`	Elimina el item de índice n (y lo devuelve como valor)
`miLista.remove(item)`	Elimina la primer coincidencia con item
`miLista.reverse()`	Invierte la posición de los elementos
`miLista.sort()`	Ordena la lista (elementos del mismo tipo)
`miLista.sort(reverse = True)`	Ordena la lista (en orden descendente)

Diccionarios

En lugar de usar un índice entero para acceder a sus elementos como en las tuplas y las listas, un diccionario define entre llaves pares de claves y valores. Las claves deben ser inmutables y únicas (no pueden repetirse). No tienen un orden definido ni un índice y por lo tanto no pueden recortarse.

```
dic_vacio={}
dic_vacio=dict()
cuadrados={1:1, 2:4, 3:9, 4:16, 5:25}
figuras={‘triangulo’:3, ‘cuadrado’:4, ‘rectangulo’:4, ‘octagono’:8}
miDic={‘nombre’: ‘Pepe’, ‘ocupacion’: ‘Carpintero’, ‘edad’: 33}
print(miDic)
{‘nombre’: ‘Pepe’, ‘ocupacion’: ‘Carpintero’, ‘edad’: 33}
#para consultar un valor del diccionario se usa su clave como índice
print(miDic[‘edad’])
33

del(miDic[‘ocupacion’])  #eliminar un par clave-valor
print(miDic)
{‘nombre’: ‘Pepe’, ‘edad’: 33}

#al asignar un nuevo valor si la clave no existe se crea el par clave-valor
miDic[‘altura’]=1.67
print(miDic)
{‘nombre’: ‘Pepe’, ‘edad’: 33, ‘altura’: 1.67}
#pero si la clave existe se modifica el valor
miDic[‘altura’]=1.82
print(miDic)
{‘nombre’: ‘Pepe’, ‘edad’: 33, ‘altura’: 1.82}

#modificar o agregar varios elementos a la vez desde otro diccionario
miDic.update({‘peso’: 82.3, ‘ocupacion’: ‘Carpintero’, ’DNI’: 11222333})
print(miDic)
{‘nombre’: ‘Pepe’, ‘edad’: 33, ‘altura’: 1.82, ‘peso’: 82.3, ‘ocupacion’:
‘Carpintero’, ‘DNI’: 11222333}
otroDic={‘deporte’: ‘natacion’, ‘edad’: 44}
miDic.update(otroDic)
print(miDic=
{‘nombre’: ‘Pepe’, ‘edad’: 44, ‘altura’: 1.82, ‘peso’: 82.3, ‘ocupacion’:
‘Carpintero’, ‘DNI’: 11222333, ‘deporte’: ‘natacion’}
#agrega deporte, pero como edad ya existe solo modifica su valor de 33 a 44

#eliminar elementos
miDic.pop(‘ocupacion’)
print(miDic)
{‘nombre’: ‘Pepe’, ‘edad’: 44, ‘altura’: 1.82, ‘peso’: 82.3, ‘DNI’: 11222333,
‘deporte’: ‘natacion’}

print(len(miDic))  #ver la cantidad de elementos
6
```

Pertenencia

por clave		por valor
ok=‘name’ in miDic.keys() print(esta) False	ok=‘name’ in miDic print(ok) False	ok=33 in miDic.values() print(ok) True

Conjuntos (set)

Un set es una colección de datos no ordenados, no indexados y **mutable** de elementos únicos. Sus elementos pueden ser de cualquier tipo, no se permiten duplicados ni modificar sus elementos, pero se pueden quitar y agregar elementos. True y 1 se consideran duplicados.
Los conjuntos pueden ser valores de un diccionario pero no claves ni elementos de otro conjunto.

```
a=set()
b=set((1, 2, 3))  #set recibe un solo argumento
print(b, len(b))
{1, 2, 3} 3
b.add(4)  #agregamos un elemento
{1, 2, 3, 4}
frutas={‘pera’, ‘ananá’, ’manzana’, ’naranja’}
print(frutas, len(frutas))
{'naranja', 'manzana', 'pera', 'ananá'} 5
#otras formas de definir un conjunto:

b.update(frutas)  #con cualquier iterable
{‘naranja’, True, 1, 2, 3, 4, ‘pera’, ‘manzana’, ‘ananá’}
c=b.union(a)  #c toma los elementos de b y de a
x=a.intersection(b)  #devuelve un nuevo conjunto con los elementos en común
a.intersection_update(b)  #a se queda solo con los elementos en común con b

#x se queda con los elementos que no son comunes a ambos conjuntos
x.symmetric_difference_update(y)
#eliminar un elemento
b.remove(‘pera’)  #si el elemento no existe se produce un error
print(b)
{‘naranja’, True, 1, 2, 3, 4, ‘manzana’, ‘ananá’}
b.discard(100)  #si el elemento no existe NO se produce un error
print(b)
{‘naranja’, True, 1, 2, 3, 4, ‘manzana’, ‘ananá’}
```

Para eliminar un elemento de un conjunto también podemos usar el método `pop`, pero se eliminará un elemento al azar (aunque pop devuelve el elemento eliminado).

```
x=c.pop()  #error si el conjunto está vacío
del c  #elimina el conjunto
```

Iterar conjuntos

```
frutas={‘pera’, True, ‘ananá’, ’manzana’, ‘ananá’, ’naranja’, 1}
for fruta in frutas:
    print(fruta)
```

Frozenset

Un frozenset es una colección de datos no ordenados, no indexados e **inmutable** de elementos únicos. Sus elementos pueden ser de cualquier tipo, no se permiten duplicados ni modificar sus elementos, y a diferencia de los sets **NO** se pueden quitar y agregar elementos.
Un frozenset puede ser un elementos de otro conjunto y sus elementos tanto valores como claves de un diccionario.

```
fruits = {"Apple", "Banana", "Cherry", "Apple", "Kiwi"}  #set
basket = frozenset(fruits)  #frozenset
```

Entrada por teclado

La función `input()` toma lo ingresado por el teclado y retorna una cadena de caracteres luego de presionar la tecla INTRO o ENTER, cadena que luego podemos convertir a otros tipos de datos.

```
#leer una cadena de caracteres después de mostrar un texto
var = input(“Mensaje: ”)

#leer una cadena y convertirla a un entero
var = int(input(“Mensaje”))

#lo mismo para otros tipos de datos
real = float(input(“Ingrese el precio: ”))

#pero ese texto de ayuda es opcional
print(‘Ingrese dos números, presione ENTER después de cada ingreso:’)
a = int(input())
b = int(input())

#al ser Python dinámicamente tipado una variable puede cambiar su tipo durante
#en tiempo de ejecución: var se define 1° como str y luego pasa a ser int.

var = input(“Ingrese un número entero: ”)  #var toma 1° un valor de str
var = int(var)  #pasa de str a int

var = int(input(“Mensaje”))  #la forma más común de verlo en Python

#también puedo generar una pausa, sin guardar lo ingresado en una variable:
input(‘Presione ENTER para continuar...’)
```

Salida por pantalla

```
#imprimir en pantalla constantes, variables y expresiones separadas por comas
cantidad=3
importe=5.252
print(cantidad, 'x', importe, '=', cantidad * importe)
3 x 5.252 = 15.756
#print separa automáticamente sus argumentos con un espacio

#normalmente print salta a una nueva línea luego de imprimir sus argumentos
#ese comportamiento se puede cambiar con el argumento de palabra clave end
print('precio:', end=' - ')  #cambia el final de nueva linea ('\n') por ' - '
print(importe)
print(cantidad)
precio: - 5.252
3

#otro argumento de palabra clave de print es sep, que permite cambiar el
#separador de argumentos predeterminado (un espacio)
print(cantidad, ' x ', importe, ' = ', cantidad * importe, sep='~')
3~ x ~5.252~ = ~15.756

#salida con formato
print("Cantidad: {} - Precio: {}".format(cant, importe))  #poco usado
Cantidad: 3 - Precio: 5.252

#salida con formato mediante fstrings, más fáciles de leer en el código
print(f"Cantidad: {cant} - Precio: {importe}")
Cantidad: 3 - Precio: 5.252
#precio formateado con dos decimales
print(f"precio: {importe:.2f}")
precio: 5.25
```

Salida en múltiples líneas

```
#en bloque
print(f'''-----   TOTAL   -----
cantidad Precio
{cantidad:4}{importe:10.2f}
''')
-----   TOTAL   -----
cantidad Precio
   3      5.25

#fstrings multilinea
msg = (
      f'-----   TOTAL   -----\n'
      f'cantidad Precio\n'
      f'{cantidad:4}{importe:10.2f}\n'
)
print(msg)
-----   TOTAL   -----
cantidad Precio
   3      5.25
```

Caracteres de escape	
\n	Salto de línea
\t	Tabulación
\”	Para imprimir comillas simples dentro de un str creado con comillas simples
\’	Para imprimir comillas simples dentro de un str creado con comillas simples
\\	Barra invertida

Distintos formatos en las fstrings	
{variable}	inserta el valor de la variable en la cadena
{expresion}	evalúa la expresión y luego inserta el resultado en la cadena
{variable:tipo}	especifica el tipo de la variable a insertar:
{variable:d}	enteros
{variable:b}	muestra un entero como binario
{variable:f}	reales
{variable:.nf}	reales con n decimales
{variable:m.nf}	reales con n decimales en un campo de m caracteres en total (incluidos los n decimales y el punto; si el valor tiene mas dígitos enteros se muestran todos sin respetar el límite m)
{variable:e}	notación científica
{variable:.ne}	notación científica con n decimales
{variable:.n%}	porcentajes (multiplica el valor por 100), con n decimales
{variable:s}	cadenas
{variable:<n}	alineado a la izquierda en una columna de n caracteres
{variable:>n}	alineado a la derecha en una columna de n caracteres
{variable:^n}	centrado en una columna de n caracteres
{variable:0>n}	rellena la columna de n caracteres en total con ceros a la izquierda

Orden de los modificadores: [alinecion][ancho][,][.precision][tipo]

Estructuras de Control

Las estructuras de control de un lenguaje permiten ejecutar código de forma condicional tras evaluar si se cumple o no una expresión lógica o relacional.

Simple	Doble	Múltiple

`if condicion:`
`    instrucciones`
	`if condicion:`
	`    instrucciones`
	`else:`
	`    instrucciones`
		`if condicion:`
		`    instrucciones`
		`elif condicion:`
		`    instrucciones`
		`else:`
		`    instrucciones`

Debe haber un if, los elif que sean necesarios y un solo else (los elif y else son opcionales)		

Condicional doble en una sola línea

```
#instruccion_True if condicion else instruccion_False
num = 34
paridad = "par" if num%2==0 else "impar"
print(f'{num} es {paridad}.')
34 es par.
```

Formato condicional

```
num = 35
print(f'{num} es {"par" if num%2==0 else "impar"}.')
35 es impar.
```

Estructuras de Iteración

Las estructuras de iteración permiten repetir instrucciones si se cumple una condición.

`while condicion:` `    instrucciones`	`for valor in secuencia:` `    instrucciones`

`N=1`
`while n<=3:`
`    print(n, end=’’)`
`    n+=1`
`1 2 3`
	`for n in (1, 2, 3):`
	`    print(n, end=’’)`
	`1 2 3`

Para los bucles es muy útil la función `range` que devuelve una secuencia de rango de valores `int`.

`range(3)` `range(0, 3)`	`range(5, 10)` `range(5, 10)`	`range(5, 10, 2)` `range(5, 10, 2)`

Cuando se usa un solo argumento éste se toma como valor final (el valor final nunca se incluye en la secuencia), y los valores van desde cero hasta ese argumento (`0` a `fin-1`), con dos argumentos (valor inicial y valor final), los valores van desde el primero hasta el segundo (`inicio` a `fin-1`), y con tres argumentos los valores van desde el primero hasta el segundo (siempre sin incluirlo), con saltos o pasos indicados por el tercero (`ini` a `fin-1`, saltando `pasos`).

Para optimizar el uso de memoria `range` tiene su propio tipo de datos, pero podemos pasar esos valores a una lista:

```
type(range(3))
<class ‘range’>
```

`list(range(3))` `[0, 1, 2]`	`list(range(5, 10))` `[5, 6, 7, 8, 9]`	`list(range(5, 10, 2))` `[5, 7, 9]`

Formas de iterar con for

`for i in range(3):`
`    print(i)`
`0`
`1`
`2`
	`for i in ‘abc’:`
	`    print(i)`
	`a`
	`b`
	`c`
		`for i in [‘abc’, 5, 3.3]:`
		`    print(i)`
		`abc`
		`5`
		`3.3`

Iteración con diccionarios

```
d = {‘nombre’: ‘Pepe’, ‘edad’: 33, ‘alt’: 1.82, ‘peso’: 82.3, ‘DNI’: 11222333}
```

Iteración por claves	**Iteración por valores**	**Iteración por claves y valores**

`for clave in d.keys():`
`    print(clave)`
`nombre`
`edad`
`alt`
`peso`
`DNI`
	`for valor in d.values():`
	`    print(valor)`
	`Pepe`
	`33`
	`1.82`
	`82.3`
	`11222333`
		`for clave, valor in d.items():`
		`    print(clave, valor)`
		`nombre Pepe`
		`edad 33`
		`alt 1.82`
		`peso 82.3`
		`DNI 11222333`

`break-continue-else`

`break`	sale del bucle
`continue`	salta al principio del bucle
`else`	ejecuta sus instrucciones al salir del bucle, excepto si sale por una instrucción `break`

Funciones

Son bloques de código que son ejecutados cuando son llamados, pueden tomar argumentos y devolver resultados, ambos opcionales. Se llaman parámetros en la definición de la función y argumentos en las llamadas a la función.
Las funciones built-in estan definidas dentro del lenguaje, no es necesario importarlas (int, float, str, tuple, list, dict, range, len, print, input, abs, min, max, map, pow, filter, round, type, upper, lower, open, etc.).
Documentación oficial: https://docs.python.org/es/library/functions.html

```
#función sin argumentos y que no retorna ningún valor
def pausa():
    input('Presione ENTER para continuar…')
pausa()
Presione ENTER para continuar...

#función que recibe un argumento pero no retorna ningún valor
def saludo(nombre):
    print(f'Hola {nombre}!')
saludo('Pepe')
Hola Pepe!

#función sin argumentos que retorna un valor
def dos_pi():
    return 2*3.14
print(dos_pi())
6.28

#función con dos argumentos que retorna un valor
def suma(a, b):
    return a + b
print(suma(2, 3))
5
```

Las funciones en Python en realidad siempre devuelven un valor, si no usamos `return` para devolver explícitamente un valor, la función devuelve `None`. El valor devuelto es único, pero puede ser de cualquier tipo, incluyendo los tipos estructurados como tuplas, listas, diccionarios, etc.
Los argumentos pueden ser:

- **posicionales**: se corresponden según su orden (el 1° argumento se corresponde con el 1° parámetro, etc.)
- **nombrados**: los puedo ubicar en cualquier lugar en la llamada a la función si uso la fórmula `argumento=valor`, si se emplean ambos en una misma llamada se deben pasar primero los posicionales y luego los nombrados
- **opcionales**: si en la definición le asigno un valor a un parámetro, el argumento es opcional y el valor asignado en la definición de la función es su valor predeterminado. Los parámetros opcionales siempre van al final de la lista de parámetros.

```
#función con dos argumentos que retorna una tupla con múltiples valores
def operaciones(a, b):
    return (a+b, a-b, a*b, None if b==0 else a/b)
print(operaciones(2, 5))
(7, -3, 10, .4)
print(operaciones(2, 0))
(2, 2, 0, None)
print(operaciones(b=2, a=0))  #argumentos nombrados, no importa el orden
(2, -2, 0, 0)
```

```
#función con un argumento opcional, si se omite toma el valor predeterminado
def ficha_paciente(paciente, donante=True):
    print(f'Paciente: {paciente} - Donante: ({"Si" if donante else "No"})')
ficha_paciente('Pepe')
Paciente: Pepe - Donante: (Si)
ficha_paciente('José', False)
Paciente: José - Donante: (No)
```

- Un parámetro al que se le asigna un valor en la definición de una función es un valor predeterminado para ese parámetro opcional
- Un argumento al que se le asigna un valor en el llamado a una función es un argumento nombrado

Argumentos por valor y por referencia

Los argumentos pueden pasarse por valor, una copia de la variable, o por referencia, la dirección de memoria de la variable. Los argumentos se pasan por valor o por referencia según su tipo. Por valor se pasan los datos simples como int, float, str, bool. Si se modifica el valor del parámetro dentro de la función, el valor de la variable pasada como argumento no se modifica. Por referencia se pasan las colecciones como tuplas, listas, diccionarios, etc. Si dentro de la función se modifica el valor de ese parámetro, ese cambio se ve reflejado también en la variable usada como argumento.

```
#los argumentos pasados por valor pueden cambiar dentro de la función sin
#afectar a la variable original
def doble(a):  #el parámetro a recibe el contenido de la variable a
    a=2*a  #el parámetro recibe el doble de su valor original
    return a
a=2  #la variable a recibe el valor 2
print(doble(a))  #print muestra el valor retornado por la función
4
print(a)  #el valor original de la variable no se modifica dentro de la función
2

#si un argumento pasado por referencia es modificado dentro de la función, ese
#cambio se ve reflejado en la variable original
def mayus(lista):
    for i in range(len(lista)):
        lista[i]=lista[i].upper()
nombres=['pepe', 'ana', 'carlos']
print(nombres)
['pepe', 'ana', 'carlos']
mayus(nombres)  #la función no devuelve nada (no hay un return)
print(nombres)  #la variable cambió su contenido en la función
['PEPE', 'ANA', 'CARLOS']
```

Cantidad variable de argumentos

Cuando no sabemos la cantidad de argumentos que le pasaremos a una función, utilizamos en la definición de la función un asterisco como prefijo del parámetro, indicando que la función recibirá una tupla de argumentos:

```
def saludar(*nombres):
    for nombre in nombres:
        print(f'Hola {nombre}!!!')
saludar('Pepe', 'Pancho', 'Lucho')
Hola Pepe!!!
Hola Pancho!!!
Hola Lucho!!!
```

En la documentación de Python se refiere usualmente a estos argumentos como *args.

Cantidad variable de argumentos nombrados

Cuando no sabemos la cantidad de argumentos nombrados que le pasaremos a una función, utilizamos en la definición de la función dos asteriscos como prefijo del parámetro, indicando que la función recibirá un diccionario de argumentos:

```
def saludar(**usuarios):
    for usuario in usuarios:
        print(f'Hola {usuario["nombre"]}!!!')
saludar('Pepe', 'Pancho', 'Lucho')
Hola Pepe!!!
Hola Pancho!!!
Hola Lucho!!!
```

En la documentación de Python se refiere usualmente a estos argumentos como **kwargs.

Recursión

La recursión se da cuando una función se llama a sí misma, por lo que se debe tener mucho cuidado para evitar que la función no termine nunca o acapare demasiada memoria o procesador.
Es un concepto común en matemáticas y programación, y para ciertos problemas permite soluciones elegantes.

```
def fibo(n):
    return n  if n==0 or n==1 else fibo(n-1) + fibo(n-2)
for i in range(10):
    print(fibo(i), end=' ')
0 1 1 2 3 5 8 13 21 34
```

Funciones lambda

Una función lambda es una pequeña función anónima que solamente puede tener una única expresión:

```
#lambda args: expr
raiz_n = lambda x, r: x ** (1/r)
print(raiz_n(2, 2))
1.4142135623730951
```

El poder de las funciones lambda se entiende mejor cuando son usadas dentro de otras funciones:

```
def potencia(n):
    return lambda x: x**n
cuadrado=potencia(2)
cubo=potencia(3)
print(cuadrado(5), cubo(5))
```

Archivos

Para abrir un archivo se usa la función `open` que requiere dos parámetros, el nombre del archivo y el modo de apertura:

Modo	Aplicación	Si el archivo existe	Si el archivo no existe
‘r’	Read o leer, predeterminado	Permite leer	Devuelve un error
‘a’	Append o agregar	Permite escribir al final (no leer)	Crea el archivo
‘w’	Write o escribir	Borra su contenido (sobreescribe)	Crea el archivo
‘x’	Crear, para escribir	Devuelve un error	Crea el archivo

Modos adicionales: para indicar si debe ser tratado como texto o como binario	
‘t’	Text o texto, modo predeterminado
‘b’	Binary o binario, para trabajar por ejemplo con archivos de imágenes

Modos compuestos	
‘r+’	Lectura y escritura
‘r+b’	Lectura y escritura con archivos binarios

Para los archivos de texto opcionalmente se puede indicar su codificación, siendo utf-8 la más usual.

El nombre del archivo también puede contener su ubicación si no se encuentra en la misma carpeta del código:

```
#dirección absoluta
f=open(‘/home/pepe/datos/notas.txt’)  #f=open(r‘D:\\pepe\datos\notas.txt’)

#dirección relativa a la carpeta donde está el código
f=open(‘datos/notas.txt’)  #f=open(r‘datos\notas.txt’)
```

Tres formas equivalentes de abrir un archivo de texto en modo lectura:

- `f=open(‘notas.txt’)`
- `f=open(‘notas.txt’, ’r’)`
- `f=open(‘notas.txt’, ’rt’)`

```
archivo=’ventas.dat’
modo=’r’
codificacion=’utf-8’  #solo para archivos de texto
f=open(archivo, modo, codificacion)
datos=f.read()  #devuelve un str con el archivo (incluidos los \n)
print(datos)
f.close()d

#con with no es necesario cerrar el archivo, al salir del mismo se cierra
#automáticamente
with open(archivo, modo, codificacion) as f:
    datos=f.read()
    print(datos)
```

```
f=open(archivo, modo, codificacion)
for linea in f.readlines():  #devuelve una lista (un str por linea, \n al final)
    print(linea, end='')

f=open(archivo, modo, codificacion)
for linea in f:  #más eficiente, usa solo la referencia al archivo para iterar
    print(linea, end='')

modo='w'
f=open(archivo, modo, codificacion)
lista=['uno', 'dos', 'tres', 'cuatro', 'cinco']
for linea in lista:
    f.write(linea+'\n')  #devuelve la cantidad de bytes escritos
f.close()

f.seek(0)  #ir al principio del archivo
```

Formato JSON

Un archivo de datos en formato JSON (JavaScript Object Notation), tiene una estructura similar a una lista de diccionarios en Python:

```
[
  {
    "usuario": "admin",
    "clave": "1234"
  },
  {
    "usuario": "ventas",
    "clave": "5678"
  },
  {
    "usuario": "compras",
    "clave": "0303"
  }
]
```

Un ejemplo de código:

```
import json

#lectura de datos
f_usuarios=open('usuarios.json', 'r')
usuarios=json.load(f_usuarios)
f_usuarios.close()

#codigo para procesar los datos de la lista creada con load

#escritura de datos
f_usuarios=open('usuarios.json', 'w')
json.dump(usuarios, f_usuarios, indent=2, ensure_ascii=False)
f_usuarios.close()
```

En `dump` los argumentos `indent=2` y `ensure_ascii=False` permiten una lectura más fácil de los datos si abrimos el archivo json con un editor de textos (no es necesario si se abrirá unicamente desde el código), y la correcta escritura de vocales acentuadas respectivamente (sin caracteres de escape).

Librerías, módulos y paquetes

Librerías

Las librerías son conjuntos de módulos y paquetes de Python. La librería estándar se distribuye junto con Python. Muchas constantes, operaciones y funciones de uso habitual ya están implementadas en la librería estándar.

Documentación oficial: https://docs.python.org/es/library/index.html

Algunas librerías						
`os`						
`random`						
`math`						
`datetime`						
`statistics`						

```
from random import randint
n=randint()
```

```
from random import randint as azar
n=azar()
```

```
from random import *
n=random()
```

```
import random
n=random.randint()
```

Módulos

Un módulo es un archivo de Python con objetos que pueden ser accedidos desde otro archivo (constantes, variables, funciones, etc.). Nos permite organizar el código.

Módulo de ejemplo

```
#modulo.py
def funcion():
    pass
```

```
#codigo.py
from modulo import funcion
funcion()
```

Paquetes

Un paquete es una carpeta con varios módulos. Para que Python la reconozca como un paquete la carpeta debe contener un archivo vacío `__init__.py`.

```
import paquete.modulo
paquete.modulo.funcion()

from paquete.modulo import funcion
funcion()
```

Si en distintos módulos tenemos dos funciones con el mismo nombre, se utiliza la última importada, a menos que usemos un alias:

```
from paquete.modulo1 import funcion as fn1
from paquete.modulo2 import funcion
funcion()
fn1()
```

Entornos virtuales

A veces es necesario utilizar distintas versiones de librerías para por ejemplo mantener código heredado, cuyas funcionalidades pueden entrar en conflicto con versiones más actualizadas.
Desde la consola podemos utilizar `pip`, el gestor de paquetes propio de Python.

```
user@machine:~$ pip list  #lista los paquetes instalados a nivel global
```

Creación de un entorno virtual con Python:

```
user@machine:~/Documentos/Entorno1$ python3 -m venv nombre_env #normalmente env

source bin/activate

user@machine:~/Documentos/Entorno1$ nombre_env/Scripts/activate
(nombre_env) ... $ pip list

instalar paquetes para este entorno virtual


user@machine:~/Documentos/Entorno1$ nombre_env/Scripts/deactivate

deactivate
```

#escribe en .txt los requisitos de librerias del proyecto
pip freeze > requirements.txt
pip install -r requirements.txt

****
auto-py-to-exe

Manejo de errores

Para manejar desde el código los errores predecibles disponemos de las instrucciones try y except. Algunos ejemplos de mensajes de errores comunes:

- `ValueError`: cuando intentamos pasar un `str` con caracteres no numéricos a `int` o `float`
- `TypeError`: cuando intentamos concatenar con `+` un `str` y un `int`
- `ZeroDivisionError`: cuando intentamos dividir por cero
- `IndexError`: cuando intentamos acceder a una posición fuera de los límites de una lista
- `FileNotFoundError`: cuando intentamos abrir un archivo inexistente
- KeyboardInterrupt: cuando presionamos durante la ejecución de un programa CTRL+C

Ver: https://docs.python.org/es/3/tutorial/errors.html

```
edad=0  #si se dispara la excepción input no le asigna ningún valor a edad
try:
    edad=int(input('Ingrese su edad: '))
except:
    print('Ingrese un número...')
if edad>=18:
    print('Mayor de edad')
else:
    print('Menor de edad')

#otra forma:
try:
    edad=int(input('Ingrese su edad: '))
    if edad>=18:
        print('Mayor de edad')
    else:
        print('Menor de edad')
except:
    print('Ingrese un número…')

try:
    edad=int(input('Ingrese su edad: '))
except ValueError:
    print('Ingrese un número...')
if edad>=18:
    print('Mayor de edad')
else:
    print('Menor de edad')

while True:
    try:
        edad=int(input('Ingrese su edad: '))
        break
    except ValueError:
        print('Ingrese un número…')
if edad>=18:
    print('Mayor de edad')
else:
    print('Menor de edad')


while True:
    try:
        a=int(input('a='))
        b=int(input('b='))
```

```
        cociente=a/b
        break
    except ValueError:
        print('Ingrese un número...')
    except ZeroDivisionError:
        print('El divisor no puede ser cero...')
print(cociente)

finally:
```

except:

print(exception)

Bases de datos

Una base de datos es una colección de información que está organizada de manera que se pueda acceder, administrar y actualizar fácilmente. Las bases de datos informáticas suelen contener conjuntos de registros o archivos de datos.

conceptos de cliente y servidor, relacionales vs no realacionales

Componentes de una Base de datos

Entre los componentes de una base de datos podemos encontrar:

- **Tablas:** Es el componente principal de las Bases de Datos Relacionales y comprende definición de tablas, campos, relaciones e índices.
- **Formularios:** se utilizan principalmente para actualizar datos.
- **Consultas:** se utilizan para ver, modificar y analizar datos.
- **Informes:** se utilizan para presentar los datos en formato impreso
- **Macros:** conjunto de instrucciones para realizar una operación determinada.

Tablas

Podemos definirlas como un conjunto de datos homogéneo que contiene información sobre un tema específico, o como una colección de registros relacionados.

- **Registro:** es una fila en la tabla que almacena información sobre una misma entidad.
- **Campo:** es la unidad elemental de información y almacena un dato simple referido a una entidad.

SQL

SQL (Structured Query Language), o en español Lenguaje Estructurado de Consulta, es el lenguaje utilizado para definir, controlar y acceder a los datos almacenados en una base de datos relacional.
Se trata de un lenguaje universal empleado en cualquier sistema gestor de bases de datos relacional y cuenta con un estándar definido a partir del cual cada gestor ha desarrollado una versión propia.
No confundir gestor de base de datos (aplicación), con base de datos (información).

Sistemas Gestores de Bases de Datos Relacionales

Es el conjunto de programas, procedimientos, lenguajes, etc. que suministran tanto a los usuarios como a los analistas, programadores o administradores, los medios necesarios para describir, recuperar y manipular los datos almacenados en una base de datos, manteniendo su integridad, confidencialidad y seguridad.
Algunos de los gestores mas conocidos:

- **MySQL** : https://www.mysql.com/
- **MariaDB** : https://mariadb.org/
- **SQLite** : https://www.sqlite.org/index.html
- **PostgreSQL** : https://www.postgresql.org/
- **Microsoft SQL Server**: https://www.microsoft.com/es-es/sql-server/sql-server-downloads
- **Oracle** : https://www.oracle.com/es/downloads/

Sentencias SQL y su clasificación

Para interactuar con nuestra BD lo vamos a hacer mediante sentencias SQL, las cuales podemos clasificar según su propósito en tres grupos:

- **DDL (Data Description Language), o Lenguaje de Descripción de Datos:** normalmente usadas por el administrador de la BD (son sentencias para crear la BD, crear, eliminar o modificar estructura de tablas, definir relaciones entre tablas, etc.).

- **DCL (Data Control Language), o Lenguaje de Control de Datos:** permiten ejercer un control sobre los datos como asignación de privilegios de acceso a los mismos (`GRANT/REVOKE`) o en el caso de gestión de transacciones (`COMMIT/ROLLBACK`).
- **DML (Data Manipulation Language) o Lenguaje de Manipulación de Datos:** son las instrucciones más usadas por el usuario ya que se trata de aquellas que requieren el manejo de datos como insertar nuevos registros, modificar datos existentes, eliminarlos y hasta recuperar datos de la BD.

Formato de las sentencias

- Preferiblemente se escriben en mayúsculas para diferenciar sus partes de los nombres de las columnas y los datos
- Comienzan con un verbo indicando la acción a realizar
- Continúan con un objeto sobre el cual se realiza la acción
- Se complementan con una serie de cláusulas, obligatorias y opcionales, que especifican lo que se quiere hacer detalladamente
- Terminan con un punto y coma

Comentarios en SQL

Los comentarios comienzan con un doble guión:

```
--comentario en sql
```

Sentencias DDL básicas (descripción)

- `CREATE`: permite crear una base de datos o una tabla.
  ```
  CREATE DATABASE escuela;
  CREATE TABLE estudiantes (id int, nombre varchar(30), curso varchar(30);
  ```
- `DROP`: para eliminar una base de datos o una tabla.
  ```
  DROP DATABASE escuela;
  DROP TABLE estudiantes;
  ```
- `ALTER`: para modificar la definición de una tabla
  ```
  --cambiar el tipo de dato de una columna
  ALTER TABLE estudiantes ALTER COLUMN curso VARCHAR(20);
  --agregar una columna
  ALTER TABLE estudiantes ADD promedio REAL;
  ```

Sentencias DCL básicas (control)

- ...

Sentencias DML básicas (manipulación)

- `INSERT`: para agregar un registro o fila a una tabla.
  ```
  INSERT INTO estudiantes (id, nombre, curso) VALUES (1, ”Miriam”, ”react”);
  ```
- `DELETE`: para eliminar filas en una tabla
  ```
  DELETE FROM estudiantes WHERE id=1;
  ```
- `UPDATE`: para modificar filas de una tabla
  ```
  UPDATE estudiantes SET curso=“java” WHERE id=1;
  ```
- `SELECT`: con esta sentencia puedo consultar y traer de la tabla lo que necesito.
  ```
  --mostrar todos los campos de todas las filas
  SELECT * FROM estudiantes;
  --mostrar solo los campos nombre y promedio de los estudiantes de python
  SELECT nombre, promedio FROM estudiantes WHERE curso=“python”;
  ```

Herramientas

Todas estas sentencias las podemos realizar de manera mas practica con alguna herramienta visual o por línea de comando. De todas maneras es necesario conocer estas sentencias ya que no siempre vamos a trabajar con alguna de estas herramientas, sino que también vamos a tener la necesidad de

manipular nuestras bases de datos desde nuestro código. Algunas herramientas que nos facilitan esta tarea pueden ser:

- **MySQL Workbench** : https://dev.mysql.com/downloads/workbench/
- **phpMyAdmin** : https://www.phpmyadmin.net/
- **DB Browser for SQLite** : https://sqlitebrowser.org/
  Es una herramienta visual para crear, diseñar y editar archivos de bases de datos compatibles con SQLite.

Base de datos SQLite

SQLite es un gestor de bases de datos relacional que tiene por objetivo ser parte de la misma aplicación con la que colabora, es decir no cumple los conceptos de cliente y servidor.
SQLite forma parte de la biblioteca estándar de Python, así que si ya tenemos instalado el interprete, ya disponemos de SQLite y lo podemos usar importándolo como cualquier otro modulo:

```
import sqlite3
conexion=sqlite3.connect("base.db")  #si no existe la base de datos la crea
```

Crear tablas

Para crear tablas, primero vamos a definir un cursor (una estructura de control utilizada para el recorrido de los registros del resultado de una consulta).
A una variable, que la podemos llamar cursor, le asignamos el método `cursor()` de nuestra conexión. Luego podemos con el método `execute()` realizar consultas, y al final recordar cerrar la conexión para liberar recursos.

```
cursor=conexion.cursor()
cursor.execute("""CREATE TABLE animales (
    codigo INTEGER PRIMARY KEY AUTOINCREMENT,
    especie TEXT,
    edad INTEGER
)""")
cursor.close()
```

Crear tablas y evitar excepciones

El código anterior funciona siempre que la tabla no este creada en la BD, pero si ya existe nos dará un error `sqlite3.OperationalError: table animales already exists`.

Con el siguiente código lo solucionamos desde la consulta:

```
cursor.execute("""CREATE TABLE IF NOT EXISTS animales (
    codigo INTEGER PRIMARY KEY AUTOINCREMENT,
    especie TEXT,
    edad INTEGER
)""")
```

Con el siguiente código lo controlamos con los bloques `try except`:

```
try:
    cursor.execute("""CREATE TABLE animales (
        codigo INTEGER PRIMARY KEY AUTOINCREMENT,
        especie TEXT,
        edad INTEGER
    )""")
except sqlite3.OperationalError:
    print("La tabla ya existe")
```

Agregar registros a una tabla

Veamos dos maneras de insertar registro o filas a una tabla:

- Llamamos a `execute` con la consulta que genera la inserción del registro:
  `cursor.execute(“INSERT INTO animales VALUES (23, ‘Loro’, 5)”)`
  `cursor.execute(“INSERT INTO animales (especie, edad) VALUES (‘Gato’, 3)”)`
- Llamamos a `execute` y le pasamos como primer parámetro un comando `insert`, con el caracter '`?`' para indicar las posiciones de los campos a sustituir, y una tupla como segundo parámetro con los datos:
  `cursor.execute(“INSERT INTO animales (especie, edad) VALUES (?,?)”, (‘Perro’, 12))`

Recuperar filas de una tabla

Mediante el método `execute()` realizamos una consulta y tenemos el resultado en el cursor:

```
#recuperar toda la tabla
SELECT * FROM tabla_a

#recuperar los registros o filas que cumplan cierto criterio
SELECT * FROM tabla_a WHERE condicion

#recuperar solamente algunos campos
SELECT columna_1, columna_2, columna_n FROM tabla_a

#ordenar por alguna columna
SELECT * FROM tabla_a ORDER BY columna_x

#ordenar por alguna columna (orden inverso o descendente)
SELECT * FROM tabla_a ORDER BY columna_x DESC

#tambien podemos combinar distintas clausulas
SELECT columna_1, columna_2 FROM tabla_a WHERE condicion ORDER BY columna_1
```

Para acceder a los datos del cursor podemos usar los siguientes métodos:

- `fetchone()` devuelve el siguiente registro del cursor como una tupla
- `fetchall()` devuelve todos los registros como una lista de tuplas (una por cada fila)

```
cursor.execute(“SELECT * FROM animales”)
print(cursor.fetchone())  #()
print(cursor.fetchall())  #[()]

cursor.execute(“select * from animales”)
for fila in cursor:
    print(fila)  #()
```

Modificar un registro

Para modificar o eliminar un registro usamos el método `execute()` con `update` o `delete`:

```
cursor.execute(“UPDATE animales SET edad=8 WHERE codigo=3;”)
conexión.commit()

cursor.execute(“DELETE FROM animales WHERE codigo=3;”)
conexión.commit()
```

#
select distinct

#
select count(distinct columna)

and or not

#
delete all records

select top
limit
select top percent

min()
max()

count()
avg()
sum()

like

in
not in

between

alias

inner join
left join
right join
self join

union

group by

having

exist

any
all

insert into select

case