Curiosidades de Python

Python permite el uso de enteros "infinitos"

Bueno, técnicamente no son infinitos, pero Python tiene un tipo de dato `int` que no tiene un límite fijo en su tamaño, a diferencia de otros lenguajes de programación donde los enteros tienen un tamaño máximo debido a limitaciones del sistema (como 32 o 64 bits).

Por ejemplo, puedes hacer operaciones con números increíblemente grandes sin preocuparte por desbordamientos:

```
>>> print (3457298746928739485792364576237469582763874659283746958726387465236456928345 45 *  109274183794871023401387 40\1923847)
377793498705668359705382171583060048808861054121781943267612710968458139906668757415471716408678429556608946 15
```

Python puede manejar "signos negativos" en exponentes

En Python, los exponentes pueden ser negativos, lo que significa que puedes realizar cálculos de números fraccionarios de manera sencilla. Python maneja esto internamente como una fracción.

Ejemplo:

```
print(2 ** -3)  # Imprime: 0.125
```

Python puede manejar "numeros fraccionarios" en exponentes

Python puede manejar números fraccionarios como exponentes, sin problema.

- **números con decimales** (`0.5`, `1/3`, etc.)
- como **fracciones exactas** usando el módulo `fractions`.

```
print(9 ** 0.5)   # raíz cuadrada de 9 → 3.0
print(8 ** (1/3)) # raíz cúbica de 8 → 2.0

from fractions import Fraction
print(16 ** Fraction(1, 4))  # raíz cuarta de 16 → 2.0
print(27 ** Fraction(2, 3))  # (raíz cúbica de 27)^2 → 9.0
```

Python tiene soporte para "tipos de datos complejos"

En Python, puedes trabajar con números complejos usando el tipo de datos `complex`. Un número complejo se representa como `a + bj`, donde `a` es la parte real y `b` es la parte imaginaria.

Ejemplo:

```
z = 3 + 4j
print(z.real)  # Imprime: 3.0
print(z.imag)  # Imprime: 4.0
```

Python tiene soporte para "numeros infinitos"

En Python, puedes trabajar con **valores infinitos** utilizando `float('inf')` para **infinito positivo** y `float('-inf')` para **infinito negativo** o utilizando `import math`

```
print(math.isinf(float('inf')))  # True
print(math.isinf(42))            #  False
print(math.inf)            #  inf
```

Python tiene una forma de comparar "números de punto flotante" con una tolerancia

Comparar números de punto flotante en Python puede ser problemático debido a pequeñas imprecisiones. Para evitar errores, puedes usar la función `math.isclose()` para comparar dos números con una tolerancia.

Ejemplo:

```
import math
x = 0.1 + 0.2
y = 0.3
print(math.isclose(x, y))  # Imprime: True
```

Python tiene "un operador de fusión de diccionarios" desde Python 3.9

A partir de Python 3.9, puedes fusionar dos diccionarios utilizando el operador `|`. Esto hace que la fusión de diccionarios sea más sencilla y directa.

Ejemplo:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
fusionado = dict1 | dict2
print(fusionado)  # Imprime: {'a': 1, 'b': 3, 'c': 4}

dict1 |= dict2   ## dict1 = dict1 | dict2
```

Python puede ejecutar instrucciones directamente desde la línea de comandos utilizando el operador -c

Esto significa que puedes escribir y ejecutar código Python en una sola línea sin necesidad de guardar un archivo.
Por ejemplo, si quieres calcular la suma de dos números desde la terminal, puedes escribir algo como esto:

```
python -c "print('Hola, mundo')"
```

Python tiene una función llamada `globals()` y `locals()`

`globals()` devuelve un diccionario con todas las variables globales, mientras que `locals()` devuelve un diccionario con las variables locales. Esto es útil cuando trabajas con metaprogramación.

Ejemplo:

```
x = 10
def mi_funcion():
    y = 5
    print(locals())  # Imprime: {'y': 5}
    print(globals())  # Imprime todas las variables globales
mi_funcion()
```

Variable _ en los intérpretes interactivos

En una sesión interactiva de Python, _ se utiliza para almacenar el último resultado calculado
Guiones bajos en nombres: Los guiones bajos también se usan en los nombres de variables o funciones para indicar intenciones especiales:

- `_nombre`: Para indicar que algo es "privado" (aunque no se aplica estrictamente).
- `__nombre`: Para evitar conflictos de nombres en clases (name mangling).
- `__nombre__`: Estas son "funciones mágicas" o métodos especiales como `__init__` o `__str__`.

Python tiene una manera secreta de evaluar expresiones

Si escribes esto en Python:

`True + True + True`
El resultado será `3`.

¿Por qué? Porque en Python, `True` se comporta como `1` y `False` como `0`. Así que si sumas `True + True + True`, en realidad estás haciendo `1 + 1 + 1`, lo que da `3`.

Incluso puedes hacer cosas como:

```
False * 10  # Resultado: 0
True * 10   # Resultado: 10
```

¡Es un truco matemático curioso que puede ser útil en algunos casos!

Python: tiene algo llamado *"Duck Typing"*

Este concepto proviene de la frase "Si parece un pato, nada como un pato y suena como un pato, probablemente sea un pato". En Python, esto significa que el tipo de un objeto no es tan importante como los métodos o atributos que posee. Por ejemplo, mientras un objeto pueda comportarse como una lista (aunque no sea exactamente una lista), Python lo tratará como tal.

Esto es lo que hace que Python sea tan flexible y fácil de usar, ya que permite a los desarrolladores centrarse más en la funcionalidad del código y menos en los detalles técnicos

```
class Pato:
    def quack(self):
        return "Cuack, cuack"

class Persona:
    def quack(self):
        return "Estoy imitando un pato: Cuack, cuack"

def hacer_quack(objeto):
    print(objeto.quack())

# Creamos instancias
pato = Pato()
persona = Persona()
```

```
# Ambas instancias pueden "quackear" porque tienen un método `quack`
hacer_quack(pato)    # Salida: Cuack, cuack
hacer_quack(persona) # Salida: Estoy imitando un pato: Cuack, cuack
```

En este ejemplo, tanto la clase `Pato` como la clase `Persona` tienen un método `quack()`. Python no verifica el tipo del objeto que pasa a la función `hacer_quack`; simplemente asume que si el objeto tiene un método `quack`, todo funcionará correctamente. ¡Eso es *Duck Typing*!

En Python, puedes usar caracteres UTF-8 en nombres de variables

lo que significa que puedes usar letras acentuadas y caracteres de otros alfabetos.

En concreto, caracteres Unicode de la categoria Letter.

Por ejemplo:

```
π = 3.1416
nombre_éxito = "¡Sí se puede!"

print(π)      # 3.1416
print(nombre_éxito)  # "¡Sí se puede!"
```

Python permite esto porque usa Unicode (no todos son válidos) para los identificadores, aunque no es recomendable en código profesional por temas de legibilidad y compatibilidad.

Categorías Unicode de "Letter" permitidas en Python

Código	Nombre	Ejemplo de caracteres
`Lu`	**Uppercase Letter**	A, B, C, Α, Б, 字
`Ll`	**Lowercase Letter**	a, b, c, α, б, 字
`Lt`	**Titlecase Letter**	Dž (Dz), Lj (Lj)
`Lm`	**Modifier Letter**	ʰ (ʰH), ˡ (ˡL)
`Lo`	**Other Letter**	漢, 字, あ, ك, ज

✅ **Todos estos caracteres se pueden usar al inicio de una variable en Python.**
✅ Después del primer carácter, también se pueden usar números (`Nd` - "**Number, Decimal Digit**").

Aunque Python permite caracteres Unicode en nombres de variables, **no todos los caracteres son válidos**, y los emojis están entre los que no se pueden usar.

Lo que sí puedes usar:

- Letras de alfabetos internacionales (π, 变量, `nombre_éxito`)
- Caracteres con acentos (`año`, `mañana`)
- Letras griegas (α, β, γ)

Lo que no puedes usar:

- Emojis (🚀, 🔥, 🐍)
- Símbolos matemáticos (`+`, `-`, `/`, `*`)
- Números al inicio (`2variable` ❌, pero `variable2` ✅)

Si realmente quieres usar un emoji, puedes hacerlo como clave en un diccionario:

```
datos = {"🚀": "Despegando..."}
print(datos["🚀"])  # "Despegando..."
```

Cómo verificar la categoría de un carácter Unicode

```
import unicodedata
caracter = "字"
categoria = unicodedata.category(caracter)
print(categoria)  # Salida: Lo (Other Letter)
```

Python tiene un operador de "walrus" (`:=`) que te permite asignar valores dentro de expresiones

El **operador walrus** (`:=`) fue introducido en Python 3.8 y permite asignar un valor a una variable dentro de una expresión. Esto te permite hacer asignaciones y evaluaciones de manera más compacta y eficiente en ciertas situaciones.

¿Cómo funciona?
El operador `:=` permite que una **variable** se **asigne** dentro de una **expresión**, lo cual normalmente no sería posible sin un bloque separado.

Ejemplo de uso básico:

```
# Sin el walrus
n = 10
if n > 5:
    print("El número es mayor que 5")

# Con el walrus
if (n := 10) > 5:
    print("El número es mayor que 5")
```

En el ejemplo anterior, `n := 10` asigna el valor `10` a la variable `n` y luego evalúa si es mayor que `5`. Lo hace todo en una sola línea.

Ejemplo práctico: Leer datos mientras se procesa una lista
Supón que quieres leer valores desde una lista hasta que encuentres un valor que no cumpla una condición. Con el operador `walrus` puedes hacerlo de manera eficiente.

```
python
CopiaModifica
# Leer números hasta que encontramos un número negativo
while (num := int(input("Introduce un número: "))) >= 0:
    print(f"Número ingresado: {num}")
```

Los archivos `.pyc` mejoran el rendimiento de Python

Cuando ejecutas un programa en Python, el **código fuente** se compila en un archivo bytecode (`.pyc`) que Python puede ejecutar más rápidamente. Estos archivos `.pyc` se almacenan en un directorio llamado `__pycache__` y permiten a Python iniciar más rápido la próxima vez que ejecutes tu código.

Python tiene un "modo de depuración" muy útil

Si algo no está funcionando en tu código, puedes usar el módulo `pdb` (Python Debugger) para realizar una depuración interactiva. Puedes insertar puntos de interrupción en tu código y examinar el estado del programa paso a paso.

Ejemplo:

```
import pdb

def sumar(a, b):
    pdb.set_trace()  # Esto detiene la ejecución para poder depurar
    return a + b

print(sumar(3, 5))
```

Al ejecutar este código, se pausará en `pdb.set_trace()` y podrás inspeccionar las variables y pasos del programa.

Python tiene un "modo silencioso" para desactivar las advertencias

Puedes desactivar las advertencias en Python usando el módulo `warnings`. Esto puede ser útil cuando quieres ejecutar código sin que aparezcan advertencias.

Ejemplo:

```
import warnings
warnings.filterwarnings("ignore")  # Desactiva las advertencias
```

Python tiene un "importador de módulos" dinámico

Python te permite cargar módulos de manera dinámica usando la función `__import__()`. Esto puede ser útil si deseas cargar un módulo de manera condicional o durante la ejecución de un programa sin saber exactamente qué módulos necesitarás de antemano.

Ejemplo:

```
module = __import__("math")
print(module.sqrt(16))  # Usa el módulo "math" para calcular la raíz cuadrada
```

Python tiene un límite de recursión (recursividad)

Python tiene un **límite de recursión** predeterminado que evita que el programa se quede atrapado en una recursión infinita. Por defecto, el límite es 1000, lo que significa que si una función recursiva llama a sí misma más de 1000 veces, Python lanzará un error `RecursionError`.

Puedes modificar este límite utilizando `sys.setrecursionlimit()`, pero siempre ten cuidado al hacerlo.

```
import sys
sys.setrecursionlimit(2000)
```

Python tiene una función llamada `help()` que te ofrece documentación interactiva

Python tiene una función interna muy útil llamada `help()`. Puedes usarla para obtener información sobre funciones, módulos o clases sin necesidad de buscar documentación externa. Solo tienes que escribir `help()` y pasarle el nombre del objeto sobre el que quieras obtener más información.

Ejemplo:

```
help(print)
```

Python tiene un "modo de documentación" integrado

Los **docstrings** son una característica especial en Python que te permiten documentar funciones, clases y módulos directamente en el código. Puedes acceder a los docstrings utilizando la función `help()` para obtener información sobre cualquier objeto o función.

Ejemplo:

```
def saludar():
    """Esta función saluda al usuario."""
    print("¡Hola!")

help(saludar)
```

Help on function saludar in module __main__:

saludar()
    Esta función saluda al usuario.

Python tiene un operador "ternario" o "condicional"

Puedes usar un **operador ternario** en Python para realizar asignaciones basadas en condiciones de una forma más compacta. La sintaxis es:

```
python
CopiaModifica
x if condición else y
```

Esto evalúa la **condición** y, si es `True`, devuelve `x`, y si es `False`, devuelve `y`.

Ejemplo:

```
python
CopiaModifica
edad = 18
resultado = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(resultado)  # Imprime: Mayor de edad
```

Las funciones pueden devolver múltiples valores en Python

A diferencia de otros lenguajes, Python permite que una función devuelva **más de un valor** a la vez, utilizando tuplas.

Ejemplo:

```
def suma_y_producto(a, b):
    return a + b, a * b

resultado = suma_y_producto(3, 4)
print(resultado)  # Imprime: (7, 12)
```

Python permite la "herencia múltiple"

En Python, puedes **heredar** de múltiples clases, lo que te permite combinar comportamientos de diferentes clases en una sola clase.

Ejemplo:

```
python
CopiaModifica
class A:
    def metodo_a(self):
        print("Método A")

class B:
    def metodo_b(self):
        print("Método B")

class C(A, B):
    pass

c = C()
c.metodo_a()  # Imprime: Método A
c.metodo_b()  # Imprime: Método B
```

Python permite crear "clases abstractas"

En Python, puedes usar el módulo `abc` para definir clases abstractas. Estas clases no pueden instanciarse directamente y sirven como plantillas para otras clases. Esto es útil cuando quieres asegurarte de que las clases derivadas implementen ciertos métodos.

Ejemplo:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        print("Guau")

perro = Perro()
perro.hablar()  # Imprime: Guau
```

Python tiene una función `dir()` para inspeccionar objetos

La función `dir()` te permite obtener una lista de los atributos y métodos de un objeto. Esto es útil para explorar las propiedades de las bibliotecas y objetos en Python.

Ejemplo:

```
lista = [1, 2, 3]
print(dir(lista))  # Muestra todos los métodos y atributos de la lista
```

Python tiene una guia de estilo para escribir código. PEP8

PEP 8 es una guía de estilo para escribir código en Python. Su objetivo es mejorar la legibilidad y la coherencia del código en la comunidad de Python.

Algunos de los aspectos más importantes de PEP 8 incluyen:
✅ **Indentación**: Usar 4 espacios por nivel de indentación (NO tabulaciones).
✅ **Longitud de línea**: Máximo 79 caracteres por línea.
✅ **Espacios en blanco**: Evitar espacios innecesarios en expresiones y estructuras.
✅ **Nombres de variables y funciones**: Usar `snake_case` (ejemplo: `mi_variable`).
✅ **Clases**: Usar `PascalCase` (ejemplo: `MiClase`).
✅ **Importaciones**: Organizar en bloques (módulos estándar, módulos de terceros, módulos propios).
✅ **Comentarios**: Ser claros y concisos.

Seguir PEP 8 ayuda a que el código sea más limpio y fácil de entender. Puedes usar herramientas como **`flake8`** o **`black`** para verificar y formatear tu código automáticamente según estas reglas

Python permite usar "pases de parámetro por referencia" para listas y diccionarios (conjuntos y otros datos "mutables")

En Python, cuando pasas una lista o un diccionario a una función, **no se copia** el objeto, sino que se pasa una **referencia** al objeto. Esto significa que si modificas la lista o diccionario dentro de la función, esos cambios afectarán al objeto original.

Ejemplo:

```
def modificar_lista(lista):
    lista.append(4)

mi_lista = [1, 2, 3]
modificar_lista(mi_lista)
print(mi_lista)  # Imprime: [1, 2, 3, 4]
```

En cambio, si pasas tipos inmutables (como `int`, `float`, o `str`), **no** se modifican dentro de la función, ya que estos valores se copian al pasarlos.

El "GIL" de Python limita el rendimiento en entornos multihilo

Python tiene un concepto llamado **Global Interpreter Lock (GIL)**, que significa que, aunque Python permite la creación de varios hilos en el programa, solo un hilo puede ejecutarse a la vez en un proceso dado. Esto puede afectar el rendimiento en programas multihilo, aunque para tareas que no son de alto rendimiento, el GIL no suele ser un problema.

Origen del nombre del lenguaje Python

Una curiosidad interesante sobre Python es el origen de su nombre. Mucha gente asume que está relacionado con la serpiente pitón, pero en realidad, su creador, Guido van Rossum, lo nombró en honor al grupo cómico británico "Monty Python". Guido era fanático de su humor absurdo y quería un nombre que fuera único, corto y un poco peculiar, como el propio lenguaje.
De hecho, muchas referencias en la documentación oficial y ejemplos de código de Python hacen guiños al grupo "Monty Python". Es un lenguaje de programación con una buena dosis de humor en su ADN.

El Zen de Python

Python tiene una estructura peculiarmente humorística conocida como **"PEP 20: The Zen of Python"**, pero también existen muchos más **PEPs (Python Enhancement Proposals)**. Algunos de ellos son súper técnicos, pero otros contienen bromas internas de la comunidad.

Por ejemplo, el **PEP 404**... ¡No existe! Fue creado como un guiño a los errores 404 ("Página no encontrada"). Oficialmente, no hubo una versión Python 2.8 porque Python 2 fue descontinuado directamente en favor de Python 3, así que los desarrolladores decidieron que el PEP 404 simbolizara esta decisión con un toque cómico.

¿Sabías que Python tiene un módulo llamado `__future__`? Parece un poco misterioso, pero en realidad su propósito es preparar el lenguaje para futuras versiones. Al usarlo, puedes activar características experimentales o de versiones más modernas en tu código actual. Es como viajar en el tiempo para probar el futuro de Python antes que nadie.

Creador de Python

Python fue creado por **Guido van Rossum**, un programador neerlandés. Él comenzó a desarrollar Python a finales de los años 80 y lanzó su primera versión oficial en 1991.

Actualmente, Python es mantenido por una comunidad de desarrolladores bajo la supervisión de la **Python Software Foundation (PSF)**, una organización sin fines de lucro que promueve y gestiona el desarrollo del lenguaje y su ecosistema.

El compilador de Python está escrito en Python.

El intérprete CPython, que es la implementación principal del lenguaje, está en su mayor parte escrito utilizando el propio Python (aunque también incluye partes en C).
Esto significa que el lenguaje puede definirse y mejorarse con sus propias herramientas, lo que contribuye a su flexibilidad y a la facilidad de desarrollo. ¡Es como si Python fuera lo suficientemente maduro como para cuidarse a sí mismo!

Por si fuera poco, también existen otras implementaciones interesantes del intérprete, como:

- **Jython**: Una implementación de Python escrita en Java.
- **IronPython**: Python en la plataforma .NET.
- **PyPy**: Una versión rápida de Python escrita... ¡en Python también!

Una curiosidad fascinante de Python es que su filosofía está plasmada en un documento llamado *The Zen of Python*. Si escribes `import this` en cualquier consola de Python, te aparecerá un poema que enumera 19 principios que guían el diseño del lenguaje. Algunos de ellos incluyen:

- "Simple es mejor que complejo."
- "La legibilidad cuenta."
- "Aunque es tentador ignorar las reglas, a menudo es mejor seguirlas."

Estos principios reflejan la importancia de la claridad y la simplicidad en Python, lo cual lo hace tan popular entre los programadores

Python tiene una biblioteca llamada `antigravity`

Python tiene una **biblioteca oculta llamada `antigravity`**, que es un **homenaje** a un cómic de XKCD. Si escribes el siguiente código en tu consola de Python:

```
import antigravity
```

¡Verás que se abre una página web con un cómic que hace referencia a cómo Python puede ser algo “ligero” como la gravedad! Es solo una broma, pero ¡es una curiosidad divertida

Puedes usar Python para crear interfaces gráficas con Tkinter

Aunque Python es conocido por su facilidad en programación de scripts, también puedes crear aplicaciones con **interfaz gráfica** (GUI) utilizando **Tkinter**, que es una de las bibliotecas estándar de Python. ¡Sí! Puedes crear ventanas, botones y otros elementos gráficos de forma muy sencilla.

Ejemplo básico con Tkinter:

```
import tkinter as tk

ventana = tk.Tk()
ventana.title("Mi primera GUI en Python")

etiqueta = tk.Label(ventana, text="¡Hola, Mundo!")
etiqueta.pack()
```

```
ventana.mainloop()  # Mantiene la ventana abierta
```

Aunque Python no está tan centrado en el desarrollo de interfaces gráficas como otros lenguajes, existen bibliotecas como **Tkinter**, **PyQt** y **Kivy** que te permiten crear aplicaciones con interfaces gráficas de usuario de manera relativamente sencilla.

Python puede hacer magia con generadores y "yield"

Python tiene un concepto poderoso llamado **generadores**, que permite generar secuencias de datos de forma perezosa (lazy evaluation). En lugar de almacenar todos los elementos en memoria, Python genera los elementos solo cuando se necesitan, lo que es muy eficiente en términos de memoria.

Ejemplo:

```
def contar_hasta_tres():
    yield 1
    yield 2
    yield 3

for numero in contar_hasta_tres():
    print(numero)  # Imprime 1, luego 2, luego 3
```

Python tiene un `with` para manejar recursos automáticamente

Python tiene una estructura llamada **`with`** que permite trabajar con recursos como archivos o conexiones de red de manera más eficiente y segura. El bloque `with` se asegura de que el recurso se libere (se cierre, por ejemplo) incluso si ocurre un error durante su uso.

Ejemplo de uso con archivos:

```
with open('archivo.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
# No es necesario cerrar el archivo manualmente; Python lo hace automáticamente.
```

Python tiene un módulo `random` para generar números aleatorios

Si necesitas generar números aleatorios o elegir elementos de una lista de forma aleatoria, Python tiene un módulo llamado **`random`**. Este módulo proporciona varias funciones para trabajar con probabilidades y aleatorización.

Ejemplo:

```
import random

numero_aleatorio = random.randint(1, 10)
print(numero_aleatorio)  # Imprime un número aleatorio entre 1 y 10
```

Python tiene una biblioteca para manejar expresiones regulares llamada `re`

El módulo **`re`** de Python permite trabajar con **expresiones regulares**, lo que facilita la búsqueda y manipulación de cadenas de texto. Con **`re`**, puedes hacer coincidencias complejas de patrones en textos, validar correos electrónicos, o incluso buscar palabras clave.

Ejemplo:

```
import re

texto = "Mi número es 123-456-7890"
patron = r'\d{3}-\d{3}-\d{4}'  # Coincide con un número de teléfono

resultado = re.search(patron, texto)
if resultado:
    print("Número encontrado:", resultado.group())
```

Python tiene una biblioteca llamada `os` que te permite interactuar con el sistema operativo

El módulo **`os`** en Python te permite interactuar con el sistema operativo y realizar tareas como manipular archivos, cambiar directorios y obtener información sobre el entorno del sistema.

Ejemplo:

```
import os
print(os.name)  # Imprime el nombre del sistema operativo (posiblemente 'posix'
o 'nt')
```

Python tiene un "módulo de configuración" llamado `configparser`

El módulo **`configparser`** te permite trabajar con archivos de configuración que suelen tener un formato de **clave-valor**, como los archivos `.ini`. Es útil cuando deseas gestionar configuraciones externas en tu aplicación.

Ejemplo:

```
import configparser

config = configparser.ConfigParser()
config.read('config.ini')

print(config['Sección1']['clave'])  # Accede al valor de una clave en una
sección
```

Python tiene una función llamada `enumerate()` que agrega índices a los elementos de una lista

Si necesitas acceder tanto al índice como al valor de los elementos en un bucle, **`enumerate()`** es la función perfecta. Esta función te devuelve tanto el índice como el valor de cada elemento durante la iteración.

Ejemplo:

```
lista = ['a', 'b', 'c']
for indice, valor in enumerate(lista):
    print(f'Índice: {indice}, Valor: {valor}')
```

Python tiene un sistema de "gestión de memoria automática"

Python maneja la **gestión de memoria automáticamente** a través de un sistema de **recolección de basura** (garbage collection). Esto significa que el lenguaje se encarga de liberar memoria no utilizada, lo que hace que la programación en Python sea más fácil y menos propensa a errores de memoria.

Python puede interactuar con bases de datos usando `sqlite3`

Python tiene soporte incorporado para interactuar con bases de datos SQL a través del módulo **`sqlite3`**, lo que te permite gestionar bases de datos sin necesidad de instalar bibliotecas adicionales.

Ejemplo:

```
import sqlite3

# Conexión a la base de datos (se creará si no existe)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Crear una tabla
cursor.execute('''CREATE TABLE IF NOT EXISTS usuarios (id INTEGER PRIMARY KEY,
nombre TEXT)''')

# Insertar un registro
cursor.execute('''INSERT INTO usuarios (nombre) VALUES ('Juan')''')

# Guardar cambios y cerrar
conn.commit()
conn.close()
```

Python tiene un "módulo `math`" con funciones matemáticas poderosas

Python tiene un módulo llamado **`math`** que incluye muchas funciones matemáticas avanzadas como **raíces cuadradas**, **funciones trigonométricas** y **constantes matemáticas**.

Ejemplo:

```
import math
print(math.sqrt(16))  # Imprime: 4.0
print(math.pi)  # Imprime: 3.141592653589793
```

Python tiene un "módulo `socket`" para trabajar con redes

Python tiene un módulo llamado **`socket`** que permite crear aplicaciones cliente-servidor, permitiendo la comunicación a través de la red, como la creación de servidores web o la conexión a otros servicios.

Ejemplo básico de un servidor:

```
import socket

servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servidor.bind(('localhost', 12345))
servidor.listen(1)

print("Esperando conexión...")
conexion, direccion = servidor.accept()
print(f"Conexión recibida de {direccion}")
conexion.send(b"¡Hola, cliente!")
conexion.close()
```

Python tiene un "módulo `uuid`" para generar identificadores únicos

El módulo **`uuid`** permite generar identificadores únicos universales (UUIDs), que son útiles cuando necesitas generar claves únicas en bases de datos, sesiones o para cualquier otra aplicación.

Ejemplo:

```
import uuid
id_unico = uuid.uuid4()
print(id_unico)  # Imprime: un identificador único como '2d40c8ff-36d5-4a27-
b63c-23ab7f6941db'
```

Python puede usarse para desarrollar juegos con Pygame

Pygame es una biblioteca que permite crear videojuegos sencillos en Python. Con Pygame, puedes crear gráficos, sonidos y gestionar eventos para hacer juegos interactivos.

Ejemplo básico de Pygame:

```
import pygame
pygame.init()
pantalla = pygame.display.set_mode((400, 300))
pygame.display.set_caption("¡Hola Pygame!")
while True:
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            pygame.quit()
            exit()
```

Python es usado en la automatización con herramientas como Selenium

Selenium es una herramienta que permite automatizar la interacción con navegadores web. Es ampliamente usada para hacer **pruebas automáticas** en sitios web, pero también es útil para tareas de automatización como rellenar formularios, hacer scraping de información, y más.

Ejemplo con Selenium:

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.python.org")
print(driver.title)  # Imprime el título de la página
driver.quit()
```

Python es el lenguaje más popular para análisis de texto y procesamiento del lenguaje natural (NLP)

Con bibliotecas como **NLTK** y **spaCy**, Python se ha convertido en el lenguaje principal para realizar tareas de procesamiento de lenguaje natural (NLP). Estas bibliotecas permiten realizar análisis de texto, extracción de información, y tareas complejas como traducción automática o análisis de sentimientos.

Ejemplo con NLTK:

```
import nltk
from nltk.tokenize import word_tokenize

texto = "¡Hola, soy un texto!"
palabras = word_tokenize(texto)
print(palabras)  # Imprime: ['¡', 'Hola', ',', 'soy', 'un', 'texto', '!']
```

Python tiene un "módulo `shutil`" para trabajar con archivos y directorios

El módulo **`shutil`** permite realizar operaciones de alto nivel con archivos y directorios, como mover, copiar y eliminar archivos. Es útil cuando necesitas hacer tareas de administración de archivos.

Ejemplo:

```
import shutil

# Copiar un archivo
shutil.copy('archivo.txt', 'copia_archivo.txt')
```

Python puede ser "extremadamente eficiente" para tareas específicas con "Cython"

Cython es una extensión de Python que permite escribir código Python más rápido utilizando una sintaxis similar a Python, pero con la capacidad de hacer llamadas directas a funciones escritas en C. Esto es útil para mejorar el rendimiento de ciertas operaciones que requieren cálculos intensivos.

Puedes usar Python para "programación paralela"

Python permite usar **múltiples procesos** de manera eficiente con módulos como **`multiprocessing`**. Esto es útil cuando deseas aprovechar múltiples núcleos de CPU para realizar tareas en paralelo y mejorar el rendimiento.

Ejemplo básico con `multiprocessing`:

```
import multiprocessing

def tarea():
    print("¡Ejecutando tarea!")

proceso = multiprocessing.Process(target=tarea)
proceso.start()
proceso.join()  # Esperar a que el proceso termine
```

Python tiene un "módulo `calendar`" para trabajar con fechas y calendarios

El módulo **`calendar`** en Python permite trabajar con fechas, imprimir calendarios y realizar cálculos con fechas de manera sencilla.

Ejemplo:

```
import calendar
print(calendar.month(2025, 3))  # Imprime el calendario del mes de marzo de 2025
```

Python es utilizado para "programación de redes"

Python tiene varias bibliotecas para trabajar con redes, como **`socket`** para crear aplicaciones cliente-servidor, **`asyncio`** para programación asíncrona y **`requests`** para realizar solicitudes HTTP.

Ejemplo básico con `socket`:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.python.org", 80))
s.send(b"GET / HTTP/1.1\r\nHost: www.python.org\r\n\r\n")
respuesta = s.recv(1024)
print(respuesta)
s.close()
```

Python es extremadamente popular en "desarrollo web"

Python es una de las opciones más populares para el desarrollo web, gracias a frameworks como **Django** y **Flask**. Django es un framework completo para crear aplicaciones web robustas y escalables, mientras que Flask es más ligero y flexible.

Ejemplo con **Flask**:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '¡Hola, Mundo!'
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Python tiene un "módulo `sys`" para interactuar con el sistema

El módulo **`sys`** te permite interactuar con el intérprete de Python y el sistema operativo. Puedes usarlo para acceder a los argumentos de la línea de comandos, cambiar el directorio de trabajo y más.

Ejemplo:

```
import sys

print(sys.argv)  # Imprime los argumentos pasados en la línea de comandos
```

Python tiene un "módulo `turtle`" para enseñar programación

`turtle` es una biblioteca que permite crear gráficos sencillos de manera interactiva. Es utilizada frecuentemente para enseñar a los niños los fundamentos de la programación.

Ejemplo:

```
import turtle
pantalla = turtle.Screen()
tortuga = turtle.Turtle()
tortuga.forward(100)  # Mueve la tortuga hacia adelante 100 unidades
tortuga.left(90)  # Gira la tortuga 90 grados
tortuga.forward(100)  # Mueve la tortuga hacia adelante 100 unidades
pantalla.mainloop()
```

Este código dibuja una forma de "L".

Python puede leer y escribir "archivos CSV" fácilmente

Con la biblioteca **`csv`**, Python hace que trabajar con **archivos CSV** (valores separados por comas) sea muy sencillo. Puedes leer, escribir y manipular archivos CSV con unas pocas líneas de código.

Ejemplo de lectura:

```
import csv
with open('archivo.csv', 'r') as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

Python tiene un "módulo `json`" para trabajar con datos JSON

El módulo `json` en Python te permite trabajar con datos en formato JSON (JavaScript Object Notation), que es comúnmente usado para el intercambio de datos entre aplicaciones web.

Ejemplo:

```
import json

# Convertir un diccionario Python en un objeto JSON
datos = {"nombre": "Juan", "edad": 25}
json_string = json.dumps(datos)
print(json_string)  # Imprime: '{"nombre": "Juan", "edad": 25}'

# Convertir un objeto JSON de nuevo a un diccionario Python
datos_nuevos = json.loads(json_string)
print(datos_nuevos)  # Imprime: {'nombre': 'Juan', 'edad': 25}
```

Python puede interactuar con archivos de "Excel" usando `openpyxl`

Para leer y escribir archivos **Excel (.xlsx)** en Python, puedes usar bibliotecas como `openpyxl`. Esto es útil para la automatización de tareas de análisis de datos o informes.

Ejemplo:

```
from openpyxl import Workbook

wb = Workbook()
ws = wb.active
ws['A1'] = "Hola"
ws['B1'] = "Mundo"
wb.save("mi_archivo.xlsx")
```

Puedes definir tus propios iteradores en Python

- Python permite la creación de **iteradores personalizados**. Un iterador es un objeto que implementa el protocolo de iteración (tener los métodos `__iter__()` y `__next__()`), lo que te permite definir cómo se recorren las colecciones.

  ```
  class Contador:
      def __init__(self, limite):
          self.limite = limite
          self.contador = 0

      def __iter__(self):
          return self

      def __next__(self):
          if self.contador < self.limite:
              self.contador += 1
              return self.contador
          else:
              raise StopIteration

  contador = Contador(3)
  for num in contador:
      print(num)  # Imprime: 1 2 3
  ```

Puedes usar `del` para borrar partes de una lista... ¡o incluso variables!

- Normalmente `del` se usa para eliminar elementos de una lista:

```
lista = [1, 2, 3, 4]
del lista[1]
print(lista)  # [1, 3, 4]
```

Puedes comparar cadenas con operadores como > y <

- En Python, puedes comparar cadenas alfabéticamente con > y <:

```
print("banana" > "apple")  # True (porque "b" viene después de "a" en el
alfabeto)
print("carro" < "casa")    # True (porque "r" viene antes de "s")
```

Python permite asignar valores a variables dentro de un `if` con `:=` (walrus)

- Desde Python 3.8, puedes usar el **operador walrus (`:=`)** para asignar valores dentro de una condición:

```
if (n := len("Python")) > 5:
    print(f"La palabra tiene {n} letras")  # Imprime: La palabra tiene 6
letras
```

Puedes ejecutar código Python dentro de una cadena con `exec()` 🕵️

- `exec()` permite ejecutar código Python desde una cadena:

```
codigo = "print('Hola, mundo!')"
exec(codigo)  # Imprime: Hola, mundo!
```

- ¡Pero cuidado! **Ejecutar código arbitrario con `exec()` es un riesgo de seguridad**

Las funciones pueden modificarse a sí mismas en tiempo de ejecución

- Puedes reasignar una función para que haga algo completamente distinto:

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

saludo()  # Hola!

saludo = lambda: print("Adiós!")  # La función cambia
saludo()  # Adiós!
```

Python tiene un "modo debug" secreto con `python -i`

- Si ejecutas un script con `python -i script.py`, Python no cierra la sesión después de ejecutarlo, lo que te permite inspeccionar variables y probar cosas en tiempo real.

Puedes ejecutar código en paralelo con `multiprocessing`

- Python tiene un módulo llamado `multiprocessing` que permite ejecutar procesos en paralelo:

```
from multiprocessing import Process

def imprimir():
    print("¡Ejecutando en otro proceso!")

p = Process(target=imprimir)
p.start()
p.join()
```

Python permite hacer listas de listas... ¡pero hay que tener cuidado!

- Si creas listas de listas de forma incorrecta, todas apuntarán al mismo objeto:

```
matriz = [[0] * 3] * 3
matriz[0][0] = 1
print(matriz)  # [[1, 0, 0], [1, 0, 0], [1, 0, 0]] ¡No era lo esperado!
```

- Esto sucede porque Python copia la referencia a la misma lista en lugar de crear nuevas listas

Puedes definir números con separadores de guiones bajos

- Python permite escribir números grandes de forma más legible usando _:

```
numero_grande = 1_000_000_000
print(numero_grande)  # 1000000000
```

- ¡Funciona igual que sin los guiones bajos

Puedes acceder a elementos de una lista con valores negativos fuera del rango

- Si accedes a una lista con un índice **negativo más allá del inicio**, Python no da error:

```
lista = [1, 2, 3, 4, 5]
print(lista[-100:100])  # [1, 2, 3, 4, 5]
```

Puedes convertir un número en palabras con `inflect`

- Python no tiene esto por defecto, pero `inflect` lo hace:

```
import inflect
p = inflect.engine()
print(p.number_to_words(123))  # "one hundred and twenty-three"
```

la biblioteca `inflect` en Python **solo** soporta **inglés**. Si necesitas convertir números a palabras en otros idiomas, puedes usar alternativas como:

`num2words` (soporta múltiples idiomas)

```
from num2words import num2words

print(num2words(123, lang="es"))  # "ciento veintitrés"
print(num2words(456, lang="fr"))  # "quatre cent cinquante-six"
print(num2words(789, lang="de"))  # "siebenhundertneunundachtzig"
```

Idiomas soportados por `num2words`: español (`es`), francés (`fr`), alemán (`de`), italiano (`it`), portugués (`pt`), ruso (`ru`), etc.

Y si el idioma no está en la función `num2words ...`

Puedes usar `num2words` junto con un traductor (`googletrans`)

```
from num2words import num2words
from googletrans import Translator

num = 123
text_es = num2words(num, lang='es')

translator = Translator()
text_ca = translator.translate(text_es, src='es', dest='ca').text
print(text_ca)  # "cent vint-i-tres"
```

Puedes hacer que `print()` borre lo que imprimió antes

- Si quieres que un `print()` reemplace lo anterior en la misma línea:

```
import time
for i in range(10):
    print(f"\rCargando... {i}%", end="", flush=True)
    time.sleep(0.5)
```

- `\r` mueve el cursor al inicio, sobrescribiendo el texto anterior. ¡Útil para barras de progreso!

Puedes abrir URLs en el navegador con una línea de código

```
import webbrowser
webbrowser.open("https://www.python.org")
```

- Esto abrirá la página en tu navegador predeterminado. ¡Útil para automatizaciones!

Puedes crear un servidor web en un segundo

- Si tienes una carpeta con archivos y quieres compartirlos en red local:

  ```
  python -m http.server 8000
  ```

- Abre `http://localhost:8000/` y verás los archivos disponibles.

Puedes generar contraseñas aleatorias en una línea

```
import secrets
print(secrets.token_hex(16))  # Genera una clave segura de 16 bytes
```

Puedes comprimir archivos sin usar programas externos

```
import shutil
shutil.make_archive("mi_zip", "zip", "carpeta_a_comprimir")
```

- Esto crea `mi_zip.zip` con el contenido de `carpeta_a_comprimir`.

Puedes escribir código que solo se ejecute en un sistema operativo específico

```
import sys
if sys.platform == "win32":
    print("¡Estás en Windows!")
elif sys.platform == "darwin":
    print("¡Estás en macOS!")
elif sys.platform == "linux":
    print("¡Estás en Linux!")
```

Puedes obtener el código fuente de cualquier función en Python

```
import inspect
print(inspect.getsource(print))
```

- Funciona mejor con funciones definidas por el usuario

Puedes imprimir colores en la terminal con código ANSI

Python permite cambiar el color del texto en la terminal con secuencias ANSI:

```
print("\033[31mEsto es rojo\033[0m y esto es normal")
print("\033[34mEsto es azul\033[0m")
print("\033[1;32mTexto verde brillante\033[0m")
```

- `\033[31m` → Rojo
- `\033[34m` → Azul
- `\033[1;32m` → Verde brillante
- `\033[0m` → Restablece el color

Puedes hacer que Python "hable" en voz alta

Si usas `pyttsx3`, Python puede leer texto en voz alta:

```
import pyttsx3
engine = pyttsx3.init()
engine.say("Hola, esto es Python hablando")
engine.runAndWait()
```

Funciona en Windows, macOS y Linux.

Puedes hacer que un programa espere una entrada secreta (como contraseñas)

La función `getpass` oculta la entrada del usuario:

```
import getpass
password = getpass.getpass("Introduce la contraseña: ")
print("Contraseña ingresada (no se muestra en pantalla)")
```

Muy útil para autenticación sin mostrar la contraseña en pantalla.

Puedes convertir una lista en una cadena sin `join()`

```
palabras = ["Hola", "mundo"]
print(*palabras)  # Hola mundo
```

`*palabras` descomprime la lista como si cada elemento fuera un argumento separado.

Puedes hacer que una función se auto-destruya después de usarse

```
def auto_destruccion():
    print("Solo me ejecutaré una vez")
    del auto_destruccion

auto_destruccion()
auto_destruccion()  # Error, la función ya no existe
```

Se elimina a sí misma con `del`

Puedes rotar elementos en una lista sin ciclos

```
lista = [1, 2, 3, 4]
print(lista[1:] + lista[:1])  # [2, 3, 4, 1]
```

Intercambia los elementos sin `for`.

Puedes encadenar comparaciones matemáticas como en álgebra

En Python, puedes hacer comparaciones encadenadas sin necesidad de operadores lógicos:

```
x = 5
print(3 < x < 10)  # True
print(3 < x > 2)   # True
```

En otros lenguajes tendrías que escribir `3 < x and x < 10`.

Las funciones tienen un atributo secreto `__defaults__`

```
def prueba(a=42):
    pass

print(prueba.__defaults__)  # (42,)
```

Muestra los valores por defecto de los parámetros.