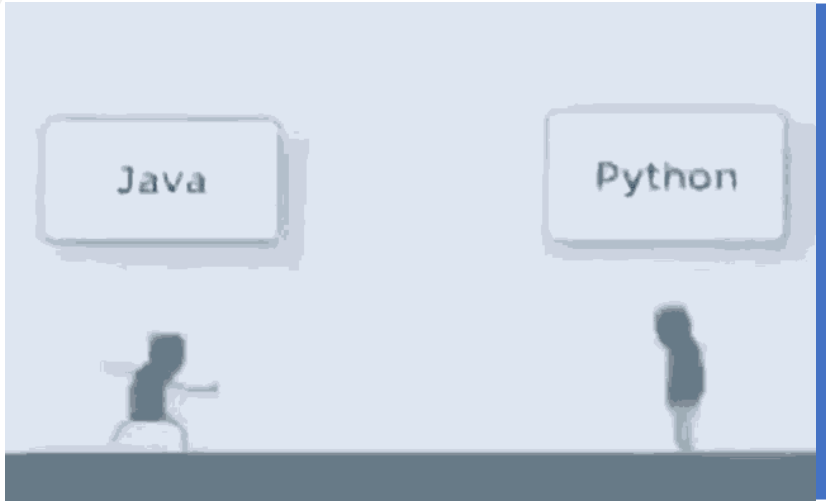
Sistemas de Gestión Empresarial

Python

Baldomero Sánchez

Bloque 3

UT 1



Introducción a Python

sentencias

Las sentencias son cada una de las ordenes ejecutables de un programa. En Python cada sentencia se debe delimitar separándolas en líneas, si queremos utilizar varias sentencias en una línea podemos usar punto y coma `;` para separarlas, aunque esta práctica no es recomendable.

```
sentencia 1
sentencia 2
sentencia 3; sentencia 4
sentencia 5; sentencia 6; sentencia 7; sentencia 8
```

Por motivos de legibilidad, se recomienda que las líneas no superen los 79 caracteres. Si una instrucción supera esa longitud, se puede dividir en varias líneas usando el carácter contrabarra `\`:

```
radio = 5
area = 3.14159265358979323846 \
       * radio ** 2
print(area)
```

sentencias

Los elementos del lenguaje se separan por espacios en blanco (normalmente, uno), aunque en algunos casos no se escriben espacios:

- Entre los nombres de las funciones y el paréntesis
- Antes de una coma (,)
- Entre los delimitadores y su contenido (paréntesis, llaves, corchetes o comillas)
- Excepto al principio de una línea, los espacios no son significativos, es decir, da lo mismo un espacio que varios.

Los espacios al principio de una línea (el sangrado) son significativos porque indican un nivel de agrupamiento. El sangrado inicial es una de las características de Python que lo distinguen de otros lenguajes, que utilizan un carácter para delimitar agrupamientos (en muchos lenguajes se utilizan las llaves { }). Por ello, **una línea no puede contener espacios iniciales, a menos que forme parte de un bloque de instrucciones o de una instrucción dividida en varias líneas**.

```
    radio = 5
```

Este código dará error debido a los espacios iniciales

comentarios

Los comentarios son zonas en el código que el interprete de Python va ignorar y no va a tener en cuenta a la hora de ejecutar el código.

Los comentarios de línea comienzan con el símbolo de almohadilla # y finalizan al final de la línea.

Python no dispone de comentarios que pueden ocupar varias líneas, pero se puede utilizar cadenas de caracteres multilínea colocando triple comilla doble (""") al principio y al final del comentario.

```
# Este es un comentario de una línea completa

sentencia1 # Este es un comentario de parte de
una línea

""" Esto puede servir como comentario
multilínea y puede
extenderse a lo largo de varias líneas """
```

palabras reservadas

Como cualquier lenguaje, Python posee un conjunto de palabras que utiliza para realizar tareas especificas y que por lo tanto no podemos utilizar para nombrar a nuestras variables. Este conjunto de palabras se conoce como “palabras reservadas”.

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

literales

Una literal es un valor constante formado por una secuencia de caracteres o números. Son ejemplos de literales los booleanos, los números, las cadenas de caracteres...

```
20
2.2
'Jorge'
```

En Python los literales representan valores de tipo numérico o de cadena de caracteres. Estos son valores fijos, constantes, y no variables.

Una cadena literal consta de cero o más caracteres encerrados entre comillas dobles ("), simples (') y triple comilla doble (""""). Las dos primeras solo pueden ser usadas en una sola línea, mientras que la triple comilla doble puede ser usada en varias líneas.

```
'foo'
"bar"
'1234'
"una linea \n otra linea"
"Hola \" Jorge\""

""" Una cadena
 de varias
líneas """
```

Una cadena debe estar delimitada por comillas del mismo tipo (es decir, ambas comillas simples, ambas comillas dobles, o triple comillas dobles).

literales

También podemos usar cadenas literales Unicode, colocando una *u* justo delante y literales de cadena de tipo bytes con una *b* delante como *b'\x89PNG'*

Un literal de tipo booleano se escribe con *True* o *False*.

Un literal numérico se puede escribir como entero, coma flotante o número complejo colocando una *j* al final.

Se pueden definir números en formato octal colocando *0o* justo antes del número y de igual manera, se pueden usar números en formato hexadecimal colocando *0x*

```
u"Hola"           (cadena literal unicode)
True, False       (booleano)
17, -34, 92       (entero)
0o777, 0o21       (entero octal)
0xAF2, 0x21B      (entero hexadecimal)
3.14              (coma flotante)
2+3j, -2j, 1-1j   (complejo)
```

En Python podemos usar _ (*underscore*) para separar las unidades de millar, de millón… si queremos. No tiene efecto sobre el valor numérico:

```
1_000_000
2_354_2
```

```
iva = 0x10
print(iva)
```

literales

Un literal de coma flotante puede tener las siguientes partes:

- Un entero decimal que puede tener un signo (precedido por "+" o "-"),
- Un punto decimal ("."),
- Una fracción (otro número decimal),
- Un exponente.
- La parte del exponente es una "e" o "E" seguida de un número entero, que puede tener signo (precedido por "+" o "-"). Un literal de coma flotante debe tener al menos un dígito y un punto decimal o "e" (o "E").

Específicamente, la sintaxis es:

```
[(+|-)][dígitos].[dígitos][(E|e)[(+|-)]dígitos]
```

En Python si escogemos un número muy grande que no es soportado por nuestro sistema obtendremos que es infinito mediante la palabra *inf*:

```
numero = 10e4567
print(numero)
```

Salida:

```
inf
```

```
3.14_15_926
-.123456789
-3.1E+12
.1e-23
```

variables

Una variable es un contenedor para almacenar un valor, como un número o una cadena de texto. Realmente, son espacios de memoria reservados para almacenar los datos y el nombre de la variable hace referencia a ese lugar.

Las variables en Python se crean (*definen*) con el nombre que escojamos para nuestra variable, seguido de un igual y el dato que almacena (se *inicializan*):

```
numero = 10
```

Podemos *definir* la variable sin inicializarla con el nombre que escojamos para nuestra variable, seguido de dos puntos y la palabra reservada que indica el tipo de valor que va a contener:

```
numero: int
```

Para nombrar nuestras variables podemos usar cualquier palabra que comience por una letra (A-Z, a-z) o _ (no pueden empezar por números) y que no constituya una palabra reservada, tampoco puede contener espacios.

En Python no hace falta indicar el tipo de dato que va a contener la variable, por eso se dice que es débilmente tipado.

Si intentamos ver el valor de una variable no inicializada obtendremos un error:

```
numero : int
print(numero)
```

NameError: name 'numero' is not defined

Indicando que la variable no está inicializada.

variables

En Python se utiliza la notación Snake Case para nombrar las variables. En notación Snake Case el nombre de la variable se escribe en minúsculas.

Si se quiere usar como nombre variable “carrito compra”, como no se pueden usar espacios, la notación Snake Case combina las palabras usando el símbolo _ *underscore*: “carrito_compra”.

Otro ejemplo: “cliente tienda hotel” , en notación Snake Case: “cliente_tienda_hotel”.

Otras notaciones: “alumno ciclo”:

Camel Case: alumnoCiclo **Pascal Case:** AlumnoCiclo
Snake Case: alumno_ciclo **Kebab Case:** alumno-ciclo

constantes

En ocasiones, dentro de nuestro programa necesitamos almacenar valores que no van a cambiar, permanecen inmutables durante toda la ejecución del mismo. Estas zonas de memoria reciben el nombre de constantes.

En Python, sin embargo, no existen las constantes.

Para este caso, se utilizan variables con la convención de poner el nombre de la misma con letras mayúsculas y así identificarla como una constante.

```
IVA = 16
print(IVA)

# Esto esta permitido
# Pero no debería hacerse
IVA = 21
```

variables de cadenas de caracteres

En Python, las variables de cadenas de caracteres son secuencias ordenadas de caracteres, a los que podemos acceder de manera individual a través de un índice numérico o un valor clave. Este proceso se denomina indexación.

Se puede acceder a los caracteres individuales de una cadena especificando el nombre de la variable que contiene la cadena seguido de un número entre corchetes [ ].

El primer carácter de la cadena tiene el índice 0, el siguiente tiene el índice 1, y así sucesivamente.

El índice del último carácter será la longitud de la cadena menos uno.

```
nombre = “Juan”
print(nombre[0])
```

```
J
```

variables de cadenas de caracteres

También podemos utilizar números negativos para extraer caracteres por el final de la cadena de texto.

En este caso, el número -1 representa el último carácter de la cadena y va disminuyendo para ir indicando los caracteres más a la izquierda, los del principio.

```
nombre = "Juan Garcia"
print(nombre[-1])

print(nombre[-6])
```

```
a

G
```

Un *string* es un tipo de dato que Python considera inmutable, esto quiere decir que no podemos modificar una parte de la cadena de datos asociada a una variable.

```
# Esto da error
nombre[0]="p"
```

variables de cadenas de caracteres

A pesar de que no pueda modificarse el contenido de un string, si puede asignarse una nueva cadena de datos a la variable.

```
# Esto no da error
nombre = “Ana Perez”
```

Podemos introducir secuencias de escape (`\t \n \a \v \\`) dentro de las cadenas. Podemos usar `\n` para definir una cadena de datos que ocupe múltiples líneas.

```
nombre = “En un Lugar\n de\n la Mancha”
print(nombre)
```

```
En un Lugar
 de
 La Mancha
```

variables de cadenas de caracteres

A pesar de que no pueda modificarse el contenido de un *string*, se puede asignarse una nueva cadena de datos a la variable.

```
# Esto no da error
nombre = “Ana Perez”
```

Podemos introducir secuencias de escape (`\t \n \\`) dentro de las cadenas. Podemos usar `\n` para definir una cadena de datos que ocupe múltiples líneas. No hace falta usar `\\` para poner `\` se puede crear una cadena cruda (*raw*) anteponiendo la letra `r` o R delante.

```
nombre = “En un Lugar\n de\n la Mancha”
print(nombre)
```

```
En un Lugar
 de
 La Mancha
```

```
nombre = r"En un Lugar\ la Mancha“
```

```
En un Lugar\ la Mancha
```

variables de cadenas de caracteres

También podemos usar cadenas de caracteres multilínea colocando triple comilla doble (""") al principio y al final de la misma.

```
nombre = """En un lugar
 de
 la Mancha"""
print(nombre)
```

```
En un lugar
 de
 la Mancha
```

Podemos usar la función `len()`, de la biblioteca estándar, para conocer la longitud de una cadena de caracteres.

```
nombre = “Miguel de Cervantes”
print(len(nombre))
```

```
19
```

slicing

Python también permite una sintaxis específica de indexación que extrae subcadenas de una cadena de texto, a esto se denomina *slicing*.

La sintaxis que se utiliza para extraer una subcadena de una cadena *s* es de la forma *s[m:n]*, esto devuelve la porción de *s* que comienza en la posición *m*, y termina en la posición *n*, sin incluir el último carácter.

```
nombre = “Juan Garcia”
print(nombre[0:5])

print(nombre[-6:-1])
```

```
Juan

Garci
```

slicing

Para recuperar el último carácter de una cadena *s* mediante *slicing* *no se indica la última posición* `s[m:]`, esto devuelve la porción de *s* que comienza en la posición *m*, hasta el final de la misma.

```
nombre = "Juan Garcia"

print(nombre[-6:])
```

```
Garcia
```

También podemos dejar la primera posición vacía `[:n]`, esto devuelve la porción de *s* que comienza en la primera posición hasta la posición *n*, sin incluir este último.

```
print(nombre[:5])

print(nombre[:])
```

```
Juan

Juan Garcia
```

stride

Stride es otra variante del *slicing,* se realiza añadiendo al final otros dos puntos **:** adicionales y un tercer índice.

Este índice designa cuantos caracteres debe salta entre carácter y carácter para formar la subcadena.

```
numero= “123456789”

print(numero[::2])

print(numero[0:5:2])

print(numero[-9:-1:3])

print(numero[:-1:3])
```

```
13579

135

147

147
```

variables numéricas

En Python, como vimos en la parte de los literales, existen tres tipos de datos numéricos: enteros (`int`), números de punto flotante (`float`) y números complejos (`complex`).

En principio, los números enteros son aquellos que no tienen parte decimal. En Python los números enteros se referencian con la palabra `int`.

```
# Numero entero
numero=123
print(numero)

# Esto no es un numero
# Es una cadena de caracteres
numero2="123"
print(numero2)
```

```
123

123
```

variables numéricas

En Python existe la función `type()`, de la biblioteca estándar que podemos usar para conocer el tipo de dato que almacena una variable.

```
# Numero entero
numero=123
print(type(numero))

# Esto no es un numero
# Es una cadena de caracteres
numero2="123"
print(type(numero2))
```

```
int


str
```

variables numéricas

También podemos usar la función `int()`, de la biblioteca estándar, para convertir una cadena de caracteres o un numero en coma flotante en número en entero.

```
numero2="123"
print(type(int(numero2)))
print(int(numero2))
```

```
int
123
```

Los números de punto flotante son aquellos que tienen una parte decimal. En Python los números de punto flotante se referencian con la palabra `float`.

```
numero3: float = 0.33
print(type(numero3))
```

```
float
```

variables numéricas

Podemos usar la función `float()`, de la biblioteca estándar, para convertir una cadena de caracteres o un numero en coma flotante en número en entero.

```
numero2="123"
print(type(float(numero2)))
```

```
<class 'float'>
```

Podemos usar la función `str()`, de la biblioteca estándar, para convertir un número entero o en como flotante a una cadena de caracteres.

```
numero = 15
numero2 = 2.3

print(str(numero))
print(str(numero2))
```

variables numéricas

Python es uno de los pocos lenguajes de programación que ofrece soporte integrado para números complejos. En Python los números de punto flotante se referencian con la palabra `complex`.

Un número complejo está formado por dos componentes distintos: una parte real y una parte imaginaria. Para definir un número complejo en Python, se define la parte real seguida de un más + (o un menos `-`) y la parte imaginaria terminando con la letra `j`.

Para acceder a la parte real podemos usar el nombre de la variable seguido de un punto `.` y la palabra `real`.

Y para la parte imaginaria podemos usar el nombre de la variable seguido de un punto `.` y la palabra `imag`.

```
numero_complejo = 1+2j

print(numero_complejo.real)
print(numero_complejo.imag)
```

```
1.0
2.0
```

funciones

Una función es una sección de un programa o una parte de código que calcula un valor o realiza alguna tarea de manera independiente al resto del programa.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)
```

Una función tiene tres componentes importantes:

- Los parámetros, que son los valores que recibe la función como entrada.
- El código de la función, que son las operaciones que hace la función.
- El resultado (o valor de retorno), que es el valor final que entrega la función.

En esencia, una función es un mini programa.

funciones

La sintaxis utilizada para definir funciones en Python es la siguiente:

```
def <nombre_funcion>([<parámetros>]):
    <sentencia(s)>
```

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)
```

El último elemento, `<sentencia(s)>`, se denomina cuerpo de la función.

El cuerpo es el bloque de sentencias en Python que se ejecutará cuando se llame a la función.

El cuerpo de una función de Python se define por la sangría.

funciones

La sintaxis utilizada para invocar una función y que se ejecute su código es la siguiente:

```
<nombre_funcion>([<argumentos>])
```

Los <argumentos> son los valores que se pasan a la función.

Se corresponden con los <parámetros> en la definición de la función en Python.

Se pueden definir funciones que no reciban ningún argumento, pero los paréntesis siguen siendo necesarios. Tanto una definición de función como una llamada a una función deben incluir siempre paréntesis, incluso si están vacíos.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)


mi_funcion("Hola", "Juan")
```

```
def funcion_saludo():
    print("Hola!!")

funcion_saludo()
```

funciones

La forma más sencilla de pasar argumentos a una función en Python es con argumentos posicionales (también llamados argumentos requeridos).

En la definición de la función, debe especificarse una lista de parámetros separada por comas dentro de los paréntesis.

El número de argumentos que le proporcionamos en la invocación de la función debe respetar el orden y el número de parámetros definidos.

Otra forma de invocar una función en Python es indicando los argumentos de la forma `<palabra clave>=<valor>`. En ese caso, cada `<palabra clave>` debe coincidir con un parámetro en la definición de la función.

```
def mi_funcion(arg1, arg2):
    print(arg1)
    print(arg2)


mi_funcion("Hola", "Juan")
```

```
mi_funcion(arg2="Hola", arg1="Juan")
```

funciones

Utilizar argumentos de palabra clave nos exime de tener que respetar el orden de los parámetros definidos en la función, pero el número de argumentos debe seguir respetándose.

Podemos combinar argumentos posicionales y de palabra clave en la misma llamada a una función. En estos casos, los argumentos posicionales deben indicarse primero.

```
def mi_funcion(arg1, arg2, arg3):
    print(arg1)
    print(arg2)
    print(arg3)


mi_funcion("Hola", arg3="Garcia", arg2="Juan")
```

funciones

Podemos especificar valores por defecto en los parámetros de la definición de una función en Python.

Para ello, en la definición utilizamos la forma `<nombre>=<valor>`, si no es aportado en como argumento entonces `<valor>` se convierte en un valor por defecto para ese parámetro.

Los parámetros definidos de esta manera se denominan parámetros por defecto u opcionales.

```
def mi_funcion(arg1, arg2="Pedro", arg3=""):
    print(arg1)
    print(arg2)
    print(arg3)


mi_funcion("Hola", "Juan")
```

funciones

Las funciones en Python pueden devolver un valor en función de la ejecución de sus instrucciones de código.

Para devolver un valor, se debe utilizar la palabra `return` dentro del cuerpo de la función.

Cuando se ejecuta esta sentencia en Python, el intérprete termina inmediatamente la ejecución de la función y regresa a la línea de código desde dónde la hubiésemos invocado.

```
def mi_funcion(arg1, arg2="Pedro", arg3=""):
    print(arg1)
    return
# lo que sigue jamás se va a ejecutar
    print(arg2)
    print(arg3)
```

funciones

Si indicamos junto con la palabra `return` un valor, dicho valor es devuelto, cuando acabe de ejecutarse el código de la función y llegué a la palabra `return`, en la misma línea en la que se produce la llamada a la función.

Si no se indica ningún valor junto la palabra `return` se devuelve la palabra reservada None.

```
def mi_funcion():
    return “Hola Juan”


mensaje=mi_función()
print(mensaje)
```

```
Hola Juan
```

funciones

Una función puede devolver varios valores a la vez especificándolos a continuación del `return` y separándolos por comas.

Para recogerlos podemos usar variables separadas por comas.

```
def mi_funcion():
    return “Hola”, “Juan”, “Adios”


mensaje1, mensaje2, mensaje3=mi_función()
print(mensaje1)
```

```
Hola
```

funciones

Cuando la primera declaración en el cuerpo de una función en Python es un comentario de la forma cadenas de caracteres multilínea de triple comilla doble (""""), se le conoce como *docstring* de la función.

Los *docstring* se utilizan para proporcionar la documentación de una función. Puede contener el propósito de la función, los argumentos que toma, información sobre los valores de retorno, o cualquier otra información que se considere útil.

Existe un PEP (*Python Enhancement Proporsal*) que establece el estilo con el que deben definirse los *docstrings*: https://www.python.org/dev/peps/pep-0257/

```
def mi_funcion():
    """Esto es un docstring de la
función."""
    return “Hola”, “Juan”, “Adios”
```

ámbito de las variables

El ámbito de una variable (*scope*) es la zona del programa en la que se define la variable.

Las variables en Python son locales por defecto. Por lo que las variables definidas y utilizadas en el bloque de código de una función, sólo tienen existencia dentro de la misma, y no interfieren con otras variables del resto del código.

En caso de que sea conveniente o necesario, una variable local puede convertirse en una variable global declarándola explícitamente como tal con la sentencia `global`.

```
# definimos una funcion llamada local
def local():
 m = 7
 print(m)

m = 5
print(m)

# llamanos a funcion local
local()
```

```
global numero
```

built-in functions

El intérprete de Python tiene un conjunto de funciones útiles que están siempre disponibles y que forman parte de la librería "estándar" de Python (*Built-in functions*).

Están disponibles a través del siguiente enlace a la documentación de Python:

https://docs.python.org/3/library/functions.html

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

built-in functions

La función `help()` es muy útil porque da acceso a la documentación de Python.

Por si sola muestra la ayuda, pero se le puede pasar algo determinado como argumento para que busque y muestre la ayuda referente a ese tema concreto.

Otras funciones Built-in que ya hemos visto son: `len()`, `float()`, `int()`, `str()`, `type()`.

```
help(float)
```

```
Help on class float in module builtins:

class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
```

operadores aritméticos

Un operador es un elemento de programa que se aplica a uno o varios operandos en una expresión o instrucción, El operador genera un resultado.

+	Suma	num1 + num2
-	Resta	num1 - num2
-	Menos unario	num1 = -num1
*	Multiplicación	num1 * num2
/	División	num1 / num2
%	Módulo (resto de la división)	num1 % num2
**	potencia de los operandos	num1 ** num2
//	división con resultado de número entero	num1 // num2
++	Post incremento	num1++
++	Pre incremento	++num1
--	Post decremento	num1--
--	Pre decremento	--num1

operadores de asignación

=	asignación	num1 = num2
+=	Suma y asignación	num1 += num2
-=	resta y asignación	num1 -= num2
*=	multiplicación y asignación	num1 *= num2
/=	división y asignación	num1 /= num2
%=	modulo y asignación	num1 %= num2
**=	potencia y asignación	num1 **= num2
//=	División entera y asignación	num1 //= num2

operadores de comparación

==	Igual a	dato1 == dato2
!=	Distinto de	dato1 != dato2
>	Mayor que	dato1 > dato2
>=	Mayor o igual a	dato1 >= dato2
<	Menor que	dato1 < dato2
<=	Menor o igual a	dato1 <= dato2
is	si ambos operandos hacen referencia al mismo objeto	dato1 is dato2
is not	si ambos operandos hacen referencia al mismo objeto	dato1 is not dato2

operadores lógicos

and	Y lógico	dato1 and dato2
or	O lógico	dato1 or dato2
not	Negación lógica	not dato1

operadores binarios

&	Y binario	variablebyte & 1
|	O binario	variablebyte | 2
~	Negación binaria	~ variablebyte
^	XOR	variablebyte ^ 3
<<	Desplazamiento de bits a la izquierda	variablebyte << 1
>>	Desplazamiento de bits a la derecha	Variablebyte >> 2
&=	Y binario y asignación	num1 &= num2
|=	O binario y asignación	num1 |= num2
^=	XOR binario y asignación	num1 ^= num2
<<=	Desplazamiento a la izquierda y asignación	num1 >>= num2
>>=	Desplazamiento a la derecha y asignación	num1 <<= num2

Bibliografía y recursos online

- https://www.mclibre.org/consultar/python/index.html
- https://docs.python.org/es/3/reference
- https://www.python.org/dev/peps/pep-0257/
- https://docs.python.org/3/library/functions.html