



PYTHON

¿Qué es Python?

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Administrado por la Python Software Foundation, posee una licencia de código abierto, denominada Python Software Foundation License. Python se clasifica constantemente como uno de los lenguajes de programación más populares.

Python es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad.



En los últimos años el lenguaje se ha hecho muy popular, gracias a varias razones como:

- La cantidad de librerías que contiene, tipos de datos y funciones incorporadas en el propio lenguaje, que ayudan a realizar muchas tareas habituales sin necesidad de tener que programarlas desde cero.
- La sencillez y velocidad con la que se crean los programas. Un programa en Python puede tener de 3 a 5 líneas de código menos que su equivalente en Java o C.
- La cantidad de plataformas en las que podemos desarrollar, como Unix, Windows, OS/2, Mac, Amiga y otros.
- Además, Python es gratuito, incluso para propósitos empresariales.

Historia

Python fue creado a finales de los ochenta por Guido van Rossum en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.

El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos Monty Python.

Guido Van Rossum es el principal autor de Python, y su continuo rol central en decidir la dirección de Python es reconocido, refiriéndose a él como Benevolente Dictador Vitalicio (en inglés: Benevolent Dictator for Life, BDFL); sin embargo el 12 de julio de 2018 declinó de dicha situación de honor sin dejar un sucesor o sucesora y con una declaración altisonante:

Entonces, ¿qué van a hacer todos ustedes? ¿Crear una democracia? ¿Anarquía? ¿Una dictadura? ¿Una federación?

Guido van Rossum

El 20 de febrero de 1991, van Rossum publicó el código por primera vez en alt.sources, con el número de versión 0.9.0. En esta etapa del desarrollo ya estaban presentes clases con herencia, manejo de excepciones, funciones y los tipos modulares, como: str, list, dict, entre otros. Además en este lanzamiento inicial aparecía un sistema de módulos adoptado de Modula-3; van Rossum



describe el módulo como «una de las mayores unidades de programación de Python». El modelo de excepciones en Python es parecido al de Modula-3, con la adición de una cláusula else. En el año 1994 se formó comp.lang.python, el foro de discusión principal de Python, marcando un hito en el crecimiento del grupo de usuarios de este lenguaje.

Python alcanzó la versión 1.0 en enero de 1994. Una característica de este lanzamiento fueron las herramientas de la programación funcional: lambda, reduce, filter y map. Van Rossum explicó que «hace 12 años, Python adquirió lambda, reduce(), filter() y map(), cortesía de Amrit Perin, un hacker informático de Lisp que las implementó porque las extrañaba».

La última versión liberada proveniente de CWI fue Python 1.2. En 1995, van Rossum continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI) en Reston, Virginia, donde lanzó varias versiones del software.

Durante su estancia en CNRI, van Rossum lanzó la iniciativa Computer Programming for Everybody (CP4E), con el fin de hacer la programación más accesible a más gente, con un nivel de 'alfabetización' básico en lenguajes de programación, similar a la alfabetización básica en inglés y habilidades matemáticas necesarias por muchos trabajadores. Python tuvo un papel crucial en este proceso: debido a su orientación hacia una sintaxis limpia, ya era idóneo, y las metas de CP4E presentaban similitudes con su predecesor, ABC. El proyecto fue patrocinado por DARPA. En el año 2007, el proyecto CP4E está inactivo, y mientras Python intenta ser fácil de aprender y no muy arcano en su sintaxis y semántica, alcanzando a los no-programadores, no es una preocupación activa.

En el año 2000, el equipo principal de desarrolladores de Python se cambió a BeOpen.com para formar el equipo BeOpen PythonLabs. CNRI pidió que la versión 1.6 fuera pública, continuando su desarrollo hasta que el equipo de desarrollo abandonó CNRI; su programa de lanzamiento y el de la versión 2.0 tenían una significativa cantidad de traslapo. Python 2.0 fue el primer y único lanzamiento de BeOpen.com. Después que Python 2.0 fuera publicado por BeOpen.com, Guido van Rossum y los otros desarrolladores de PythonLabs se unieron en Digital Creations.

Python 2.0 tomó una característica mayor del lenguaje de programación funcional Haskell: listas por comprensión. La sintaxis de Python para esta construcción es muy similar a la de Haskell, salvo por la preferencia de los caracteres de puntuación en Haskell, y la preferencia de Python por palabras



claves alfabéticas. Python 2.0 introdujo además un sistema de recolección de basura capaz de recolectar referencias cíclicas.

Posterior a este doble lanzamiento, y después que van Rossum dejara CNRI para trabajar con desarrolladores de software comercial, quedó claro que la opción de usar Python con software disponible bajo GNU GPL era muy deseable. La licencia usada entonces, la Python License, incluía una cláusula estipulando que la licencia estaba gobernada por el estado de Virginia, por lo que, bajo la óptica de los abogados de Free Software Foundation (FSF), se hacía incompatible con GPL. Para las versiones 1.61 y 2.1, CNRI y FSF hicieron compatibles la licencia de Python con GPL, renombrándola Python Software Foundation License. En el año 2001, van Rossum fue premiado con FSF Award for the Advancement of Free Software.

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '%s [%label=%s]' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s';' % ast[1]
        else:
            print ''
    else:
        print ']';
        children = []
        for n, childenumerate(ast[1:]):
            children.append(dotwrite(child))
        print ', %s -> {' % nodename
        for n in children
            print '%s' % name,
```

Código Python con coloreado de sintaxis

Python 2.1 fue un trabajo derivado de las versiones 1.6.1 y 2.0. Es a partir de este momento que Python Software Foundation (PSF) pasa a ser dueño del proyecto, organizada como una organización sin ánimo de lucro fundada en el año 2001, tomando como modelo la Apache Software Foundation. Incluido en este lanzamiento fue una implementación del scoping más parecida a las reglas de static scoping (del cual Scheme es el originador).

Una innovación mayor en Python 2.2 fue la unificación de los tipos en Python (tipos escritos en C), y clases (tipos escritos en Python) dentro de una jerarquía. Esta unificación logró un modelo de objetos de Python puro y consistente.



También fueron agregados los generadores que fueron inspirados por el lenguaje Icon.

Las adiciones a la biblioteca estándar de Python y las decisiones sintácticas fueron influenciadas fuertemente por Java en algunos casos: el package logging, introducido en la versión 2.3, está basado en log4j; el parser SAX, introducido en 2.0; el package threading, cuya clase Thread expone un subconjunto de la interfaz de la clase homónima en Java.

Python 2, es decir Python 2.7.x, fue oficialmente descontinuado el 1 de enero de 2020 (primero planeado para 2015) después de lo cual no se publicarán parches de seguridad y otras mejoras para él. Con el final del ciclo de vida de Python 2, solo tienen soporte la rama Python 3.6.x y posteriores.

En la actualidad, Python se aplica en los campos de inteligencia artificial y machine learning.

Principales influencias de otros lenguajes

El núcleo de la sintaxis de Python y grandes aspectos de su filosofía fueron heredados directamente del lenguaje de programación ABC. Por ejemplo, el siguiente código muestra una función para obtener el conjunto de todas las palabras en un documento en ABC y en Python:

Recuperar las palabras de un documento en ABC

```
HOW TO RETURN words document:  
    PUT {} IN collection  
    FOR line IN document:  
        FOR word IN split line:  
            IF      word      not.in  
collection:  
                INSERT      word      IN  
collection  
    RETURN collection
```

Recuperar las palabras de un documento en Python

```
def words(document):  
    collection = set()  
    for line in document:  
        for word in line.split():  
            if word not in  
collection:  
                collection.add(word)  
    return collection
```

En ABC no existe propiamente el tipo de dato conjunto, sino algo como multiconjuntos, es decir, si se inserta un elemento dos veces, aparecerá dos veces en esa colección, pero en Python el if puede ser eliminado porque la operación add sobre los conjuntos en Python no hace nada si el elemento ya está en el conjunto. Se puede observar perfectamente las similitudes entre ambos códigos, la sangría, el ciclo for, el operador in, pero también sus



diferencias, como por ejemplo en ABC no se utilizan paréntesis y en Python sí, además las palabras claves en ABC son escritas en mayúsculas, así como los nombres de los procedimientos o funciones; ABC hace distinción entre procedimiento y función, otra diferencia con Python.

Las principales ideas de ABC que influyeron o se incluyeron en Python fueron según el propio Guido:

- La sangría para agrupar el código
- El carácter : para indicar que comienza un bloque indentado (después de pruebas con usuarios)
- El diseño simple de las instrucciones: if, while, for,...
- Tuplas, listas, diccionarios (fuertemente modificados en Python)
- Tipos de datos inmutables
- No imponer límites, como tamaño de un array, etc...
- El "prompt" >>>

Otras ideas que influyeron en la concepción de Python fue tener un lenguaje que pudiera ser tan potente como C pero también expresivo para ejecutar "scripts" como sh. De hecho la sintaxis de Python copia muchísimo de C. Por ejemplo, las palabras claves (if, else, while, for, etc.) son las mismas que en C, los identificadores tienen las mismas reglas para nombrarlos que C, y la mayoría de los operadores estándar tienen el mismo significado que en C. Una de las mayores diferencias es que Python en lugar de usar llaves para agrupar código usa sangría, la otra gran diferencia es que Python usa tipado dinámico.

Los generadores e iteradores fueron inspirados por Icon, y fusionados con las ideas de la programación funcional en un modelo unificado. Modula-3 fue la base del modelo de excepciones y del sistema de módulos. Perl contribuyó en las expresiones regulares, usadas para la manipulación de "string".[\[7\]](#) Las adiciones a biblioteca estándar de Python standard library y algunas opciones sintácticas fueron influenciadas por Java algunos ejemplos son: el paquete de logging, introducido en la versión 2.3, el paquete threading para aplicaciones multihilos, el parser SAX, introducido en la versión 2.0, y la sintaxis del decorador que usa @, incluida en la versión 2.4



Características del lenguaje

Propósito general

Se pueden crear todo tipo de programas. No es un lenguaje creado específicamente para la web, aunque entre sus posibilidades sí se encuentra el desarrollo de páginas.

Multiplataforma

Hay versiones disponibles de Python en muchos sistemas informáticos distintos. Originalmente se desarrolló para Unix, aunque cualquier sistema es compatible con el lenguaje siempre y cuando exista un intérprete programado para él.

Interpretado

Quiere decir que no se debe compilar el código antes de su ejecución. En realidad, sí que se realiza una compilación, pero esta se realiza de manera transparente para el programador. En ciertos casos, cuando se ejecuta por primera vez un código, se producen unos bytecodes que se guardan en el sistema y que sirven para acelerar la compilación implícita que realiza el intérprete cada vez que se ejecuta el mismo código.

Interactivo

Python dispone de un intérprete por línea de comandos en el que se pueden introducir sentencias. Cada sentencia se ejecuta y produce un resultado visible, que puede ayudarnos a entender mejor el lenguaje y probar los resultados de la ejecución de porciones de código rápidamente.

Orientado a Objetos

La programación orientada a objetos está soportada en Python y ofrece en muchos casos una manera sencilla de crear programas con componentes reutilizables.



Funciones y librerías

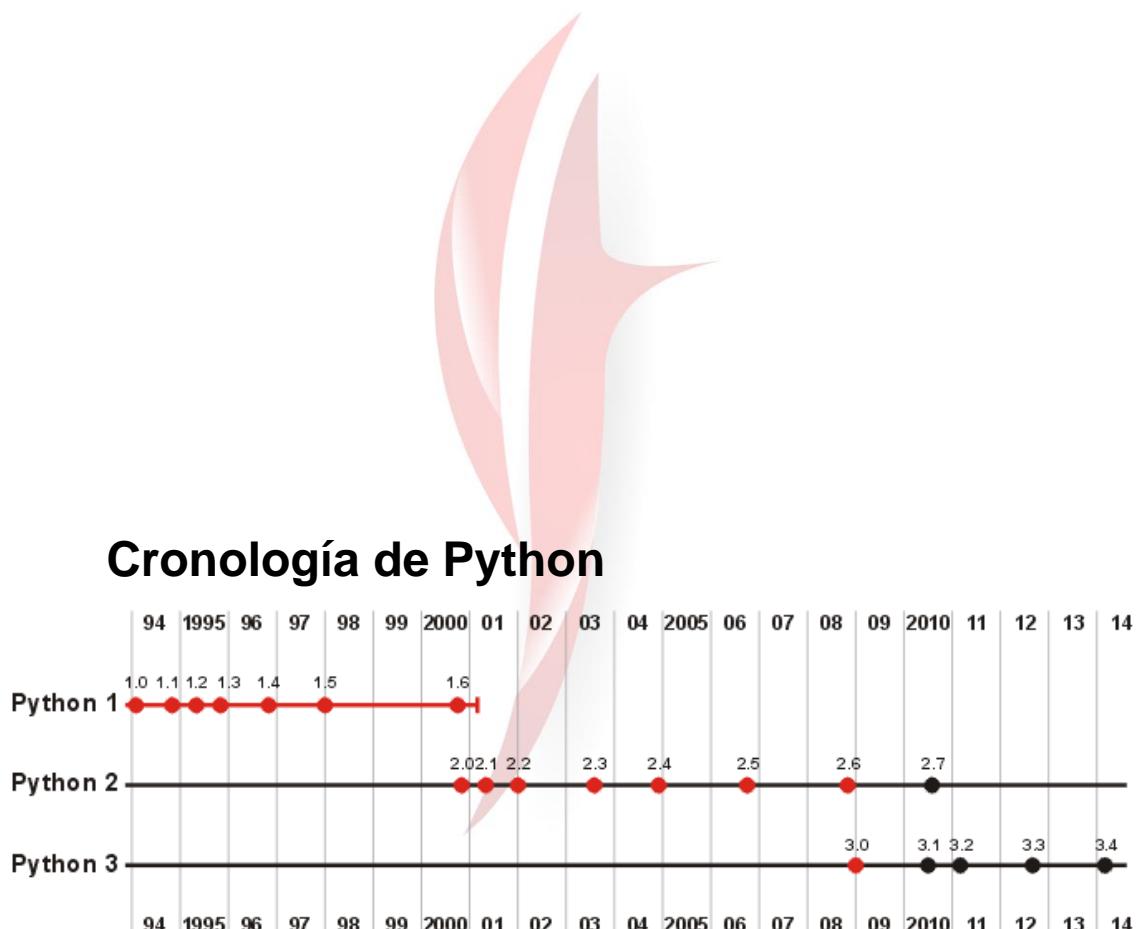
Dispone de muchas funciones incorporadas en el propio lenguaje, para el tratamiento de strings, números, archivos, etc. Además, existen muchas librerías que podemos importar en los programas para tratar temas específicos como la programación de ventanas o sistemas en red o cosas tan interesantes como crear archivos comprimidos en .zip.

Sintaxis clara

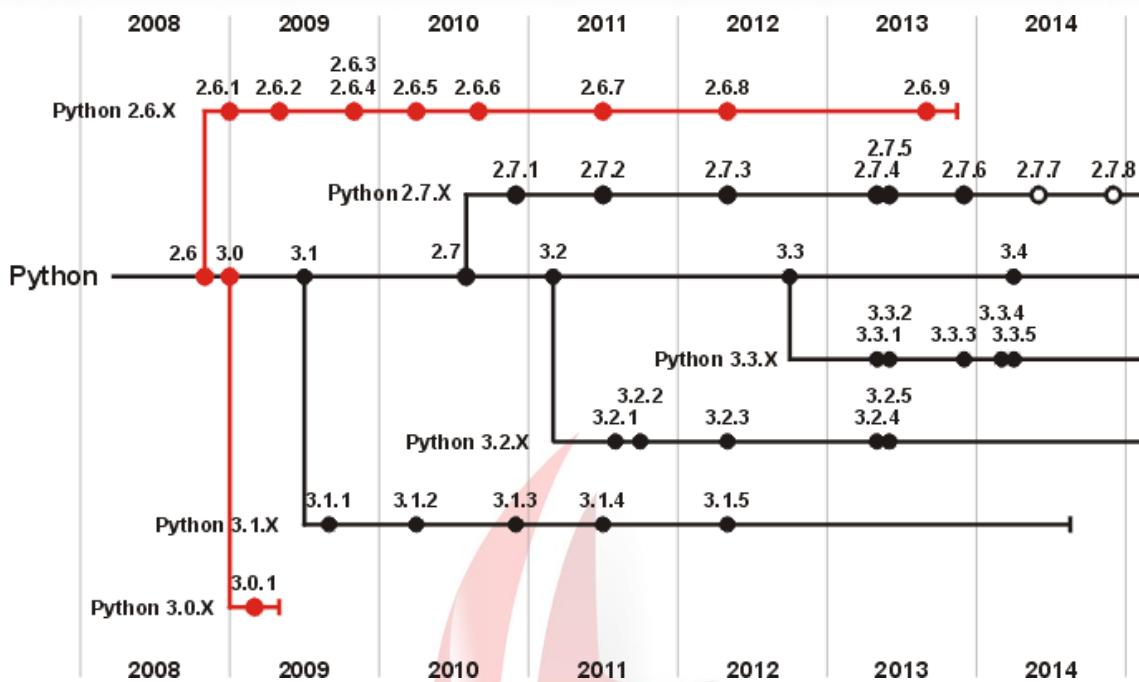
Por último, destacar que Python tiene una sintaxis muy visual, gracias a una notación identada (con márgenes) de obligado cumplimiento. En muchos lenguajes, para separar porciones de código, se utilizan elementos como las llaves o las palabras clave begin y end. Para separar las porciones de código en Python se debe tabular hacia dentro, colocando un margen al código que iría dentro de una función o un bucle. Esto ayuda a que todos los programadores adopten unas mismas notaciones y que los programas de cualquier persona tengan un aspecto muy similar.



CRONOLOGÍA DE VERSIONES



Las versiones indicadas en rojo se consideran obsoletas.



Las versiones indicadas en rojo se consideran obsoletas.

Fechas de publicación:

- Comienzo de la implementación - December, 1989
- Publicación interna en CWI - 1990
- Python 0.9.0 - 20 de febrero de 1991 (publicado en alt.sources)
 - Python 0.9.1 - febrero de 1991
 - Python 0.9.2 - otoño de 1991
 - Python 0.9.4 - 24 de diciembre de 1991
 - Python 0.9.5 - 2 de enero de 1992 (solo para Macintosh)
 - Python 0.9.6 - 6 de abril de 1992
 - Python 0.9.7 beta - 1992
 - Python 0.9.8 - 9 de enero de 1993
 - Python 0.9.9 - 29 de julio de 1993
- Python 1.0 - enero de 1994
 - Python 1.5 - 31 de diciembre de 1997
 - Python 1.6 - 5 de septiembre de 2000
- Python 2.0 - 16 de octubre de 2000
 - Python 2.1 - 17 de abril de 2001
 - Python 2.2 - 21 de diciembre de 2001
 - Python 2.3 - 29 de julio de 2003
 - Python 2.4 - 30 de noviembre de 2004
 - Python 2.5 - 19 de septiembre de 2006



- Python 2.6 - 1 de octubre de 2008
- Python 2.7 - 3 de julio de 2010
- Python 3.0 - 3 de diciembre de 2008
 - Python 3.1 - 27 de junio de 2009
 - Python 3.2 - 20 de febrero de 2011
 - Python 3.3 - 29 de septiembre de 2012
 - Python 3.4 - 16 de marzo de 2014
 - Python 3.5 - 8 de febrero de 2015
 - Python 3.6 - 23 de diciembre de 2016
 - Python 3.7 - 12 de junio de 2018
 - Python 3.8 - 14 de octubre de 2019
 - Python 3.9 - 5 de octubre de 2020

Primera publicación

El 20 de febrero de 1991, van Rossum publicó el código por primera vez en alt.sources, con el número de versión 0.9.0. En esta etapa del desarrollo ya estaban presentes clases con herencia, manejo de excepciones, funciones y los tipos modulares, como: str, list, dict, entre otros. Además en este lanzamiento inicial aparecía un sistema de módulos adoptado de Modula-3; Van Rossum describe el módulo como “una de las mayores unidades de programación de Python”. El modelo de excepciones en Python es parecido al de Modula-3, con la adición de una cláusula else. En el año 1994 se formó comp.lang.python, el foro de discusión principal de Python, marcando un hito en el crecimiento del grupo de usuarios de este lenguaje.

Versión 1.0

Python llega a la versión 1.0 en enero de 1994. Las características más importantes incluidas en esta publicación fueron las herramientas de la programación funcional lambda, reduce, filter y map. Van Rossum ha comentado que "Python adquiere lambda, reduce(), filter() and map(), cortesía de un hacker de Lisp que las extrañaba y envió parches que funcionaban".

La última versión liberada proveniente de CWI fue Python 1.2. En 1995, van Rossum continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI) en Reston, Virginia, donde lanzó varias versiones del software.

Para la versión 1.4, Python adquirió varias características nuevas. Es notable entre estas los argumentos por nombre inspirados por Modula-3 (que también



son similares a los de Common Lisp) y soporte "built-in" para los números complejos.

Durante su estancia en CNRI, Van Rossum lanzó la iniciativa Computer Programming for Everybody (CP4E), con el fin de hacer la programación más accesible a más gente, con un nivel de 'alfabetización' básico en lenguajes de programación, similar a las habilidades básicas en inglés y matemáticas necesarias por muchos empleadores. Python tuvo un papel crucial en este proceso: debido a su orientación hacia una sintaxis limpia, ya era idóneo, y las metas de CP4E presentaban similitudes con su predecesor, ABC. El proyecto fue patrocinado por DARPA. El proyecto CP4E está inactivo, y mientras tanto Python intenta ser fácil de aprender y no muy arcano en su sintaxis y semántica, pero alcanzar a los no-programadores ya no es una preocupación activa.

BeOpen

En el año 2000, el equipo principal de desarrolladores de Python se cambió a BeOpen.com para formar el equipo BeOpen PythonLabs. CNRI pidió que la versión 1.6 fuera publicada hasta el punto de desarrollo en que el equipo abandonó CNRI. Consecuentemente su agenda de lanzamiento para la versión 1.6 y la versión 2.0 tenían una significativa cantidad de solapamiento. Python 2.0 fue el primer y único lanzamiento de BeOpen.com. Después que Python 2.0 fuera publicado por BeOpen.com, Guido van Rossum y los otros desarrolladores de PythonLabs se unieron a Digital Creations.

La publicación de Python 1.6 incluía una nueva licencia de CNRI que era substancialmente más larga que la licencia de CWI que había sido usada en las publicaciones anteriores. La nueva licencia incluía una cláusula estipulando que la licencia estaba gobernada por las leyes del estado de Virginia. La Free Software Foundation (FSF) argumentó la cláusula era incompatible con GNU GPL. Entonces BeOpen, CNRI, y FSF acordaron cambiar Python hacia una licencia de software libre que lo haría compatible con GPL. Python 1.6.1 es básicamente el mismo que Python 1.6, con unos arreglos de bugs, y con la nueva licencia compatible con GPL.

Versión 2.0

Python 2.0 tomó una importante característica del lenguaje de programación funcional Haskell: generación de listas. La sintaxis de Python para esta construcción es muy similar a la de Haskell, salvo por la preferencia de los



caracteres de puntuación en Haskell, y la preferencia de Python por palabras claves. Python 2.0 introdujo además un sistema de recolección de basura capaz de recolectar referencias cíclicas.

Python 2.1 fue un trabajo derivado de Python 1.6.1, así como también de Python 2.0. Su licencia fue renombrada a: Python Software Foundation License. Todo el código, documentación y especificaciones añadidas, desde la fecha del lanzamiento de la versión alfa de Python 2.1, pertenece a Python Software Foundation (PSF), una organización sin ánimo de lucro fundada en el año 2001, tomando como modelo la Apache Software Foundation. Este lanzamiento incluyó un cambio en el lenguaje para soportar ámbitos anidados (más conocido en programación como "nested scopes") como lo soportan otros lenguajes de "static scoping" (del cual Scheme es el originador). (Esta característica fue deshabilitada por defecto, y no requerida, hasta Python 2.2.)

Una gran innovación en Python 2.2 fue la unificación de los tipos en Python (tipos escritos en C), y clases (tipos escritos en Python) dentro de una jerarquía. Esta unificación logró un modelo orientado a objetos de Python puro y consistente. También fueron agregados los generadores que fueron inspirados por el lenguaje Icon.

Versión 3.0

Python 3.0 (también conocido como "Python 3000" o "Py3K") fue diseñado para rectificar ciertas fallas fundamentales en el diseño del lenguaje (los cambios requeridos no podían ser implementados mientras se mantuviera compatibilidad hacia atrás con la serie 2.x). El principio que guía Python 3 es: "reducir la duplicación de características eliminando viejas formas de hacer las cosas (reduce feature duplication by removing old ways of doing things)".



LENGUAJES DE PROGRAMACIÓN



Lenguaje de programación

En términos generales, un lenguaje de programación es una herramienta que permite desarrollar software o programas para computadora. Los lenguajes de programación son empleados para diseñar e implementar programas encargados de definir y administrar el comportamiento de los dispositivos físicos y lógicos de una computadora. Lo anterior se logra mediante la creación e implementación de algoritmos de precisión que se utilizan como una forma de comunicación humana con la computadora.

A grandes rasgos, un lenguaje de programación se conforma de una serie de símbolos y reglas de sintaxis y semántica que definen la estructura principal del lenguaje y le dan un significado a sus elementos y expresiones.

Programación es el proceso de análisis, diseño, implementación, prueba y depuración de un algoritmo, a partir de un lenguaje que compila y genera un código fuente ejecutado en la computadora.



La función principal de los lenguajes de programación es escribir programas que permiten la comunicación usuario-máquina. Unos programas especiales (compiladores o intérpretes) convierten las instrucciones escritas en código fuente, en instrucciones escritas en lenguaje máquina (0 y 1).

Los intérpretes leen la instrucción línea por línea y obtienen el código máquina correspondiente.

En cuanto a los compiladores, traducen los símbolos de un lenguaje de programación a su equivalencia escrito en lenguaje máquina (proceso conocido como compilar). Por último, se obtiene un programa ejecutable.

Para entender mejor la forma como se estructura un lenguaje de programación, observa la siguiente imagen (en este apunte se utilizará el lenguaje C).

En particular, este lenguaje está caracterizado por ser de uso general, de sintaxis compacta y portable. Así, un lenguaje de programación es una herramienta informática que permite desarrollar programas para computadoras.

Características del lenguaje C

El lenguaje C es muy empleado porque puede ser utilizado para desarrollar programas de diversa naturaleza, como lenguajes de programación, manejadores de bases de datos o sistemas operativos. Su sintaxis es compacta, ya que emplea pocas funciones y palabras reservadas, comparado con otros lenguajes, como Java; además, es portable, toda vez que se utiliza en varios sistemas operativos y hardware.

Antecedentes

Profesor de matemáticas e inventor en la universidad de Cambridge, Inglaterra, a mediados del siglo XIX, Charles Babbage fue el primero en concebir la idea de un lenguaje de programación, al predecir varias de las teorías en las que se basan las computadoras actuales.

Babbage desarrolló la idea de una máquina analítica programable que, por limitaciones tecnológicas de su época, no pudo ser construida. Junto con él, su colaboradora Ada Lovelace es considerada como la primera programadora de la historia, ya que escribió los primeros programas para la máquina concebida por Babbage en tarjetas perforadas, siguiendo una lógica de programación muy similar a la empleada en nuestros días. Estos programas nunca pudieron verse ejecutados debido a que la máquina no fue construida.



Las técnicas empleadas por Babbage y Ada fueron seguidas por los primeros programadores de computadoras, quienes se valieron de tarjetas perforadas para introducir sus programas en las computadoras.

En 1823, con el apoyo del gobierno británico, se aprobó el proyecto de construcción de una máquina de diferencias. Esta máquina era un dispositivo mecánico diseñado para realizar sumas de forma repetitiva. Babbage abandonó el proyecto para dedicarse a su máquina analítica, influenciado por la creación de un fabricante de telas francés, Joseph Marie Jacquard, que había desarrollado una máquina tejedora con la capacidad de reproducir patrones de tejidos, leyendo información codificada en tarjetas perforadas de papel rígido.

Desde entonces, Babbage se propuso construir una máquina que efectuara cálculos matemáticos de precisión, empleando 20 dígitos, y que pudiera ser programada mediante tarjetas perforadas. Aun cuando esta idea quedó sólo en el proyecto, fue una contribución muy importante para el diseño y funcionamiento de las computadoras actuales.

Charles Babbage es considerado el padre de la informática. A pesar de que su máquina nunca pudo ser desarrollada, sus ideas y diseños sirvieron para la construcción y el progreso de las primeras computadoras modernas.

Cuando surgió la primera computadora, la ENIAC (Electronic Numerical Integrator And Calculator), su programación se basaba en componentes físicos, o sea, se programaba invirtiendo directamente el hardware de la máquina: se cambiaban de sitio los cables para conseguir así la programación. La entrada y salida de datos se realizaba mediante tarjetas perforadas.

Para crear un lenguaje de programación es necesaria una herramienta que lo traduzca. Se describe a continuación cómo ha ido evolucionando esta herramienta en los últimos 50 años.

Clasificación

Los circuitos microprogramables son sistemas digitales, lo que significa que trabajan con dos únicos niveles de tensión simbolizados con el cero (0) y el uno (1). Por eso, el lenguaje de máquina utiliza sólo dichos signos.

Un lenguaje de bajo nivel es trasladado fácilmente a lenguaje de máquina (la palabra bajo se refiere a la abstracción reducida entre el lenguaje y el hardware).



Y los lenguajes de programación de alto nivel se caracterizan por expresar los programas de una manera sencilla.

Lenguaje máquina

Es el sistema de códigos interpretable directamente por un circuito microprogramable, como el microprocesador de una computadora. Este lenguaje se compone de un conjunto de instrucciones que determinan acciones que serán realizadas por la máquina. Y un programa de computadora consiste en una cadena de estas instrucciones de lenguaje de máquina (más los datos). Normalmente estas instrucciones son ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos. El lenguaje máquina es específico de cada máquina o arquitectura de la máquina, aunque el conjunto de instrucciones disponibles pueda ser similar entre ellas.

Lenguajes de bajo nivel

Un lenguaje de programación de bajo nivel es el que proporciona poca o ninguna abstracción del microprocesador de una computadora. Consecuentemente, su trasladado al lenguaje máquina es fácil. El término ensamblador (del inglés assembler) se refiere a un tipo de programa informático encargado de traducir un archivo fuente, escrito en un lenguaje ensamblador, a un archivo objeto que contiene código máquina ejecutable directamente por la máquina para la que se ha generado.

Lenguaje de alto nivel

Los lenguajes de programación de alto nivel se caracterizan porque su estructura semántica es muy similar a la forma como escriben los humanos, lo que permite codificar los algoritmos de manera más natural, en lugar de codificarlos en el lenguaje binario de las máquinas, o a nivel de lenguaje ensamblador.

Ejemplos de lenguajes de alto nivel

Revisa la siguiente información para saber cuáles son y en qué consisten los lenguajes de alto nivel.

C++, Fortran, Java, Perl, PHP y Python.



Interpretados y Compilados

Son las instrucciones de la máquina las que realmente pueden impulsar la ejecución de la máquina, pero no es realista que los desarrolladores ordinarios escriban instrucciones de la máquina directamente, por lo que han aparecido lenguajes de alto nivel de computadora. Los lenguajes de alto nivel permiten la programación en lenguaje natural (generalmente inglés), pero los programas en lenguajes de alto nivel deben eventualmente traducirse a instrucciones de máquina para su ejecución.

Los lenguajes de alto nivel se pueden dividir en dos tipos: compilados e interpretados de acuerdo con la forma en que se ejecuta el programa.

Compilación

Un lenguaje compilado es un lenguaje de programación cuyas implementaciones son normalmente compiladores (traductores que generan código de máquina a partir del código fuente) y no intérpretes (ejecutores paso a paso del código fuente, donde no se lleva a cabo una traducción en la preejecución).

El término es un tanto vago. En principio, cualquier lenguaje puede ser implementado con un compilador o un intérprete. Sin embargo, es cada vez más frecuente una combinación de ambas soluciones: un compilador puede traducir el código fuente en alguna forma intermedia (muchas veces llamado Bytecode), que luego se pasa a un intérprete que lo ejecuta.

Ventajas y desventajas

Los programas compilados a código nativo en tiempo de compilación tienden a ser más rápidos que los traducidos en tiempo de ejecución, debido a la sobrecarga del proceso de traducción. Sin embargo, las nuevas tecnologías como la compilación en tiempo de ejecución, y mejoras generales en el proceso de traducción están empezando a reducir esta brecha. En algún punto intermedio, tiende a ser más eficiente la solución mixta usando bytecode.

Los lenguajes de programación de bajo nivel son típicamente compilados, en especial cuando la eficiencia es la principal preocupación, en lugar de soporte de plataformas cruzadas. Para los lenguajes de bajo nivel, hay más correspondencias uno a uno entre el código programado y las operaciones de hardware realizadas por el código máquina, lo que hace que sea más fácil para los programadores controlar más finamente la CPU y uso de memoria.



Interpretación

Lenguaje interpretado . Es el lenguaje cuyo código no necesita ser preprocesado mediante un compilador, eso significa que el ordenador es capaz de ejecutar la sucesión de instrucciones dadas por el programador sin necesidad de leer y traducir exhaustivamente todo el código.

Para que esto sea posible hace falta un intermediario, un programa encargado de traducir cada instrucción escrita con una semántica 'humana' a Código máquina (instrucciones de la CPU del ordenador), este programa recibe el nombre de interprete (en inglés parser).

El interprete se encarga de leer una a una las instrucciones textuales del programa conforme estas necesitan ser ejecutadas y descomponerlas en instrucciones del sistema, además se encarga de automatizar algunas de las tareas típicas de un programador como declaraciones de variables o dependencias, de esta manera el proceso de programar se suele agilizar mucho lo cual repercute en la eficiencia del que tiene que escribir el código.

Ventajas

La principal ventaja de un lenguaje interpretado es que es independiente de la máquina y del sistema operativo ya que no contiene instrucciones propias de un procesador sino que contiene llamadas a funciones que el interprete deberá reconocer. Basta que exista un interprete de un lenguaje para dicho sistema y todos los programas escritos en ese lenguaje funcionaran.

Además un lenguaje interpretado permite modificar en tiempo de ejecución el código que se está ejecutando así como añadirle nuevo, algo que resulta idóneo cuando queremos hacer pequeñas modificaciones en una aplicación y no queremos tener que recompilarla toda cada vez.

Desventajas

- **Velocidad.** Es el aspecto más notable y el cual se debe evaluar a fondo al crear software con este tipo de lenguajes, pues se debe equilibrar la portabilidad con la velocidad que se está sacrificando. A menos que las prestaciones de los equipos informáticos sean bastante altas, en el caso cual, se podría despreciar este aspecto.
- **Portabilidad.** El problema radica en que en la actualidad, casi todos los lenguajes compilados, existen para todas las plataformas, no así las



máquinas virtuales o frameworks, aunque en el caso de Java, se ha hecho un excelente trabajo en cuanto a eso.





ENTORNOS DE DESARROLLO



Cualquier editor de texto podría servir para programar en cualquier lenguaje, pero necesitaría programas externos para integrar funcionalidades específicas para Python como:

- Coloreado de sintaxis propia de Python.
- Documentación de módulos Python.
- Soporte de entornos de ejecución.
- Validadores de sintaxis.
- Validadores de estilos y seguimiento del PEP-8.
- Depurador de código Python.

Los IDEs para Python que se presentan a continuación se pueden separar en 3 grupos ordenados desde más específicos a más generalistas:

- Entornos de desarrollo orientados a desarrollo Python.
- Entornos que soportan múltiples lenguajes (generalistas) que usan complementos para dar soporte a python



- Editores de texto potentes que añadiendo muchos complementos manualmente soportan Python.

PYTHON IDLE



IDLE (Integrated DeveLopment Environment for Python) es un entorno gráfico de desarrollo elemental que permite editar y ejecutar programas en Python. IDLE es también un entorno interactivo en el que se pueden ejecutar instrucciones sueltas de Python.

En Windows, IDLE se distribuye junto con el intérprete de Python, es decir, al instalar Python en Windows también se instala IDLE.

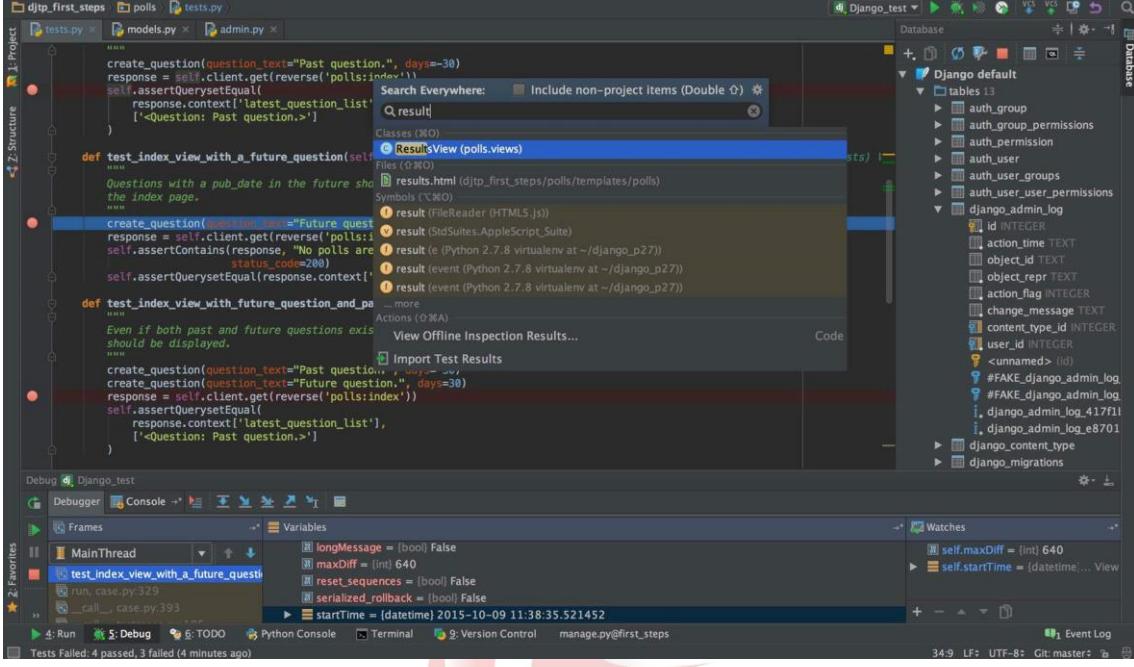
En Linux, IDLE se distribuye como una aplicación separada que se puede instalar desde los repositorios de cada distribución.



PYCHARM



PyCharm es un IDE o entorno de desarrollo integrado multiplataforma utilizado para desarrollar en el lenguaje de programación Python. Proporciona análisis de código, depuración gráfica, integración con VCS / DVCS y soporte para el desarrollo web con Django, entre otras bondades. PyCharm es desarrollado por la empresa JetBrains y debido a la naturaleza de sus licencias tiene dos versiones, la Community que es gratuita y orientada a la educación y al desarrollo puro en Python y la Professional, que incluye más características como el soporte a desarrollo web con varios precios.



A screenshot of the PyCharm IDE interface. The top navigation bar shows 'Django_test' and various icons. The main area displays Python test code for a Django application named 'djtp_first_steps'. The code includes tests for creating past and future questions and checking their display on the index page. A search bar at the top right says 'Search Everywhere: Q result'. To the right, a 'Database' browser is open, connected to 'Django default' and showing the structure of the 'django_admin_log' table, which contains logs of administrative actions. At the bottom, there's a 'Frames' tool window showing the call stack, and a 'Watches' tool window monitoring variables like 'self.maxDiff' and 'self.startTime'.

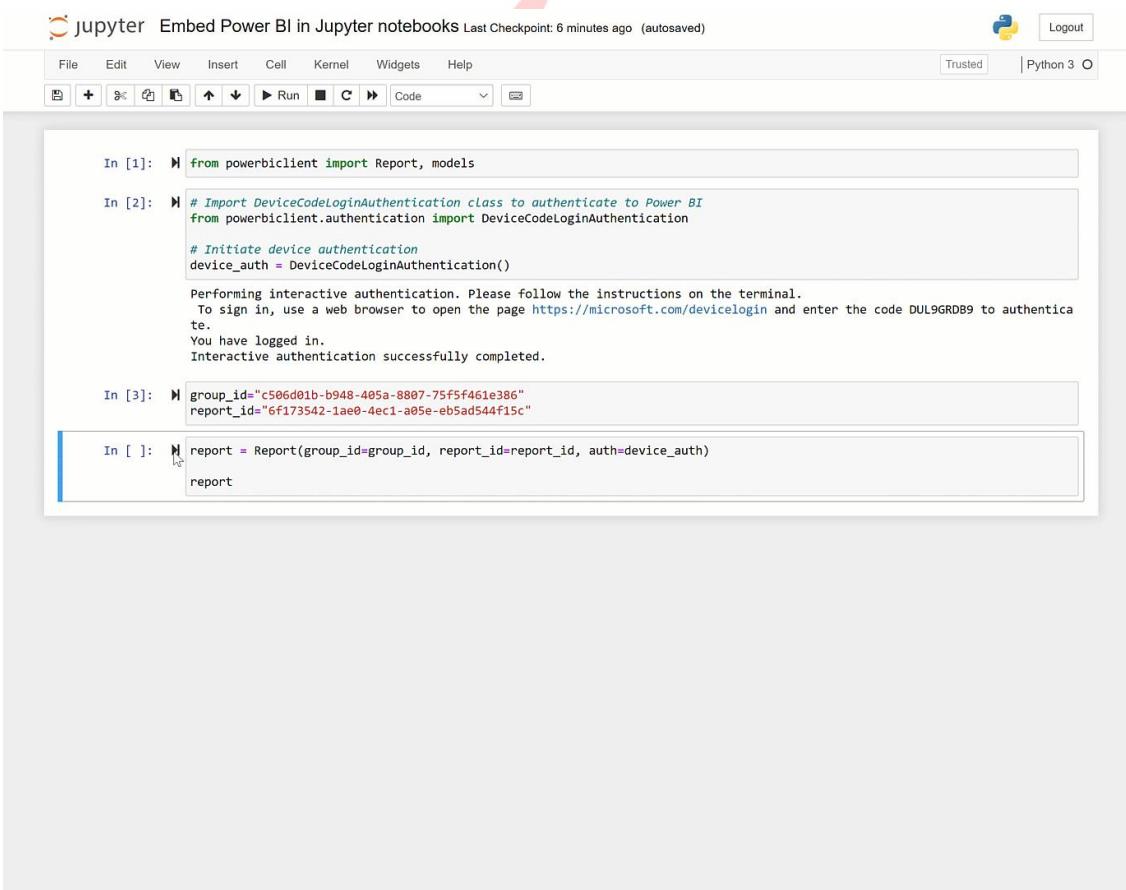
JUPYTER NOTEBOOK



Jupyter Notebook (anteriormente IPython Notebooks) es un entorno informático interactivo basado en la web para crear documentos de Jupyter notebook. El término "notebook" puede hacer referencia coloquialmente a muchas entidades diferentes, principalmente la aplicación web Jupyter, el servidor web Jupyter

Python o el formato de documento Jupyter según el contexto. Un documento de Jupyter Notebook es un documento JSON, que sigue un esquema versionado y que contiene una lista ordenada de celdas de entrada/salida que pueden contener código, texto (usando Markdown), matemáticas, gráficos y texto enriquecidos, generalmente terminado con la extensión ".ipynb".

Jupyter Notebook puede conectarse a muchos núcleos para permitir la programación en muchos idiomas. Por defecto, Jupyter Notebook se conecta con el núcleo IPython. A partir de la versión 2.3.89 (octubre del 2014), hay, actualmente, 49 núcleos compatibles con Jupyter para muchos lenguajes de programación, incluidos Python, R, Julia y Haskell.



The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** "jupyter Embed Power BI in Jupyter notebooks Last Checkpoint: 6 minutes ago (autosaved)"
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, Python 3
- Cells:**
 - In [1]: `from powerbiclient import Report, models`
 - In [2]:

```
# Import DeviceCodeLoginAuthentication class to authenticate to Power BI
from powerbiclient.authentication import DeviceCodeLoginAuthentication

# Initiate device authentication
device_auth = DeviceCodeLoginAuthentication()
```

Performing interactive authentication. Please follow the instructions on the terminal.
To sign in, use a web browser to open the page <https://microsoft.com/devicelogin> and enter the code DUL9GRDB9 to authenticate.
You have logged in.
Interactive authentication successfully completed.
 - In [3]: `group_id="c506d01b-b948-405a-8807-75f5f461e386"`
`report_id="6f173542-1ae0-4ec1-a05e-eb5ad544f15c"`
 - In []: `report = Report(group_id=group_id, report_id=report_id, auth=device_auth)`
`report`

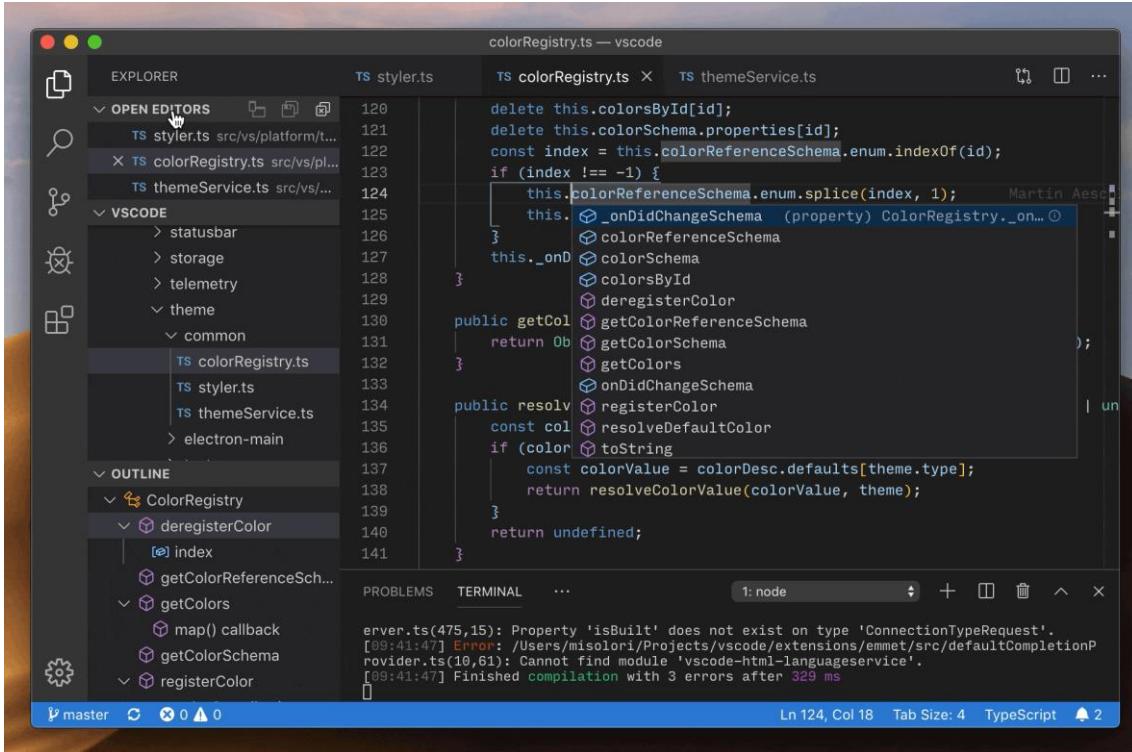


VISUAL STUDIO CODE



Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. También es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos de teclado y las preferencias. Es gratuito y de código abierto, aunque la descarga oficial está bajo software privativo e incluye características personalizadas por Microsoft.

Visual Studio Code se basa en Electron, un framework que se utiliza para implementar Chromium y Node.js como aplicaciones para escritorio, que se ejecuta en el motor de diseño Blink. Aunque utiliza el framework Electron, el software no usa Atom y en su lugar emplea el mismo componente editor (Monaco) utilizado en Visual Studio Team Services (anteriormente llamado Visual Studio Online).



A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows the file tree with files like 'styler.ts', 'colorRegistry.ts', and 'themeService.ts'. The main editor window displays a portion of 'colorRegistry.ts' with code completion suggestions for 'colorReferenceSchema' and 'onDidChangeSchema'. The bottom status bar shows the file path 'colorRegistry.ts — vscode', line 'Ln 124, Col 18', tab size '4', TypeScript, and two notifications.

GOOGLE COLABORATORY

Google Colaboratory





Google Colaboratory es una herramienta que te permite ejecutar scripts de Python a través de los servidores de Google. Esto te permite ejecutar celdas de código como si se tratara de un cuaderno de Jupyter Notebook. Pero no sólo eso, Google Colab es perfecto para implementar algoritmos de aprendizaje máquina, ya que no te limitas a los recursos de tu computadora. Esto quiere decir que tienes a tu disposición el GPU y las TPUs de Google para potencializar el cómputo de tu proyecto.

Al igual que un cuaderno de Jupyter, puedes utilizar librerías de aprendizaje máquina y procesamiento de imágenes. Puedes utilizar librerías como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Todo ello con Python 2.7 y 3.6, que aún no está disponible para R y Scala. Usar los servicios de Google Colab es gratuito, al igual que otras herramientas de ofimática. Aunque esta versión tiene ciertas limitaciones que se pueden consultar en su página de preguntas frecuentes, es una herramienta muy poderosa y un buen complemento para tu instalación de Python.



INSTALACIÓN DE PYTHON

Interprete de Python

El intérprete de Python es un programa encargado de analizar cualquier código escrito en el lenguaje de programación Python, interpretarlo y ejecutarlo sobre la máquina virtual de Python.

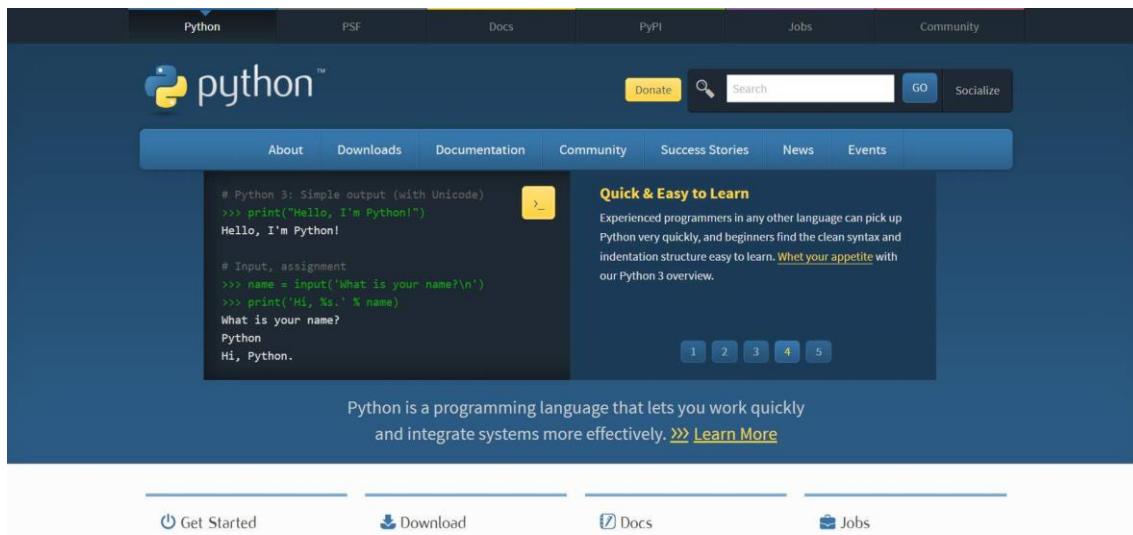
Existen diferentes versiones de intérpretes orientadas a mejorar alguna funcionalidad en particular, aunque el más comúnmente utilizado es CPython.

- CPython: intérprete oficial, utilizado por defecto, escrito en C y recomendado a usar.
- PyPy: intérprete con un JIT (Just in Time compiler) orientado a dar el mejor rendimiento que CPython.
- IronPython: implementación del intérprete de Python utilizando el framework .Net. Permite una buena integración con aplicaciones .Net pudiendo compartir librerías.
- buJython: implementación del intérprete en Java. Permite ejecutar código Python en la máquina virtual de Java con código Java.

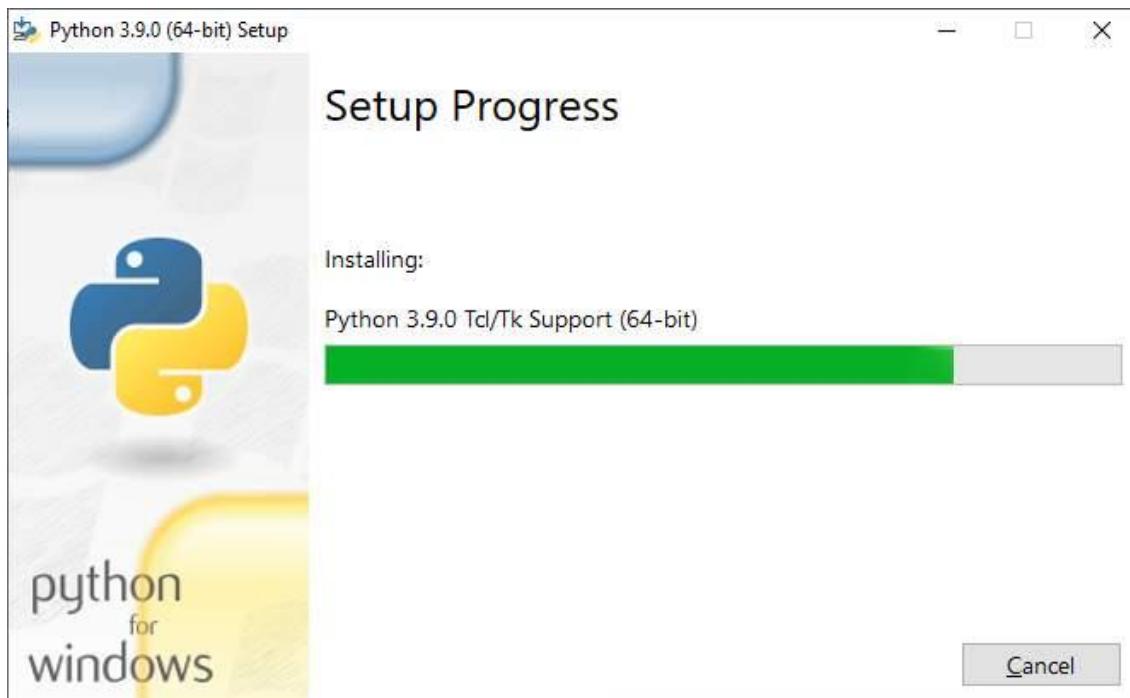
Instalación Windows

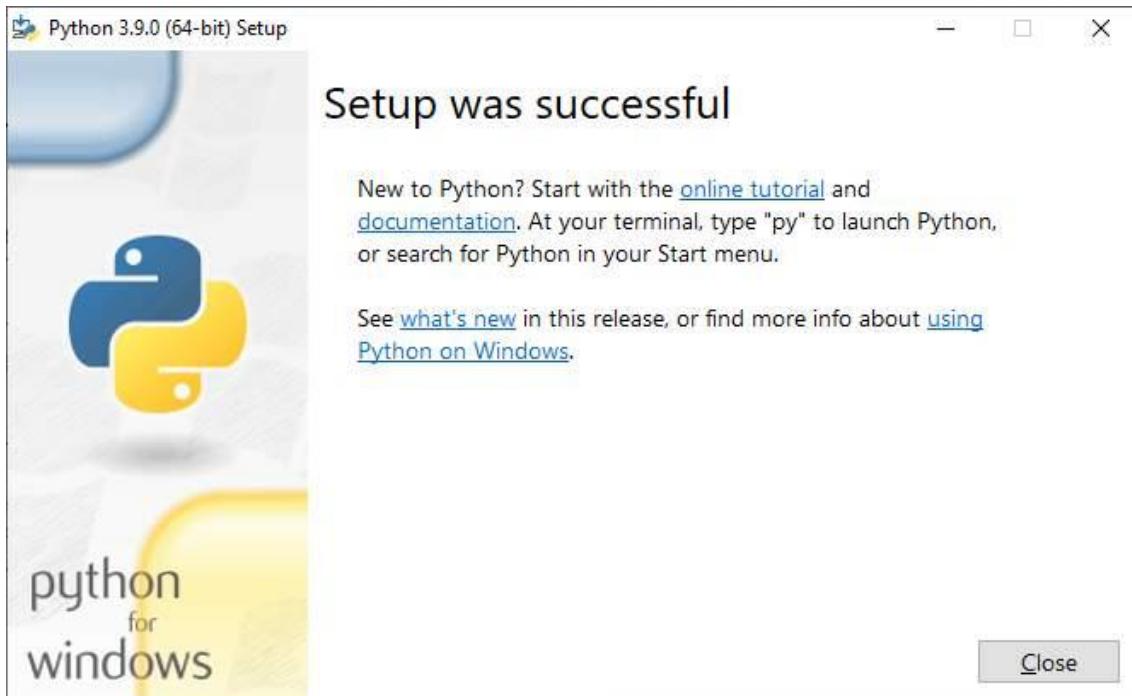
El intérprete de Python cuenta con múltiples componentes, por un lado el intérprete y por otro la máquina virtual, pero todos los componentes se instalan fácilmente desde la web oficial de Python si no están ya instalados en el sistema.

Puedes descargar Python desde [aqui](#)



La instalación en Windows es sencilla, el propio instalador de Python sirve de guia para instalar todas las dependencias y requisitos.





Es importante seleccionar que Python se incluya en el path para facilitar la ejecución de módulos desde consola.

De esta forma se ha instalado el intérprete de Python y el IDE simple IDLE, el cual permite realizar los primeros programas en Python.

Instalación Linux

Python se encuentra instalado por defecto en la mayoría de distribuciones de linux, pero quizás la versión instalada no es la que se desea.

Para instalar una versión diferente en linux se puede buscar en el gestor de dependencias y paquetes del sistema por la versión específica a instalar, o se puede instalar directamente desde el código fuente compilando la versión específica.

```
# Ubuntu (apt-get)
$ sudo apt-get update
$ sudo apt-get install python3.9

# Centos/Fedora (yum)
$ sudo yum install gcc openssl-devel bzip2-devel libffi-devel
## Instalación desde código fuente
```

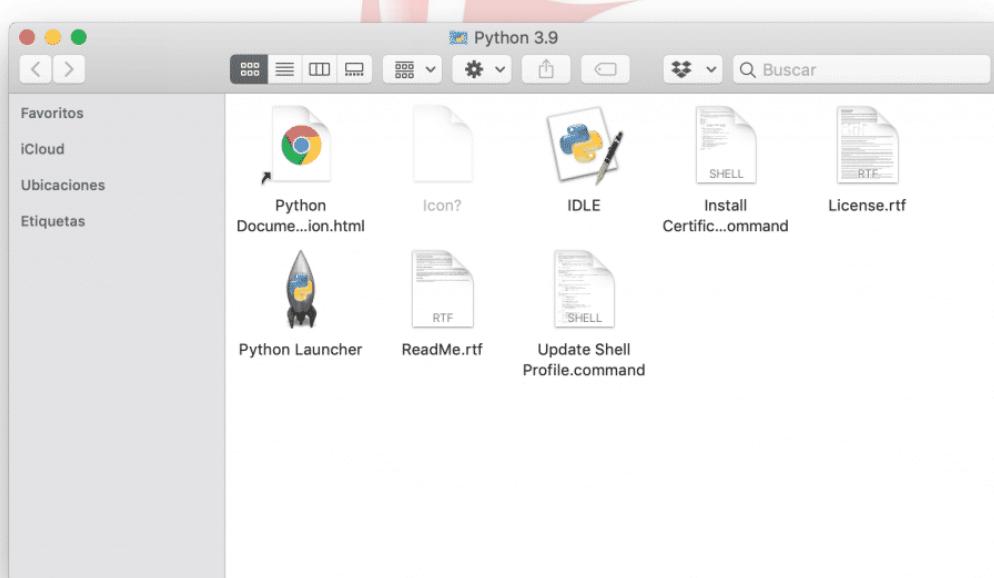


```
$ cd /tmp
$ wget https://www.python.org/ftp/python/3.9.0/Python-3.9.0.tgz
$ tar xzf Python-3.9.0.tgz
$ cd Python-3.9.0
$ sudo ./configure --enable-optimizations
$ sudo make altinstall
```

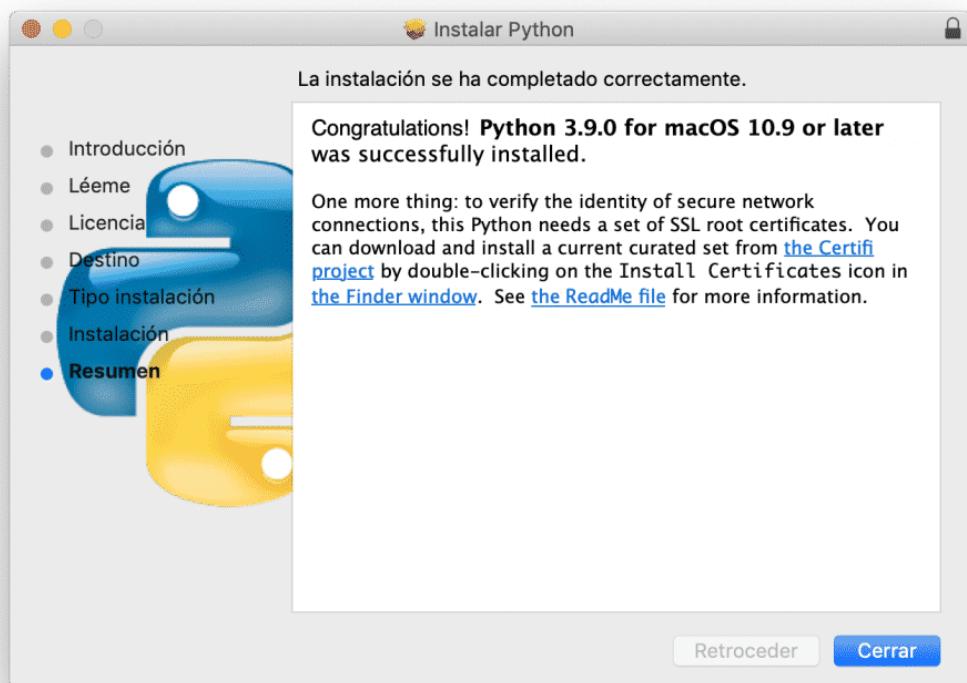
Instalación Mac OS

Python se encuentra instalado en MacOS X por defecto aunque seguramente no esté instalada la última versión o la versión que se desee utilizar.

Se puede instalar Python en Mac OS X descargando el archivo .pkg desde la página oficial de python dejado en la parte de arriba









PRIMER CONTACTO

Primer código

Ahora que tenemos instalado el interprete podemos comenzar a escribir Python podemos de dos maneras:

- Modo interactivo
- Mediante un archivo exclusivo para Python

Vamos a realizarlo de las dos maneras, pues Python viene tanto con el modo interactivo como con un editor para escribir en archivos con extensión "py".

Modo interactivo

Modo interactivo, también conocido como REPL nos proporciona una forma rápida de ejecutar bloques o una sola línea de código Python. El código se ejecuta a través del shell de Python, que viene con la instalación de Python. El modo interactivo es útil cuando solo desea ejecutar comandos básicos de Python.



o si es nuevo en la programación de Python y solo quiere ensuciarse las manos con este hermoso lenguaje.

Para acceder al shell de Python, abra la terminal de su sistema operativo y luego escriba «python». Presione la tecla enter y aparecerá el shell de Python. Este es el mismo ejecutable de Python que usa para ejecutar scripts, que viene instalado de forma predeterminada en Mac y sistemas operativos basados en Unix.

```
C:\Windows\system32>python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Los >>> indica que el shell de Python está listo para ejecutarse y enviar sus comandos al intérprete de Python. El resultado se muestra inmediatamente en el shell de Python tan pronto como el intérprete de Python interpreta el comando.

Para ejecutar sus declaraciones de Python, simplemente escríbalas y presione la tecla Intro. Obtendrá los resultados de inmediato, a diferencia del modo script. Por ejemplo, para imprimir el texto «Hola mundo», podemos escribir lo siguiente:

```
>>> print("Hello World")
Hello World
>>>
```

Usando archivo

La instalación de Python incluye un IDE (un programa en donde escribir el código) llamado IDLE, muy básico, pero que nos será útil en nuestros primeros pasos en el lenguaje. Si ya tienes tu editor de código favorito también puedes usarlo.

Para crear nuestro primer programa vamos a abrir IDLE y seleccionar el menú File > New File para crear un nuevo documento. Luego, escribiremos lo siguiente.

```
print("¡Hola, mundo!")
```

Este es un auténtico código de Python. ¡Solo una línea! En ella llamamos a la función incorporada print() y le pasamos una cadena de caracteres como argumento para que imprima en la pantalla. Se dice que es incorporada ya que es una herramienta que el lenguaje nos pone siempre a disposición en nuestros programas. Existen muchas otras que iremos conociendo en el camino.



Para poder ejecutar este pequeño código primero debemos guardarlo. Para ello, en IDLE vamos a ir al menú File > Save y lo guardaremos en el escritorio como hola.py.

Ahora bien, recordemos que Python es un lenguaje interpretado. Esto quiere decir que no hay un programa compilador que transforme nuestro código fuente (hola.py) y lo convierta en un archivo ejecutable (hola.exe, por ejemplo); más bien, hay un programa llamado intérprete al cual le indicamos que queremos ejecutar un archivo determinado. Todos los editores de código pueden hacer esto automáticamente (por ejemplo, en IDLE, presionando F5), no obstante, en este tutorial vamos a hacerlo de la forma manual, esto es, invocando al intérprete desde la terminal. Esto nos dará un panorama más amplio sobre cómo funciona todo en el mundo de Python.

Entonces, como decíamos, vamos a abrir la terminal. Todo sistema operativo tiene algún atajo para esto. En Windows, puedes presionar CTRL + R y escribir cmd, o bien buscar el programa de nombre "Símbolo del sistema". El primer paso es ubicarnos en la ruta en donde hemos guardado nuestro archivo (el escritorio) vía el comando cd. Hecho esto, ejecutamos nuestro script de Python escribiendo python o py seguido del nombre del archivo.

```
> cd Desktop  
> python hola.py  
¡Hola, mundo!
```



INSTALACIÓN DE JUPYTER LAB



Jupyter Notebook y JupyterLab son herramientas de desarrollo de código abierto muy poderosas para trabajar interactivamente con múltiples lenguajes de programación y será muy útiles para el análisis de datos, el análisis exploratorio, la visualización de datos, etc.

Jupyter Notebook y JupyterLab se pueden instalar en cualquier sistema operativo (Windows, Linux, Mac).

Cómo Instalar Jupyter Lab

Prerequisitos

Es necesario que su computadora tenga instalado Python, y este se encuentre agregado al PATH de Windows. Además, para la instalación es necesario que tenga acceso a Internet.

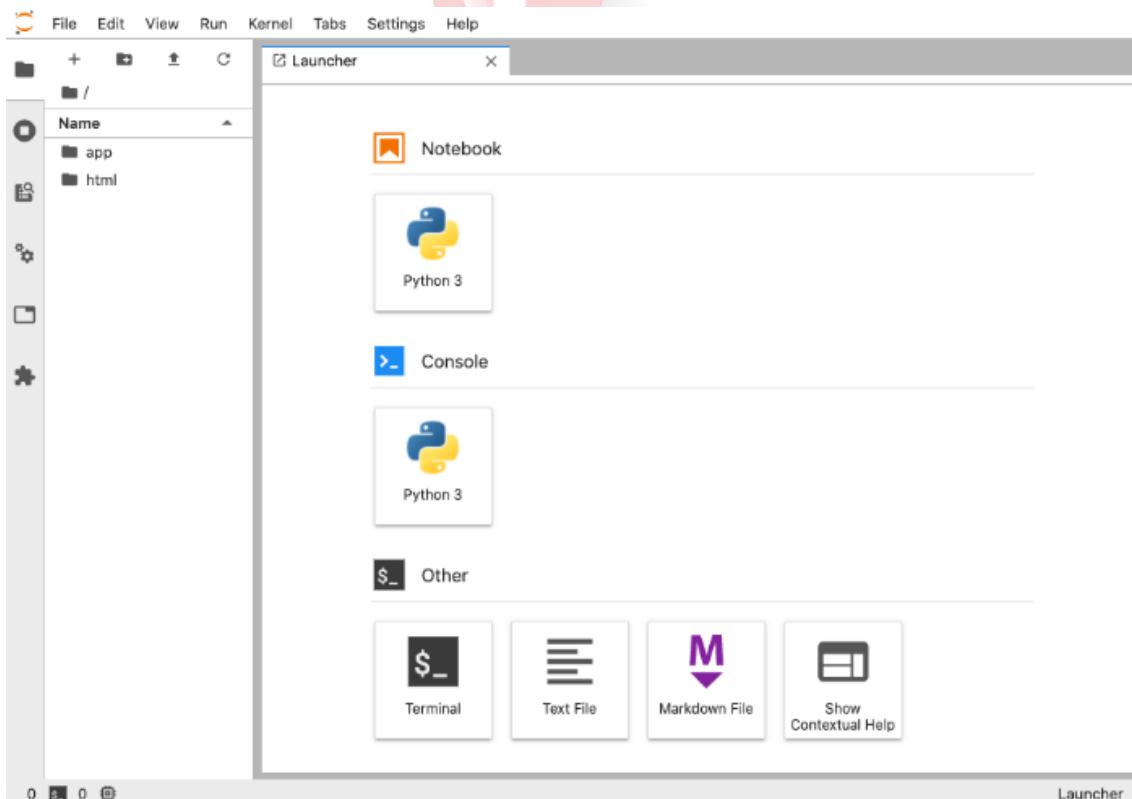


Instalación

- Abrir consola de comandos de Windows. En la barra de búsqueda de Windows, escribir cmd y seleccionar Símbolo del sistema.
- En la línea de comandos escribir la siguiente instrucción: pip install **jupyter-lab** y presionar Enter.
- Python descargará e instalará las librerías necesarias para utilizar jupyter.

Abrir la aplicación web

Una vez desplegado el servidor con JupyterLab instalado, podremos acceder a la aplicación web entrando con un navegador en la IP que se haya asignado al nuevo servidor cloud. Solicitará entonces la clave de acceso, que es la misma clave de root creada para el servidor. Posteriormente nos encontraremos con la interfaz de la aplicación, que tiene un aspecto como el que podemos ver en la siguiente imagen.



Crear un nuevo Notebook



Desde el Launcher, en la parte principal de la pantalla, podemos crear fácilmente un nuevo Notebook, pulsando el botón «Python 3» que hay en la sección «Notebook». Un notebook Jupyter se distribuye a lo largo de diversas celdas, donde podemos generar el contenido con toda la gama de elementos soportados por Jupyter.

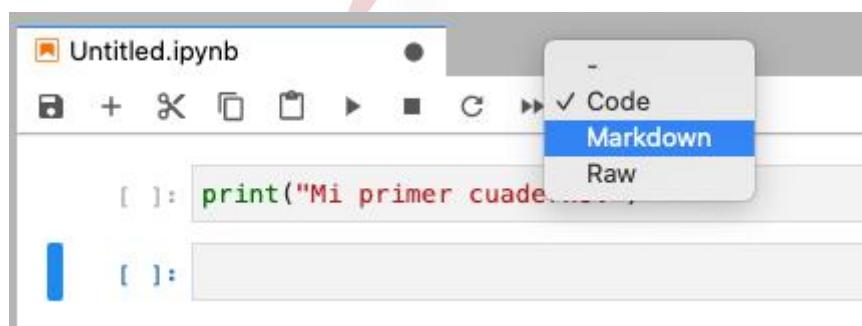
Modos de trabajo en notebooks

Para familiarizarnos en el sistema de creación de notebooks tenemos que entender los dos modos de trabajo que disponemos.

Modo comando: Nos permite crear celdas del notebook, copiar y pegar, insertar arriba, abajo, etc. Podemos activar las diversas opciones con el menú contextual que se abre al hacer clic derecho sobre el rectángulo azul de la celda activa. Dentro del modo comando, disponemos de diversos shortcuts básicos, entre otros:

- Cursor arriba y abajo, para cambiar la celda activa
- «A» para insertar antes
- «B» para insertar después
- «X» para cortar
- «C» para copiar
- «V» para pegar

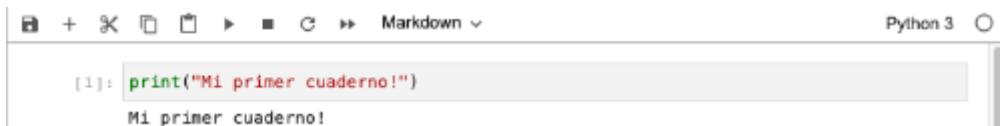
Modo edición: Accedemos a este modo haciendo doble clic sobre el rectángulo de contenido de una celda. Como hemos abierto un entorno de trabajo con Python, el contenido predeterminado que podremos escribir en las celdas es código Python. Sin embargo podemos cambiar el contenido a insertar con un desplegable en la parte de la barra de herramientas del notebook.



Una vez que tengamos el contenido preparado, podemos pulsar el botón de «play» para ejecutarlo. Veremos la salida justo en la parte de abajo de la celda



de código. En el caso de un bloque markdown, lo que aparecerá será el renderizado de ese contenido, mostrando las imágenes, enlaces y otros elementos que hayamos incluido.



A screenshot of a Jupyter Notebook interface. The top bar shows icons for file operations and a dropdown menu set to 'Markdown'. On the right, it says 'Python 3'. The main area contains a code cell with the command `[1]: print("Mi primer cuaderno!")`. Below the cell, the output is displayed as 'Mi primer cuaderno!'

Kernel

El «kernel» en la arquitectura de Jupyter se refiere al proceso de ejecución asociado a un documento. En este proceso se ejecutarán las sentencias de código incluidas en el documento.

Cada documento tiene un kernel de ejecución independiente. Sin embargo, dentro de un mismo documento de Jupyter se usa siempre el mismo kernel. Esto quiere decir que podemos escribir cualquier número de bloques de código en el documento y que todos ellos comparten el mismo flujo de ejecución. El código se ejecutará en el lenguaje que se haya escogido al crear el documento, que en nuestro caso será Python 3.

Aunque se declaren una vez en una celda, los bloques de código pueden ejecutarse varias veces, pulsando repetidas veces el botón de «play». Esto produce la ejecución repetida de las mismas sentencias de ese bloque en el kernel, manteniendo la memoria de todas las ejecuciones pasadas. Para ayudarnos a entender el flujo de ejecución de los bloques, a la izquierda de cada celda hay un número entre corchetes, que indica el orden de ejecución de los scripts de código.

En la siguiente imagen puedes ver como el valor de la variable «x» se puede compartir entre todas las celdas, ya que se ejecuta todo el código en el mismo kernel. Además, vemos que el orden de ejecución (mostrado en los corchetes de la izquierda) hace que, al mostrar el valor de la variable x en el bloque del medio, ésta tenga un valor de 30.

```
[1]: x = 10
      print(x)
      10

[4]: y = 20
      print(y)
      print(x)
      20
      30

[3]: x = x + 20
```

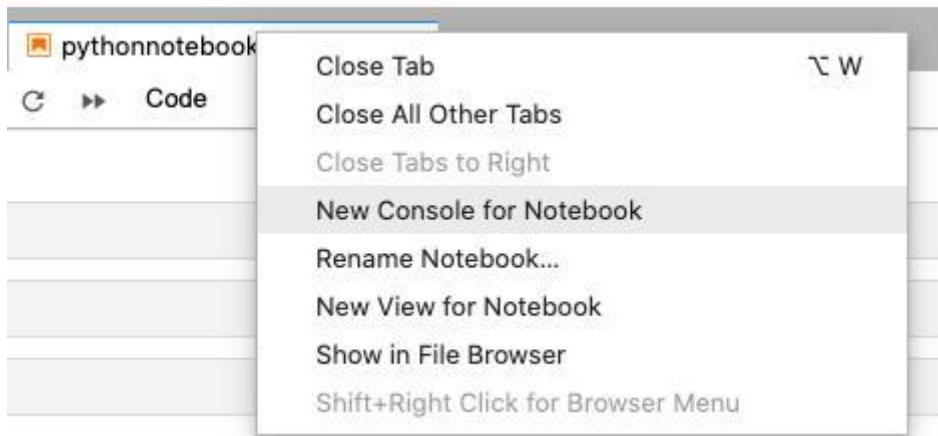
Para ayudarnos a controlar el flujo de ejecución de los bloques de código del documento existe un menú en la parte de arriba llamado «Kernel», que permite hacer acciones como reiniciar el kernel o reiniciar y eliminar todas las salidas anteriores.

Añadir otros Kernel a JupyterLab

Como mencionamos anteriormente, existen diversos lenguajes con los que podemos trabajar en los notebooks de Jupyter. En principio tenemos solamente disponible Python, pero podríamos usar kernels de otros lenguajes como R, Scala, Javascript... Para conseguir esta utilidad necesitamos, no obstante, instalar el soporte al kernel deseado por medio de la línea de comandos del servidor. Generalmente los distintos kernel disponibles son ofrecidos por terceros desarrolladores y las instrucciones de instalación pueden variar. Por ejemplo, para instalar el kernel de Javascript podemos dirigirnos a este repositorio de GitHub, donde encontraremos las instrucciones detalladas.

Code Consoles

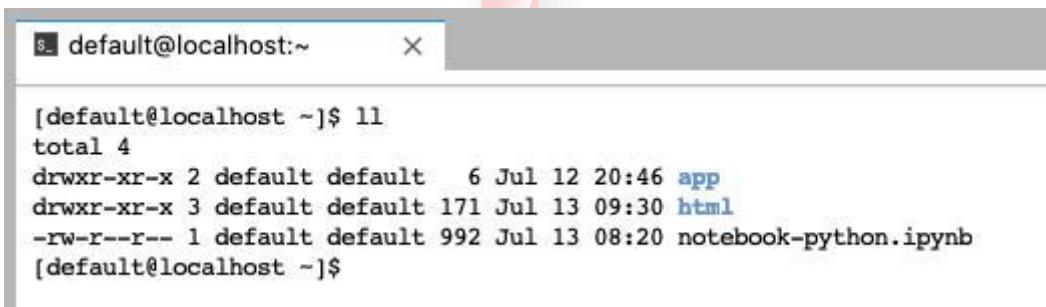
Otra de las actividades que podemos realizar dentro de JupyterLab es trabajar con consolas de código, que permiten escribir y ejecutar código en un lenguaje de programación en modo interactivo. Desde el menú «File > New > Console» podemos crear una nueva sesión con la consola. Podremos ver entonces el intérprete de Python y escribir código en un campo de texto. Para ejecutar el código pulsamos la combinación de teclas «Shift + Enter». También podemos abrir una consola que comparta el mismo kernel que un Jupyter notebook. Una manera rápida de conseguirlo es desde el menú contextual que aparece al hacer clic con el botón derecho en la pestaña del documento que deseemos usar como kernel de base.



Terminales

Desde JupyterLab también podemos abrir terminales de línea de comandos que se ejecutarán dentro del sistema operativo donde se encuentra instalada la aplicación.

Podemos abrir un terminal nuevo desde «File > New > Terminal». Entonces veremos que aparece una ventana donde podemos ejecutar comandos de consola sobre el sistema operativo. En la consola podremos encontrar las mismas carpetas y archivos que hemos generado dentro de la misma aplicación web.



```
[default@localhost ~]$ ll
total 4
drwxr-xr-x 2 default default 6 Jul 12 20:46 app
drwxr-xr-x 3 default default 171 Jul 13 09:30 html
-rw-r--r-- 1 default default 992 Jul 13 08:20 notebook-python.ipynb
[default@localhost ~]$
```



PYTHON EN LA NUBE

Google Colaboratory

Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de las GPUs y TPUs de Google, con librerías como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Todo ello con bajo Python 2.7 y 3.6, que aún no está disponible para R y Scala.

Aunque tiene algunas limitaciones, que pueden consultarse en su [página de FAQ](#), es una herramienta ideal, no solo para practicar y mejorar nuestros conocimientos en técnicas y herramientas de Data Science, sino también para el desarrollo de aplicaciones (pilotos) de machine learning y deep learning, sin tener que invertir en recursos hardware o del Cloud.

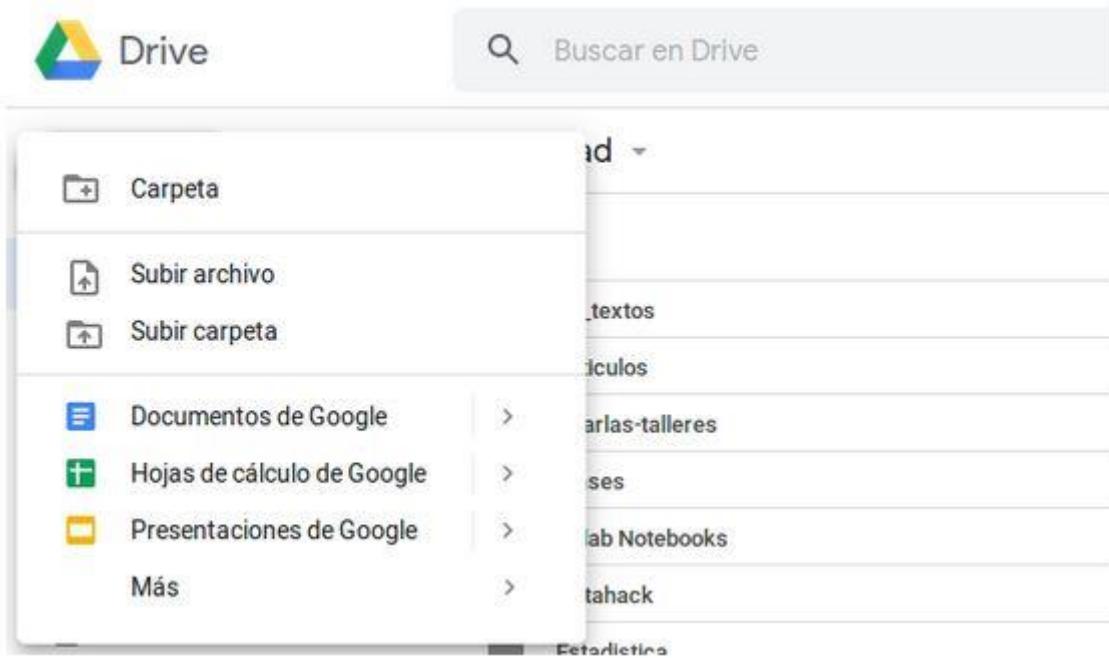
Con Colab se pueden crear notebooks o importar los que ya tengamos creados, además de compartirlos y exportarlos cuando queramos. Esta fluidez a la hora de manejar la información también es aplicable a las fuentes de datos que usemos en nuestros proyectos (notebooks), de modo que podremos trabajar con información contenida en nuestro propio Google Drive, unidad de



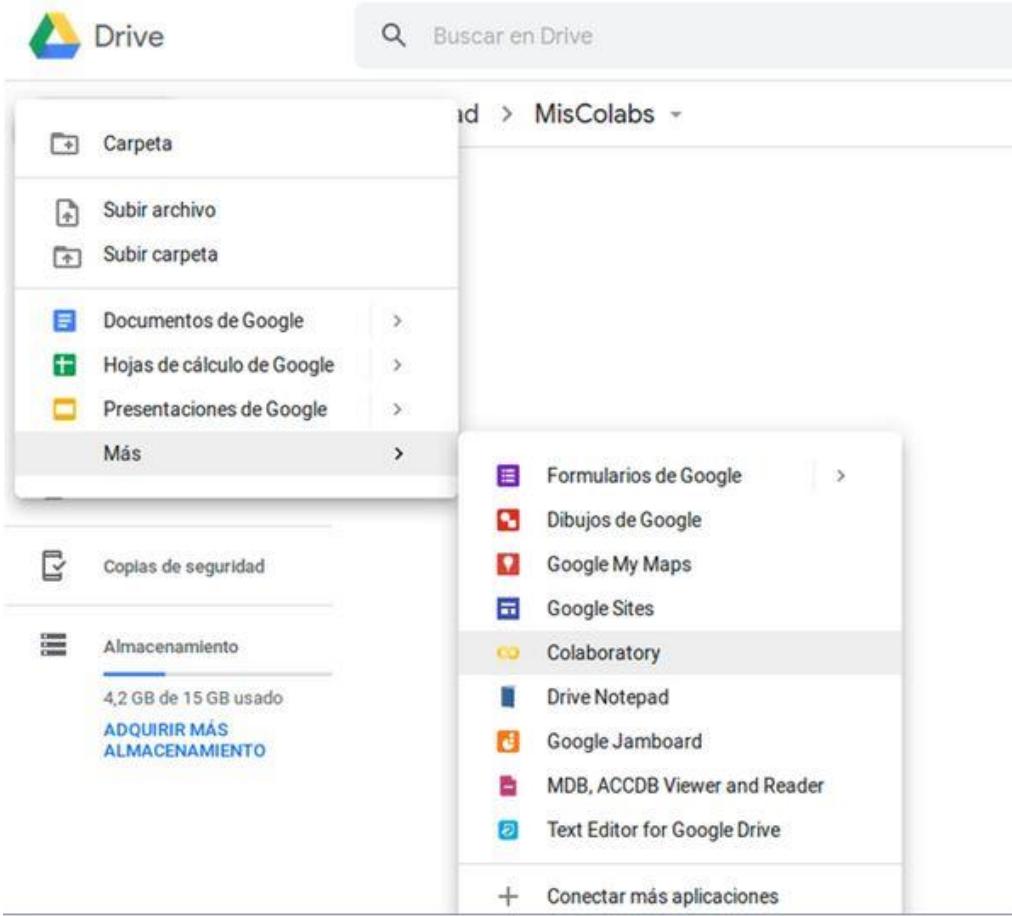
almacenamiento local, github e incluso en otros sistemas de almacenamiento cloud, como el S3 de Amazon.

Empezando a trabajar con Colab

Para poder tener nuestro espacio de trabajo en Colab, tendremos que tener una cuenta de google y acceder al servicio de Google Drive. Una vez dentro, le daremos a Nuevo > Carpeta, poniéndole el nombre que consideremos, por ejemplo: "MisColabs".



Para crear nuestro primer Colab, entraremos dentro de la carpeta que hemos creado y daremos a Nuevo > Más > Colaboratory, a continuación se abrirá un nuevo notebook.

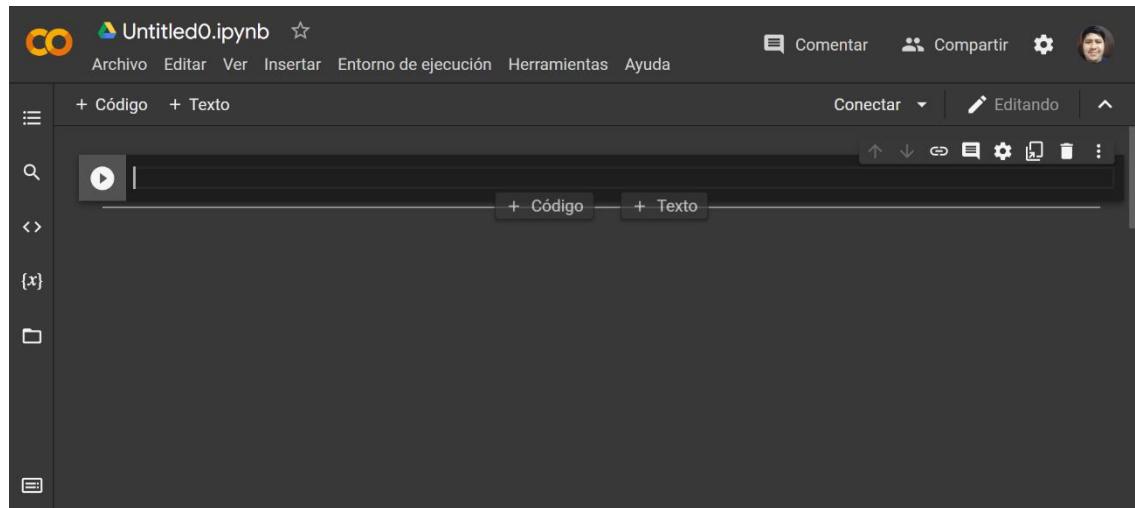


The screenshot shows the Google Drive interface. A context menu is open over a folder named "MisColabs". The menu includes options like "Carpeta", "Subir archivo", "Subir carpeta", and "Documentos de Google", "Hojas de cálculo de Google", "Presentaciones de Google", "Más", "Copias de seguridad", and "Almacenamiento". The "Más" section is expanded, showing "Formularios de Google", "Dibujos de Google", "Google My Maps", "Google Sites", "Colaboratory" (which is highlighted), "Drive Notepad", "Google Jamboard", "MDB, ACCDB Viewer and Reader", and "Text Editor for Google Drive". At the bottom of the expanded menu, there is a "+ Conectar más aplicaciones" option.

Otra opción sería ir directamente a [Google Colab](#).

Lo siguiente sería cambiar el nombre del notebook, haciendo clic en el nombre del notebook (esquina superior-izquierda) o yendo al menú Archivo > Cambiar

nombre.



Una vez hecho esto, hay que establecer el entorno de ejecución: menú Entorno de ejecución > Cambiar tipo de entorno de ejecución, tras lo que se abrirá la siguiente ventana:

En la mismo indicaremos las versión de Python (2 ó 3) y la unidad de procesamiento que se usará para ejecutar el código del Notebook: CPU (None), GPU ó TPU.



INSTALACIÓN DE VISUAL CODE EN MAC



Visual Studio Code es un editor de código ligero con soporte para muchos lenguajes de programación a través de extensiones

Instalación

Para instalar la última versión, use Homebrew:

```
brew cask install visual-studio-code
```

integración macOS

Inicie VS Code desde la línea de comando.

Después de eso, puede iniciar VS Code desde su terminal:

```
code . abrirá VS Code en el directorio actual
```

```
code myfile.txtse abrirá myfile.txten VS Code
```



Python

Python: resaltado de código Python

Para habilitar el formato automático en "Guardar", es decir ⌘ + S, configure lo siguiente:

1. Cambie el formateador predeterminado a en Black lugar de Autopep8. Crítico para evitar grandes diferencias. Vaya a Preferencias -> Configuración de usuario y actualice la configuración `python.formatter.provider` a Black
2. Habilitar Format on Save configuración: Editor: Formato al guardar configuración en Código -> Preferencias -> Configuración



VARIABLES



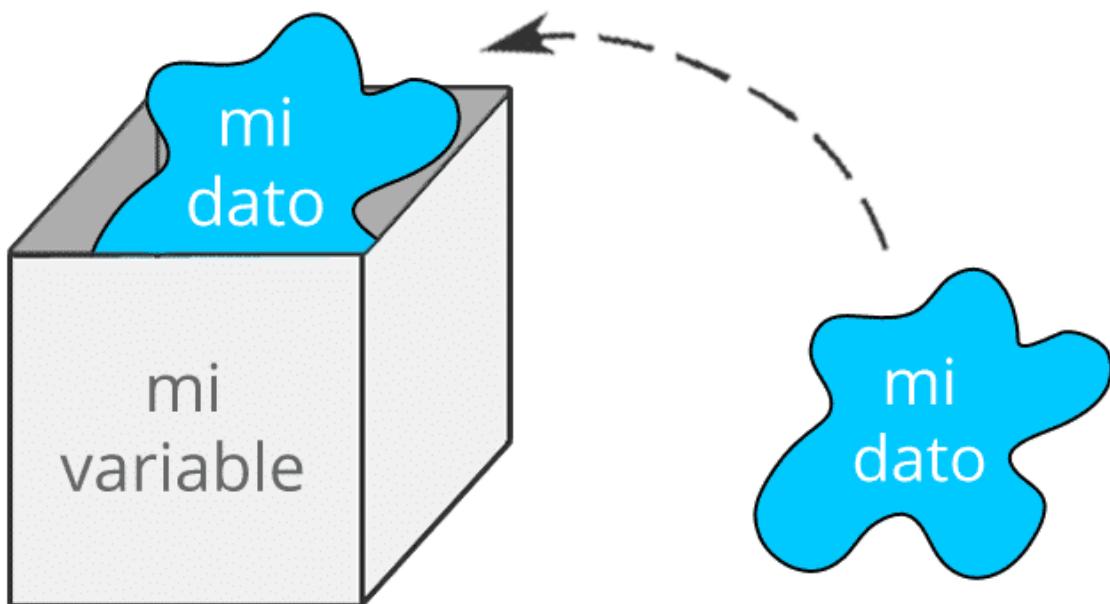
International Business Machines Corporation, conocida mundialmente como IBM, empresa especializada en tecnología y consultoría, propone una cómo definir una variable en programación:

Una variable en programación es un elemento de datos con nombre cuyo valor puede cambiar durante el curso de la ejecución de un programa. El nombre de la variable debe seguir el convenio de denominación de un identificador (carácter alfabético o número y el signo de subrayado). Cuando se define más de una variable en una sola declaración, el nombre debe ir separado por comas. Cada declaración de variable debe finalizar con un signo de punto y coma. Los nombres de variables no pueden coincidir con una palabra reservada.

En palabras más sencillas, si queremos explicar qué es una variable en programación podemos decir que es una unidad de almacenamiento y recuperación de datos con valores que pueden cambiar, la cual se identifica con un nombre único en el código del programa.



Si utilizáramos una analogía para describir este concepto, diríamos que una variable en programación es la base de la pirámide, ya que sostiene los componentes fundamentales de todos los programas computacionales.



Características de una variable en programación

La plataforma de investigación, Lifeder, reconoce que las variables en programación cuentan con tres características principales.

1. Asignación de memoria

Cuando definimos qué es una variable de programación, hicimos énfasis en que es un elemento del coding para guardar información, como valores y datos. En la programación, la computadora le asigna a la variable una posición en su memoria, en función al tipo de datos que almacene la misma.

En otras palabras, una variable en programación es un lugar en la memoria del ordenador, ya que, cuando este ejecuta un programa o una aplicación, la variable tendrá acceso a este bloque de memoria.

2. Declaración



Cuando hablamos de cómo declarar una variable, nos referimos a asignarle un nombre y un tipo; de esta manera, el sistema le podrá otorgar una abstracción de la memoria para almacenar los datos y su valor.

Dependerá del tipo de lenguaje de programación para determinar si se necesita declarar la clasificación de la variable antes de usarla. Por ejemplo, en el caso de Javascript y Lenguaje C, si se necesita declarar; sin embargo, en otros lenguajes de programación, como Python, no.

3. Alcance

Según las características de las variables en programación, por medio del alcance se podrá determinar hasta dónde se puede leer o manipular la información o el valor de una variable.

Por ejemplo, en el caso de las variables globales, se tendrá mayor alcance, pues tienen la capacidad de funcionar a lo largo de todo el programa. Por otro lado, en cuanto a las variables locales, el alcance solo llega a su propia función.

Tipos de variables en programación

Una vez que hemos definido qué es una variable en programación, vamos a diferenciar sus diferentes tipos. Aquí te presentaremos dos clasificaciones distintas.

Tipos de valores en variables

En primer lugar, según el concepto de qué es una variable en programación, estos son elementos que se encargan de almacenar datos. Entonces, en el curso online de Fundamentos de Javascript, el profesor Sergio Agamez Negrete, divide las variables en programación según el tipo de valor que contienen.

1. Númeroica

Un ejemplo de variables de programación son las numéricas en las que, tal como su nombre lo indica, podemos almacenar números. No necesitas agregar comillas, sólo el dígito. Asimismo, si quieras guardar números decimales, solo necesitas utilizar el punto (.) entre los números enteros y los fraccionados, ya que hay lenguajes de programación en que las comas (,) se usan para añadir otros datos a la variable.



2. Cadena de texto

En el caso de una variable en programación que almacena texto, este necesita estar entre comillas. Pueden ser comillas simples o dobles, pero se recomienda el uso de las sencillas, porque hay momentos en los que las comillas dobles no se pueden utilizar.

Existen dos tipos de cadena de texto: cadena de longitud fija y cadena de longitud variable. En la primera, se estipula cuántos caracteres va a contener, mientras que la segunda no define la extensión.

3. Booleana

¿Qué es una variable en programación con valor booleano? Este término se refiere a un valor lógico, que, en el caso de la programación, representa la dicotomía de verdadero y falso. Usualmente, para representar este valor se coloca “true” o “false”, haciendo alusión en inglés a verdadero y falso, respectivamente.

4. Elementos HTML

También, puedes utilizar una variable en programación para seleccionar elementos dentro del HTML de tu programa o aplicación. Solo necesitas colocarles un identificador (id). Además, en el caso de Javascript, necesitas utilizar un selector para asignar los elementos a la variable. Esto último, depende del lenguaje de programación y el software que utilices.

5. Arreglos

¡Crear estas variables de programación es muy fácil! Un arreglo es, básicamente, una variable donde puedes almacenar más de un elemento, incluyendo otras variables. Solo necesitas redactar la lista entre corchetes [] y separar cada elemento con una coma.

Para referirse a un solo elemento del arreglo se debe hacer desde su índice, el cual se le asigna de forma automática. El índice consta de los ítems del arreglo numerados desde el 0. Entonces, si quieras seleccionar solo un elemento de esta variable en programación, necesitas seleccionar el dígito que le corresponde por su posición.



6. Objetos

¿Qué es una variable en programación objeto? ¡Es muy similar a la anterior! La diferencia es que se va a colocar entre llaves {} y dentro de esta se pueden agregar propiedades y valores. Por ejemplo, {color: verde}, en el cual color corresponde a la propiedad y verde al valor.

7. Funciones

Finalmente, una variable en programación también puede almacenar funciones. ¿Qué es una función? Dentro de la disciplina de la programación web, una función se define como una forma en que los algoritmos y expresiones se agrupan en códigos simbólicos para determinar acciones. En este caso, para ejecutar la variable se necesita colocar la palabra reservada function(){}.

Declarando una variable

Se le llama declaración a la creación de una nueva variable, mientras que en algunos lenguajes la variable se debe crear primero y posteriormente añadirle contenido, en Python la variable se declara asignándole un valor, en otras palabras no es necesario declararlas previamente, con solo asignarle algún valor se creará en el momento.

Declaración de una variable en C++

```
#include <iostream>

using namespace std;

int main() {

    int entero = 15;
    float flotante = 10.45;
    double mayor = 14.24424;
    char letra = 'a';

    return 0;
}
```

Declaración de una variable en Python

```
variable1 = 5 # Esta será una variable de tipo entero, el intérprete
de Python lo entiende
```



```
Variable2 = "Curso de Python Nivel intermedio en Codigazo"
```

```
# La variable2 será de tipo texto
```





MOSTRANDO DATOS



En Informática, la "salida" de un programa son los datos que el programa proporciona al exterior. Aunque en los inicios de la informática la salida más habitual era una impresora, hace muchos años que el dispositivo de salida más habitual es la pantalla del ordenador.

Salida por pantalla en el entorno interactivo

En un entorno interactivo (por ejemplo, en IDLE), para que python nos muestre el valor de una variable basta con escribir su nombre.

```
>>> a = 2
>>> a
2
```

También se puede conocer el valor de varias variables a la vez escribiéndolas entre comas (el entorno interactivo las mostrará entre paréntesis), como muestra el siguiente ejemplo:



```
>>> a = b = 2
>>> c = "pepe"
>>> a
2
>>> c, b
('pepe', 2)
```

La función print()

En los programas, para que python nos muestre texto o variables hay que utilizar la función print().

La función print() permite mostrar texto en pantalla. El texto a mostrar se escribe como argumento de la función:

```
print("Hola")
```

```
Hola
```

Las cadenas se pueden delimitar tanto por comillas dobles ("") como por comillas simples ('').

```
print('Hola')
```

```
Hola
```

Nota: En estos apuntes se utilizan normalmente comillas dobles.

La función print() admite varios argumentos seguidos. En el programa, los argumentos deben separarse por comas. Los argumentos se muestran en el mismo orden y en la misma línea, separados por espacios:

```
print("Hola", "Adiós")
```

```
Hola Adiós
```

Cuando se trata de dos cadenas seguidas, se puede no escribir comas entre ellas, pero las cadenas se escribirán seguidas, sin espacio en blanco entre ellas:

```
print("Hola" "Adiós")
```

```
HolaAdios
```

Al final de cada print(), Python añade automáticamente un salto de línea:

```
print("Hola")
print("Adiós")
Hola
Adiós
```

Para generar una línea en blanco, se puede escribir una orden print() sin argumentos.

```
print("Hola")
print()
print("Adiós")
Hola

Adiós
```

Si se quiere que Python no añada un salto de línea al final de un print(), se debe añadir al final el argumento end="":

```
print("Hola", end="")
print("Adiós")
HolaAdios
```

En el ejemplo anterior, las dos cadenas se muestran pegadas. Si se quieren separar los argumentos en la salida, hay que incluir los espacios deseados (bien en la cadena, bien en el argumento end):

```
print("Hola. ", end="")
print("Adiós")
Hola. Adiós

print("Hola.", end=" ")
print("Adiós")
Hola. Adiós
```

El valor del parámetro end puede ser una cadena f:

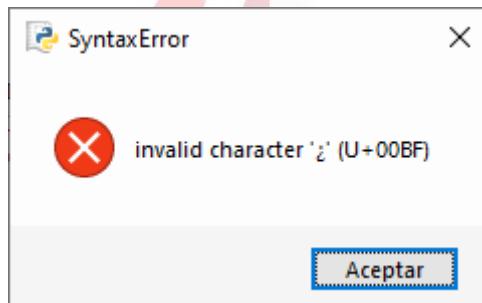
```
texto = " y "
print("Hola", end=f"{texto}")
print("Adiós")
Hola y Adiós
```



Como las comillas indican el principio y el final de una cadena, si se escriben comillas dentro de comillas se produce un error de sintaxis.

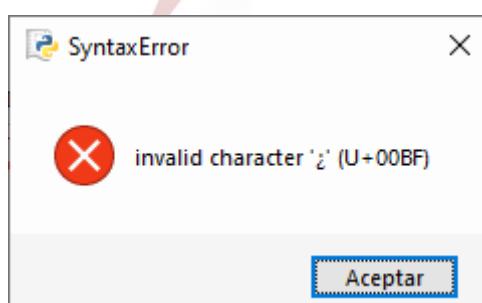
```
print("Un tipo le dice a otro: "¿Cómo estás?")  
File "prueba.py", line 1  
    print("Un tipo le dice a otro: "¿Cómo estás?")  
                                ^  
SyntaxError: invalid character in identifier
```

En IDLE se muestra una ventana indicando el error:



```
print('Un tipo le dice a otro: '¿Cómo estás?')  
File "prueba.py", line 1  
    print('Un tipo le dice a otro: '¿Cómo estás?')  
                                ^  
SyntaxError: invalid character in identifier
```

En IDLE se muestra una ventana indicando el error:



Nota: Si nos fijamos en la forma como el editor colorea la instrucción, podemos darnos cuenta de que hay un error en ella. Como las cadenas empiezan y acaban



con cada comilla, el editor identifica dos cadenas y un texto en medio que no sabe lo que es.

Para incluir comillas dentro de comillas, se puede escribir una contrabarra (\) antes de la comilla para que Python reconozca la comilla como carácter, no como delimitador de la cadena:

```
print("Un tipo le dice a otro: \"¿Cómo estás?\"")  
print('Y el otro le contesta: \'¡Pues anda que tú!\'')  
Un tipo le dice a otro: "¿Cómo estás?"  
Y el otro le contesta: '¡Pues anda que tú!'
```

O escribir comillas distintas a las utilizadas como delimitador de la cadena:

```
print("Un tipo le dice a otro: '¿Cómo estás?'")  
print('Y el otro le contesta: "¡Pues anda que tú!"')  
Un tipo le dice a otro: '¿Cómo estás?'  
Y el otro le contesta: "¡Pues anda que tú!"
```

La función print() permite incluir variables o expresiones como argumento, lo que nos permite combinar texto y variables:

```
nombre = "Pepe"  
edad = 25  
print("Me llamo", nombre, "y tengo", edad, "años.")  
Me llamo Pepe y tengo 25 años.  
semanas = 4  
print("En", semanas, "semanas hay", 7 * semanas, "días.")  
En 4 semanas hay 28 días.
```

La función print() muestra los argumentos separados por espacios, lo que a veces no es conveniente. En el ejemplo siguiente el signo de exclamación se muestra separado de la palabra.

```
nombre = "Pepe"  
print("¡Hola,", nombre, "!")  
¡Hola, Pepe !  
nombre = "Pepe"  
print(f"¡Hola, {nombre}!")
```



¡Hola, Pepe!





SALTOS DE LINEA - FSTRING

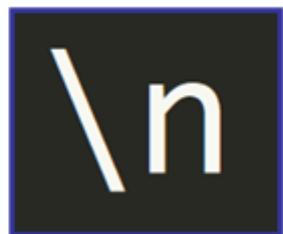
Salto de línea

El carácter de salto de línea es usado en Python para marcar el fin de una línea y el comienzo de una nueva línea. Saber cómo usarlo es esencial si quieres mostrar (imprimir) mensajes y trabajar con archivos y cadenas de caracteres en Python.

Línea

Este es el carácter de salto de línea en Python:





Está compuesto por dos caracteres:

- Una barra invertida \
- La letra n

Si encuentras este carácter en una cadena de caracteres en Python, esto representa el fin de la línea actual y el inicio de una linea nueva:



También puedes usar este carácter en las f-string (un tipo específico de cadenas de caracteres en Python):

```
print(f"¡Hola \n Mundo!")
```

Salto de Línea en Llamadas a print()

Por defecto, las llamadas a la función print() añaden un salto de línea al final de la cadena de caracteres.

Esto es lo que ocurre detrás de escena:

```
print("¡Hola, Mundo!")
```



```
"¡Hola, Mundo!\n"
```

Esto ocurre porque, según la [Documentación de Python](#), la función print() se define de esta forma:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Además, la documentación menciona que esta función:

Imprime objects al flujo de texto file, separándolos por sep y seguidos por end.

El valor por defecto del parámetro end es el carácter \n, como lo podemos ver en la definición de la función que se muestra arriba. Este carácter será añadido al final de la cadena de caracteres.

Si solo llamas a la función print() una vez, no notarás que se incluye este carácter porque solo se muestra una línea:

```
>>> print("¡Hola, Mundo!")
¡Hola, Mundo!
```

Pero si llamas a la función más de una vez en secuencia en un archivo Python:

```
1 print("¡Hola, Mundo!")  
2 print("¡Hola, Mundo!")  
3 print("¡Hola, Mundo!")  
4 print("¡Hola, Mundo!")  
5
```

Cada mensaje será mostrado en su propia línea porque \n ha sido añadido al final de cada línea:



```
¡Hola, Mundo!  
¡Hola, Mundo!  
¡Hola, Mundo!  
¡Hola, Mundo!
```

Qué son las f-Strings en Python?

Las cadenas en Python generalmente se encierran entre comillas dobles ("") o comillas simples (''). Para crear fstring, solo necesita agregar una f o una F antes de las comillas de apertura de la cadena.

Por ejemplo, "This" es una cadena mientras que f"This" es una f-String.

Cómo imprimir variables usando Python f-Strings

Cuando se utilizan f-Strings para mostrar variables, sólo es necesario especificar los nombres de las variables dentro de un conjunto de llaves rizadas. Y en tiempo



de ejecución, todos los nombres de variables serán reemplazados por sus respectivos valores. {}

Si tiene varias variables en la cadena, debe incluir cada uno de los nombres de las variables dentro de un conjunto de llaves rizadas.

La sintaxis se muestra a continuación:

```
f"This is an f-string {var_name} and {var_name}."
```

Aquí hay un ejemplo.

Tiene dos variables, language y school , encerrado en llaves dentro de la fstring.

```
language = "Python"  
school = "freeCodeCamp"  
print(f"I'm learning {language} from {school}.")
```

Echemos un vistazo a la salida:

```
#Output  
I'm learning Python from freeCodeCamp.
```

Observe cómo las variables language y school han sido reemplazadas por Python y freeCodeCamp, respectivamente.

Cómo evaluar expresiones con cadenas f de Python

Como las f-Strings se evalúan en tiempo de ejecución, también puede evaluar expresiones válidas de Python sobre la marcha.

En el siguiente ejemplo, num1 y num2 hay dos variables. Para calcular su producto, puede insertar la expresión num1 * num2 dentro de un conjunto de llaves.

```
num1 = 83  
num2 = 9  
print(f"The product of {num1} and {num2} is {num1 * num2}.")
```



Observe cómo se reemplaza por el producto de y en la salida. num1 *
num2num1num2

```
#Output
```

```
The product of 83 and 9 is 747.
```

Espero que ahora puedas ver el patrón.

En cualquier f-String, , sirven como marcadores de posición para variables y expresiones, y se reemplazan con los valores correspondientes en tiempo de ejecución.{var_name}{expression}

Dirígete a la siguiente sección para obtener más información sobre f-Strings.



COMENTARIOS



La capacidad de comentar el código fuente figura en todos los lenguajes de programación. Un comentario es una línea de texto no ejecutable, esto quiere decir que el compilador o intérprete no la tomará como una línea de código. Los comentarios en Python, así como en otros lenguajes de programación, sirven para dejar pequeñas explicaciones sobre qué es lo que hace el programa. Sabemos que es muy difícil recordar cada aspecto de nuestro programa, sobre todo cuando trabajamos en proyectos largos y complicados. Python, al ser un lenguaje sencillo en cuestiones de sintaxis, nos ayuda a documentar apropiadamente nuestro código sin mucho esfuerzo. Esta es una práctica necesaria y los buenos desarrolladores harán un gran uso de los comentarios. Sin esto, el código fuente puede volverse confuso realmente rápido.

En Python hacer comentarios de dos formas:

- Escribiendo el símbolo de numeral (#) al comienzo de la línea de texto donde queremos nuestro comentario.
- Escribiendo triple comilla (') al principio y al final del comentario, en este caso los comentarios pueden ocupar mas de una línea.



Veamos un par de ejemplos:

Comentarios de una sola línea en Python.

```
# Esto es un comentario en Python  
  
>>> funcion_1()  
  
>>> funcion_2() # Los comentarios también pueden estar después de un texto ejecutable  
  
# Este es otro comentario en Python.  
  
# Puedes tener tantos comentarios de línea simple seguidos.
```

Comentarios multilínea en Python.

```
"""Este es un comentario multilínea.  
  
Podemos escribir tantas líneas queramos a modo de documentación."""  
  
funcion_3()  
  
funcion_4()  
  
'''También podemos hacer comentarios multilíneas con comillas simples.'''
```

Docstrings

Python cuenta con otra herramienta para ayudar a los desarrolladores a documentar su código, las docstrings. Estas son simplemente comentarios multilínea ubicados después de la declaración de una función. Estas proveen una forma conveniente de documentar funciones, módulos, clases y métodos en Python.

Ejemplo:

```
def funcion_1(a, b):  
  
    """Esta función retorna la suma de ambos argumentos."  
  
    return a + b
```

Ten en cuenta algunas convenciones al hacer comentarios en tu código fuente en Python:



- No satures tu código con comentarios de una sola línea.
- Los comentarios de una sola línea deben estar separados al menos dos espacios del signo de comentario (#).
- Úsalos cuando de verdad los creas necesario, no comentes código que es fácil de entender.

Ejemplo de comentarios en Python en un caso real

Veamos directamente el uso de los docstrings y comentarios en Python. El siguiente script le pide al usuario que ingrese el primer nombre de un estudiante en clase, hasta que se ingrese una string vacía. Luego la función imprime el número de estudiantes con cada nombre que se ingresó.

```
def nombres():
    '''Pide al usuario que ingrese el primer nombre de
    un estudiante en clase
    hasta que se ingrese una string vacía. Luego la
    función imprime el número
    de estudiantes con cada nombre que se ingresó.
    '''

    nombre = ''
    nombres = []
    nombre = input('Enter next nombre: ')

    while nombre != '': # Mientras que nombre no sea
        una string vacia
        nombres.append(nombre)
        nombre = input('Enter next nombre: ')

    # Creamos un diccionario con la estructura
    nombre:ocurrencias
    ocurrencias_nombre = {n:nombres.count(n) for n in
    nombres}

    # Iteramos a traves de las claves del diccionario
    for k in ocurrencias_nombre.keys():
        singular = True if ocurrencias_nombre[k] < 2
    else False
```



```
if singular:  
    print('Hay {} estudiante llamado  
{})'.format(ocurrencias_nombre[k], k))  
else:  
    print('Hay {} estudiantes llamados  
{})'.format(ocurrencias_nombre[k], k))  
  
nombres()
```

Debemos mantener nuestro código bien documentado y solo con comentarios que sean necesarios. Recuerda siempre agregar una docstring a tus funciones para tener siempre claro su objetivo.





ASIGNACIÓN MULTIPLE



Las asignaciones múltiples son un sistema en el que podremos asignar 2 o más valores en 2 o más variables en una sola línea de código, de forma que podremos tener un código más limpio y compacto.

Casos de uso de asignaciones múltiples

Usar asignaciones múltiples con listas

Por ejemplo, podemos tener una lista de 3 valores que queramos asignar en tres variables distintas. Para asignarlas, tendremos que escribir en la misma línea los nombres de las tres variables separadas por comas, después el signo de igualdad y por último la lista:

```
number_list = [4, 1, 9]
```

```
a, b, c = number_list
```



Los valores se asignarán en orden. En este caso, 'a' tendrá un valor de 4, 'b' de 1 y 'c' de 9. En el caso de que tengamos menos variables a las que asignar todos los valores de la lista, estos valores quedarán sin asignar, en el caso contrario, es decir, declarar más variables que valores tiene la lista, se lanzará una excepción:

```
number_list = [4, 1, 9]

a, b, c, d = number_list
Traceback (most recent call last):
  File "asignaciones-multiples.py", line 4, in <module>
    a, b, c, d = number_list
ValueError: not enough values to unpack (expected 4, got 3)
```

Usar asignaciones múltiples en Sets

También podemos usar las asignaciones múltiples en sets de igual forma que con las listas:

```
fruit_list = {'apple', 'orange', 'lemon'}
```



```
apple, orange, lemon = fruit_list
```

Usar asignaciones múltiples en Tuplas

Al igual que en listas y sets, también podemos usarlas con tuplas:

```
books = ('El imperio final', 'El pozo de la ascensión',
         'El heroe de las eras')

book1, book2, book3 = books
```



Usar asignaciones múltiples en Diccionarios

Si tenemos un diccionario y queremos asignar sus claves en variables, podremos hacerlo como en los ejemplos anteriores:

```
user = {  
    'name': 'Alberto',  
    'age': 33  
}  
  
name, age = user  
  
print(name) # Resultado: name  
print(age) # Resultado: age
```

En el caso de que lo que queramos obtener sean los valores y no las claves, podemos usar la función values() de nuestro diccionario para obtener una lista con los valores y después asignarla a las variables:

```
user = {  
    'name': 'Alberto',  
    'age': 33  
}  
  
name, age = user.values()  
  
print(name) # Resultado: Alberto  
print(age) # Resultado: 33
```



Asignaciones con funciones y métodos

Python nos permite retornar más de un valor en nuestras funciones y métodos. En este caso también podemos utilizar las asignaciones múltiples para recuperar los valores:

```
def get_profile():
    username = 'Alber'
    email = 'alber@cosasdedebs.com'
    photo = '../files/photo.png'

    return username, email, photo

username, email, photo = get_profile()
```

Asignaciones con el operador *

Puede aparecer el caso en el que tengamos una lista en la que queramos asignar algunos valores en variables y el resto sobrante guardarla en una sola lista. Para asignar ese resto podemos utilizar el operador * de la siguiente forma:

Caso 1

```
number_list2 = [4, 3, 0, 1, 5]
```

```
x, *y, z = number_list2
```

Resultado:

```
4
[3, 0, 1]
5
```

En este caso, el primer valor se asigna a la variable 'x' y el último en la variable 'z'. Los valores restantes se guardarán en la variable 'y' en forma de lista.

Caso 2



```
number_list2 = [4, 3, 0, 1, 5]
```

```
*x, y, z = number_list2
```

Resultado:

```
[4, 3, 0]  
1  
5
```

En este ejemplo, 'y' y 'z' guardan los dos últimos valores de la lista respectivamente y en 'x' se guarda el resto.

Caso 3

```
number_list2 = [4, 3, 0, 1, 5]
```

```
x, y, *z = number_list2
```

Resultado:

```
4  
3  
[0, 1, 5]
```

En este caso, 'x' e 'y' guardan los dos primeros valores respectivamente y el resto se almacenará en la variable z.



ENTRADA DE DATOS

Función input()

La función `input()` permite a los usuarios introducir datos de distintos tipos desde la entrada estándar (normalmente se corresponde con la entrada de un teclado).

```
print('Ingresa tu nombre: ')
x = input()
print('Hola, ' + x)
```

Cuando recibimos los datos mediante la función `input` los recibimos en tipo cadena, es muy importante saberlo pues si queremos realizar operaciones matemáticas debemos convertirlo.



```
print('Ingresa 2 numeros:')

x = int(input())
y = int(input())
suma = y + x
print(suma)
```

También podemos ingresar el mensaje dentro del input().

```
x = input('Ingresa tu nombre: ')
print('Hola, ' + x)
```





PROGRAMACIÓN ESTRUCTURADA



La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: secuencia, selección (if y switch) e iteración (bucles for y while); asimismo, se considera innecesario y contraproducente el uso de la instrucción de transferencia incondicional (GOTO), que podría conducir a código espagueti, mucho más difícil de seguir y de mantener, y fuente de numerosos errores de programación.

Surgió en la década de 1960, particularmente del trabajo de Böhm y Jacopini,¹ y un famoso escrito de 1968: «La sentencia goto, considerada perjudicial», de Edsger Dijkstra. Sus postulados se verían reforzados, a nivel teórico, por el teorema del programa estructurado y, a nivel práctico, por la aparición de lenguajes como ALGOL, dotado de estructuras de control consistentes y bien formadas.



Fundamentación teórica

El teorema del programa estructurado proporciona la base teórica de la programación estructurada. Señala que la combinación de las tres estructuras básicas, secuencia, selección e iteración, son suficientes para expresar cualquier función computable. Esta observación no se originó con el movimiento de la programación estructurada. Estas estructuras son suficientes para describir el ciclo de instrucción de una unidad central de procesamiento, así como el funcionamiento de una máquina de Turing. Por lo tanto, un procesador siempre está ejecutando un «programa estructurado» en este sentido, incluso si las instrucciones que lee de la memoria no son parte de un programa estructurado. Sin embargo, los autores usualmente acreditan el resultado a un documento escrito en 1966 por Böhm y Jacopini, posiblemente porque Dijkstra había citado este escrito. El teorema del programa estructurado no responde a cómo escribir y analizar un programa estructurado de manera útil. Estos temas fueron abordados durante la década de 1960 y principio de los años 1970, con importantes contribuciones de Dijkstra, Robert W. Floyd, Tony Hoarey y David Gries.

Debate

P. J. Plauger, uno de los primeros en adoptar la programación estructurada, describió su reacción con el teorema del programa estructurado:

Nosotros los conversos ondeamos esta interesante pizca de noticias bajo las narices de los recalcitrantes programadores de lenguaje ensamblador que mantuvieron trotando adelante retorcidos bits de lógica y diciendo, 'Te apuesto que no puedes estructurar esto'. Ni la prueba por Böhm y Jacopini, ni nuestros repetidos éxitos en escribir código estructurado, los llevaron un día antes de lo que estaban listos para convencerse.

Donald Knuth aceptó el principio de que los programas deben adaptarse con asertividad, pero no estaba de acuerdo (y aún está en desacuerdo)[cita requerida] con la supresión de la sentencia GOTO. En su escrito de 1974 «Programación estructurada con sentencias Goto», dio ejemplos donde creía que un salto directo conduce a código más claro y más eficiente sin sacrificar demostratividad. Knuth propuso una restricción estructural más flexible: debe ser posible establecer un diagrama de flujo del programa con todas las bifurcaciones hacia adelante a la izquierda, todas las bifurcaciones hacia atrás a la derecha, y sin bifurcaciones que se crucen entre sí. Muchos de los expertos en teoría de



grafos y compiladores han abogado por permitir solo grafos de flujo reducible[¿quién?][¿cuándo?].

Los teóricos de la programación estructurada se ganaron un aliado importante en la década de 1970 después de que el investigador de IBM Harlan Mills aplicara su interpretación de la teoría de la programación estructurada para el desarrollo de un sistema de indexación para el archivo de investigación del New York Times. El proyecto fue un gran éxito de la ingeniería, y los directivos de otras empresas lo citaron en apoyo de la adopción de la programación estructurada, aunque Dijkstra criticó las maneras en que la interpretación de Mills difería de la obra publicada.

Habrá que esperar a 1987 para que la cuestión de la programación estructurada llamará la atención de una revista de ciencia de la computación. Frank Rubin lo hizo en ese año, con el escrito: «¿“La sentencia GOTO considerada dañina” se considera dañina?». A este le siguieron numerosas objeciones, como una respuesta del propio Dijkstra que criticaba duramente a Rubin y las concesiones que otros autores hicieron cuando le respondieron.

Características

- Los programas desarrollados con la programación estructurada son más sencillos de entender, ya que tienen una estructura secuencial y desaparece la necesidad de rastrear los complejos saltos de líneas (propios de la sentencia Goto) dentro de los bloques de código para intentar comprender la lógica interna.
- Como consecuencia inmediata de lo anterior, otra ventaja es que los programas resultantes tendrán una estructura clara, gracias a que las sentencias están ligadas y relacionadas entre sí.
- La fase de prueba y depuración de los programas se optimiza, ya que es mucho más sencillo hacer el seguimiento de los fallos y errores y, por tanto, detectarlos y corregirlos.
- El coste del mantenimiento de los programas que usan la programación estructurada es más reducido. ¿Por qué? Pues porque modificar o extender los programas es más fácil al estar formados por una estructura secuencial.
- Al ser más sencillos los programas, son más rápidos de crear y los programadores aumentan su rendimiento.

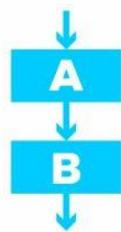


Las 3 estructuras básicas

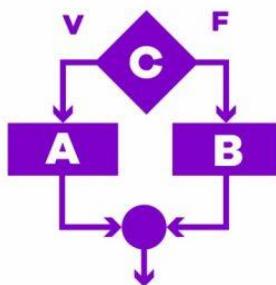
Ya nos ha quedado claro que la programación estructurada es una forma de programar más sencilla que se basa únicamente en la combinación de tres órdenes. Pero, ¿cuáles son esos tipos de estructuras de control que son capaces de expresarlo todo?

- **Secuencia.** La estructura secuencial es la que se da de forma natural en el lenguaje, porque las sentencias se ejecutan en el orden en el que aparecen en el programa, es decir, una detrás de la otra.
- **Selección o condicional.** La estructura condicional se basa en que una sentencia se ejecuta según el valor que se le atribuye a una variable booleana. ¡Un pequeño inciso! Una variable booleana es aquella que tiene dos valores posibles. Por tanto, esta estructura se puede ejecutar de dos formas distintas, dependiendo del valor que tenga su variable. Como apunte para los verdaderos amantes de la programación: para las estructuras condicionales o de selección, Python dispone de la sentencia if, que puede combinarse con elif y/o else.
- **Iteración (ciclo o bucle).** La estructura de repetición ejecuta una o un conjunto de sentencias siempre que una variable booleana sea verdadera. Para los bucles o iteraciones, los lenguajes de programación usan las estructuras while y for.

Secuencia



Selección o condicional



Iteración (ciclo o bucle)





TIPO NÚMERO



Python define tres tipos de datos numéricos básicos: enteros, números de punto flotante (simularía el conjunto de los números reales, pero ya veremos que no es así del todo) y los números complejos.

Números enteros

El tipo de los números enteros es int. Este tipo de dato comprende el conjunto de todos los números enteros, pero como dicho conjunto es infinito, en Python el conjunto está limitado realmente por la capacidad de la memoria disponible. No hay un límite de representación impuesto por el lenguaje.

Pero tranquilidad, que para el 99% de los programas que desarrolles tendrás suficiente con el subconjunto que puedes representar.

Un número de tipo int se crea a partir de un literal que represente un número entero o bien como resultado de una expresión o una llamada a una función.



Ejemplos:

```
>>> a = -1 # a es de tipo int y su valor es -1  
  
>>> b = a + 2 # b es de tipo int y su valor es 1  
  
>>> print(b)
```

1

También podemos representar los números enteros en formato binario, octal o hexadecimal.

Los números octales se crean anteponiendo 0o a una secuencia de dígitos octales (del 0 al 7).

Para crear un número entero en hexadecimal, hay que anteponer 0x a una secuencia de dígitos en hexadecimal (del 0 al 9 y de la A la F).

En cuanto a los números en binario, se antepone 0b a una secuencia de dígitos en binario (0 y 1).

```
>>> diez = 10  
  
>>> diez_binario = 0b1010  
  
>>> diez_octal = 0o12  
  
>>> diez_hex = 0xa  
  
>>> print(diez)  
  
10  
  
>>> print(diez_binario)  
  
10  
  
>>> print(diez_octal)  
  
10  
  
>>> print(diez_hex)  
  
10
```



Números de punto flotante

Vamos a hacer un experimento que te va a dejar a cuadros. A continuación introduce la expresión $1.1 + 2.2$ y mira cuál es el resultado.

>>> [1.1 + 2.2](#)

3.3000000000000003

Representación de los números de punto flotante

Tenemos que repasar un poco de teoría que voy a tratar de simplificar porque la explicación completa da para un artículo entero.

Al igual que ocurre con los números enteros, los números reales son infinitos y, por tanto, es imposible representar todo el conjunto de números reales con un ordenador.

Para representar el mayor número posible de los números reales con las limitaciones de memoria (tamaños de palabra de 32 y 64 bits), se adaptó la notación científica de representación de números reales al sistema binario (que es el sistema que se utiliza en programación para representar los datos e instrucciones).

En esta notación científica, los números se representan así:

Número	Notación científica
101,1	$1,011 \times 10^2$
0,032	$3,2 \times 10^{-2}$



Vaya tela, ¿no? Pero es una muy buena solución que ha llegado hasta nuestros días.

El caso es que la suma de la representación en punto flotante en binario del número 1,1 y de la representación en punto flotante en binario del número 2,2, dan como resultado 3,3000000000000003

Pero hay más casos, como por ejemplo la representación del número 1/3. En algún momento, el ordenador tiene que truncar el número periódico resultante.

La explicación final es que los números de punto flotante se representan en el hardware del ordenador como fracciones de base 2 (binarias). Y el problema está en que la mayoría de las fracciones decimales no se pueden representar de forma exacta como fracciones binarias porque tienen infinitos números decimales. Una consecuencia es que, en general, los números decimales de punto flotante que usas en tus aplicaciones son una aproximación de los números binarios de punto flotante realmente almacenados en la máquina.

Números de Punto flotante en Python

Pues una vez vista esta simplificada introducción a los números de punto flotante, te diré que este tipo de datos en Python es float.

Puedes usar el tipo float sin problemas para representar cualquier número real (siempre teniendo en cuenta que es una aproximación lo más precisa posible). Por tanto para longitudes, pesos, frecuencias, ..., en los que prácticamente es lo mismo 3,3 que 3,3000000000000003 el tipo float es el más apropiado.

Cuando un número float vaya a ser usado por una persona, en lugar de por el ordenador, puedes darle formato al número de la siguiente manera:

```
>>> real = 1.1 + 2.2 # real es un float  
  
>>> print(real)  
  
3.3000000000000003 # Representación aproximada de 3.3  
  
>>> print(f'{real:.2f}')  
  
3.30 # real mostrando únicamente 2 cifras decimales
```

Al igual que los números enteros, un float se crea a partir de un literal, o bien como resultado de una expresión o una función.



```
>>> un_real = 1.1 # El literal debe incluir el carácter .  
>>> otro_real = 1/2 # El resultado de 1/2 es un float  
>>> not_cient = 1.23E3 # float con notación científica (1230.0)
```

Y para terminar esta sección, te adelanto que, si por cualquier motivo sí que necesitas una mayor precisión a la hora de trabajar con los números reales, Python tiene otros tipos de datos, como Decimal.

El tipo Decimal es ideal a la hora de trabajar, por ejemplo, con dinero o tipos de interés. Este tipo de dato trunca la parte decimal del número para ser más preciso, pero no es el objetivo de este tutorial hablar sobre el tipo de dato Decimal.

Números complejos

El último tipo de dato numérico básico que tiene Python es el de los números complejos, complex.

Los números complejos tienen una parte real y otra imaginaria y cada una de ellas se representa como un float.

Para crear un número complejo, se sigue la siguiente estructura <parte_real>+<parte_imaginaria>j. Y se puede acceder a la parte real e imaginaria a través de los atributos real e imag:

```
>>> complejo = 1+2j
```

```
>>> complejo.real
```

1.0

```
>>> complejo.imag
```

2.0



OPERADORES ARITMETICOS



Los valores numéricos son además el resultado de una serie de operadores aritméticos y matemáticos:

Operador Suma

El operador + suma los valores de tipo de datos numéricos.

```
>>> 3 + 2
```

```
5
```

Operador Resta

El operador - resta los valores de tipo de datos numéricos.

```
>>> 4 - 7
```

```
-3
```



Operador Negación

El operador - asigna un valor negativo a un tipo de datos numéricos.

```
>>> -7
```

```
-7
```

Operador Multiplicación

El operador * multiplica los valores de tipo de datos numéricos.

```
>>> 2 * 6
```

```
12
```

Operador Exponente

El operador ** calcula el exponente entre valores de tipo de datos numéricos.

```
>>> 2 ** 6
```

```
64
```

Operador división

El operador división el resultado que se devuelve es un número real.

```
>>> 3.5 / 2
```

```
1.75
```

Operador división entera

El operador división entera el resultado que se devuelve es solo la parte entera.

```
>>> 3.5 // 2
```

```
1.0
```

No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que quiere que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo, 3 / 2 y 3 // 2 sería el mismo: 1.



Si quisiéramos obtener los decimales necesitaríamos que al menos uno de los operandos fuera un número real, bien indicando los decimales:

```
r = 3.0 / 2
```

O bien utilizando la función [float\(\)](#) para convertir a entero coma flotante o real:

```
r = float(3) / 2
```

Esto es así porque cuando se mezclan tipos de números, Python convierte todos los operandos al tipo más complejo de entre los tipos de los operandos.

Operador Módulo

El operador módulo no hace otra cosa que devolver el resto de la división entre los dos operandos. En el ejemplo, $7 / 2$ sería 3, con 1 de resto, luego el módulo es 1.

```
>>> 7 % 2
```

```
1
```

Orden de precedencia

El orden de precedencia de ejecución de los operadores aritméticos es:

1. Exponente: `**`
2. Negación: `-`
3. Multiplicación, División, División entera, Módulo: `*`, `/`, `//`, `%`
4. Suma, Resta: `+`, `-`

Eso quiere decir que se debe usar así:

```
>>> 2**1/12
```

```
0.1666666666666666
```

```
>>>
```

Más igualmente usted puede omitir este orden de precedencia de ejecución de los operadores aritméticos usando paréntesis () anidados entre cada nivel calculo, por ejemplo:

```
>>> 2**(1/12)
```



1.0594630943592953

>>>





OPERADORES ARITMETICOS DE ASIGNACIÓN



Operadores de asignación

El operador de asignación se utiliza para asignar un valor a una variable. Como te he mencionado en otras secciones, este operador es el signo =.

Además del operador de asignación, existen otros operadores de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor.

Por ejemplo, $x += 1$ es lo mismo que $x = x + 1$. Los operadores compuestos realizan la operación que hay antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual.

A continuación, aparece la lista de todos los operadores de asignación compuestos:



Operador	Ejemplo	Equivalencia
<code>+=</code>	<code>x += 2</code>	<code>x = x + 2</code>
<code>-=</code>	<code>x -= 2</code>	<code>x = x - 2</code>
<code>*=</code>	<code>x *= 2</code>	<code>x = x * 2</code>
<code>/=</code>	<code>x /= 2</code>	<code>x = x / 2</code>
<code>%=</code>	<code>x %= 2</code>	<code>x = x % 2</code>
<code>//=</code>	<code>x //= 2</code>	<code>x = x // 2</code>
<code>**=</code>	<code>x **= 2</code>	<code>x = x ** 2</code>
<code>&=</code>	<code>x &= 2</code>	<code>x = x & 2</code>



Operador	Ejemplo	Equivalencia
<code> =</code>	<code>x = 2</code>	<code>x = x 2</code>
<code>^=</code>	<code>x ^= 2</code>	<code>x = x ^ 2</code>
<code>>=</code>	<code>x >= 2</code>	<code>x = x >> 2</code>
<code><=</code>	<code>x <= 2</code>	<code>x = x << 2</code>



EJERCICIOS OPERADORES ARITMÉTICOS



Ejercicio 1

Realizar la siguiente operación $[2+(8 \times 3 - 6) + 4 \times 5 - (28 / 6)] / 25$.

Solución:

```
resultado = (2 + (8 * 3 - 6) + 4 * 5 - (28 / 6)) / 25
print(resultado)
```

Ejercicio 2

Obtener el promedio o media aritmética de 200, 150 y 300 y limitar los decimales a 3 e imprimir el resultado.

Solución:

```
promedio = (200+150+300) / 3
```



```
print(f"{promedio:.3f}")
```

Ejercicio 3

Dada la siguiente formula $z=x^3+8-y^2$, solicitar por la terminal los valores para X y Y y calcular el valor de Z e imprimir el resultado.

Solución:

```
x= int(input("Ingresa el valor de x: "))
y= int(input("Ingresa el valor de y: "))

z= x**3+8-y**2

print(z)
```

Ejercicio 4

Esteban envía a Mario las cantidades de tres elementos que debe ir en una fórmula, el problema es que las cantidades están en un solo dígito 256477, 7 de la sustancia "a", 47 de la sustancia "b" y 256 de la sustancia "c", separar mediante operaciones matemáticas estos números, luego imprimir el nombre de la sustancia junto con la cantidad.

Solución:

```
elementos = 256477

a = elementos%10

elementos // = 10

b = elementos%100

elementos // = 100

c= elementos

print(f"Sustancia a: {a}")
print(f"Sustancia b: {b}")
print(f"Sustancia c: {c}")
```



OPERADORES RELACIONALES



Los operadores relacionales, o también llamados comparison operators nos permiten saber la relación existente entre dos variables. Se usan para saber si por ejemplo un número es mayor o menor que otro. Dado que estos operadores indican si se cumple o no una operación, el valor que devuelven es True o False.

Veamos un ejemplo con $x=2$ e $y=3$

Operador	Nombre	Ejemplo
<code>==</code>	Igual	<code>x == y = False</code>
<code>!=</code>	Distinto	<code>x != y = True</code>
<code>></code>	Mayor	<code>x > y = False</code>
<code><</code>	Menor	<code>x < y = True</code>
<code>>=</code>	Mayor o igual	<code>x >= y = False</code>
<code><=</code>	Menor o igual	<code>x <= y = True</code>

```
x=2; y=3
print("Operadores Relacionales")
print("x==y =", x==y) # False
print("x!=y =", x!=y) # True
print("x>y =", x>y) # False
print("x<y =", x<y) # True
print("x>=y =", x>=y) # False
print("x<=y =", x<=y) # True
```



Operador ==

El operador `==` permite comparar si las variables introducidas a su izquierda y derecha son iguales. Muy importante no confundir con `=`, que es el operador de asignación.

```
print(4==4)          # True
print(4==5)          # False
print(4==4.0)        # True
print(0==False)      # True
print("asd"]=="asd") # True
print("asd"]=="asdf") # False
print(2=="2")        # False
print([1, 2, 3] == [1, 2, 3]) # True
```

Operador !=

El operador `!=` devuelve True si los elementos a comparar son iguales y False si estos son distintos. De hecho, una vez definido el operador `==`, no sería necesario ni explicar `!=` ya que hace exactamente lo contrario. Definido primero, definido el segundo. Es decir, si probamos con los mismos ejemplo que el apartado anterior, veremos como el resultado es el contrario, es decir False donde era True y viceversa.

```
print(4!=4)          # False
print(4!=5)          # True
print(4!=4.0)        # False
print(0!=False)      # False
print("asd"!="asd") # False
print("asd"!="asdf") # True
print(2!="2")        # True
print([1, 2, 3] != [1, 2, 3]) # False
```

Operador >

El operador `>` devuelve True si el primer valor es mayor que el segundo y False de lo contrario.

```
print(5>3) # True
print(5>5) # False
```

Algo bastante curioso, es como Python trata al tipo booleano. Por ejemplo, podemos ver como True es igual a 1, por lo que podemos comparar el tipo True como si de un número se tratase.

```
print(True==1)    # True
print(True>0.999) # True
```



Para saber más: De hecho, el tipo bool en Python hereda de la clase int. Si quieres saber más acerca del tipo bool en Python puedes leer la [PEP285](#)

También se pueden comparar listas. Si los elementos de la lista son numéricos, se comparará elemento a elemento.

```
print([1, 2] > [10, 10]) # False
```

Operador <

El operador < devuelve True si el primer elemento es mayor que el segundo. Es totalmente válido aplicar operadores relacionales como < sobre cadenas de texto, pero el comportamiento es un tanto difícil de ver a simple vista. Por ejemplo abc es menor que abd y A es menor que a

```
print("abc" < "abd") # True
print("A" < "a") # True
```

Para el caso de A y a la explicación es muy sencilla, ya que Python lo que en realidad está comparando es el valor entero Unicode que representa tal carácter. La función ord() nos da ese valor. Por lo tanto cuando hacemos "A" < "a" lo que en realidad hacemos es comparar dos números.

```
print(ord('A')) # 65
print(ord('a')) # 97
```

Operador >=

Similar a los anteriores, >= permite comparar si el primer elemento es mayor o igual que el segundo, devolviendo True en el caso de ser cierto.

```
print(3>=3) # True
print([3,4] >= [3,5]) # False
```

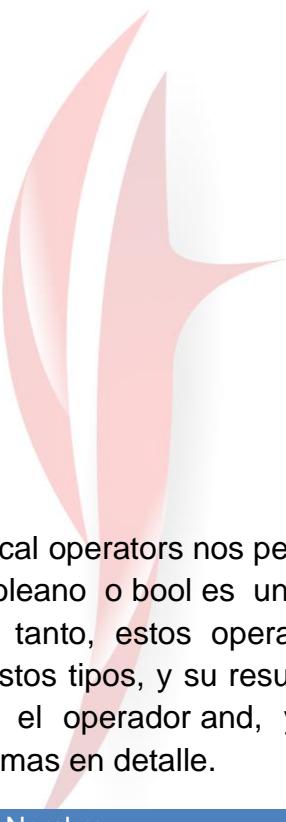
Operador <=

De la misma manera, <= devuelve True si el primer elemento es menor o igual que el segundo. Nos podemos encontrar con cosas interesantes debido a la precisión numérica existente al representar valores, como el siguiente ejemplo.

```
print(3<=2.999999999999999)
```



OPERADORES LÓGICOS



Los operadores lógicos o logical operators nos permiten trabajar con valores de tipo booleano. Un valor booleano o bool es un tipo que solo puede tomar valores True o False. Por lo tanto, estos operadores nos permiten realizar diferentes operaciones con estos tipos, y su resultado será otro booleano. Por ejemplo, True and True usa el operador and, y su resultado será True. A continuación lo explicaremos mas en detalle.

Operador	Nombre	Ejemplo
and	Devuelve True si ambos elementos son True	True and True = True
or	Devuelve True si al menos un elemento es True	True or False = True
not	Devuelve el contrario, True si es Falso y viceversa	not True = False

Operador and

El operador and evalúa si el valor a la izquierda y el de la derecha son True, y en el caso de ser cierto, devuelve True. Si uno de los dos valores es False, el resultado será False. Es realmente un operador muy lógico e intuitivo que incluso usamos en la vida real. Si hace sol y es fin de semana, iré a la playa. Si ambas



condiciones se cumplen, es decir que la variable haceSol=True y la variable finDeSemana=True, iré a la playa, o visto de otra forma irALaPlaya=(haceSol and finDeSemana).

```
print(True and True) # True
print(True and False) # False
print(False and True) # False
print(False and False) # False
```

Operador or

El operador or devuelve True cuando al menos uno de los elementos es igual a True. Es decir, evalúa si el valor a la izquierda o el de la derecha son True.

```
print(True or True) # True
print(True or False) # True
print(False or True) # True
print(False or False) # False
```

Es importante notar que varios operadores pueden ser usados conjuntamente, y salvo que existan paréntesis que indiquen una cierta prioridad, el primer operador que se evaluará será el and. En el ejemplo que se muestra a continuación, podemos ver que tenemos dos operadores and. Para calcular el resultado final, tenemos que empezar por el and calculando el resultado en grupos de dos y arrastrando el resultado hacia el siguiente grupo. El resultado del primer grupo sería True ya que estamos evaluando True and True. Después, nos guardamos ese True y vamos a por el siguiente y último elemento, que es False. Por lo tanto hacemos True and False por lo que el resultado final es False

```
print(True and True and False)
#      | ----- |
#      True and False
#      | ----- |
#      False
```

También podemos mezclar los operadores. En el siguiente ejemplo empezamos por False and True que es False, después False or True que es True y por último True or False que es True. Es decir, el resultado final de toda esa expresión es True.

```
print(False and True or True or False)
#      False and True = False
#                  Fase or True = True
#                  True or False = True
#      True
```



Operador not

Y por último tenemos el operador not, que simplemente invierte True por False y False por True. También puedes usar varios not juntos y simplemente se irán aplicando uno tras otro. La verdad que es algo difícil de ver en la realidad, pero simplemente puedes contar el número de not y si es par el valor se quedará igual. Si por lo contrario es impar, el valor se invertirá.

```
print(not True) # False
print(not False) # True
print(not not not not True) # True
```

Dado que estamos tratando con booleanos, hemos considerado que usar True y False es lo mejor y más claro, pero es totalmente válido emplear 1 y 0 respectivamente para representar ambos estados. Y por supuesto los resultados no varían.

```
print(not 0) # True
print(not 1) # False
```

Ejemplos

Antes de entrar con los ejemplos, es importante notar que el orden de aplicación de los operadores puede influir en el resultado, por lo que es importante tener muy claro su prioridad de aplicación. De mayor a menor prioridad, el primer sería not, seguido de and y or.

Para saber más: Puedes leer la especificación oficial si quieres saber el orden de aplicación de todos los operadores [en este enlace](#)

Pero, imaginemos que no sabemos el orden de aplicación de los operadores. Vamos a intentar con ingeniería inversa descubrirlo con unos cuantos ejemplos. En el ejemplo que se muestra a continuación, habría dos posibilidades. La primera sería (False and False) or True cuyo resultado sería True y la segunda False and (False or True) cuyo resultado sería False. Dado que el resultado que Python da es True, podemos concluir que el equivalente es la primera opción, es decir que and tiene prioridad sobre el or

```
print(False and False or True)
# True
```

Lo mismo con el siguiente ejemplo. Como ya sabemos de antes que and es evaluado primero, la siguiente expresión sería en realidad equivalente a True or (False and False) por lo que su resultado es True

```
print(True or False and False)
# True
```



Como ya hemos indicado anteriormente, se puede usar 0 y 1 para representar False y True respectivamente. Veamos a usar esa representación para abreviar. Podemos simplificar la siguiente expresión, ya que el not es el operador que primer se aplica. Por lo tanto nos quedaría 0 and 0 or 1 and 1 or 1 and 0. Y como ya sabemos, después se aplica el and, por lo que nos quedaría (0 and 0) or (1 and 1) or (1 and 0). Y ya nos quedaría sólo aplicar el or de la expresión resultante 0 or 1 or 1, por lo que el resultado final es 1 o True. Date cuenta que la expresión se podría simplificar, y una vez en un or nos encontramos que un lado es True, podríamos dejar de calcular el resto de la expresión ya que el resultado ya sería True

```
print(0 and not 1 or 1 and not 0 or 1 and 0)  
# True
```





EJERCICIOS OPERADORES RELACIONALES Y LÓGICOS



Ejercicio 1

Solicitar la edad de una persona y verificar si es mayor a 18 años.

Solución:

```
edad = int( input("Eres mayor de edad? Ingresa tu edad: ") )  
resultado = edad > 18  
print(resultado)
```

Ejercicio 2

Demostrar que los valores booleanos de "" y 0 son iguales.



Solución:

```
a = bool("Hola")
b = bool(1)

resultado = a == b

print(a)
```

Ejercicio 3

Solicitar 3 notas menores a 10 y verificar si todas son mayores que 7.

Solución:

```
nota1 = int(input("Ingresa la nota 1: "))
nota2 = int(input("Ingresa la nota 2: "))
nota3 = int(input("Ingresa la nota 3: "))

#resultado = nota1 and nota2 and nota3 > 7
resultado = nota1 > 7 and nota2> 7 and nota3 > 7

print(resultado)
```

Ejercicio 4

El servicio militar solicita postulantes que sean mayores de 18 años y con una estatura mayor o igual a 1.75m , solicitar estos datos para verificar si cumplen esta condición.

Solución:

```
print("Cumples con las condiciones?")
edad = int(input("Ingresa tu edad: "))
estatura = int(input("Ingresa tu estatura en cm: "))

resultado = edad >= 18 and estatura >= 175

print(f"Cumples con las condiciones? {resultado}")
```



MÓDULO DATETIME



En Python, la fecha y la hora no son un tipo de datos propio, pero se puede importar un módulo llamado datetime para trabajar con la fecha y la hora. El módulo Python Datetime viene integrado en Python, por lo que no es necesario instalarlo externamente.

El módulo Python Datetime proporciona clases para trabajar con fecha y hora. Estas clases proporcionan una serie de funciones para tratar con fechas, horas e intervalos de tiempo. Date y datetime son un objeto en Python, por lo que cuando los manipulas, en realidad estás manipulando objetos y no cadenas o marcas de tiempo.

El módulo DateTime se clasifica en 6 clases principales:

Clase Date

El objeto de la clase Date representa la fecha ingenua que contiene año, mes y fecha de acuerdo con el calendario gregoriano actual. Esta fecha se puede



extender indefinidamente en ambas direcciones. El 1 de enero del año 1 se llama día 1 y el 2 de enero o el año 2 se llama día 2 y así sucesivamente.

Sintaxis:

```
class datetime.date(year, month, day)
```

Los argumentos deben estar en el siguiente rango:

- MINYEAR(1) <= año <= MAXYEAR(9999)
- 1 <= mes <= 12
- 1 <= día <= número de días en el mes y año dados

Nota: Si el argumento no es un entero, generará un `TypeError` y si está fuera del rango se generará un `ValueError`.

Clase Time

La clase de hora representa la hora local del día que es independiente de cualquier día en particular. Esta clase puede tener el objeto `tzinfo` que representa la zona horaria de la hora dada. Si el `tzinfo` es `None`, entonces el objeto `time` es el objeto ingenuo, de lo contrario es el objeto consciente.

Sintaxis:

```
class datetime.time(hora=0, minuto=0, segundo=0, microsegundo=0,  
tzinfo=Ninguno, *, fold=0)
```

Todos los argumentos son opcionales. `tzinfo` puede ser `Ninguno`, de lo contrario, todos los atributos deben ser enteros en el siguiente rango:

- 0 <= hora < 24
- 0 <= minuto < 60
- 0 <= segundo < 60
- 0 <= microsegundo < 1000000
- doblar [0, 1]

Clase Datetime

La clase `DateTime` del módulo `DateTime`, como su nombre indica, contiene información tanto sobre la fecha como sobre la hora. Al igual que un objeto de fecha, `DateTime` asume el calendario gregoriano actual extendido en ambas direcciones; al igual que un objeto de tiempo, `DateTime` asume que hay



exactamente $3600 * 24$ segundos en cada día. Pero a diferencia de la clase date, los objetos de la clase DateTime son objetos potencialmente conscientes, es decir, también contienen información sobre la zona horaria.

Sintaxis:

```
class datetime.datetime(año, mes, día, hora=0, minuto=0, segundo=0,  
microsegundo=0, tzinfo=Ninguno, *, fold=0)
```

Los argumentos de año, mes y día son obligatorios. tzinfo puede ser Ninguno, el resto de todos los atributos deben ser un entero en el siguiente rango:

- MINYEAR(1) <= año <= MAXYEAR(9999)
- 1 <= mes <= 12
- 1 <= día <= número de días en el mes y año dados
- 0 <= hora < 24
- 0 <= minuto < 60
- 0 <= segundo < 60
- 0 <= microsegundo < 1000000
- doblar [0, 1]

Nota: Pasar un argumento que no sea entero generará un TypeError y pasar argumentos fuera del rango generará ValueError.

Clase Timedelta

La clase Timedelta se utiliza para calcular las diferencias entre fechas y representa una duración. La diferencia puede ser tanto positiva como negativa.

Sintaxis:

```
class datetime.timedelta(days=0, seconds=0, microseconds=0,  
millisegundos=0, minutes=0, hours=0, weeks=0)
```

Clase Tzinfo

Proporciona objetos de información de zona horaria.



Clase Timezone

Clase que implementa la clase base abstracta tzinfo como un desplazamiento fijo del UTC (Nuevo en la versión 3.2).





CLASE DATE



El objeto de la clase Date representa la fecha ingenua que contiene año, mes y fecha de acuerdo con el calendario gregoriano actual. Esta fecha se puede extender indefinidamente en ambas direcciones. El 1 de enero del año 1 se llama día 1 y el 2 de enero o el año 2 se llama día 2 y así sucesivamente.

Sintaxis:

```
class datetime.date(year, month, day)
```

Los argumentos deben estar en el siguiente rango:

- MINYEAR(1) <= año <= MAXYEAR(9999)
- 1 <= mes <= 12
- 1 <= día <= número de días en el mes y año dados

Nota: Si el argumento no es un entero, generará un TypeError y si está fuera del rango se generará un ValueError.



Ejemplo:

```
# Python program to

# demonstrate date class


# import the date class

from datetime import date


# initializing constructor

# and passing arguments in the

# format year, month, date

my_date = date(2020, 12, 11)

print("Date passed as argument is", my_date)

# Uncommenting my_date = date(1996, 12, 39)

# will raise an ValueError as it is

# outside range


# uncommenting my_date = date('1996', 12, 11)

# will raise a TypeError as a string is

# passed instead of integer
```

Salida

```
Date passed as argument is 2020-12-11
```



Atributos de clase

Veamos los atributos proporcionados por esta clase:

Attribute	Description
Name	
min	The minimum representable date
max	The maximum representable date
resolution	The minimum possible difference between date objects
year	The range of year must be between MINYEAR and MAXYEAR
month	The range of month must be between 1 and 12
day	The range of day must be between 1 and number of days in the given month of the given year

Ejemplo 1: Obtener una fecha mínima y máxima representable

```
from datetime import date

# Getting min date
mindate = date.min
print("Min Date supported", mindate)

# Getting max date
maxdate = date.max
print("Max Date supported", maxdate)
```

Salida

```
Min Date supported 0001-01-01
Max Date supported 9999-12-31
```



Ejemplo 2: Acceso al atributo year, month, and date desde la clase date

```
from datetime import date

# creating the date object
Date = date(2020, 12, 11)

# Accessing the attributes
print("Year:", Date.year)
print("Month:", Date.month)
print("Day:", Date.day)
```

Salida

Year: 2020

Month: 12

Day: 11

Funciones de clase

La clase Date proporciona varias funciones para trabajar con el objeto date, como podemos obtener la fecha de hoy, la fecha de la marca de tiempo actual, la fecha del ordinal gregoriano proleptico, donde el 1 de enero del año 1 tiene el ordinal 1, etc. Veamos la lista de todas las funciones proporcionadas por esta clase:



Nombre de la función	Descripción
ctime()	Devolver una cadena que represente la fecha
fromisocalendar()	Devuelve una fecha correspondiente al calendario ISO.
fromisoformat()	Devuelve un objeto date de la representación de cadena de la fecha.
fromordinal()	Devuelve un objeto de fecha del ordinal gregoriano proleptico, donde el 1 de enero del año 1 tiene el ordinal 1
fromtimestamp()	Devuelve un objeto de fecha de la marca de tiempo POSIX
isocalendario()	Devuelve un año, una semana y un día laborables de la tupla.
isoformat()	Devuelve la representación de cadena de la fecha.
isoweekday()	Devuelve el día de la semana como entero donde el lunes es 1 y el domingo es 7
reemplazar()	Cambia el valor del objeto date con el parámetro dado
strftime()	Devuelve una representación de cadena de la fecha con el formato dado.
timetuple()	Devuelve un objeto de tipo time.struct_time
hoy()	Devuelve la fecha local actual.
toordinal()	Devuelve el ordinal gregoriano proleptico de la fecha, donde el 1 de enero del año 1 tiene el ordinal 1
días laborables()	Devuelve el día de la semana como entero donde el lunes es 0 y el domingo es 6

Veamos ciertos ejemplos de las funciones anteriores

Ejemplo 1: Obtener la fecha actual y también cambiar la fecha a cadena



```
# Python program to
# print current date

from datetime import date

# calling the today
# function of date class
today = date.today()

print("Today's date is", today)

# Converting the date to the string
Str = date.isoformat(today)
print("String Representation", Str)
print(type(Str))
```

Salida

```
Today's date is 2021-07-23
String Representation 2021-07-23
<class 'str'>
```

Ejemplo 2: Obtener el día de la semana a partir del día y el ordinal gregoriano proleptico



```
# Python program to
# print current date

from datetime import date

# calling the today
# function of date class
today = date.today()

# Getting Weekday using weekday()
# method
print("Weekday using weekday():", today.weekday())

# Getting Weekday using isoweekday()
# method
print("Weekday using isoweekday():", today.isoweekday())

# Getting the proleptic Gregorian
# ordinal
print("proleptic Gregorian ordinal:", today.toordinal())

# Getting the date from the ordinal
print("Date from ordinal", date.fromordinal(737000))
```

Salida

```
Weekday using weekday(): 4
Weekday using isoweekday(): 5
proleptic Gregorian ordinal: 737994
Date from ordinal 2018-11-02
```



CLASE TIME



La clase de hora representa la hora local del día que es independiente de cualquier día en particular. Esta clase puede tener el objeto tzinfo que representa la zona horaria de la hora dada. Si el tzinfo es None, entonces el objeto time es el objeto ingenuo, de lo contrario es el objeto consciente.

Sintaxis:

```
clase datetime.time(hora=0, minuto=0, segundo=0, microsegundo=0,  
tzinfo=Ninguno, *, fold=0)
```

Todos los argumentos son opcionales. tzinfo puede ser Ninguno, de lo contrario, todos los atributos deben ser enteros en el siguiente rango:

- $0 \leq \text{hora} < 24$
- $0 \leq \text{minuto} < 60$
- $0 \leq \text{segundo} < 60$
- $0 \leq \text{microsegundo} < 1000000$



- doblar [0, 1]

Ejemplo:

```
datetime import time

# calling the constructor

my_time = time(12, 14, 36)

print("Entered time", my_time)

# calling constructor with 1
# argument

my_time = time(minute = 12)

print("\nTime with one argument", my_time)

# Calling constructor with
# 0 argument

my_time = time()

print("\nTime without argument", my_time)

# Uncommenting time(hour = 26)
# will raise an ValueError as
# it is out of range

# uncommenting time(hour ='23')
# will raise TypeError as
# string is passed instead of int
```



Salida

```
Entered time 12:14:36
```

```
Time with one argument 00:12:00
```

```
Time without argument 00:00:00
```

Atributos de clase

Veamos los atributos proporcionados por esta clase:

Nombre del atributo	Descripción
min	Representación mínima posible del tiempo
max	Máxima representación posible del tiempo
resolution	La mínima diferencia posible entre objetos de tiempo
hour	El rango de horas debe estar entre 0 y 24 (sin incluir 24)
minute	El rango de minutos debe estar entre 0 y 60 (sin incluir 60)
second	El rango de segundo debe estar entre 0 y 60 (sin incluir 60)
microsecond	El rango de microsegundos debe estar entre 0 y 1000000 (sin incluir 1000000)
tzinfo	El objeto que contiene información de zona horaria
fold	Representa si el pliegue se ha producido en el tiempo o no

Ejemplo 1: Obtener tiempo mínimo y máximo representable



```
from datetime import time

# Getting min time
mintime = time.min
print("Min Time supported", mintime)

# Getting max time
maxtime = time.max
print("Max Time supported", maxtime)
```

Salida

```
Min Time supported 00:00:00
Max Time supported 23:59:59.999999
```

Ejemplo 2: Acceso al atributo de hora, minutos, segundos y microsegundos desde la clase de tiempo



```
from datetime import time

# Creating Time object
Time = time(12,24,36,1212)

# Accessing Attributes
print("Hour:", Time.hour)
print("Minutes:", Time.minute)
print("Seconds:", Time.second)
print("Microseconds:", Time.microsecond)
```

Salida

```
Hour: 12
Minutes: 24
Seconds: 36
Microseconds: 1212
```

Funciones de clase

La clase de tiempo proporciona varias funciones como que podemos obtener tiempo de cadena o convertir tiempo a cadena, formatear el tiempo de acuerdo con nuestra necesidad, etc. Veamos una lista de todas las funciones proporcionadas por la clase de tiempo.



Listado de funciones de clase de tiempo

Nombre de la función	Descripción
dst()	Devuelve tzinfo.dst() es tzinfo no es Ninguno
fromisoformat()	Devuelve un objeto time de la representación de cadena de la hora.
isoformat()	Devuelve la representación de cadena del tiempo del objeto time
reemplazar()	Cambia el valor del objeto time con el parámetro dado
strftime()	Devuelve una representación de cadena de la hora con el formato dado.
tzname()	Devuelve tzinfo.tzname() es tzinfo no es Ninguno
utcoffset()	Devuelve tzinfo.utcoffset() es tzinfo no es Ninguno

Veamos ciertos ejemplos de las funciones anteriores

Ejemplo 1: Convertir objeto de tiempo en cadena y viceversa

```
from datetime import time

# Creating Time object
Time = time(12,24,36,1212)

# Converting Time object to string
Str = Time.isoformat()
print("String Representation:", Str)
print(type(Str))

Time = "12:24:36.001212"
```



```
# Converting string to Time object
Time = time.fromisoformat(Str)
print("\nTime from String", Time)
print(type(Time))
```

Salida

String Representation: 12:24:36.001212

<class 'str'>

Time from String 12:24:36.001212

<class 'datetime.time'>

Ejemplo 2: Cambiar el valor de un objeto de tiempo ya creado y dar formato a la hora

```
from datetime import time

# Creating Time object
Time = time(12,24,36,1212)
print("Original time:", Time)

# Replacing hour
Time = Time.replace(hour = 13, second = 12)
print("New Time:", Time)

# Formatting Time
```



```
Ftime = Time.strftime("%I:%M %p")
print("Formatted time", Ftime)
```

Salida

```
Original time: 12:24:36.001212
New Time: 13:24:12.001212
Formatted time 01:24 PM
```





CLASE DATETIME



La clase DateTime del módulo DateTime, como su nombre indica, contiene información tanto sobre la fecha como sobre la hora. Al igual que un objeto de fecha, DateTime asume el calendario gregoriano actual extendido en ambas direcciones; al igual que un objeto de tiempo, DateTime asume que hay exactamente 3600 * 24 segundos en cada día. Pero a diferencia de la clase date, los objetos de la clase DateTime son objetos potencialmente conscientes, es decir, también contienen información sobre la zona horaria.

Sintaxis:

```
class datetime.datetime(año, mes, día, hora=0, minuto=0, segundo=0,  
microsegundo=0, tzinfo=Ninguno, *, fold=0)
```

Los argumentos de año, mes y día son obligatorios. tzinfo puede ser Ninguno, el resto de todos los atributos deben ser un entero en el siguiente rango:



- MINYEAR(1) <= año <= MAXYEAR(9999)
- 1 <= mes <= 12
- 1 <= día <= número de días en el mes y año dados
- 0 <= hora < 24
- 0 <= minuto < 60
- 0 <= segundo < 60
- 0 <= microsegundo < 1000000
- fold [0, 1]

Nota: Pasar un argumento que no sea entero generará un `TypeError` y pasar argumentos fuera del rango generará `ValueError`.

Ejemplo: Creación de una instancia de la clase `DateTime`

```
# Python program to

# demonstrate datetime object


from datetime import datetime


# Initializing constructor

a = datetime(2022, 10, 22)

print(a)


# Initializing constructor

# with time parameters as well

a = datetime(2022, 10, 22, 6, 2, 32, 5456)

print(a)
```

Salida

```
2022-10-22 00:00:00
2022-10-22 06:02:32.005456
```



Atributos de clase

Veamos los atributos proporcionados por esta clase:

Nombre del atributo	Descripción
min	El DateTime mínimo representable
máximo	El datetime máximo representable
resolución	La diferencia mínima posible entre objetos datetime
Year	El rango de año debe estar entre MINYEAR y MAXYEAR
Month	El rango de mes debe estar entre 1 y 12
Day	El rango de día debe estar entre 1 y el número de días en el mes dado del año dado.
Hour	El rango de horas debe estar entre 0 y 24 (sin incluir 24)
Minute	El rango de minutos debe estar entre 0 y 60 (sin incluir 60)
Second	El rango de segundo debe estar entre 0 y 60 (sin incluir 60)
Microsecond	El rango de microsegundos debe estar entre 0 y 1000000 (sin incluir 1000000)
Tzinfo	El objeto que contiene información de zona horaria
fold	Representa si el pliegue se ha producido en el tiempo o no

Ejemplo 1: Obtención del objeto DateTime representable mínimo y máximo

```
from datetime import datetime

# Getting min datetime
mindatetime = datetime.min
print("Min DateTime supported", mindatetime)

# Getting max datetime
maxdatetime = datetime.max
print("Max DateTime supported", maxdatetime)
```

Salida

Min DateTime supported 0001-01-01 00:00:00



Max DateTime supported 9999-12-31 23:59:59.999999

Ejemplo 2: Acceso a los atributos del objeto de fecha y hora

```
from datetime import datetime

# Getting Today's Datetime
today = datetime.now()

# Accessing Attributes
print("Day: ", today.day)
print("Month: ", today.month)
print("Year: ", today.year)
print("Hour: ", today.hour)
print("Minute: ", today.minute)
print("Second: ", today.second)
```

Salida

```
Day: 26
Month: 7
Year: 2021
Hour: 16
Minute: 24
Second: 7
```

Funciones de clase

La clase DateTime proporciona varias funciones para tratar con los objetos DateTime, como podemos convertir el objeto DateTime en cadena y la cadena en objetos DateTime, también podemos obtener el día de la semana para el día



particular de la semana del mes en particular, también podemos establecer la zona horaria para un objeto DateTime en particular, etc.

Lista de métodos de clase DateTime

Nombre de la función	Descripción
astimezone()	Devuelve el objeto DateTime que contiene información de zona horaria.
combinar()	Combina los objetos de fecha y hora y devuelve un objeto DateTime
ctime()	Devuelve una representación de cadena de fecha y hora.
fecha()	Devolver el objeto de clase Date
fromisoformat()	Devuelve un objeto datetime de la representación de cadena de la fecha y la hora.
fromordinal()	Devuelve un objeto de fecha del ordinal gregoriano proléptico, donde el 1 de enero del año 1 tiene el ordinal 1. La hora, el minuto, el segundo y el microsegundo son 0
fromtimestamp()	Fecha y hora de devolución desde la marca de tiempo POSIX
isocalendario()	Devuelve un año, una semana y un día laborables de la tupla.
isoformat()	Devolver la representación de cadena de fecha y hora
isoweekday()	Devuelve el día de la semana como entero donde el lunes es 1 y el domingo es 7
ahora()	Devuelve la fecha y hora locales actuales con el parámetro tz
reemplazar()	Cambia los atributos específicos del objeto DateTime
strftime()	Devuelve una representación de cadena del objeto DateTime con el formato dado.
strptime()	Devuelve un objeto DateTime correspondiente a la cadena de fecha.
tiempo()	Devolver el objeto de clase Time
timetuple()	Devuelve un objeto de tipo time.struct_time
timetz()	Devolver el objeto de clase Time
hoy()	Devolver DateTime local con tzinfo como Ninguno
toordinal()	Devuelve el ordinal gregoriano proléptico de la fecha, donde el 1 de enero del año 1 tiene el ordinal 1
tzname()	Devuelve el nombre de la zona horaria.
utcfromtimestamp()	Devolver UTC desde la marca de tiempo POSIX
utcoffset()	Devuelve el desplazamiento UTC
utcnow()	Devolver la fecha y hora UTC actuales



días laborables()	Devuelve el día de la semana como entero donde el lunes es 0 y el domingo es 6
--------------------------	--------------------------------------------------------------------------------

Ejemplo 1: Obtener la fecha de hoy

```
from datetime import datetime

# Getting Today's Datetime
today = datetime.now()

print("Today's date using now() method:", today)

today = datetime.today()

print("Today's date using today() method:", today)
```

Salida

Fecha de hoy usando el método now(): 2021-07-26 22:23:22.725573

Fecha de hoy usando el método today(): 2021-07-26 22:23:22.725764

Ejemplo 2: Obtención de DateTime de la marca de tiempo y el ordinal

```
from datetime import datetime

# Getting Datetime from timestamp
date_time = datetime.fromtimestamp(1887639468)

print("Datetime from timestamp:", date_time)

# Getting Datetime from ordinal
```



```
date_time = datetime.fromordinal(737994)
print("Datetime from timestamp:", date_time)
```

Salida

```
Datetime from timestamp: 2029-10-25 16:17:48
Datetime from ordinal: 2021-07-23 00:00:00
```





FORMATO DATETIME

Formatear fechas y horas

En esta sección vamos a ver dos métodos especiales de las clases date, time y datetime. Uno para representarlos en una cadena de caracteres (o string) formateado a nuestro gusto, y otro para crear instancias de estos objetos a partir de un string.

Método `.strftime()`

La fecha se representa de forma distinta en diferentes países. Por ejemplo en Estados Unidos se usa el formato mes/día/año, mientras que en Europa el formato utilizado es día/mes/año. Lo mismo sucede con las horas, en algunos lugares se usa el formato de 24h y en otros prefieren el formato de 12h. Para ello tenemos el método `strftime()`, el cual retorna un string con la fecha o hora representada en un formato definido por nosotros mismos.



El siguiente código Python muestra cómo formatear un objeto datetime con el método strftime() a partir de un string que representa nuestro formato deseado. Si quieres saber cómo crear ese string a medida, al final del artículo he añadido una tabla con los códigos que podemos utilizar. Señalar también que estos mismos conceptos son aplicables a los objetos date y time.

```
>>> x = datetime(2021, 7, 22, 21, 15)  
  
>>> x.strftime("%A, %d of %B %Y at %I:%M %p")  
  
>>> print(x)
```

Thursday, 22 of July 2021 at 09:15 PM

Por defecto, los resultados que obtenemos con el método strftime() son en inglés. Si queremos mostrar los resultados en español tenemos que especificarlo con la librería locale. El siguiente ejemplo muestra cómo hacerlo.

```
>>> import locale  
  
>>> locale.setlocale(locale.LC_TIME, "es_ES")  
  
'es_ES'
```

Una vez completado el paso anterior los resultados ya se muestran en español.

```
>>> x = datetime(2021, 7, 22, 21, 15)  
  
>>> x.strftime("%A, %d de %B de %Y a las %I:%M %p")  
  
>>> print(x)
```

jueves, 22 de julio de 2021 a las 09:15

Método .strptime()

Este método hace justo lo contrario que el método strftime(). Es decir, crea un objeto de tipo date, time o datetime a partir de dos strings: uno que representa los valores de fecha y/o tiempo que va a tener el objeto, y otro que especifica el formato del anterior.

```
>>> x = datetime.strptime("22/07/2021, 21:15", "%d/%m/%Y, %H:%M")
```



>>> x

datetime.datetime([2021, 7, 22, 21, 15](#))

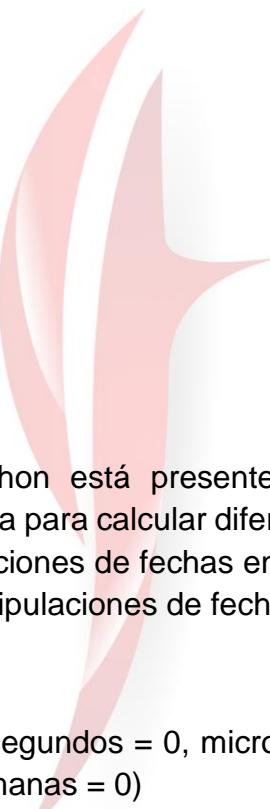
Tabla de códigos de formato

A continuación te dejo un tabla con los distintos códigos que podemos utilizar para formatear el tiempo con los métodos strftime() y strftime().

Código	Significado	Ejemplos
%a	Día de la semana abreviado	lu., ma., ...
%A	Día de la semana completo	lunes, martes, ...
%w	Día de la semana como número decimal	0, 1, ... 6
%d	Día del mes como número decimal con cero	01, 02, ..., 31
%b	Mes abreviado	ene., feb., ...
%B	Mes completo	enero, febrero, ...
%m	Mes como número decimal con cero	01, 02, ... 12
%Y	Año en formato de cuatro dígitos	0001, 0002, ..., 2020, 2021, ...
%H	Hora en formato 24h. con dos dígitos	00, 01, ..., 23
%I	Hora en formato 12h. con dos dígitos	01, 02, ..., 12
%M	Minutos en formato de dos dígitos	00, 01, ..., 59
%S	Segundos en formato de dos dígitos	00, 01, ..., 59



TIME DELTA



La `timedelta()` función de Python está presente en la biblioteca de fecha y hora, que generalmente se usa para calcular diferencias en las fechas y también se puede usar para manipulaciones de fechas en Python. Es una de las formas más sencillas de realizar manipulaciones de fechas.

Sintaxis:

```
datetime.timedelta(días = 0, segundos = 0, microsegundos = 0, milisegundos = 0, minutos = 0, horas = 0, semanas = 0)
```

Devoluciones: Fecha



Código # 1:

```
from datetime import datetime, timedelta

ini_time_for_now = datetime.now()

print ("initial_date", str(ini_time_for_now))

future_date_after_2yrs = ini_time_for_now + \
                         timedelta(days = 730)

future_date_after_2days = ini_time_for_now + \
                          timedelta(days = 2)

print('future_date_after_2yrs:', str(future_date_after_2yrs))
print('future_date_after_2days:', str(future_date_after_2days))
```

Salida:

```
fecha_inicial 2019-02-27 12: 41: 45.018389
Future_date_after_2yrs: 2021-02-26 12: 41: 45.018389
future_date_after_2days: 2019-03-01 12:41: 45.018389
```



Código # 2:

```
from datetime import datetime, timedelta

ini_time_for_now = datetime.now()

print ("initial_date", str(ini_time_for_now))

past_date_before_2yrs = ini_time_for_now - \
                        timedelta(days = 730)

past_date_before_2hours = ini_time_for_now - \
                         timedelta(hours = 2)

print('past_date_before_2yrs:', str(past_date_before_2yrs))
print('past_date_after_2days:', str(past_date_before_2hours))
```

Salida:

```
fecha_inicial 2019-02-27 12: 41: 46.104662
past_date_before_2yrs: 2017-02-27 12: 41: 46.104662
past_date_after_2days: 2019-02-27 10: 41: 46.104662
```



Código # 3:

```
from datetime import datetime, timedelta

ini_time_for_now = datetime.now()

print ("initial_date", str(ini_time_for_now))

new_final_time = ini_time_for_now + \
                  timedelta(days = 2)

print ("new_final_time", str(new_final_time))

print('Time difference:', str(new_final_time - \
                               ini_time_for_now))
```

Salida:

```
fecha_inicial 2019-02-27 12: 41: 47.386595
new_final_time 2019-03-01 12: 41: 47.386595
Diferencia horaria: 2 días, 0:00:00
```



SINTAXIS Y SENTENCIAS COMPUESTAS

s



Indentación

Python utiliza la indentación para delimitar la estructura permitiendo establecer bloques de código. No existen comandos para finalizar las líneas ni llaves con las que delimitar el código. Los únicos delimitadores existentes son los dos puntos (:) y la indentación del código.

Si buscamos el significado de indentación en Wikipedia indica que “en los lenguajes de programación de computadoras, la indentación es un tipo de notación secundaria utilizado para mejorar la legibilidad del código fuente por parte de los programadores, teniendo en cuenta que los compiladores o intérpretes raramente consideran los espacios en blanco entre las sentencias de un programa”.

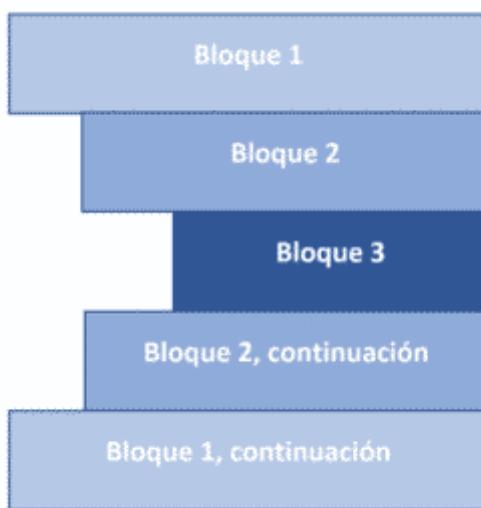
Normalmente se suelen dejar como indentación o sangrado 4 espacios en blanco con lo que se indicaría el inicio del bloque, si en las posteriores líneas no introdujéramos el sangrado, significaría el final de dicho bloque de código, con lo



cual para finalizar un bloque de código, solo tenemos que dejar de introducir el sangrado, no tenemos que usar ninguna llave ni símbolo.

Tenemos que ser muy conscientes del sangrado que realizamos ya que de ello dependerá la lógica de nuestro código.

A continuación podemos ver un ejemplo de indentación de bloques de código:



Al usar la indentación nos ahorraremos el uso de símbolos como las llaves {} que en Python se usan para definir diccionarios y para formatear cadenas de texto.

Ejemplo de indentado

```
if pwd == 'manzana':  
    print('Iniciando sesión ...')  
else:  
    print('Contraseña incorrecta.')  
  
print('¡Todo terminado!')
```

Las líneas `print('Iniciando sesión ...')` y `print('Contraseña incorrecta')` son dos bloques de código separados. Estos resultan tener solo una línea de longitud, pero Python le permite escribir bloques de código que constan de cualquier número de declaraciones.

Para indicar un bloque de código en Python, debes indentar cada línea del bloque en la misma cantidad. Los dos bloques de código en nuestro ejemplo sentencia `if` están endentados con cuatro espacios que es una cantidad típica de sangría para Python.



En la mayoría de los otros lenguajes de programación, la indentación se usa solo para ayudar a que el código se vea bonito. Pero en Python, es necesario para indicar a qué bloque de código pertenece una declaración. Por ejemplo, la sentencia final print ("¡Todo terminado!") No está endentada, por lo que no es parte del bloque else.

Los programadores familiarizados con otros lenguajes a menudo se irritan ante la idea de que la indentación importa: a muchos programadores les gusta la libertad de formatear su código como les plazca. Sin embargo, las reglas de indentación en Python son bastante simples y la mayoría de los programadores ya utilizan indentación para hacer que su código sea legible. Python simplemente lleva esta idea un paso más allá y le da significado a la sangría.

Sentencias compuestas

Las sentencias compuestas contienen (grupos de) otras sentencias; estas afectan o controlan la ejecución de esas otras sentencias de alguna manera. En general, las sentencias compuestas abarcan varias líneas, aunque en representaciones simples una sentencia compuesta completa puede estar contenida en una línea.

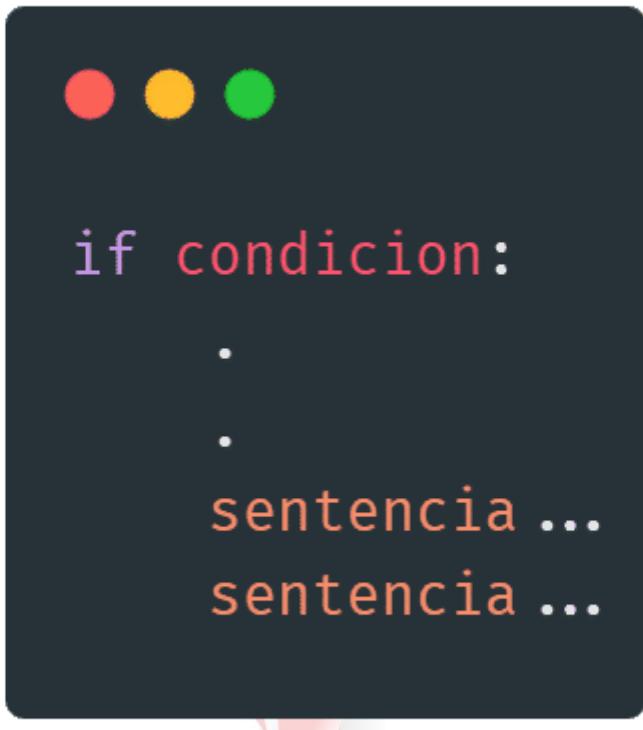
Una sentencia compuesta consta de una o más “cláusulas”. Una cláusula consta de un encabezado y una “suite”.

Encabezado

Los encabezados de cláusula de una declaración compuesta particular están todos en el mismo nivel de indentación. Cada encabezado de cláusula comienza con una palabra clave de identificación única y termina con dos puntos.

Suite

Una suite es un grupo de sentencias controladas por una cláusula. Una suite puede ser una o más sentencias simples separadas por punto y coma en la misma línea como el encabezado, siguiendo los dos puntos del encabezado, o puede ser una o puede ser una o más declaraciones indentadas en líneas posteriores.



```
if condicion:  
    .  
    .  
    .  
    sentencia ...  
    sentencia ...
```

Tipos de sentencias compuestas

Sentencia IF

La sentencia condicional if se usa para tomar decisiones, este evalúa básicamente una operación lógica, es decir una expresión que de como resultado True o False, y ejecuta la pieza de código siguiente siempre y cuando el resultado sea verdadero.

Sentencia WHILE



El ciclo while nos permite realizar múltiples iteraciones basándonos en el resultado de una expresión lógica que puede tener como resultado un valor True o False.

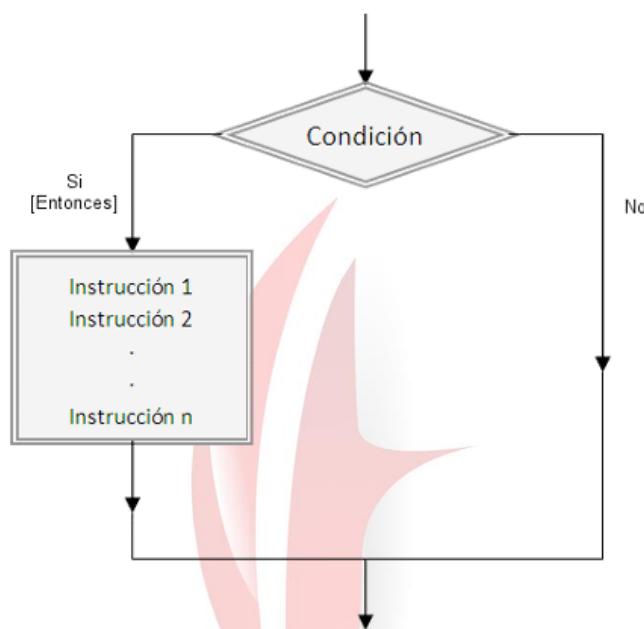
Sentencia FOR

El bucle for se utiliza para recorrer los elementos de un objeto iterable (lista, tupla, conjunto, diccionario, ...) y ejecutar un bloque de código. En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable, sobre el cuál se pueden aplicar una serie de operaciones.





SENTENCIA CONDICIONAL IF



La sentencia condicional if se usa para tomar decisiones, este evalúa básicamente una operación lógica, es decir una expresión que de como resultado True o False, y ejecuta la pieza de código siguiente siempre y cuando el resultado sea verdadero.

Una sentencia if en Python esencialmente dice:

"Si la expresión evaluada, resulta ser verdadera(True), entonces ejecuta una vez el código en la expresión. Si sucede el caso contrario y la expresión es falsa, entonces No ejecutes el código que sigue."

La sintaxis general para la sentencia if básica es como lo siguiente:

```
if condición:  
    bloque de código
```

Una sentencia if consiste en:

- La palabra reservada if , da inicio al condicional if .



- La siguiente parte es la condición. Esta puede evaluar si la declaración es verdadera o falsa. En Python estas son definidas por las palabras reservadas (True or False).
- Paréntesis () Los paréntesis son opcionales, no obstante, ayudan a mejorar la legibilidad del código cuando más de una condición está presente.
- Dos puntos : cuya función es separar la condición de la declaración de ejecución siguiente.
- Una nueva línea.
- Un nivel de indentación de cuatro espacios, que es una convención en Python. El nivel de indentación es asociado con la estructura de la declaración que sigue.
- Finalmente, la estructura de la sentencia. Este es el código que será ejecutado, únicamente si la sentencia a ser evaluada es verdadera. Es posible tener múltiples líneas en la estructura de código que pueden ser ejecutadas; en este caso es necesario tener cautela en cuanto a que todas las líneas tengan el mismo nivel de indentación.

Ejemplo:

```
a = 1
b = 2

if b > a:
    print(" b es mayor que a")
```

En el ejemplo anterior, fueron creadas dos variables; a y b, a las cuales se les asignaron los valores de 1 y 2, respectivamente.

La frase en la sentencia "print", permitirá a la consola imprimir la sentencia solo en caso de cumplirse la condición $b > a$. Puesto que esta condición fue evaluada como verdadera(True), la instrucción fue ejecutada. Si no hubiese sido verdadera (False), la instrucción no se habría ejecutado.



Es decir; si en cambio hubiésemos tenido:

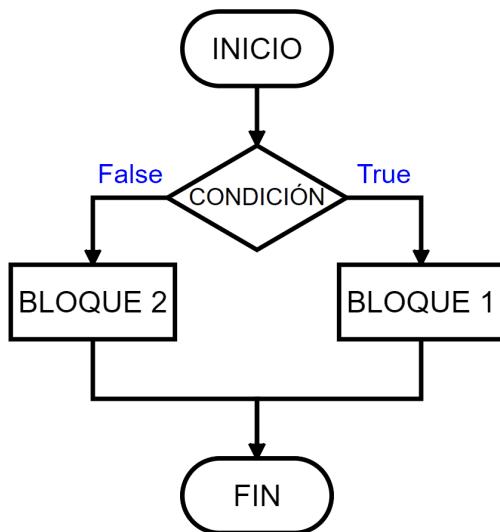
```
a = 1  
b = 2  
  
if a > b  
    print("a es mayor que b")
```

La consola no hubiera ejecutado la instrucción(sin importar cuál fuere) debido a que la condición a evaluar no es verdadera.





SENTENCIA ELSE



La estructura de control if ... else ... permite que un programa ejecute unas instrucciones cuando se cumple una condición y otras instrucciones cuando no se cumple esa condición. En inglés "if" significa "si" (condición) y "else" significa "si no".

La orden en Python se escribe así:

```
● ● ●  
if condición:  
    #bloque de sentencias 1 aquí van las órdenes que se ejecutan  
    #si la condición es cierta y que pueden ocupar varias líneas  
else:  
    #bloque de sentencias 1 aquí van las órdenes que se ejecutan  
    #si la condición es falsa y que también pueden ocupar varias líneas
```

The screenshot shows a dark-themed Python code editor. At the top left, there are three colored circular icons: red, yellow, and green. The main area contains a snippet of Python code demonstrating an if...else... conditional structure. The code uses triple quotes for multi-line strings and includes comments explaining the purpose of each section.

La ejecución de esta construcción es la siguiente:



- Si el resultado es True se ejecuta solamente el bloque de sentencias.
- Si el resultado es False se ejecuta solamente el bloque de sentencias.

La primera línea contiene la condición a evaluar. Esta línea debe terminar siempre por dos puntos (:).

A continuación, viene el bloque de órdenes que se ejecutan cuando la condición se cumple (es decir, cuando la condición es verdadera). Es importante señalar que este bloque debe ir sangrado, puesto que Python utiliza el sangrado para reconocer las líneas que forman un bloque de instrucciones. El sangrado que se suele utilizar en Python es de cuatro espacios, pero se pueden utilizar más o menos espacios. Al escribir dos puntos (:) al final de una línea, IDLE sangrará automáticamente las líneas siguientes. Para terminar un bloque, basta con volver al principio de la línea.

Después viene la línea con la orden else, que indica a Python que el bloque que viene a continuación se tiene que ejecutar cuando la condición no se cumpla (es decir, cuando sea falsa). Esta línea también debe terminar siempre por dos puntos (:). La línea con la orden else no debe incluir nada más que el else y los dos puntos.

En último lugar está el bloque de instrucciones sangrado que corresponde al else.

Ejemplo:

Aquí vemos un ejemplo bastante sencillo para entender la sentencia ELSE, la variable a guarda el resultado de la suma de 2+3, posteriormente evaluamos si la suma es igual a 4, si esto es correcto mostrar el mensaje "Es igual a cuatro", caso contrario va imprimir "No cumple la condición".

```
a = 2 + 3
if a == 4: #condicion si a es exactamente cuatro, entonces(:)
    print ("A es igual a cuatro") # Imprimir
else:
    print ("No se cumple la condicion")
#Resultado: "No se cumple la condicion"
```



Recortando sentencia IF

Si solo tiene una instrucción para ejecutar, puede colocarla en la misma línea que la instrucción if.



```
if a > b: print("a is greater than b")
```

Recortando sentencia IF-ELSE

Si solo tiene una instrucción para ejecutar, una para si y otra para else, puede ponerlo todo en la misma línea:

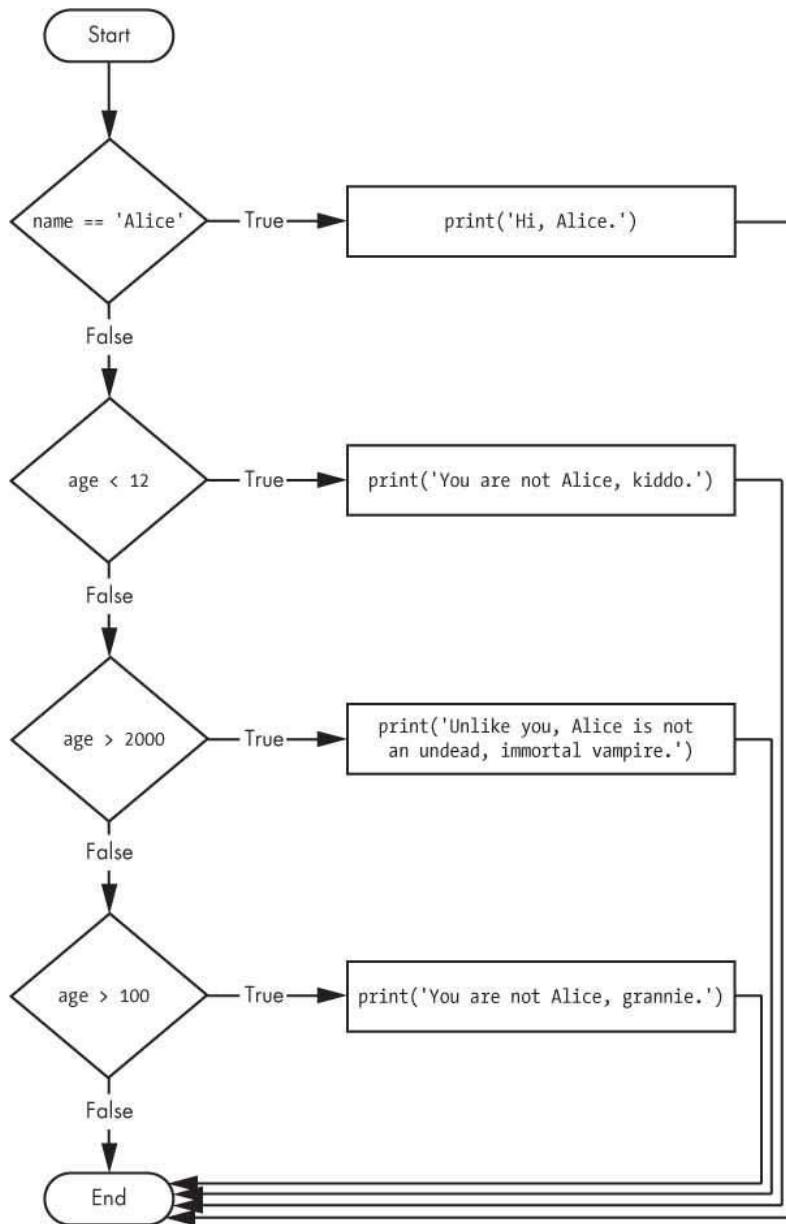


```
a = 2
b = 330
print("A") if a > b else print("B")
```



SENTENCIA CONDICIONAL

ELIF





La estructura de control switch – case no existe en Python. Una forma directa de simularlo es encadenando diversas sentencias if elif para todos los casos necesarios. Otra alternativa común es el uso de diccionarios, donde a cada caso se asocia una función con su código correspondiente.

En Python no existe la estructura de control switch porque sus desarrolladores no se pusieron de acuerdo y no llegaron a una solución satisfactoria para su implementación. Entre los motivos principales destaco, por un lado, la sintaxis, pues no les convencía ninguna propuesta, y por otro, en mi opinión, la poca voluntad o disposición general para hacerlo.

Switch con sentencias IF

Lo primero que solemos pensar como alternativa al switch es en utilizar diversas condicionales if. También suele pasar que mucha gente que está iniciándose en la programación, con independencia del lenguaje, suele desconocer la existencia de estructuras switch y tender de manera natural a esta solución con condicionales.

Se trata de escribir en secuencia distintas condicionales, donde cada una de esas condicionales gestiona un caso diferente. Vamos a verlo con un ejemplo.

Supongamos que necesitas un programa en Python que te muestre el día de la semana a partir de un número del 1 al 7, es decir, que muestre «lunes» si el día es 1, «martes» si el día es 2, y así sucesivamente. Si el día no está entre 1 y 7 mostraremos un error:

```
dia = 4

if dia == 1:
    print('lunes')
if dia == 2:
    print('martes')
if dia == 3:
    print('miércoles')
if dia == 4:
    print('jueves')
if dia == 5:
    print('viernes')
if dia == 6:
    print('sábado')
if dia == 7:
    print('domingo')
if dia < 1 or dia > 7:
    print('error')
```

Fíjate en este ejemplo y entenderás enseguida por qué no es una opción aceptable:



```
dia = 4

if dia == 1:
    print('lunes')
else:
    if dia == 2:
        print('martes')
    else:
        if dia == 3:
            print('miércoles')
        else:
            if dia == 4:
                print('jueves')
            else:
                if dia == 5:
                    print('viernes')
                else:
                    if dia == 6:
                        print('sábado')
                    else:
                        if dia == 7:
                            print('domingo')
                        else:
                            print('error')
```

Switch con sentencias IF

Tal vez esta sea la solución preferida por muchos. Ten en cuenta que esta solución implica que todas las ramas son excluyentes entre sí, es decir, solo se puede ejecutar una. Veamos como transformar el ejemplo anterior antes de ver sus ventajas e inconvenientes. Para lograrlo puedes hacer lo siguiente encadenando sentencias if elif:

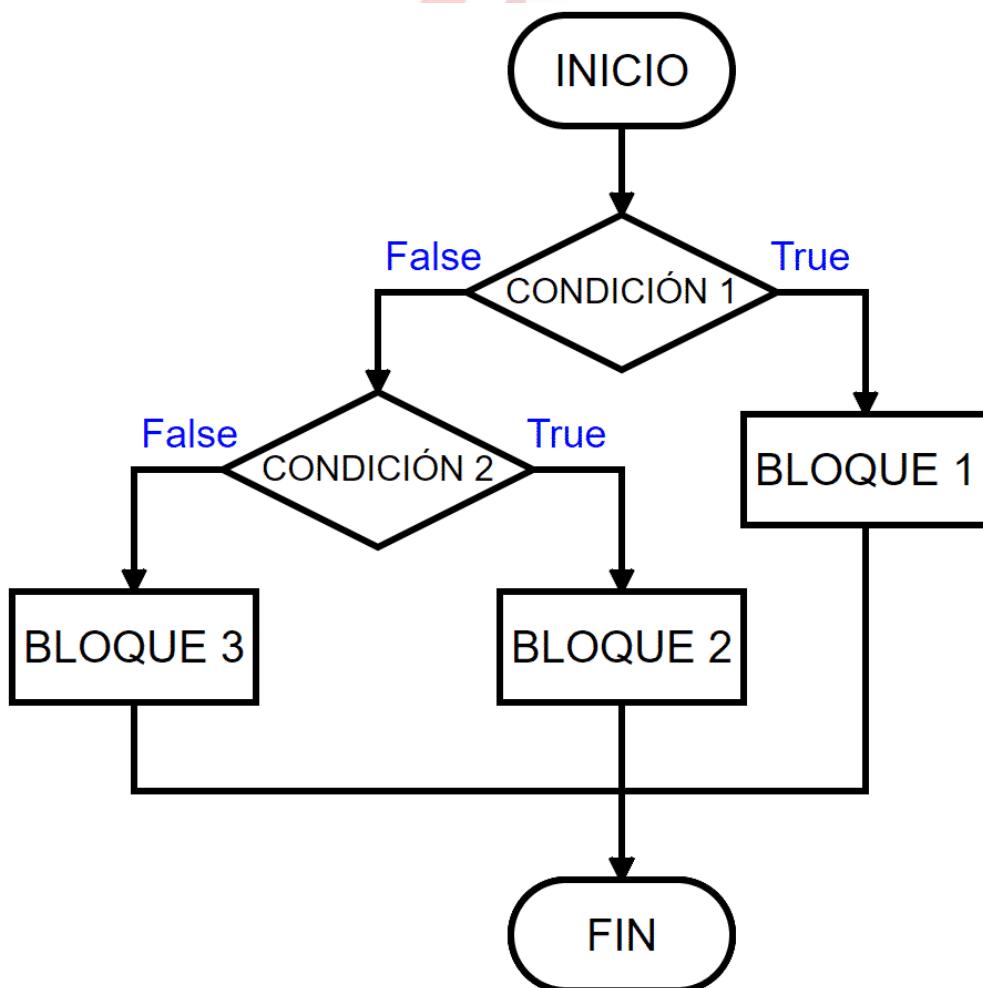
```
dia = 4

if dia == 1:
    print('lunes')
elif dia == 2:
    print('martes')
elif dia == 3:
    print('miércoles')
elif dia == 4:
    print('jueves')
elif dia == 5:
    print('viernes')
elif dia == 6:
    print('sábado')
elif dia == 7:
    print('domingo')
else:
    print('error')
```



CONDICIONALES ANIDADAS

Los condicionales, permiten escribir código en su interior y en realidad, nada de impide incluso al interior de un condicional, poner otros (u otros). A eso se le llama condiciones anidados, pues una estructura condicional dentro de otra. De hecho, puedes anidar cuantos condicionales requieras, aunque no se recomienda más de dos o tres niveles.



El diagrama de flujo que se presenta contiene dos estructuras condicionales. La principal se trata de una estructura condicional compuesta y la segunda es una



estructura condicional simple y está contenida por la rama del falso de la primer estructura, también puede anidarse cuando el resultado es verdadero.

Es común que se presenten estructuras condicionales anidadas aún más complejas.

Ejemplo 1:

El siguiente código verifica si una persona es mayor de edad pero también verifica si es un persona real.

```
edad=int(input("Humano ingresa tu edad:"))

if 0 <= edad < 100:
    if edad >= 18:
        print("El humano es mayor de edad")
    else:
        print("El humano es menor de edad")
else:
    print("Me quieres ver la cara? Esa edad no existe.")
```

Ejemplo 2:

El siguiente es un algoritmo simple donde se verifica si una contraseña ingresada es correcta y contiene condicionales en ambos casos, si es que no cumple y también si cumple, veamos:



```
password = input("Ingrese la contraseña: ")

if (len(password) >= 8):
    print('Tu contraseña es suficientemente larga.')

    if(password == 'miClaveSegura'):
        print("Además es la contraseña correcta.")
    else:
        print("Pero es incorrecta.")
else:
    print('Tu contraseña es muy corta e insegura.')

if (password != 'miClaveSegura'):
    print("Además, es incorrecta (por supuesto).")
```

Anidación doble

Aunque en algunos lenguajes puede llegar a confundir el código, una anidación doble o de segundo nivel es cuando colocamos una segunda condicional dentro de la anidada, veamos el siguiente ejemplo:

Ejercicio:

Realizar un programa en Python que permita verificar si la nota de un estudiante es Reprobado,Aceptable o Excelente, el dato debe ser ingresado por el usuario. La nota ingresada debe ser menor o igual a 10 y mayor o igual a 0.

- Reprobado = Abajo de 5
- Aceptable = Entre 5.1 y 8.5
- Excelente = Arriba de 8.5



```
nota = float(input("Ingresa tu nota: "))

if nota <= 10:
    if nota >= 0:
        if nota > 8.5:
            print("Su nota es Excelente..")
        elif nota >= 5.1:
            print("Su nota es aceptable..")
        else:
            print("Su nota es reprobado..")
    else:
        print("La nota no es válida...")
else:
    print("La nota no es válida...")
```



APLICANDO OPERADORES LÓGICOS



En esta sección vamos a aplicar los operadores lógico que hemos visto en el tema anterior, su función es la misma pues nos devuelve un dato booleano y sentencia if evalúa este tipo de datos para ejecutar un bloque de código dependiendo del resultado.

Operador AND

El operador and evalúa si el valor a la izquierda y el de la derecha son True, y en el caso de ser cierto, devuelve True. Si uno de los dos valores es False, el resultado será False. Es realmente un operador muy lógico e intuitivo que incluso usamos en la vida real. Si hace sol y es fin de semana, iré a la playa. Si ambas condiciones se cumplen, es decir que la variable `haceSol=True` y la variable `finDeSemana=True`, iré a la playa, o visto de otra forma `irALaPlaya=(haceSol and finDeSemana)`.

Ejemplo: El servicio militar solicita postulantes que sean mayores de 18 años y con una estatura mayor o igual a 1.75m , solicitar estos datos para verificar si



cumplen esta condición, si cumplen imprimir el mensaje indicando que cumple las condiciones sino imprimir lo contrario.

```
print("Preselección de postulantes")

edad = int(input("Ingresa tu edad: "))
estatura =int(input("Ingresa tu estatura en cm: "))

if edad >= 18 and estatura >= 175:
    print("Felicitaciones cumples con las condiciones, has aprobado la preselección..")
else:
    print("Lo sentimos no cumples con las condiciones para la preselección..")
```

Operador OR

El operador or devuelve True cuando al menos uno de los elementos es igual a True. Es decir, evalúa si el valor a la izquierda o el de la derecha son True.

Ejemplo: Una tienda de ropa tiene descuentos de 25% si los clientes cumplen con una de las siguientes condiciones:

- Es cliente nuevo
- Cuenta con una tarjeta de cliente recurrente

Realizar un programa en Python que permita al cliente consultar si tiene el descuento de 25%.

```
print("Consulta el 25% de descuento..")

c_n = input("Ud es cliente nuevo s/n: ")
c_r = input("Tiene tarjeta de clientes s/n: ")

if c_n == "s" or c_r == "s":
    print("Felicitaciones, todas su compras tendran un descuento del 25%")
else:
    print("Lo sentimos, no puede acceder al descuento")
```

Operador NOT

Y por último tenemos el operador not, que simplemente invierte True por False y False por True. También puedes usar varios not juntos y simplemente se irán aplicando uno tras otro. La verdad que es algo difícil de ver en la realidad, pero simplemente puedes contar el número de not y si es par el valor se quedará igual. Si por lo contrario es impar, el valor se invertirá.



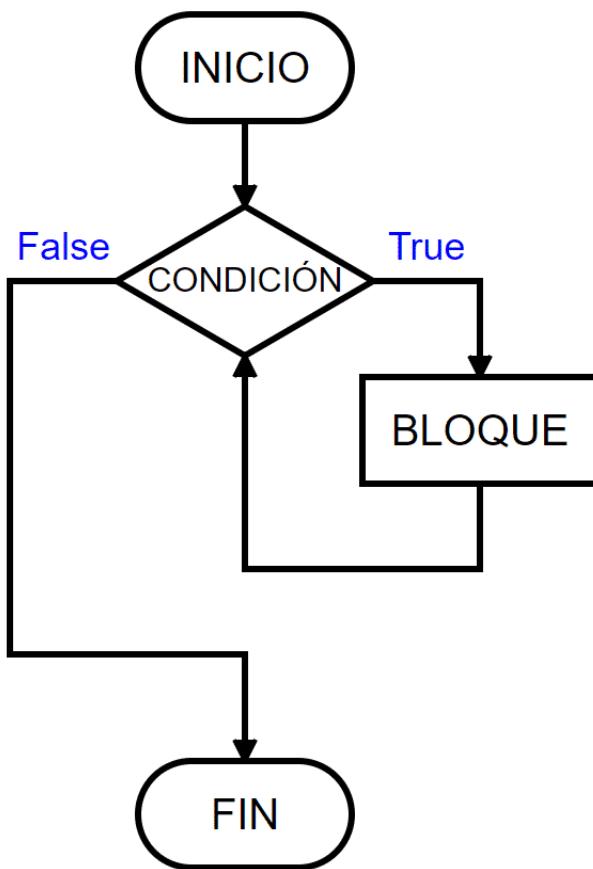
Ejemplo: Verificar si una palabra no contiene la letra e imprimir correcto.

```
p = input("ingrese cualquier palabra: ")  
  
if not "a" in p:  
    print("Correcto!!")
```





SENTENCIA WHILE



La estructura repetitiva mientras (en inglés while), es aquella en que el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando se ejecuta la instrucción mientras, la primera cosa que sucede es evaluar la condición (una expresión booleana). Si la condición se evalúa falsa, no se entra al ciclo y se sigue con el flujo normal del problema. Si la condición es verdadera, entonces se entra al ciclo y se ejecuta el cuerpo del bucle (instrucciones dentro del mientras), después se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez mientras la condición sea verdadera.

Sintaxis WHILE en Python



```
while <expr>:  
    <statement(s)>
```

En el bucle while, primero se comprueba la condición. En el bloque de código se introduce sólo si el evalúa a la condición a True . Después de una iteración, la expresión de prueba se comprueba de nuevo. Este proceso continúa hasta que se evalúa a la condición a False.

En Python, el cuerpo del bucle while se determina a través de la sangría(indentación). El bloque de código comienza con la indentación sea de 2 o 4 espacios y la primera línea sin sangría marca el final.

Ejemplo

Escribir un programa que permita sumar los números consecutivos desde el 1 hasta el 10.

```
# Inicializar variable suma y contador  
sum = 0  
i = 1  
  
while i <= 10:  
    sum = sum + i  
    i = i+1      # Actualiza el contador  
# Imprime la suma  
print("La suma es", sum)
```

Sentencia WHILE con ELSE

Python permite una cláusula opcional al final de un bucle. Esta es una característica única de Python, que no se encuentra en la mayoría de los otros lenguajes de programación. La sintaxis se muestra a continuación:

```
while <expr>:  
    <statement(s)>  
else:  
    <additional_statement(s)>
```



Ejemplo:

```
n = 5
while n > 0:
    n -= 1
    print(n)
else:
    print('Loop done.')
```





SENTENCIA BREAK - CONTINUE

```
→ while <expr>:  
    <statement>  
    <statement>  
    break ——————  
    <statement>  
    <statement>  
    continue  
    <statement>  
    <statement>  
  
    <statement> ←
```

Python proporciona dos palabras clave que terminan una iteración de bucle prematuramente:while

La instrucción de Break de Python termina inmediatamente un bucle por completo. La ejecución del programa procede a la primera instrucción después del cuerpo del bucle.

La instrucción Continue de Python finaliza inmediatamente la iteración de bucle actual. La ejecución salta a la parte superior del bucle y la expresión de control se vuelve a evaluar para determinar si el bucle se ejecutará de nuevo o terminará.



Break

En Python, la instrucción break le proporciona la oportunidad de cerrar un bucle cuando se activa una condición externa. Debe poner la instrucción break dentro del bloque de código bajo la instrucción de su bucle, generalmente después de una instrucción if condicional.

Veamos un ejemplo en el que se utiliza la instrucción break en un bucle for:

```
number = 0

while number <= 10:
    if number == 5:
        break      # break here
    print('Number is ' + str(number))
    number += 1

print('Fuera del bucle')
```

En este pequeño programa, la variable number se inicia en 0. Luego, una instrucción for construye el bucle siempre que la variable number sea inferior a 10. En el bucle existe una instrucción if que presenta la condición de que si la variable number es equivalente al entero 5, entonces el bucle se romperá.

En el bucle también existe una instrucción print() que se ejecutará con cada iteración del bucle for hasta que se rompa el bucle, ya que está después de la instrucción break. Para saber cuándo estamos fuera del bucle, hemos incluido una instrucción print() final fuera del bucle for.

Continue

La instrucción continue da la opción de omitir la parte de un bucle en la que se activa una condición externa, pero continuar para completar el resto del bucle. Es decir, la iteración actual del bucle se interrumpirá, pero el programa volverá a la parte superior del bucle.

La instrucción continue se encuentra dentro del bloque de código abajo de la instrucción del bucle, generalmente después de una instrucción if condicional.

Usando el mismo programa de bucle for que en la sección anterior Instrucción break, emplearemos la instrucción continue en vez de la instrucción break:



```
number = 0

while number <= 10:
    if number == 5:
        continue      # continua
    print('Number is ' + str(number))
    number += 1

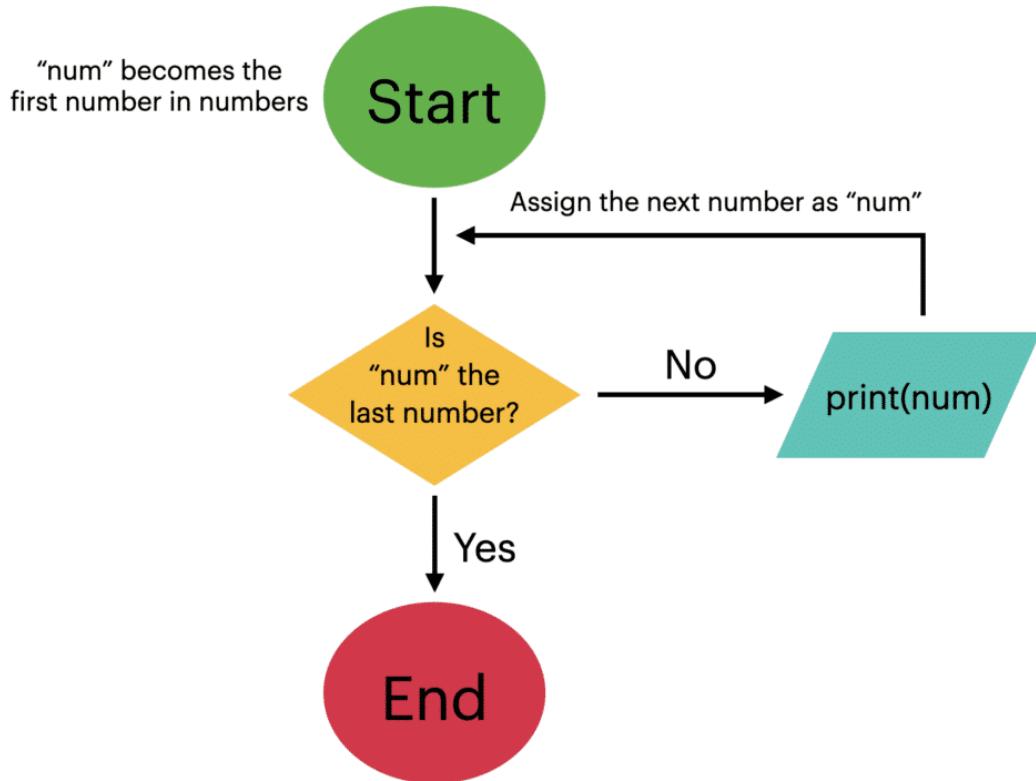
print('Fuera del bucle')
```





SENTENCIA REPETITIVA FOR

numbers = [1, 2, 3, 4, 5]



El for es un tipo de bucle, parecido al while pero con ciertas diferencias. La principal es que el número de iteraciones de un for esta definido de antemano, mientras que en un while no. La diferencia principal con respecto al while es en la condición. Mientras que en el while la condición era evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el for no existe tal condición, sino un iterable que define las veces que se ejecutará el código.

Sintaxis FOR en Python

```
for <elem> in <iterable>:  
    <Tu código>
```



En el siguiente ejemplo vemos un bucle for que se ejecuta 5 veces, y donde la i incrementa su valor “automáticamente” en 1 en cada iteración.

```
for i in [1, 2, 3, 4, 5]:  
    print(i, end=", ") # prints: 1, 2, 3, 4, 5
```

Iterables e Iteradores

Para entender al cien por cien los bucles for, y como Python fue diseñado como lenguaje de programación, es muy importante entender los conceptos de iterables e iteradores. Empecemos con un par de definiciones:

Iterables

Los iterables son aquellos objetos que como su nombre indica pueden ser iterados, lo que dicho de otra forma es, que puedan ser indexados. Si piensas en un array (o una list en Python), podemos indexarlo con lista[1] por ejemplo, por lo que sería un iterable.

```
#for <variable> in <iterable>:  
#   <Código>
```

Tiene bastante sentido, porque si queremos iterar una variable, esta variable debe ser iterable, todo muy lógico. Pero llegados a este punto, tal vez de preguntes ¿pero cómo se yo si algo es iterable o no?. Bien fácil, con la siguiente función `isinstance()` podemos saberlo. No te preocupes si no entiendes muy bien lo que estamos haciendo, fíjate solo en el resultado, True significa que es iterable y False que no lo es.

```
from collections import Iterable  
lista = [1, 2, 3, 4]  
cadena = "Python"  
numero = 10 print(isinstance(lista, Iterable)) #True  
print(isinstance(cadena, Iterable)) #True  
print(isinstance(numero, Iterable)) #False
```

Por lo tanto las listas y las cadenas son iterables, pero numero, que es un entero no lo es. Es por eso por lo que no podemos hacer lo siguiente, ya que daría un error. De hecho el error sería `TypeError: int' object is not iterable`.



```
numero = 10
#for i in numero:
#    print(i)
```

Iteradores

Los iteradores son objetos que hacen referencia a un elemento, y que tienen un método next que permite hacer referencia al siguiente.

Para entender los iteradores, es importante conocer la función iter() en Python. Dicha función puede ser llamada sobre un objeto que sea iterable, y nos devolverá un iterador como se ve en el siguiente ejemplo.

```
lista = [5, 6, 3, 2]
it = iter(lista)
print(it)    #<list_iterator object at 0x106243828>
print(type(it)) #<class 'list_iterator'>
```

Vemos que al imprimir it es un iterador, de la clase list_iterator. Esta variable iteradora, hace referencia a la lista original y nos permite acceder a sus elementos con la función next(). Cada vez que llamamos a next() sobre it, nos devuelve el siguiente elemento de la lista original. Por lo tanto, si queremos acceder al elemento 4, tendremos que llamar 4 veces a next(). Nótese que el iterador empieza apuntando fuera de la lista, y no hace referencia al primer elemento hasta que no se llama a next() por primera vez.

```
lista = [5, 6, 3, 2]
it = iter(lista)
print(next(it))
#  [5, 6, 3, 2]
#   ^
#   /
#   it
print(next(it))
#  [5, 6, 3, 2]
#   ^
#   /
#   it
```



```
print(next(it))
# [5, 6, 3, 2]
# ^
# /
# it
```





PROGRAMACIÓN FUNCIONAL



Programación Funcional

En informática, la programación funcional es un paradigma de programación declarativa basado en el uso de verdaderas funciones matemáticas. En este estilo de programación las funciones son ciudadanas de primera clase, porque sus expresiones pueden ser asignadas a variables como se haría con cualquier otro valor; además de que pueden crearse funciones de orden superior.

Ventajas de usar un paradigma funcional

El paradigma funcional es popular porque ofrece varias ventajas sobre otros paradigmas de programación. El código funcional es:

Alto nivel

Está describiendo el resultado que desea en lugar de especificar explícitamente los pasos necesarios para llegar allí. Las declaraciones individuales tienden a ser concisas pero tienen mucho impacto.



Transparente

El comportamiento de una función pura depende únicamente de sus entradas y salidas, sin valores intermedios. Eso elimina la posibilidad de efectos secundarios, lo que facilita la depuración.

Paralelizable

Las rutinas que no causan efectos secundarios pueden ejecutarse más fácilmente en paralelo entre sí.

Utilidad

La programación funcional se caracteriza por dividir la mayor cantidad posible de tareas en funciones, de esta forma estas tareas pueden ser usadas por otras funciones con diferentes objetivos.

El objetivo es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitar el concepto de estado del cómputo. La secuencia de computaciones llevadas a cabo por el programa se rige única y exclusivamente por la reescritura de definiciones más amplias a otras cada vez más concretas y definidas.

Simulación de estados

Hay tareas (como por ejemplo, el mantenimiento del saldo de una cuenta bancaria) que a menudo parecen implementadas con estados. La programación funcional pura actúa sobre esas tareas, tareas de entrada/salida de datos tales como entrada de datos por parte del usuario y mostrar resultados por pantalla, de una forma diferente.

Cuestiones de eficiencia

Los lenguajes de programación en este paradigma son típicamente menos eficientes en el uso de CPU y memoria que los lenguajes imperativos como pueden ser C y Pascal. Esto está relacionado con el hecho de que algunas estructuras de datos de tamaño indefinido como los vectores tienen una programación muy sencilla usando el hardware existente, el cual es una máquina de Turing bastante evolucionada.



Muchos lenguajes de programación admiten cierto grado de programación funcional. En algunos lenguajes, prácticamente todo el código sigue el paradigma funcional. Haskell es uno de esos ejemplos. Python, por el contrario, admite la programación funcional pero también contiene características de otros modelos de programación.

Programación Funcional - Python

La programación funcional generalmente juega un papel bastante pequeño en el código de Python. Pero es bueno estar familiarizado con él. Como mínimo, probablemente lo encontrará de vez en cuando al leer código escrito por otros. Incluso puede encontrar situaciones en las que sea ventajoso utilizar las capacidades de programación funcional de Python en su propio código.

Sin características o bibliotecas especiales de Python, podemos comenzar a codificar de una manera más funcional.

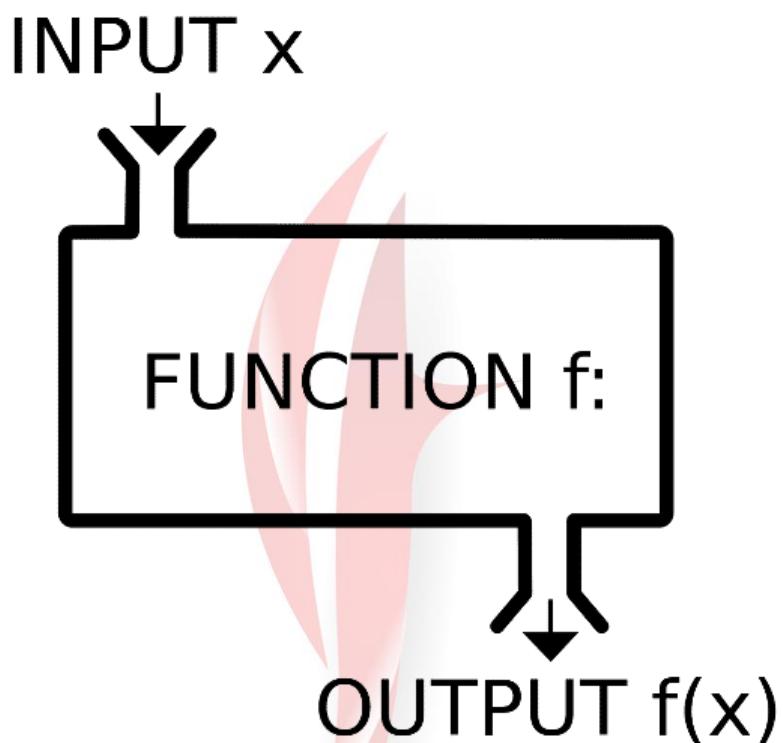
Python tiene soporte parcial para la programación funcional como un lenguaje multiparadigma. Algunas soluciones Python de programas matemáticos se pueden lograr más fácilmente con un enfoque funcional.

El cambio más difícil de hacer cuando comienza a usar un enfoque funcional es reducir la cantidad de clases que usa. Las clases en Python tienen atributos mutables que dificultan la creación de funciones puras e inmutables.

En su lugar, intente mantener la mayor parte de su código en el nivel del módulo y solo cambie a clases si es necesario.



DEFINIENDO FUNCIÓN



Las funciones se conocen con varios nombres en los lenguajes de programación, por ejemplo, como subrutinas, rutinas, procedimientos, métodos o subprogramas.

Una función es un elemento de estructuración en los lenguajes de programación para agrupar un conjunto de declaraciones para que puedan utilizarse en un programa más de una vez. La única forma de lograr esto sin funciones sería reutilizar el código copiándolo y adaptándolo a diferentes contextos, lo cual sería una mala idea. Debe evitarse el código redundante (código repetido en este caso). El uso de funciones generalmente mejora la comprensibilidad y la calidad de un programa. También reduce el costo de desarrollo y mantenimiento del software.



¿Qué es una función en Python?

En Python, una función es un grupo de declaraciones relacionadas que realizan una tarea específica. Las funciones ayudan a dividir nuestro programa en partes más pequeñas y modulares. A medida que nuestro programa crece más y más, las funciones lo hacen más organizado y manejable.

Además, evita la repetición y hace que el código sea reutilizable. Como ya sabe, Python le ofrece muchas funciones integradas como `print()`, etc., pero también puede crear sus propias funciones. Estas funciones se denominan funciones definidas por el usuario.

Definiendo una función

Puede definir funciones para proporcionar la funcionalidad requerida. Aquí hay reglas simples para definir una función en Python.

- Los bloques de funciones comienzan con la palabra clave `def` seguida del nombre de la función y paréntesis "()".
- Cualquier parámetro de entrada o argumento debe colocarse entre estos paréntesis. También puede definir parámetros dentro de estos paréntesis.
- La primera declaración de una función puede ser una declaración opcional: la cadena de documentación de la función o `docstring`.
- El bloque de código dentro de cada función comienza con dos puntos (:) y está sangrado.
- La instrucción `return [expresión]` sale de una función y, opcionalmente, devuelve una expresión a la persona que llama. Una declaración de devolución sin argumentos es lo mismo que devolver `None`.

Sintaxis de una función en Python

```
def nombrefuncion( parametros ):  
    bloque de código  
    return [expresión]
```



Ejemplo:

```
def my_function():
    print("Hola desde una función")
```





LLAMANDO FUNCIÓN



Llamando un función

Las funciones son bloques de código que se pueden reutilizar simplemente llamando a la función. Esto permite la reutilización de código simple y elegante sin volver a escribir explícitamente secciones de código. Esto hace que el código sea más legible, facilita la depuración y limita los errores de escritura.

Una vez has definido una función, el código no se ejecutará por sí solo. Para ejecutar el código dentro de la función, tienes que hacer una invocación de la función, o una llamada a la función.

Puedes llamar a la función tantas veces como lo deseas, para llamar a la función solo necesitas hacer lo siguiente:

```
nombre_funcion(argumentos)
```



Ejemplo #1:

```
def saludo():  
    print("Hola, buenos días!")  
  
saludo()
```

Salida: Hola, buenos días!

En el anterior código hemos definido una función llamada "saludo", dentro de esta imprimimos por pantalla un texto, luego en la parte de abajo la hemos llamado y esta se ejecuta.

Ejemplo #2:

```
def operar():  
    sum = 1 + 5  
    print(sum)  
  
operar()
```

Salida: 6

En el anterior código hemos definido una función llamada "operar", dentro de esta declaramos una variable que será el resultado de una suma entre 2 numeros "sum" y luego este resultado será imprimido por pantalla.

Saliendo de la función, vamos a llamarla para que esta se ejecute imprimiendo así, el resultado de la suma.

Nota: En python, la definición de la función siempre debe estar presente antes de la llamada a la función. De lo contrario, obtendremos un error.



Por ejemplo:

```
#Llamando función
```

```
operar()
```

```
#Definiendo función
```

```
def operar():
```

```
    sum = 1 + 5
```

```
    print(sum)
```

Salida: #Error: name 'operar' is not defined





PASO DE PARÁMETROS



Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder.

Parámetro o argumento

Puede que también hayas escucha la la palabra argumento para referirse a un parámetro, a continuación veremos las definiciones correctas.

Parámetros:

Un parámetro es la variable definida entre paréntesis durante la definición de la función. Simplemente se escriben cuando declaramos una función.



Argumentos:

Un argumento es un valor que se pasa a una función cuando se llama. Puede ser una variable, valor u objeto pasado a una función o método como entrada. Se escriben cuando estamos llamando a la función.

Tenemos por un lado la declaración de la función por medio de un nombre y el algoritmo de la función seguidamente. Luego para que se ejecute la función la llamamos desde el bloque principal de nuestro programa.

Ahora veremos que una función puede tener parámetros para recibir datos. Los parámetros nos permiten comunicarle algo a la función y la hace más flexible.

Ejemplo #1:

```
def saludo(nombre):  
    print("Hola, "+nombre+ " buenos días!")  
  
saludo("Julio")
```

Salida: Hola, Julio buenos días!

En el anterior código hemos definido una función llamada "saludo", esta la creamos con un parámetro que lo denominamos "nombre" y dentro de esta imprimimos por pantalla un texto con la variable que ingresamos como parámetro.

Luego en la parte de abajo la hemos llamado con un argumento tipo string y esta se ejecuta.

Ejemplo #2:

```
def operar(num):  
    sum = 1 + num  
    print(sum)
```



```
operar(46)
```

Salida: 47

En el anterior código hemos definido una función llamada "operar", esta la creamos con un parámetro que lo denominamos "num" y dentro de esta hacemos una operación aritmética, imprimimos por pantalla el resultado esta.

Luego en la parte de abajo la hemos llamado con un argumento tipo entero y esta se ejecuta.

Valor por defecto

En Python se pueden definir parámetros y asignarles un dato en la misma cabecera de la función. Luego cuando llamamos a la función podemos o no enviarle un valor al parámetro. Los parámetros por defecto nos permiten crear funciones más flexibles y que se pueden emplear en distintas circunstancias.

```
def resta(a=None, b=None):  
    if a == None or b == None:  
        print("Error, debes enviar dos números a la función")  
    return # indicamos el final de la función aunque no devuelva nada  
    print(a-b)  
  
resta()
```

Salida: Error, debes enviar dos números a la función



PARÁMETROS INDETERMINADOS



Quizá en alguna ocasión no sabemos de antemano cuantos elementos vamos a enviar a una función. En estos casos podemos utilizar los parámetros indeterminados por posición y por nombre.

Parámetro *args

En Python, el parámetro especial `*args` en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

- Lo que realmente indica que el parámetro es de este tipo es el símbolo `*`, el nombre `args` se usa por convención.
- El parámetro recibe los argumentos como una tupla.
- Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.
- El número de argumentos al invocar a la función es variable.
- Son parámetros posicionales, por lo que, a diferencia de los parámetros con nombre, su valor depende de la posición en la que se pasen a la función.



Ejemplo #1

La siguiente función toma dos parámetros y imprime como resultado la suma de los mismos

```
def sum(x, y):  
    print(x + y)  
  
sum(2, 3)
```

Salida: 5

Pero, ¿qué ocurre si posteriormente decidimos o nos damos cuenta de que necesitamos sumar un valor más?

```
def sum(x, y):  
    print(x + y)  
  
sum(2, 3,5)  
  
Traceback (most recent call last): File "<input>", line 1, in  
<module>TypeError: sum() takes 2 positional arguments but 3 were given
```

La mejor solución, la más elegante y la más al estilo Python es hacer uso de *args en la definición de esta función. De este modo, podemos pasar tantos argumentos como queramos. Pero antes de esto, tenemos que re implementar nuestra función sum:



```
def sum(*args):
    value = 0
    for n in args:
        value += n
    print(value)

sum()
sum(2, 3)
sum(2, 3, 4)
sum(2, 3, 4, 6, 9, 21)
```

Salida: 5

9

45

Parámetro **kwargs

En Python, el parámetro especial `**kwargs` en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

Las principales diferencias con respecto `*args` son:

- Lo que realmente indica que el parámetro es de este tipo es el símbolo `**`, el nombre `kwargs` se usa por convención.
- El parámetro recibe los argumentos como un diccionario.



Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

Para recibir un número indeterminado de parámetros por nombre (clave-valor o en inglés keyword args), debemos crear un diccionario dinámico de argumentos definiendo el parámetro con dos asteriscos:

```
def indeterminados_nombre(**kwargs):
    for kwarg in kwargs:
        print(kwarg, "=>", kwargs[kwarg])

indeterminados_nombre(n=5, c="Hola", l=[1,2,3,4,5])
```

Salida: n => 5

c => Hola

l => [1, 2, 3, 4, 5]



PROGRAMACIÓN ORIENTADA A OBJETOS



La programación orientada a objetos (POO) es un método para estructurar un programa mediante la agrupación de propiedades y comportamientos relacionados en objetos individuales.

Conceptualmente, los objetos son como los componentes de un sistema. Piense en un programa como una especie de línea de montaje de fábrica. En cada paso de la línea de ensamblaje, un componente del sistema procesa algún material, transformando finalmente la materia prima en un producto terminado.

Un objeto contiene datos, como los materiales sin procesar o pre-procesados en cada paso de una línea de ensamblaje, y el comportamiento, como la acción que realiza cada componente de la línea de ensamblaje.



POO en Python

La programación orientada a objetos es un paradigma de programación que proporciona un medio para estructurar programas de modo que las propiedades y los comportamientos se agrupan en objetos individuales .

Por ejemplo, un objeto podría representar a una persona con propiedades como nombre, edad y dirección y comportamientos como caminar, hablar, respirar y correr. O podría representar un correo electrónico con propiedades como una lista de destinatarios, asunto y cuerpo y comportamientos como agregar archivos adjuntos y enviar.

Dicho de otra manera, la programación orientada a objetos es un enfoque para modelar cosas concretas del mundo real, como automóviles, así como relaciones entre cosas, como empresas y empleados, estudiantes y profesores, etc. OOP modela entidades del mundo real como objetos de software que tienen algunos datos asociados y pueden realizar ciertas funciones.

La conclusión clave es que los objetos están en el centro de la programación orientada a objetos en Python, no solo representando los datos, como en la programación procedimental, sino también en la estructura general del programa.

Ventajas

Modularidad para facilitar la resolución de problemas

Cuando se trabaja con lenguajes de programación orientados a objetos se tiene una mejor idea de dónde buscar el error cuando algo no está funcionando bien. No tienes que ir línea por línea a través de todo tu código.

Esa es la belleza de la encapsulación los objetos son autónomos. Además, esta modularidad permite que un equipo trabaje en múltiples objetos simultáneamente mientras minimiza la posibilidad de que una persona pueda duplicar la funcionalidad de otra.

Reutilización de código mediante herencia

Suponga que, además de su objeto “auto”, un colega necesita un objeto “auto de carrera” y otro necesita un objeto “limusina”. Todos construyen sus objetos por separado pero descubren puntos en común entre ellos. De hecho, cada objeto es solo un tipo diferente de automóvil. Aquí es donde la técnica de herencia



ahorra tiempo: se puede crear una clase genérica (auto) y luego definir las subclases (auto de carrera y limusina) que heredarán los rasgos de la clase genérica.

Flexibilidad a través del polimorfismo

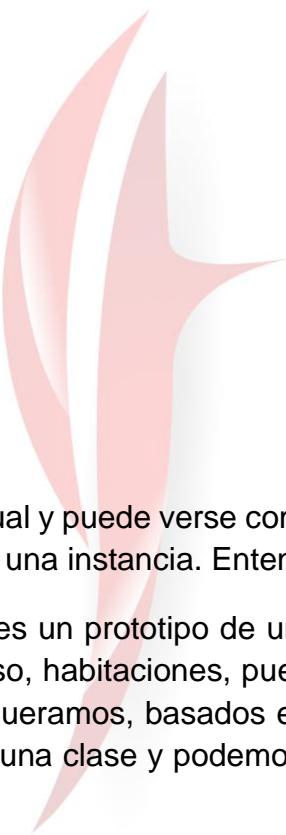
El polimorfismo de la programación orientada a objetos trata de que una sola función puede cambiar de forma para adaptarse a cualquier clase en la que se encuentre. Por ejemplo, podría crear una función en la clase principal “auto” llamada “conducir”, no “conducirAuto” ni “conducirAutoDeCarrera”, sino simplemente “conducir”. Esta función funcionaría tanto para autos de carrera como para limusina u otros. De hecho, incluso podría tener “AutoDeCarrera.conducir (miConductorAutoDeCarrera)” o “limusina.conducir (miConductorPersonal)”.

Resolución efectiva de problemas

Muchas personas evitan aprender la programación orientada a objetos porque la curva de aprendizaje parece más pronunciada que la de la programación de arriba hacia abajo. Pero tómese el tiempo para dominar la programación orientada a objetos y descubrirá que es el enfoque más fácil e intuitivo para desarrollar grandes proyectos. La programación orientada a objetos se trata, en última instancia, de tomar un gran problema y dividirlo en partes solucionables. Para cada problema, escribe una clase que hace lo que necesita. Y luego, lo mejor de todo, puede reutilizar esas clases, lo que hace que sea aún más rápido resolver el siguiente problema.



CLASES



Una clase es una entidad virtual y puede verse como un modelo de un objeto. La clase se creó cuando se creó una instancia. Entendámoslo con un ejemplo.

Supongamos que una clase es un prototipo de un edificio. Un edificio contiene todos los detalles sobre el piso, habitaciones, puertas, ventanas, etc. Podemos hacer tantos edificios como queramos, basados en estos detalles. Por lo tanto, el edificio puede verse como una clase y podemos crear tantos objetos de esta clase.

Por otro lado, el objeto es la instancia de una clase. El proceso de creación de un objeto se puede llamar creación de instancias.

Creando clases en Python

En Python, se puede crear una clase usando la palabra clave `class`, seguida del nombre de la clase. La sintaxis para crear una clase se proporciona a continuación.



Sintaxis:

```
class ClassName:  
    #statement_suite
```

Como puedes ver, la definición de una clase comienza con la palabra clave `class`, y `ClassName` sería el nombre de la clase (identificador). Ten en cuenta que el nombre de la clase sigue las mismas reglas que los nombres de variables en Python, es decir, sólo pueden comenzar con una letra o un subrayado `_`, y sólo pueden contener letras, números o guiones bajos. Además, según PEP 8 (Guía de estilo para la programación en Python), se recomienda que los nombres de las clases estén capitalizadas.

Ejemplo

Ahora vamos a definir una clase `Person` (persona), que por el momento no contendrá nada, excepto la declaración de `pass`. Según la documentación de Python:

La sentencia `pass` no hace nada. Puede ser utilizada cuando se requiere una sentencia sintácticamente pero programa no requiere acción alguna.

Código:

```
class Person:  
    pass
```



OBJETOS



Los objetos son abstracción de Python para data. Toda la data en un programa Python es representado por objectos o por relaciones entre objectos. (En cierto sentido, y en el código modelo de Von Neumann de una «computadora almacenada del programa» también es un código representado por los objetos.)

Cada objeto tiene una identidad, un tipo y un valor. Una identidad de objeto nunca cambia una vez es creada; usted puede pensar eso como la dirección de objeto en memoria. El operador `in` compara la identidad de dos objetos; la función `id()` devuelve un número entero representando la identidad (actualmente implementado como su dirección).

El tipo de un objeto también es inmutable. El tipo de un objeto determina las operaciones que admite el objeto (por ejemplo, «¿tiene una longitud?») Y también define los valores posibles para los objetos de ese tipo.

La función `«type()»` devuelve el tipo de un objeto (que es un objeto en sí mismo).

El valor *de algunos objetos puede cambiar. Se dice que los objetos cuyo valor puede cambiar son *mutables; los objetos cuyo valor no se puede cambiar una



vez que se crean se llaman immutable. La mutabilidad de un objeto está determinada por su tipo; por ejemplo, los números, las cadenas y las tuplas son inmutables, mientras que los diccionarios y las listas son mutables.

Ejemplo:

Vamos a tomar el ejemplo del punto anterior con la clase Person() , y vamos a crear el primer objeto, para crear una instancia (objeto) de esta clase, simplemente haremos lo siguiente:

```
class Person:  
    pass  
  
jorge = Person()
```

Esto significa que hemos creado un nuevo objeto jorge del tipo Person. Date cuenta que para crear un objeto solo debemos escribir el nombre de la clase, seguido de unos paréntesis.

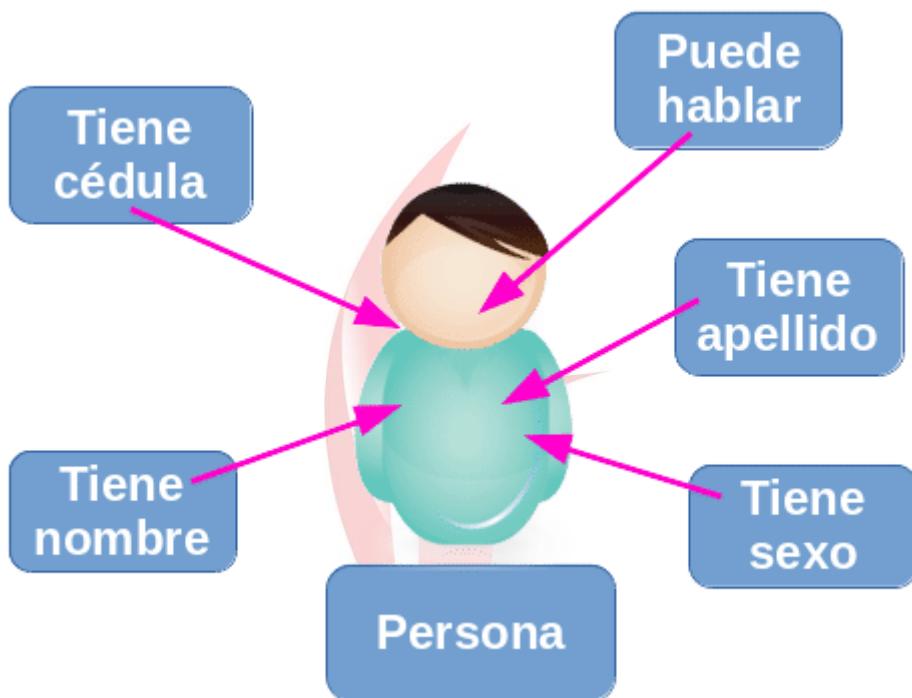
Podemos identificar de qué clase es jorge, y qué espacio ocupa en la memoria escribiendo: print jorge. En ese caso, obtendrás algo similar a:

```
print(jorge)
```

Salida: <__main__.Person instance at 0x109a1cb48>



ATRIBUTOS



Los atributos o propiedades de los objetos son las características que puede tener un objeto, como el color. Si el objeto es Persona, los atributos podrían ser: cedula, nombre, apellido, sexo, etc...

Los atributos describen el estado de un objeto. Pueden ser de cualquier tipo de dato.

```
class Persona:  
    """Clase que representa una Persona"""\n    cedula = "V-13458796"  
    nombre = "Leonardo"  
    apellido = "Caballero"  
    sexo = "M"
```



Tipos de atributos

ATRIBUTOS DINÁMICOS

Dado que Python es muy flexible los atributos pueden manejarse de distintas formas, por ejemplo se pueden crear dinámicamente (al vuelo) en los objetos.

```
class Galleta:  
    pass  
  
galleta = Galleta()  
galleta.sabor = "salado"  
galleta.color = "marrón"  
  
print(f"El sabor de esta galleta es {galleta.sabor} " f"y el color {galleta.color}")
```

ATRIBUTOS DE CLASE

Aunque la flexibilidad de los atributos dinámicos puede llegar a ser muy útil, tener que definir los atributos de esa forma es tedioso. Es más práctico definir unos atributos básicos en la clase. De esa manera todas las galletas podrían tener unos atributos por defecto:

```
class Galleta:  
    chocolate = False  
  
galleta = Galleta()  
if galleta.chocolate:  
    print("La galleta tiene chocolate")  
else:    print("La galleta no tiene chocolate")
```



ATRIBUTOS DE INSTANCIA

Los atributos de instancia son creados usando la sintaxis `instance.attr` o `self.attr` (`self` es realmente una convención, simplemente es un identificador que hace referencia a la propia instancia). Como se ha dicho son locales para esa instancia y por tanto solo accesibles desde una de ellas.

```
class Dog:  
    def __init__(self, name):  
        self.name = name #Variable de instancia único para cada objeto
```





MÉTODOS



Los métodos describen el comportamiento de los objetos de una clase. Estos representan las operaciones que se pueden realizar con los objetos de la clase, la ejecución de un método puede conducir a cambiar el estado del objeto.

Se definen de la misma forma que las funciones normales pero deben declararse dentro de la clase y su primer argumento siempre referencia a la instancia que la llama, de esta forma se afirma que los métodos son funciones, adjuntadas a objetos.

Si el objeto es Persona, los métodos pueden ser: hablar, caminar, comer, dormir, etc.

```
class Persona:  
    """Clase que representa una Persona"""\n    cedula = "V-13458796"  
    nombre = "Leonardo"  
    apellido = "Caballero"  
    sexo = "M"  
  
    def hablar(self, mensaje):  
        """Mostrar mensaje de saludo de Persona"""\n        return mensaje
```

La única diferencia sintáctica entre la definición de un método y la definición de una función es que el primer parámetro del método por convención debe ser el nombre `self`.

SELF

Self representa la instancia de la clase. Mediante el uso de la palabra clave "self" podemos acceder a los atributos y métodos de la clase en python. Vincula los atributos con los argumentos dados.

La razón por la que necesitas usarte a ti mismo. se debe a que Python no utiliza la sintaxis @ para hacer referencia a los atributos de instancia. Python decidió hacer métodos de una manera que hace que la instancia a la que pertenece el método se pase automáticamente, pero no se reciba automáticamente: el primer parámetro de los métodos es la instancia a la que se llama al método.

```
class check:  
    def __init__(self):  
        print("Address of self = ", id(self))  
  
obj = check()  
print("Address of class object = ", id(obj))
```

Método de Init

El método `__init__()` es un método especial, el cual se ejecuta al momento de instanciar un objeto. El comportamiento de `__init__()` es muy similar a los «constructores» en otros lenguajes. Los argumentos que se utilizan en la definición de `__init__()` corresponden a los parámetros que se deben ingresar al instanciar un objeto.

Las ventajas de implementar el método `__init__` en lugar del método inicializar son:



- El método `__init__` es el primer método que se ejecuta cuando se crea un objeto.
- El método `__init__` se llama automáticamente. Es decir es imposible de olvidarse de llamarlo ya que se llamará automáticamente.
- Quien utiliza POO en Python (Programación Orientada a Objetos) conoce el objetivo de este método.

Otras características del método `__init__` son:

- Se ejecuta inmediatamente luego de crear un objeto.
- El método `__init__` no puede retornar dato.
- El método `__init__` puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- El método `__init__` es un método opcional, de todos modos es muy común declararlo.

Ejercicio #1:

Confeccionar una clase que represente un empleado. Definir como atributos su nombre y su sueldo. En el método `__init__` cargar los atributos por teclado y luego en otro método imprimir sus datos y por último uno que imprima un mensaje si debe pagar impuestos (si el sueldo supera a 3000)

```
class Empleado:  
  
    def __init__(self):  
        self.nombre=input("Ingrese el nombre del empleado:")  
        self.sueldo=float(input("Ingrese el sueldo:"))  
  
    def imprimir(self):  
        print("Nombre:",self.nombre)  
        print("Sueldo:",self.sueldo)  
  
    def paga_impuestos(self):  
        if self.sueldo>3000:  
            print("Debe pagar impuestos")  
        else:  
            print("No paga impuestos")  
  
    # bloque principal  
  
empleado1=Empleado()  
empleado1.imprimir()  
empleado1.paga_impuestos()
```



Métodos especiales

Las clases en Python cuentan con múltiples métodos especiales, los cuales se encuentran entre dobles guiones bajos `__<metodo>__()`.

Los métodos especiales más utilizados son `__init__()`, `__str__()` y `__del__()`.

MÉTODO `__STR__`

El método `__str__()` es un método especial, el cual se ejecuta al momento en el cual un objeto se manda a mostrar, es decir es una cadena representativa de la clase, la cual puede incluir formatos personalizados de presentación del mismo.

Vamos a utilizar como ejemplo la clase Persona..

```
def __str__(self):
    """Devuelve una cadena representativa de Persona"""
    return "%s: %s, %s %s, %s." % (self.__doc__[25:34], str(self.cedula),
                                    self.nombre, self.apellido, self.getGenero(self.sexo))
```



OBJETOS DENTRO DE OBJETOS



Hasta ahora no lo hemos comentado, pero al ser las clases un nuevo tipo de dato resulta más que obvio que se pueden poner en colecciones e incluso utilizarlos dentro de otras clases.

A continuación veremos algo interesante, podemos trabajar con objetos dentro de las clases, primero vamos a crear la clase película y posteriormente mediante la clase catalogar vamos a guardar objetos dentro de una lista.



Código:

```
class Pelicula:

    # Constructor de clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print('Se ha creado la película:', self.titulo)

    def __str__(self):
        return '{} ({})'.format(self.titulo, self.lanzamiento)

class Catalogo:

    peliculas = [] # Esta lista contendrá objetos de la clase
Pelicula

    def __init__(self, peliculas=[]):
        self.peliculas = peliculas

    def agregar(self, p): # p será un objeto Pelicula
        self.peliculas.append(p)

    def mostrar(self):
        for p in self.peliculas:
            print(p) # Print toma por defecto str(p)
```

Ahora vamos a crear un objeto de la clase Pelicula, y posteriormente creamos otro objeto de la clase Catalogo donde debemos enviarle un objeto de la clase pelicula, para poder catalogar.

```
p = Pelicula("El Padrino", 175, 1972)
c = Catalogo([p]) # Añado una lista con una película desde el principio
c.mostrar()
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974)) # Añadimos otra
c.mostrar()
```

Salida:

```
Se ha creado la película: El Padrino
El Padrino (1972)
Se ha creado la película: El Padrino: Parte 2
El Padrino (1972)
El Padrino: Parte 2 (1974)
```



ERRORES



En nuestra vida como programadores es muy común cometer errores en nuestro código, lo importante es no desesperarnos y saber como resolverlos. Para eso será importante tener un buen manejo de errores en python para poder encontrar la solución.

Entre los errores comunes en python de python tenemos:

- **ERRORES SINTÁCTICOS:** Se producen cuando existe un problema de sintaxis en nuestros comandos, como digitar mal un comando, y Python nos alerta de esto con el mensaje de error SyntaxError: invalid syntax.
- **ERRORES EN TIEMPO DE EJECUCIÓN:** Ocurren mientras el programa se está ejecutando y algo inesperadamente ocurre mal. Generalmente Python nos informa ese tipo de error como por ejemplo una recursión infinita causando el error maximum recursion depth exceeded.
- **ERRORES SEMÁNTICOS:** Se dan cuando el programa compila y se ejecuta normalmente, pero no hace lo que se pretendía que hiciera y Python en este caso no nos va informar donde está el error. Y para eso debemos valernos de la depuración.



¿Cómo manejar los errores en Python?

Python al igual que muchos otros lenguajes de programación nos permiten encontrar y solucionar los errores de código dependiendo de su naturaleza. Veremos que para manejar los errores en python podemos hacerlo mediante la correcta lectura de los TraceBack, usando print statement, empleando el debugger del IDE que estemos usando, y realizando un correcto manejo de excepciones con los comandos Try-Except-Raise-Else-Finally.

ERRORES SINTÁCTICOS

Cuando empleamos un lenguaje de programación como por ejemplo Python, será común caer en este tipo de error, pues los errores de sintaxis, o sintácticos, ocurren cuando el programador escribe los comandos dentro del código que no son acordes con las reglas de escritura estipuladas para el empleo del lenguaje de programación.

Los errores de sintaxis siempre son detectados por el intérprete de Python antes de ejecutar el programa, por lo que son errores relativamente simples de resolver.

Estos errores de tipografía nos pueden indicar si el nombre de algun comando es incorrecto, nombres de variables incorrectas, si falta algún paréntesis, palabras claves mal escritas, etc.

A continuación veremos un listado con todas las verificaciones que podremos realizar para la corrección de errores sintácticos o de Sintaxis (Syntax Error en Python):

- Se corrigen leyendo el mensaje de error arrojado por Python, a pesar que muchas veces no nos indique exactamente donde está el error.
- Muchas veces el error ocurre NO en la línea que indica Python, sino en la línea inmediatamente anterior.
- Verificar que no se crean variables con palabras reservadas de Python
- Verificar los dos puntos al final de instrucciones como for, while, def, if y clases.
- Verificar la identación y que sea consistente.
- Verificar que los strings estén encerrados entre comillas. O cadenas multilíneas pues puede generar el error invalid token.
- Verificar cerrar todos los paréntesis, llaves y corchetes
- Verificar si no se confundió el = por ==



Ejemplo de Syntax Error en Python

El siguiente Script tiene errores de syntax escritos de forma proposital. La idea aquí es que puedas leer los errores arrojados por el interprete de python y consigas darle un manejo adecuado a estos errores.

```
"""
Errores de sintaxis
"""

def main():
    divisor = 5
    for i in range(100):
        if i % divisor = 0
            print(f'{i} is divisible by {divisor}')
```

Si observamos tenemos los siguientes errores sintácticos en python que el propio interprete nos dice la ubicación de los errores:

- En la linea 10, en el condicional hace falta un =, dado que en la lógica debemos usar el comando de igualdad y no el comando de asignación. if i % divisor == 0
- En la linea 10, dado que estamos en un condicional, es necesario finalizar con dos puntos (:), if i % divisor == 0:
- En la linea 11, el syntax error de python nos indica que falta un '}', el cual está faltando al final del mensaje del print con formato. print(f'{i} is divisible by {divisor}')

Interrupción por Timer



ERRORES EN TIEMPO DE EJECUCIÓN

En los errores de tiempo de ejecución (excepciones) Python compila y ejecuta el programa pues no hay errores de sintaxis.

Si el programa no hace nada, asegúrese de estar llamando las funciones en su principal y por supuesto de estar imprimiendo lo que desea ver con su código.

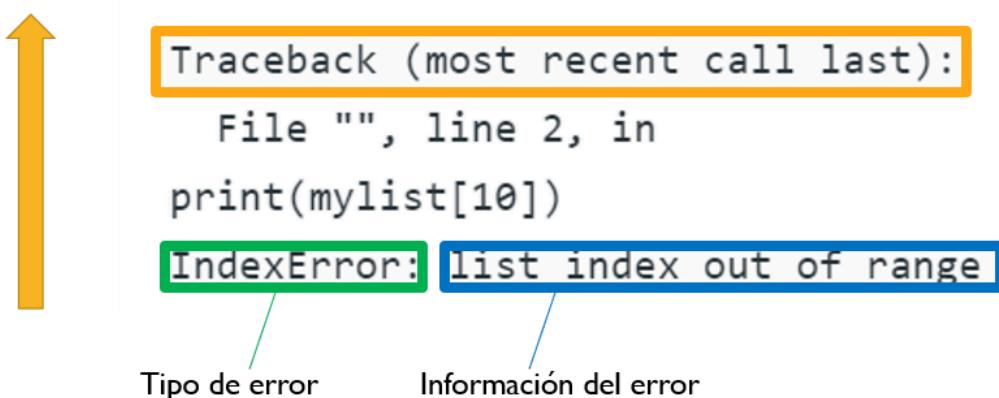
Si el programa se detiene y parece no hacer nada puede que este atrapado dentro de un ciclo infinito, para verificar este problema, añada un print dentro del ciclo (esto se conoce como print «statement») y otra fuera del ciclo para ver si sale.

Si existe recursión infinita puede aparecer el error RuntimeError:máximo recursión Depth exceeded.

Si el programa arroja una excepción, en ese caso Python imprime un mensaje que incluye el nombre de la excepción y la línea donde ocurrió: Name error; Type Error; Key Error; Attribute Error; Index Error.

Cuando Python nos arroja un error, este va a mostrar un mensaje de error conocido como TraceBack, en este mensaje estarán detallados las informaciones del error o la excepción que han impedido que Python execute o llegue al final del código programado.

La visión general de un TRACEBACK puede ser visto a continuación:



La forma correcta de leer estos errores en Python (Traceback) es como lo indica la flecha amarilla, siempre de abajo hacia arriba.



Inicialmente, observe que en el recuadro verde, siempre va a aparecer el error o la excepción encontrada por Python. Este tipo de error puede ser por ejemplo el index error (no encuentra un indice en una lista, tupla, etc) pero también pueden existir otro tipo de errores tales como:

- NameError
- IndexError
- KeyError
- TypeError
- ValueError
- ImportError /ModuleNotFoundError

Seguidamente, en el recuadro azul, podremos ver un resumen sobre el error detectado.

Si continuamos hacia arriba en el traceback, podremos notar la linea de código donde fue detectado el error y la linea de comando dentro del programa.

Finalmente en la primera linea aparece el Traceback donde aparece most recent call last, que indica que el error fue encontrado en el último módulo ejecutado en el programa de Python.

El error de ese traceback de ejemplo fue generado con el siguiente código:

```
# Python program to demonstrate

# traceback

mylist = [1, 2, 3]

print(mylist[10])
```

Ejemplo de Excepción Recursion Error

Python, establece un límite máximo de 1000 donde permite que una función se llame a ella misma (recursión). Es decir que solo puede hacer el llamado 1000 veces. Si excedemos este límite, va a aparecer una excepción mostrado en un traceback mostrando el tipo de error: Recursion Error.

El siguiente código provoca ese error:



```
#Funcion por recursion
```

```
def factorial(n):  
  
    if n == 0 or n == 1:  
  
        return 1  
  
    return n * factorial(n-1)  
  
# n = 3  
  
# 3! = 3 * 2!  
  
# 2! = 2 * 1!  
  
# 1! = 1 Base  
  
f = factorial(2000)  
  
print(f)
```

Para corregir el error, basta con configurarle a Python el límite máximo de recursión para otro valor, para eso será necesario llamar el módulo sys y posteriormente configurar el límite con el método setrecursionlimit:

```
import sys  
  
#Funcion por recursion  
  
def factorial(n):  
  
    if n == 0 or n == 1:  
  
        return 1  
  
    return n * factorial(n-1)  
  
# n = 3
```

```
# 3! = 3 * 2!
#
# 2! = 2 * 1!
#
# 1! = 1 Base
sys.setrecursionlimit(4000)
f = factorial(2000)
print(f)
```

ERRORES SEMÁNTICOS

Los errores semánticos en Python, también llamados errores lógicos, son los más difíciles de depurar dado que ni el compilador ni el sistema nos muestran un fallo o error. Si hay un error semántico en el script de python, este se ejecutará correctamente en el sentido de que el compilador no va a generar ningún mensaje de error. Sin embargo, el programa o script no hará lo correcto. Hará otra cosa. En otras palabras, el script hará lo que le usted le dijo que hiciera, no lo que usted quería que hiciera.

Existen dos formas directas de realizar el manejo de errores con python relacionados a los errores semánticos:

- Usar los depuradores que le ofrece su IDE.
- Usar print statement

Ejemplo de error semántico

Para poder exemplificar el manejo de errores semánticos en Python, vamos a crear un script que sea capaz de encontrar las raíces de una ecuación cuadrática, para eso solucionaremos inicialmente la ecuación para posteriormente proceder a desarrollar el script.

La idea es solucionar una ecuación en la forma:

$$a\backslash x^2+b\backslash x + c = 0 \quad a \neq 0$$



Suponga que tenemos la siguiente ecuación:

$$x^2 - 5x + 6 = 0$$

Solucionando, tenemos que:

$$\begin{aligned}x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\x &= \frac{-(-5) \pm \sqrt{(-5)^2 - 4(1)(6)}}{2(1)} \\x &= \frac{(2)(1) - (-5) \pm \sqrt{(-5)^2 - 4(1)(6)}}{2(1)} \\x &= \frac{5 \pm \sqrt{25 - 24}}{2} \\x &= \frac{5 \pm \sqrt{1}}{2} \\x &= 25 \pm 1\end{aligned}$$

Por lo tanto, las raíces de la ecuación cuadrática son:

$$x_1 = \frac{5+1}{2} = 3$$

$$x_2 = \frac{5-1}{2} = 2$$

El siguiente Script en Python (el cual tiene un error semántico de forma proposital) busca encontrar las raíces de la ecuación cuadrática.

Crear una función que sea capaz de encontrar las raíces de una ecuación cuadrática de la forma:

by: Sergio Andres Castaño Giraldo

```
def quadratic_equation(a, b, c):
```

```
    # calculate the discriminant
```

```
    d = (b**2) - (4*a*c)
```

```
# find two solutions

x1 = (-b-(d)**0.5)/(2*a)

x2 = (-b+(d)**0.5)/(2*a)

return x1, x2

if __name__ == '__main__':

    print('Python Program to Solve Quadratic Equation')

    a = float(input('Coefficients a: '))

    b = float(input('Coefficients b: '))

    c = float(input('Coefficients c: '))

    x1, x2 = quadratic_equation(a, b, c)

    print(f'The solution are {x1} and {x2}')
```

Si vemos, el error del script se encuentra dentro de la función quadratic_equation al momento de calcular el discriminante.

Donde tenemos:

$$d = (b^2) - (4*a*c)$$

debemos sustituirlo por:

$$d = (b^{**2}) - (4*a*c)$$

Dado que b debe estar elevado al cuadrado y NO multiplicado por 2.



EXCEPCIONES

Excepciones

Python utiliza un objeto especial llamado excepción para controlar cualquier error que pueda ocurrir durante la ejecución de un programa.

Cuando ocurre un error durante la ejecución de un programa, Python crea una excepción. Si no se controla esta excepción la ejecución del programa se detiene y se muestra el error (traceback).

```
>>> print(1 / 0) # Error al intentar dividir por 0.  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

Tipos de excepciones

Las principales excepciones definidas en Python son:



- `TypeError` : Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.
- `ZeroDivisionError` : Ocurre cuando se intenta dividir por cero.
- `OverflowError` : Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.
- `IndexError` : Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.
- `KeyError` : Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.
- `FileNotFoundException` : Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.
- `ImportError` : Ocurre cuando falla la importación de un módulo.

Excepciones

Consultar la documentación de Python para ver la lista de excepciones predefinidas.

[PYTHON DOCS](https://docs.python.org/3/library/exceptions.html)



MANEJO DE EXCEPCIONES



Cuando se producen estas excepciones, el intérprete de Python detiene el proceso actual y lo pasa al proceso de llamada hasta que se controla. Si no se maneja, el programa se bloqueará.

Por ejemplo, consideremos un programa donde tenemos una función que llama función, que a su vez llama a función. Si se produce una excepción en la función pero no se controla en , la excepción pasa a ABCCCBA.

Si nunca se maneja, se muestra un mensaje de error y nuestro programa se detiene repentinamente.

En Python, las excepciones se pueden controlar mediante una instrucción try.

La operación crítica que puede generar una excepción try se coloca dentro de la cláusula. El código que controla las excepciones está escrito en la cláusula except.

Cuando se produce un error, o una excepción como lo llamamos, Python normalmente se detendrá y generará un mensaje de error.



Ejemplo:

El bloque generará una excepción, porque no está definido: try x

```
try:  
    print(x)  
except:  
    print("An exception occurred")
```

Dado que el bloque try genera un error, se ejecutará el bloque except.

Sin el bloque try, el programa se bloqueará y generará un error:

ELSE

Puede utilizar la palabra clave else para definir un bloque de código que se ejecutará si no se han producido errores:

```
try:  
    print("Hello")  
except:  
    print("Something went wrong")  
else:  
    print("Nothing went wrong")
```

FINALLY

El bloque finally, si se especifica, se ejecutará independientemente de si el bloque de prueba genera un error o no.

```
try:  
    print(x)  
except:  
    print("Something went wrong")  
finally:  
    print("The 'try except' is finished")
```

Ejemplo #1

```
try:  
    n = float(input("Introduce un número: "))  
    m = 4  
    print("{} / {} = {}".format(n,m,n/m))  
except:  
    print("Ha ocurrido un error, introduce bien el número")
```

SALIDA:



```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
```

Ejemplo #2

```
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{} / {} = {}".format(n,m,n/m))
        break # Importante romper la iteración si todo ha salido bien
    except:
        print("Ha ocurrido un error, introduce bien el número")
```

Salida:

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: 10
10.0/4 = 2.5
```



MULTIPLES EXCEPCIONES

Múltiples excepciones

En una misma pieza de código pueden ocurrir muchos errores distintos y quizás nos interese actuar de forma diferente en cada caso.

Para esas situaciones algo que podemos hacer es asignar una excepción a una variable. De esta forma es posible analizar el tipo de error que sucede gracias a su identificador:

```
try:  
    n = input("Introduce un número: ") # no transformamos a número  
    5/n  
except Exception as e: # guardamos la excepción como una variable e  
    print("Ha ocurrido un error =>", type(e).__name__)
```

SALIDA:

```
try:  
    n = input("Introduce un número: ") # no transformamos a número  
    5/n  
except Exception as e: # guardamos la excepción como una variable e  
    print("Ha ocurrido un error =>", type(e).__name__)
```

Cada error tiene un identificador único que curiosamente se corresponde con su tipo de dato. Aprovechándonos de eso podemos mostrar la clase del error utilizando la sintaxis:

```
print( type(e) )  
  
Salida:  
  
<class 'TypeError'>
```

Como vemos siempre nos indica eso de "class" delante. Lo importante ahora es que podemos mostrar solo el nombre del tipo de dato (la clase) consultando su propiedades especial name.

```
try:  
    n = float(input("Introduce un número divisor: "))  
    5/n  
except TypeError:  
    print("No se puede dividir el número entre una cadena")  
except ValueError:  
    print("Debes introducir una cadena que sea un número")  
except ZeroDivisionError:  
    print("No se puede dividir por cero, prueba otro número")  
except Exception as e:  
    print("Ha ocurrido un error no previsto", type(e).__name__)
```

SALIDA:

```
Introduce un número divisor: 0  
No se puede dividir por cero, prueba otro número
```

Generando Excepciones

También podemos ser nosotros los que levantemos o lancemos una excepción. Volviendo a los ejemplos usados en el apartado anterior, podemos ser nosotros los que levantemos ZeroDivisionError o NameError usando raise. La sintaxis es muy fácil.

```
raise ZeroDivisionError("Información de la excepción")
```

O podemos lanzar otra de tipo NameError.

```
raise NameError("Información de la excepción")
```



Se puede ver como el string que hemos pasado se imprime a continuación de la excepción. Se puede llamar también sin ningún parámetro como se hace a continuación.

```
raise ZeroDivisionError
```

Ejemplo #1

```
def mi_funcion(algo=None):
    try:
        if algo is None:
            raise ValueError("Error! No se permite un valor nulo")
    except ValueError:
        print("Error! No se permite un valor nulo (desde la
excepción)")

mi_funcion()
```

SALIDA:

```
Error! No se permite un valor nulo (desde la excepción)
```



MULTIPLES EXCEPCIONES

Múltiples excepciones

En una misma pieza de código pueden ocurrir muchos errores distintos y quizás nos interese actuar de forma diferente en cada caso.

Para esas situaciones algo que podemos hacer es asignar una excepción a una variable. De esta forma es posible analizar el tipo de error que sucede gracias a su identificador:

```
try:  
    n = input("Introduce un número: ") # no transformamos a número  
    5/n  
except Exception as e: # guardamos la excepción como una variable e  
    print("Ha ocurrido un error =>", type(e).__name__)
```

SALIDA:

```
try:  
    n = input("Introduce un número: ") # no transformamos a número  
    5/n  
except Exception as e: # guardamos la excepción como una variable e  
    print("Ha ocurrido un error =>", type(e).__name__)
```

Cada error tiene un identificador único que curiosamente se corresponde con su tipo de dato. Aprovechándonos de eso podemos mostrar la clase del error utilizando la sintaxis:

```
print( type(e) )  
  
Salida:  
  
<class 'TypeError'>
```

Como vemos siempre nos indica eso de "class" delante. Lo importante ahora es que podemos mostrar solo el nombre del tipo de dato (la clase) consultando su propiedades especial name.

```
try:  
    n = float(input("Introduce un número divisor: "))  
    5/n  
except TypeError:  
    print("No se puede dividir el número entre una cadena")  
except ValueError:  
    print("Debes introducir una cadena que sea un número")  
except ZeroDivisionError:  
    print("No se puede dividir por cero, prueba otro número")  
except Exception as e:  
    print("Ha ocurrido un error no previsto", type(e).__name__)
```

SALIDA:

```
Introduce un número divisor: 0  
No se puede dividir por cero, prueba otro número
```

Generando Excepciones

También podemos ser nosotros los que levantemos o lancemos una excepción. Volviendo a los ejemplos usados en el apartado anterior, podemos ser nosotros los que levantemos ZeroDivisionError o NameError usando raise. La sintaxis es muy fácil.

```
raise ZeroDivisionError("Información de la excepción")
```

O podemos lanzar otra de tipo NameError.

```
raise NameError("Información de la excepción")
```



Se puede ver como el string que hemos pasado se imprime a continuación de la excepción. Se puede llamar también sin ningún parámetro como se hace a continuación.

```
raise ZeroDivisionError
```

Ejemplo #1

```
def mi_funcion(algo=None):
    try:
        if algo is None:
            raise ValueError("Error! No se permite un valor nulo")
    except ValueError:
        print("Error! No se permite un valor nulo (desde la
excepción)")

mi_funcion()
```

SALIDA:

```
Error! No se permite un valor nulo (desde la excepción)
```



MANEJO DE FICHEROS



Al igual que en otros lenguajes de programación, en Python es posible abrir ficheros y leer su contenido. Los ficheros o archivos pueden ser de lo más variado, desde un simple texto a contenido binario. Para simplificar nos centraremos en leer un fichero de texto. Si quieras aprender como escribir en un fichero te lo explicamos en este otro post.

Imagínate entonces que tienes un fichero de texto con unos datos, como podría ser un .txt o un .csv, y quieres leer su contenido para realizar algún tipo de procesado sobre el mismo. Nuestro fichero podría ser el siguiente.

```
# contenido del fichero ejemplo.txt
Contenido de la primera linea
Contenido de la segunda linea
Contenido de la tercera linea
Contenido de la cuarta linea
```



Podemos abrir el fichero con la función open() pasando como argumento el nombre del fichero que queremos abrir.

```
fichero = open('ejemplo.txt')
```

Leer ficheros

Método read()

Con open() tendremos ya en fichero el contenido del documento listo para usar, y podemos imprimir su contenido con read(). El siguiente código imprime todo el fichero.



```
print(fichero.read())
#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
#Contenido de la cuarta línea
```

Método readline()

Es posible también leer un número de líneas determinado y no todo el fichero de golpe. Para ello hacemos uso de la función readline(). Cada vez que se llama a la función, se lee una línea.



```
● ● ●  
fichero = open('ejemplo.txt')  
print(fichero.readline())  
print(fichero.readline())  
# Contenido de la primera línea  
# Contenido de la segunda línea
```

Es muy importante saber que una vez hayas leído todas las líneas del archivo, la función ya no devolverá nada, porque se habrá llegado al final. Si quieres que readline() funcione otra vez, podrías por ejemplo volver a leer el fichero con open().

Otra forma de usar readline() es pasando como argumento un número. Este número leerá un determinado número de caracteres. El siguiente código lee todo el fichero carácter por carácter.

```
● ● ●  
fichero = open('ejemplo.txt')  
caracter = fichero.readline(1)  
while caracter != "":  
    #print(caracter)  
    caracter = fichero.readline(1)
```



Método readlines()

Existe otro método llamado `readlines()`, que devuelve una lista donde cada elemento es una línea del fichero.

```
● ● ●

fichero = open('ejemplo.txt')
lineas = fichero.readlines()
print(lineas)
#['Contenido de la primera linea\n', 'Contenido de la segunda linea\n',
# 'Contenido de la tercera linea\n', 'Contenido de la cuarta linea']
```

De manera muy sencilla podemos iterar las líneas e imprimirlas por pantalla.

```
● ● ●

fichero = open('ejemplo.txt')
lineas = fichero.readlines()
for linea in lineas:
    print(linea)
#Contenido de la primera linea
#Contenido de la segunda linea
#Contenido de la tercera linea
#Contenido de la cuarta linea
```



Argumentos open()

Hasta ahora hemos visto la función `open()` con tan sólo un argumento de entrada, el nombre del fichero. Lo cierto es que existe un segundo argumento que es importante especificar. Se trata del modo de apertura del fichero.

Argumentos `open()`

Argumentos de `open`, documentación...

DOCS PYTHON

- ‘r’: Por defecto, para leer el fichero.
- ‘w’: Para escribir en el fichero.
- ‘x’: Para la creación, fallando si ya existe.
- ‘a’: Para añadir contenido a un fichero existente.
- ‘b’: Para abrir en modo binario.

Por lo tanto lo estrictamente correcto si queremos leer el fichero sería hacer lo siguiente. Aunque el modo `r` sea por defecto, es una buena práctica indicarlo para darle a entender a otras personas que lean nuestro código que no queremos modificarlo, tan solo leerlo.



```
fichero = open('ejemplo.txt', 'r')
```

Cerrando fichero

Otra cosa que debemos hacer cuando trabajamos con ficheros en Python, es cerrarlos una vez que ya hemos acabado con ellos. Aunque es verdad que el fichero normalmente acabará siendo cerrado automáticamente, es importante especificarlo para evitar tener comportamientos inesperados.



Por lo tanto si queremos cerrar un fichero sólo tenemos que usar la función `close()` sobre el mismo. Por lo tanto tenemos tres pasos:

- Abrir el fichero que queramos. En modo texto usaremos ‘r’.
- Usar el fichero para recopilar o procesar los datos que necesitábamos.
- Cuando hayamos acabado, cerramos el fichero.



```
fichero = open('ejemplo.txt', 'r')
# Usar la variable fichero
# Cerrar el fichero
fichero.close()
```





ESCRIBIENDO FICHEROS



A continuación, te explicamos como escribir datos en un fichero usando Python. Imagínate que tienes unos datos que te gustaría guardar en un fichero para su posterior análisis. Te explicamos como guardarlos en un fichero, por ejemplo, .txt.

Lo primero que debemos de hacer es crear un objeto para el fichero, con el nombre que queramos. Al igual que vimos en el post de leer ficheros, además del nombre se puede pasar un segundo parámetro que indica el modo en el que se tratará el fichero. Los más relevantes en este caso son los siguientes.

- ‘w’: Borra el fichero si ya existiese y crea uno nuevo con el nombre indicado.
- ‘a’: Añadirá el contenido al final del fichero si ya existiese (append en Inglés)
- ‘x’: Si ya existe el fichero se devuelve un error.



Por lo tanto con la siguiente línea estamos creando un fichero con el nombre datos_guardados.txt.

```
● ● ●  
# Abre un nuevo fichero  
fichero = open("datos_guardados.txt", 'w')
```

Si por lo contrario queremos añadir el contenido al ya existente en un fichero de antes, podemos hacerlo en el modo append como hemos indicado.

```
● ● ●  
# Abre un nuevo y añade el contenido al  
fichero = open("datos_guardados.txt", 'a')
```

Método write()

Ya hemos visto como crear el fichero. Veamos ahora como podemos añadir contenido. Empecemos escribiendo un texto.

```
● ● ●  
fichero = open("datos_guardados.txt", 'w')  
fichero.write("Contenido a escribir")  
fichero.close()
```

Por lo tanto, si ahora abrimos el fichero datos_guardados.txt, veremos como efectivamente contiene una línea con Contenido a escribir.



Es muy importante el uso de close() ya que si dejamos el fichero abierto, podríamos llegar a tener un comportamiento inesperado que queremos evitar. Por lo tanto, siempre que se abre un fichero es necesario cerrarlo cuando hayamos acabado.

Compliquemos un poco más las cosas. Ahora vamos a guardar una lista de elementos en el fichero, donde cada elemento de la lista se almacenará en una línea distinta.

```
● ● ●

# Abrimos el fichero
fichero = open("datos_guardados.txt", 'w')

# Tenemos unos datos que queremos guardar
lista = ["Manzana", "Pera", "Plátano"]

# Guardamos la lista en el fichero
for linea in lista:
    fichero.write(linea + "\n")

# Cerramos el fichero
fichero.close()
```

Método writelines()

También podemos usar el método writelines() y pasarle una lista. Dicho método se encargará de guardar todos los elementos de la lista en el fichero.



```
fichero = open("datos_guardados.txt", 'w')
lista = ["Manzana", "Pera", "Plátano"]

fichero.writelines(lista)
fichero.close()
```

Tal vez te hayas dado cuenta de que en realidad lo que se guarda es ManzanaPeraPlátano, todo junto. Si queremos que cada elemento se almacene en una línea distinta, deberíamos añadir el salto de línea en cada elemento de la lista como se muestra a continuación.



```
fichero = open("datos_guardados.txt", 'w')
lista = ["Manzana\n", "Pera\n", "Plátano\n"]

fichero.writelines(lista)
fichero.close()
```

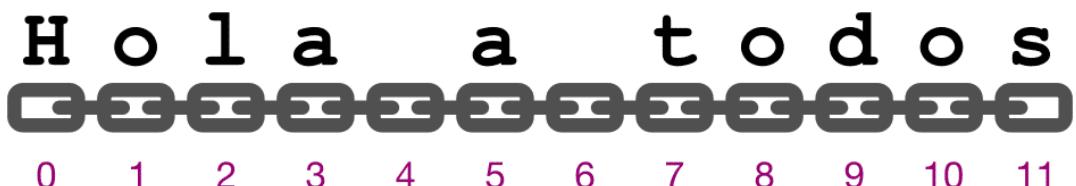


CADERAS DE CARACTERES

Cadenas de caracteres

En computación, una cadena de caracteres o cadena de texto o simplemente cadena (string en inglés) es una secuencia ordenada de símbolos, con una longitud arbitraria (con tantos símbolos como queramos).

Se llama cadena, haciendo la analogía con una cadena física creada por elementos llamados eslabones, donde cada eslabón dentro de la cadena se encuentra acomodado en una secuencia consecutiva, uno detrás de otro. Como las cadenas son una secuencia ordenada de valores unos seguidos de otros, podemos hacer referencia a la posición de cada símbolo dentro de la cadena por medio de un número o índice, hay que tener en cuenta que en computación los índices generalmente se consideran desde la posición 0 y no desde el 1.



Ejemplo de la cadena 'Hola a todos', debajo se pueden observar los indices de cada elemento de la cadena, comenzando desde 0.

En los lenguajes de programación para indicar qué algún valor es una cadena, el valor se escribe entre comillas dobles ("") o entre comillas sencillas ('').

Esta conversión de carácter a número se llama codificación y el proceso inverso es decodificación. ASCII y Unicode son algunas de las codificaciones populares utilizadas.

En Python, una cadena es una secuencia de caracteres Unicode. Unicode se introdujo para incluir todos los caracteres en todos los idiomas y brindar uniformidad en la codificación. Puede obtener información sobre Unicode en Python Unicode .

Cadenas en Python

Las cadenas en Python o strings son un tipo inmutable que permite almacenar secuencias de caracteres. Para crear una, es necesario incluir el texto entre comillas dobles ". Puedes obtener más ayuda con el comando help(str).

```
# definiendo string en Python
# todos los siguientes son equivalentes
my_string = 'Hola'
print(my_string)

my_string = "Hola"
print(my_string)

my_string = '''Hola'''
print(my_string)

# la cadena de comillas triples puede extender varias lineas
my_string = """Hola, bienvenido a
                    el mundo de Python"""
print(my_string)
```

Salida: Hola

Hola

Hola



Hola, bienvenido a el mundo de Python

Las cadenas no están limitadas en tamaño, por lo que el único límite es la memoria de tu ordenador. Una cadena puede estar también vacía.

```
s = "
```

Una situación que muchas veces se puede dar, es cuando queremos introducir una comilla, bien sea simple ' o doble " dentro de una cadena. Si lo hacemos de la siguiente forma tendríamos un error, ya que Python no sabe muy bien donde empieza y termina.

```
#s = "Esto es una comilla doble " de ejemplo" # Error!
```

Para resolver este problema debemos recurrir a las secuencias de escape. En Python hay varias, pero las analizaremos con más detalle en otro capítulo. Por ahora, la más importante es \", que nos permite incrustar comillas dentro de una cadena.



```
s = "Esto es una comilla doble \" de ejemplo"  
print(s) #Esto es una comilla doble " de ejemplo
```

También podemos incluir un salto de línea dentro de una cadena, lo que significa que lo que esté después del salto, se imprimirá en una nueva línea.

```
● ● ●  
s = "Primer linea\nSegunda linea"  
print(s)  
#Primer linea  
#Segunda linea
```



CONCATENACIÓN



Concatenación

Una de las operaciones más comunes en cualquier lenguaje de programación es la concatenación y/o combinación de cadenas. En este tutorial descubrirás distintos modos de concatenar en python dos o más strings.

Concatenar strings en Python usando el operador ‘+’

Como veremos a lo largo de este post, en Python existen varias formas de concatenar dos o más objetos de tipo string. La más sencilla es usar el operador +. Concatenar dos o más strings con el operador + da como resultado un nuevo string.



Ejemplo #1:

```
nombre = 'Juan José'  
apellidos = 'Lozano Gómez'  
nombre_completo = nombre + apellidos  
nombre_completo  
'Juan JoséLozano Gómez'
```

Como podemos observar, para concatenar dos cadenas con el operador + simplemente necesitamos dos objetos de tipo string. Estos objetos pueden estar almacenados en memoria (usando una variable como en el ejemplo), ser un literal e incluso ser el valor devuelto por una función.

Ejemplo #2:

```
saludo = 'Hola '  
hola = saludo + 'Intecssa'  
hola  
'Hola Intecssa'
```

Como he indicado en el párrafo anterior, para concatenar varios strings en Python necesitamos que todos los elementos sean de este tipo.

Ejemplo #3:

```
suma = 1 + 2  
suma  
3  
>>> res = 'El resultado de 1 + 2 es: ' + suma  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>  
TypeError: can only concatenate str (not "int") to str
```

Si tratamos de concatenar, por ejemplo, un string con un int, el intérprete lanzará un error.

Ejemplo #4:

```
suma = 1 + 2  
suma  
3  
>>> res = 'El resultado de 1 + 2 es: ' + str(suma)  
>>> res  
'El resultado de 1 + 2 es: 3'
```

Para poder concatenar los valores anteriores tienes que convertir suma a un objeto de tipo string. Para ello puedes usar el método str(), que devuelve una representación de tipo string del objeto que se le pasa como parámetro.



Ejemplo #5:

```
print('Suma: ' + str(1) + ' + 2 = ' + str(1 + 2))  
Suma: 1 + 2 = 3
```

También es posible concatenar más de dos strings a la vez.





ÍNDICES

Indexación de cadenas

A menudo, en los lenguajes de programación, se puede acceder directamente a los elementos individuales de un conjunto ordenado de datos mediante un índice numérico o un valor clave. Este proceso se conoce como indexación.

En Python, las cadenas son secuencias ordenadas de datos de caracteres y, por lo tanto, se pueden indexar de esta manera. Se puede acceder a los caracteres individuales de una cadena especificando el nombre de la cadena seguido de un número entre corchetes ([]).

La indexación de cadenas en Python se basa en cero: el primer carácter de la cadena tiene índice 0, el siguiente tiene índice 1, y así sucesivamente. El índice del último carácter será la longitud de la cadena menos uno.



Forward direction indexing

0	1	2	3	4	5	
String	P	y	t	h	o	n
	-6	-5	-4	-3	-2	-1

Backward direction indexing

La indexación permite que las referencias de direcciones negativas accedan a los caracteres desde la parte posterior de la Cadena, por ejemplo, -1 se refiere al último carácter, -2 se refiere al penúltimo carácter, y así sucesivamente.

Al acceder a un índice fuera del rango, se producirá un IndexError . Solo se permite pasar enteros como índice, flotante u otros tipos que causen un TypeError .

Se puede acceder a los caracteres individuales por índice de la siguiente manera:

```
>>> s = 'cadena'  
>>> s[0]  
'c'  
>>> s[1]  
'a'  
>>> s[3]  
'd'  

```



Aquí accedimos a los elementos índice por índice, hasta con números negativos.

Intentar indexar más allá del final de la cadena da como resultado un error:

```
>>> s[6]
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    s[6]
IndexError: string index out of range
```





SLICING

Python también permite una forma de sintaxis de indexación que extrae subcadenas de una cadena, conocida como segmentación de cadenas. Si s es una cadena, una expresión de la forma s[m:n] devuelve la parte que s comienza con la posición m y hasta la posición, pero sin incluir la n:



```
>>> s = 'cadena'  
>>> s[2:5]  
'den'
```

Nuevamente, el segundo índice especifica el primer carácter que no está incluido en el resultado: el carácter 'a'(s[5]) en el ejemplo anterior. Eso puede parecer poco intuitivo, pero produce este resultado que tiene sentido: la expresión s[m:n] devolverá una subcadena que tiene una $n - m$ longitud de caracteres, en este caso, $5 - 2 = 3$.

Si omite el primer índice, el segmento comienza al principio de la cadena. Así, s[:m] y s[0:m] son equivalentes:



```
>>> s = 'cadena'  
  
>>> s[:4]  
'cade'  
>>> s[0:4]  
'cade'
```

De manera similar, si omite el segundo índice como en `s[n:]`, el segmento se extiende desde el primer índice hasta el final de la cadena. Esta es una alternativa agradable y concisa a la más engorrosa `s[n:len(s)]`:

```
>>> s = 'cadena'  
  
>>> s[2:]  
'dena'  
>>> s[2:len(s)]  
'dena'
```

Si el primer índice de un segmento es mayor o igual que el segundo índice, Python devuelve una cadena vacía. Esta es otra forma ofuscada de generar una cadena vacía, en caso de que estuviera buscando una:

```
>>> s[2:2]  
''  
>>> s[4:2]  
''
```



Indexación negativa

Los índices negativos también se pueden usar con el corte. -1 se refiere al último carácter, -2 al penúltimo, etc., al igual que con la indexación simple. El siguiente diagrama muestra cómo dividir la subcadena 'ade' de la cadena 'cadena' utilizando índices tanto positivos como negativos:

```
>>> s = 'cadena'  
  
>>> s[-5:-2]  
'ade'  
>>> s[1:4]  
'ade'  
>>> s[-5:-2] == s[1:4]  
True
```



FORMATO PARA STRINGS

Formato %

Una de las maneras habituales de formatear cadenas en Python es emplear el operador %.

He aquí la sintaxis básica:

```
"This is a string %s" % "string value goes here"
```

Puedes crear cadenas y usar %s adentro de la cadena que funciona como un marcador. Entonces puedes escribir % seguido del valor actual de la cadena que deseas usar.

A continuación un ejemplo básico usando el formato de cadenas con % .

```
print("Hi, my name is %s" % "Jessica")
```

```
Hi, my name is Jessica
```



Este método es ocasionalmente designado como la forma "tradicional". Sin embargo, Python 3 introdujo el `str.format()` junto con formato para literales de cadena.

Método Str.Format()

La sintaxis del método `str.format()` es la siguiente:

`"template string {}".format(arguments)`

Al interior de la cadena de plantilla podemos usar `{}` como marcador para los argumentos. Los argumentos son valores que serán mostrados en la cadena.

En este ejemplo, queremos imprimir lo siguiente "Hello, my name is Jessica. I am a musician turned programmer."

En la cadena, tendremos un total de tres `{}` marcadores para los tres valores: "Jessica", "musician", y "programmer". Estos son llamados también campos de formato.

`"Hello, my name is {}. I am a {} turned {}."`

Dentro de estos paréntesis con el `str.format()`, usaremos los valores de "Jessica", "musician", y "programmer".

`.format("Jessica", "musician", "programmer")`

He aquí el código completo:

```
print("Hello, my name is {}. I am a {} turned {}.".format("Jessica", "musician", "programmer"))
```

```
Hello, my name is Jessica. I am a musician turned programmer.
```

Argumentos posicionales

Puedes acceder al valor de estos argumentos usando un número como índice dentro de `{}`.

En este ejemplo, los argumentos de "trumpet" y "drums" dentro del `.format()`.

`.format("trumpet", "drums")`



Podemos acceder a estos valores dentro de la cadena efectuando una referencia a los números índices. {0} representa el primer argumento de "trumpet" , mientras que {1} es el segundo argumento de "drums".

```
"Steve plays {0} and {1}."
```

Siendo el predicado completo el siguiente:

```
print("Steve plays {0} and {1}.".format("trumpet", "drums"))
```

Steve plays trumpet and drums.

Podemos modificar este ejemplo y cambiar los números índices en la cadena. Es de notar que la oración ha cambiado y el lugar de los argumentos es diferente.

```
print("Steve plays {1} and {0}.".format("trumpet", "drums"))
```

Steve plays drums and trumpet.

Argumentos de palabras clave

Estos argumentos consisten de un par key value, es decir, de un dúo entre clave y valor. Podemos acceder al valor del argumento(value), usando la clave key al interior del {}.

En este ejemplo, tenemos dos claves llamadas organization y adjective. Usaremos estas claves adentro de la cadena.

```
"{organization} is {adjective}!"
```

Al interior de .format(), se encuentran los pares key value.

Finalmente, redactamos la sentencia de código completa.

```
print("{organization} is {adjective}!".format(organization="freeCodeCamp",  
adjective="awesome"))
```



freeCodeCamp is awesome!

¿Cómo mezclar palabras clave y argumentos posicionales?

En str.format() esto es posible.

En este ejemplo, crearemos una narrativa corta acerca de ir a Disneylandia.

Primero necesitamos crear unas cuantas variables, por nombre, número, adjetivo y tipo de paseo en Disneylandia.

```
name = "Sam"  
adjective = "amazing"  
number = 200  
disney_ride = "Space Mountain"
```

Cuando queremos crear nuestra propia cadena de texto usando palabras clave y argumentos posicionales. Añadimos \n para crear una nueva línea después de cada oración.

```
"I went to {0} with {name}.\nIt was {adjective}.\nWe waited for {hours} hours to ride {ride}."
```

Dentro del paréntesis de str.format(), asignaremos nuestras variables a las claves de name, adjective, hours y disney_ride. {0} tendrá el valor de "Disneyland".

```
.format("Disneyland", name=name, adjective=adjective, hours=number,  
ride=disney_ride)
```

A continuación el código completo y el resultado final:

```
name = "Sam"  
adjective = "amazing"  
number = 200  
disney_ride = "Space Mountain"  
  
print("I went to {0} with {name}.\nIt was {adjective}.\nWe waited for {hours} hours to ride  
{ride}."
```

```
.format("Disneyland", name=name, adjective=adjective, hours=number,  
ride=disney_ride))
```

I went to Disneyland with Sam.

It was amazing.

We waited for 200 hours to ride Space Mountain.

¿Qué son los formatos de literales de cadenas?

Los literales de cadenas (o f-strings) nos permiten introducir expresiones al interior de una cadena. Funcionan introduciendo un f o F para indicar al programa que debe usar f-string.

La sintaxis es la siguiente:

```
variable = "some value"  
f>this is a string {variable}
```

Veamos un ejemplo básico que imprime la oración Maria and Jessica have been friends since grade school.

```
name = "Jessica"  
print(f"Maria and {name} have been friends since grade school.")
```

Funciona también si usas la mayúscula F previo a la cadena.

```
name = "Jessica"  
print(F"Maria and {name} have been friends since grade school.")
```

Maria and Jessica have been friends since grade school.

Finalmente, también es posible emplear f-string para dar formato a un diccionario.

En este ejemplo, tenemos un diccionario que representa los rankings de los jugadores de basquetbol junto con el número de juegos que han ganado.

```
rankings = {"Gonzaga": 31, "Baylor": 28, "Michigan": 25, "Illinois": 24, "Houston": 21}
```

Podemos usar un for loop y el método items() para cada uno de los pares key value del diccionario rankings.



```
for team, score in rankings.items():
```

Dentro de for loop, podemos usar f-string para dar formato a los resultados.

El uso de : para {team:10} y {score:10d} da la instrucción de crear un campo con 10 caracteres de longitud.

La d interior de {score:10d} se refiere a un entero decimal.

```
print(f'{team:10} ==> {score:10d}')
```

Veamos nuestro código completo:

```
rankings = {"Gonzaga": 31, "Baylor": 28, "Michigan": 25, "Illinois": 24, "Houston": 21}

for team, score in rankings.items():
    print(f'{team:10} ==> {score:10d}')
```

Gonzaga	==>	31
Baylor	==>	28
Michigan	==>	25
Illinois	==>	24
Houston	==>	21



FORMAT

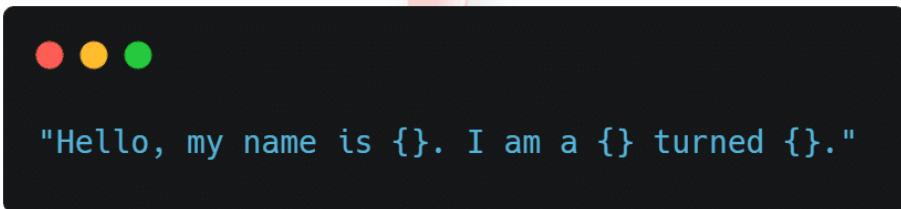
La sintaxis del método str.format() es la siguiente:



```
"template string {}".format(arguments)
```

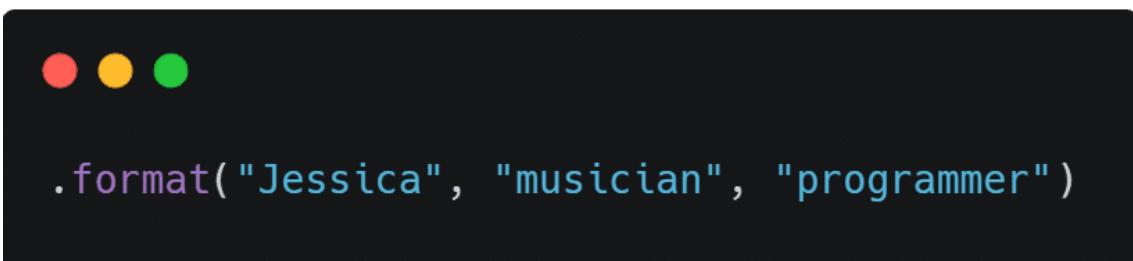
Al interior de la cadena de plantilla podemos usar {} como marcador para los argumentos. Los argumentos son valores que serán mostrados en la cadena.

En este ejemplo, queremos imprimir lo siguiente "Hello, my name is Jessica. I am a musician turned programmer." En la cadena, tendremos un total de tres {} marcadores para los tres valores: "Jessica", "musician", y "programmer". Estos son llamados también campos de formato.



```
"Hello, my name is {}. I am a {} turned {}."
```

Dentro de estos paréntesis con el str.format(), usaremos los valores de "Jessica", "musician", y "programmer".



```
.format("Jessica", "musician", "programmer")
```



Código completo:



```
print("Hello, my name is {}. I am a {} turned {}".format("Jessica", "musician", "programmer"))
```

Argumentos posicionales

Puedes acceder al valor de estos argumentos usando un número como índice dentro de {}. En este ejemplo, los argumentos de "trumpet" y "drums" dentro del .format().



```
.format("trumpet", "drums")
```

Podemos acceder a estos valores dentro de la cadena efectuando una referencia a los números índices. {0} representa el primer argumento de "trumpet" , mientras que {1} es el segundo argumento de "drums".



```
print("Steve plays {0} and {1}.format("trumpet", "drums"))
```

Llave valor

Estos argumentos consisten de un par key value, es decir, de un dúo entre clave y valor. Podemos acceder al valor del argumento(value), usando la clave key al interior del {}.

En este ejemplo, tenemos dos claves llamadas organization y adjective. Usaremos estas claves adentro de la cadena.



```
print("{organization} is {adjective}!".format(organization="freeCodeCamp", adjective="awesome"))
```

Formateo de números con format()

Puede dar formato a los números utilizando el especificador de formato que se proporciona a continuación:

Tipos de formato de número

Tipo	Significado
d	entero decimal
c	Carácter Unicode correspondiente
b	formato binario
o	formato octal
x	Formato hexadecimal (minúsculas)
X	Formato hexadecimal (mayúsculas)
n	Igual que 'd'. Excepto que usa la configuración local actual para el separador de números
e	Notación exponencial. (e minúscula)
E	Notación exponencial (E mayúscula)
f	Muestra el número de punto fijo (Predeterminado: 6)
F	Igual que 'f'. Excepto que muestra 'inf' como 'INF' y 'nan' como 'NAN'
g	formato general. Redondea el número a p dígitos significativos. (Precisión por defecto: 6)
G	Igual que 'g'. Excepto cambia a 'E' si el número es grande.



Formato de número con relleno para int y floats



```
# números enteros con ancho mínimo
print("{:5d}".format(123))
# ancho no funciona para números más largos que el
relleno
print("{:5d}".format(123475))
# relleno para números flotantes
print("{:9.2f}".format(12.4757282))
# números enteros con ancho mínimo lleno de ceros
print("{:05d}".format(123))
# números flotantes con ancho mínimo lleno de ceros
print("{:09.2f}".format(12.4757282))
```

Formateo de números con alineación

Tipos de formato de números con alineación

Tipo	Significado
<	Alineado a la izquierda con el espacio restante
^	Centro alineado con el espacio restante
>	Alineado a la derecha con el espacio restante
=	Fuerza el signo (+) (-) a la posición más a la izquierda



```
● ● ●

# relleno de cadenas con alineación a la izquierda
print("{:<8}".format("Python"))
# relleno de cadenas con alineación a la derecha
print("{:>8}".format("Python"))
# relleno de cadenas con alineación central
print("{:^8}".format("Python"))

# relleno de cadenas con alineación central
# y carácter de relleno '*'

print("{:f^8}".format("Python"))
print("{:=10d}".format(-124))
```

Formateo de cadenas con format()

```
● ● ●

# relleno de cadenas con alineación a la izquierda
print("{:<8}".format("Python"))
# relleno de cadenas con alineación a la derecha
print("{:>8}".format("Python"))
# relleno de cadenas con alineación central
print("{:^8}".format("Python"))

# relleno de cadenas con alineación central
# y carácter de relleno '*'

print("{:f^8}".format("Python"))
print("{:=10d}".format(-124))
```

```
● ● ●

# truncando cadenas a 4 letras
print("{:.4}".format("Python"))
# truncando cadenas a 4 letras
# y relleno
print("{:6.4}".format("Python"))
# truncar cadenas a 4 letras,
# relleno y alineación central
print("{:_>6.4}".format("Python"))
```



RECORRIENDO CADENAS

While

El bucle while se usa como el bucle for para un conjunto dado de declaraciones hasta que una condición dada sea Verdadera. Proporcionamos la longitud de la cadena usando la función len() para iterar sobre una cadena.

En el bucle while, el límite superior se pasa como la longitud de la cadena, atravesada desde el principio. El bucle comienza desde el índice 0 de la cadena hasta el último índice e imprime cada carácter.

```
● ● ●  
cadena = "String"  
i=0;  
while i < len(cadena):  
    print(cadena[i])  
    i=i+1
```

For

El bucle for se utiliza para iterar sobre estructuras como listas, cadenas, etc. Las cadenas son inherentemente iterables, lo que significa que la iteración sobre una cadena da cada carácter como salida.



```
for i in "String":  
    print(i)
```

Función especial enumerate()

La función enumerate() se puede utilizar con cadenas. Se utiliza para llevar un recuento del número de iteraciones realizadas en el bucle. Lo hace agregando un contador al iterable. Devuelve un objeto que contiene una lista de tuplas sobre las que se puede realizar un bucle.



```
for i , j in enumerate("string"):  
    print(i , j)
```



EJERCICIOS

Ejercicio #1

Escribir un programa en Python que solicite el nombre y apellido del usuario, realizar la concatenación de estos datos y a continuación solicitar su fecha de nacimiento (solo el día) e imprimir n veces el dato concatenado sin utilizar bucles, donde n es el dia de nacimiento.



```
nombre,apellido = input("Ingresa tu nombre: "),input("Ingresa tu apellido: ")
nombre_completo = nombre +" "+ apellido
dia_nacimiento = int(input("Ingresa tu dia de nacimiento: "))
nombre_completo += "\n"
print(nombre_completo)
r = nombre_completo*dia_nacimiento
print(r)
```

Ejercicio #2

Escribir un programa que pida al usuario que introduzca una frase en la consola, a continuación muestre la misma frase con las vocales en mayúsculas. Para esto debe crear una función que reciba la cadena y devuelva la cadena convertida.



```
● ● ●

def vocales_mayusculas(cadena):
    vocales = "aeiouáéíóú"
    nueva_cadena = ""
    for c in cadena:
        if c in vocales:
            nueva_cadena += c.upper()
        else:
            nueva_cadena += c
    return nueva_cadena

frase = input("Ingresa una frase: ")
print(vocales_mayusculas(frase))
```

Ejercicio #3

Escribir un programa que pida al usuario que introduzca una frase en la consola y una vocal, y después muestre por pantalla la misma frase pero con la vocal introducida en minúscula.

```
● ● ●

frase = input("Ingresa una frase: ")
vocal = input("Ingresa una vocal: ")
nueva_cadena = ""
vocales = "AEIOUÁÉÍÓÚ"

for c in frase:
    if c in vocales:
        if c == vocal:
            nueva_cadena += c.lower()
        else:
            nueva_cadena += c
    else:
        nueva_cadena += c

print(nueva_cadena)
```

Ejercicio #4

Escribir un programa que pregunte el nombre del usuario en la consola y después de que el usuario lo introduzca muestre por pantalla < NOMBRE > tiene < n > letras,

donde < NOMBRE > es el nombre de usuario en mayúsculas y < n > es el número de letras que tienen el nombre.

```
● ● ●

nombre = input("Ingresa tu nombre: ")
cantidad = len(nombre)
#mensaje = nombre.upper() + " tiene "+ str(cantidad)+ " letras."
print(nombre.upper() + " tiene "+ str(len(nombre))+ " letras.")
```

Ejercicio #5

Crear una función que devuelva la longitud de una cadena, esta función tendrá la misma función que len() por ello no se debe hacer uso de la función len().

```
● ● ●

def longitud(cadena):
    c = 0
    while True:
        try:
            if cadena[c]:
                c+=1
        except IndexError:
            break
    return c
longitud("Python Fundamentals")

# Valor de verdad
#bool("")
```

Continued

Continued

Ejercicio #6

Escribir un programa que pida al usuario que introduzca una frase en la consola y muestre por pantalla la frase invertida.



```
frase = input("Ingresa una frase: ")
l_f = longitud(frase)
i = -1
invertida = ""
while i >= -l_f:
    invertida += frase[i]
    i-=1

print(invertida)
```

Ejercicio #7

Escribir una función que permita verificar si una cadena es un número, la función debe devolver un valor booleano(True o False).

```
def esnumero(cadena):
    try:
        n = int(cadena)
        r = True
    except ValueError:
        try:
            n = float(cadena)
            r = True
        except ValueError:
            r = False
    return r

esnumero("2.5647")
```

Ejercicio #8

Escribir una función que verifique si una palabra se encuentra en una frase, para ello debe recibir 2 parámetros (frase,palabra). La función debe devolver True o False, no utilizar "in" en sentencias condicionales.

```
#print("Hola" in "Hola mundo")
def verificar(f,p):
    r = False
    aux = ""
    for i in range(len(f)):
        if f[i] == " ":
            if aux == p:
                r = True
            aux = ""
        else:
            aux += f[i]
        if i == len(f)-1:
            if aux == p:
                r = True
    return r

verificar("Escribir una función, que verifique","función")
```

Ejercicio #9

Escribir una función que verifique si una palabra se encuentra en una frase, para ello debe recibir 2 parámetros (frase, palabra). La función debe devolver True o False, no utilizar "in" en sentencias condicionales.



```
# "12-HPLaptop-1000" a,b,c

def separar(datos):
    aux = ""
    bandera = 1
    for caracter in datos:
        if caracter == "-":
            if bandera == 1:
                a = aux
            elif bandera == 2:
                b = aux
            bandera += 1
            aux = ""
        else:
            aux += caracter
    if bandera == 3 and aux != "" and aux != " " :
        c = aux
        r = a,b,c
    else :
        r = "Los datos enviados no son correctos.."
    return r
print(separar("12-HPLaptop-1000"))
```



MÉTODOS DE CADENA

El tipo de dato str es una clase incorporada cuyas instancias incluyen variados métodos —más de treinta— para analizar, transformar, separar y unir el contenido de las cadenas de caracteres. En esta lección veremos algunos pero puedes encontrarlos todos en el siguiente enlace.

Python Docs
Cadenas de caracteres, Métodos

[PYTHON DOCS](#)

Métodos de análisis

Count()

El método count() retorna el número de veces que se repite un conjunto de caracteres especificado.



```
● ● ●  
s = "Hola mundo"  
s.count("Hola") #4
```

Find()



El método `find()` retorna la ubicación (índice tomando en cuenta que empieza desde cero y de izquierda a derecha) en la que encuentra el argumento indicado, devuelve -1 si no lo encuentra.

```
● ● ●

-----
s = "Hola mundo mundo"
s.find("mundo") #5

-----
#Si quiere comenzar de derecha a
#izquierda utilice rfind()

-----
s = "Hola mundo mundo"
s.rfind("mundo") #11
-----
```

isspace()

El método `isspace()` devuelve True si solo hay caracteres de espacio en blanco en la cadena. Si no, devuelve False.

```
● ● ●

-----
s = "      "
s.isspace() # True
-----
```

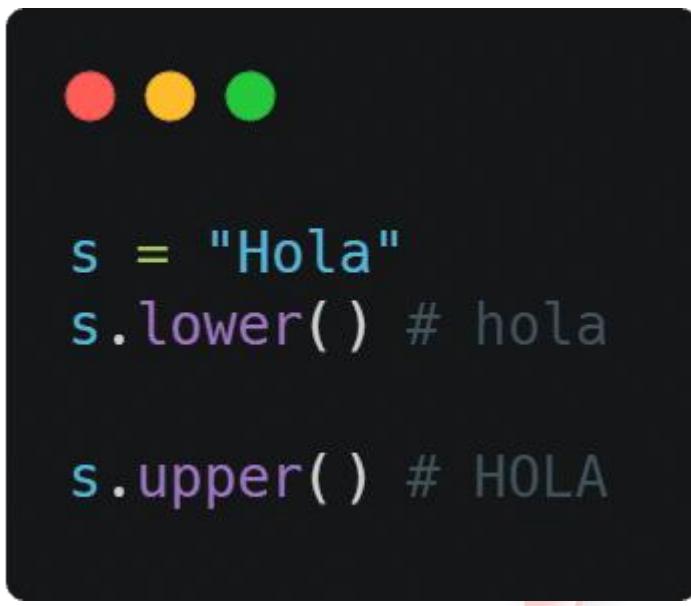


Métodos de transformación

Recuérdese que las cadenas son inmutables; por ende, todos los métodos a continuación no actúan sobre el objeto original sino que retornan uno nuevo.

LOWER() - UPPER()

lower() y upper() retornan una copia de la cadena con todas sus letras en minúsculas o mayúsculas según corresponda.



```
s = "Hola"  
s.lower() # hola  
  
s.upper() # HOLA
```

CAPITALIZA() - SWAPCASE()

capitalize() retorna la cadena con su primera letra en mayúscula.
swapcase(), por su parte, cambia las mayúsculas por minúsculas y viceversa.



```
s = "hola MUNDO"
s.capitalize() # Hola mundo

s = "HoLa"
s.swapcase() # hOlA
```

MÉTODOS DE ALINEACIÓN

Métodos de separación y unión

Split()

El método de división de una cadena según un carácter separador más empleado es split(), cuyo separador por defecto son espacios en blanco y saltos de línea.



```
"Hola mundo!\nHello world!".split()
#[ 'Hola', 'mundo!', 'Hello', 'world!' ]
```

Join()



El método `join()` —sumamente útil—, que debe ser llamado desde una cadena que actúa como separador para unir dentro de una misma cadena resultante los elementos de una lista.

```
  ● ● ●  
" ".join( [ "Hola" , "mundo" ] )  
  
#'Hola mundo'
```





LISTAS

Listas

Las listas son como matrices de tamaño dinámico, declaradas en otros lenguajes (vector en C++ y ArrayList en Java). Las listas no necesitan ser siempre homogéneas, lo que las convierte en la herramienta más poderosa de Python . Una sola lista puede contener tipos de datos como enteros, cadenas y objetos. Las listas son mutables y, por lo tanto, pueden modificarse incluso después de su creación.

length = 5					
	'p'	'r'	'o'	'b'	'e'
index	0	1	2	3	4
negative index	-5	-4	-3	-2	-1



Las listas en Python están ordenadas y tienen un conteo definido. Los elementos de una lista se indexan de acuerdo con una secuencia definida y la indexación de una lista se realiza con 0 como primer índice. Cada elemento de la lista tiene su lugar definido en la lista, lo que permite la duplicación de elementos en la lista, donde cada elemento tiene su propio lugar y credibilidad distintos. También funcionan como las cadenas en cuestión de índices positivos y negativos.

Crear una lista

Las listas en Python se pueden crear simplemente colocando la secuencia dentro de los corchetes []. A diferencia de Sets , una lista no necesita una función integrada para la creación de una lista. Lo importante de una lista es que los elementos de una lista no necesitan ser del mismo tipo. Crear una lista es tan simple como poner diferentes valores separados por comas entre corchetes.

```
list1 = [1, 2, 3, 4, 5 ]  
print("\n Lista 1: ")  
print(list1)  
  
list2 = ['a', 'b', 'c', 'd']  
print("\n Lista 2: ")  
print(list2)
```

Salida:

Lista 1:

[1, 2, 3, 4, 5]

Lista 2:

['a', 'b', 'c', 'd']

Crear una lista con múltiples elementos distintos o duplicados

```
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]  
print("\nLista con elementos repetidos: ")  
print(List)  
  
List = [1, 2, 'Hola', 4, 'que', 6, 'tal']  
print("\nLista con elementos diferentes: ")  
print(List)
```



Salida:

Lista con elementos repetidos:

[1, 2, 4, 4, 3, 3, 3, 6, 5]

Lista con elementos diferentes:

[1, 2, 'Hola', 4, 'que', 6, 'tal']

Se puede acceder a los elementos de la lista por índice

Se puede acceder a los elementos individuales de una lista mediante un índice entre corchetes. Esto es exactamente análogo a acceder a caracteres individuales en una cadena. La indexación de la lista se basa en cero como lo es con las cadenas.

Considere la siguiente lista:

```
>>> a = ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis']
>>> a[0]
'uno'
>>> a[2]
'tres'
>>> a[5]
'seis'
```

Prácticamente todo lo relacionado con la indexación de cadenas funciona de manera similar para las listas. Por ejemplo, un índice de lista negativo cuenta desde el final de la lista:

```
>>> a[-1]
'seis'
>>> a[-2]
'cinco'
>>> a[-5]
'tres'
```

Iterando a través de una lista

Usando un bucle for, podemos iterar a través de cada elemento de una lista.

```
for fruit in ['manzana', 'banana', 'mango']:
    print("Fruta: ", fruit)
```

Salida:

Fruta manzana

Fruta banana

Fruta mango



INSERTANDO Y MODIFICANDO ELEMENTOS

Agregar elementos

Método append()

Los elementos se pueden agregar a la lista mediante el uso de la función incorporada `append()`. Solo se puede agregar un elemento a la vez a la lista usando el método `append()`, para agregar múltiples elementos con el método `append()`, se usan bucles. Las tuplas también se pueden agregar a la lista con el uso del método de adición porque las tuplas son inmutables. A diferencia de los conjuntos, las listas también se pueden agregar a la lista existente con el uso del método `append()`.



Ejemplo:

```
● ● ●

# Creando lista
List = []
print("Iniciando lista vacía: ")
print(List)

# Añadiendo elementos a la lista
List.append(1)
List.append(2)
List.append(4)
print("\nLista después de añadir tres elementos: ")
print(List)

# Añadiendo elementos con iterador
for i in range(1, 4):
    List.append(i)
print("\nLista después de añadir 1-3: ")
print(List)
```

Método append()

El método `append()` solo funciona para la adición de elementos al final de la Lista, para la adición de elementos en la posición deseada, se utiliza el método `insert()`. A diferencia de `append()` que toma solo un argumento, el método `insert()` requiere dos argumentos (posición, valor).



Ejemplo:

```
● ● ●

#Creando lista
List = [1,2,3,4]
print("Lista inicial: ")
print(List)

# Insertando elementos en posición específica
List.insert(3, 12)
List.insert(0, 'dato')
print("\nLista después de la inserción: ")
print(List)
```

Método extend()

Aparte de los métodos `append()` e `insert()`, hay un método más para la adición de elementos, `extend()` , este método se usa para agregar varios elementos al mismo tiempo al final de la lista.

Nota: los métodos `append()` y `extend()` solo pueden agregar elementos al final.

Ejemplo:

```
● ● ●

# Creando lista
List = [1, 2, 3, 4]
print("Lista incial: ")
print(List)

# Agregando múltiples elementos
List.extend([8, 'dato1', 'dato2'])
print("\nLista después de agregar elementos: ")
print(List)
```



Modificando elementos

Es posible modificar un elemento de una lista en Python con el operador de asignación `=`. Para ello, lo único que necesitas conocer es el índice del elemento que quieras modificar o el rango de índices:

```
● ● ●

vocales = ['o', 'o', 'o', 'o', 'u']
# Actualiza el elemento del índice 0
vocales[0] = 'a'
print(vocales)
# Actualiza los elementos entre las posiciones 1 y 2
vocales[1:3] = ['e', 'i']
print(vocales)
```





ELIMINANDO POR ELEMENTO

Método remove()

Los elementos se pueden eliminar de la lista utilizando la función remove() incorporada, pero surge un error si el elemento no existe en la lista. El método Remove() solo elimina un elemento a la vez, para eliminar un rango de elementos, se usa el iterador. El método remove() elimina el elemento especificado.

Puntos importantes:

Los siguientes son algunos puntos importantes a tener en cuenta al usar el método remove ():

- Cuando hay elementos duplicados en la lista, el primer elemento que coincide con el elemento en particular se eliminará de la lista.
- Si el elemento dado no está presente en la lista, debe cometer un error al decir que el elemento no está en la lista.
- El método remove() no devuelve ningún valor.



- La resta () toma el valor como argumento, por lo que el valor debe pasar con el tipo de datos correcto.

Ejemplo:

```
#Creando lista
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Lista inicial: ")
print(List)

# Eliminando elementos
List.remove(5)
List.remove(6)
print("\nLista después de eliminar dos elementos: ")
print(List)
```

Salida:

Lista inicial:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Lista después de eliminar dos elementos:

[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

También podemos usar un iterador para ir eliminando los elementos:

```
#Creando lista
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Lista inicial: ")
print(List)

# Eliminando elementos
for i in range(1, 5):
    List.remove(i)
print("\nLista después de eliminar un rango de elementos: ")
print(List)
```

Salida:

Lista inicial:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Lista después de eliminar un rango de elementos:

[6, 7, 8, 9, 10, 11, 12]



ELIMINANDO ELEMENTO POR POSICIÓN



Método pop()

La función Pop() también se puede usar para eliminar y devolver un elemento de la lista, pero por defecto elimina solo el último elemento de la lista, para eliminar un elemento de una posición específica de la Lista, se pasa el índice del elemento como un argumento para el método pop()).

Puntos importantes:

- índice: el método pop() tiene solo un argumento llamado índice.
- Para eliminar un elemento de la lista, debe pasar el índice del elemento. El índice comienza en 0. Para obtener el primer elemento de la lista, pase el índice como 0. Para eliminar el último elemento, puede ejecutar el índice como -1.
- El argumento de índice es opcional. Si no se pasa, el valor predeterminado se considera -1 y se devuelve el último elemento de la lista.



- Si el índice dado no está presente o está fuera de rango, el método pop() arroja un error que dice IndexError: índice pop.
- El método pop() devolverá el elemento extraído según el índice dado. La lista final también se actualiza y no contendrá el elemento.

Ejemplo:

```
#Creando lista
List = [1,2,3,4,5]

# Eliminando último elemento
List.pop()
print("\nLista después de usar pop(): ")
print(List)

# Eliminando elemento en posición específica
List.pop(2)
print("\nLista después de eliminar un elemento: ")
print(List)
```

Salida:

Lista después de usar pop():
[1, 2, 3, 4]

Lista después de eliminar un elemento:
[1, 2, 4]



LISTAS

EJERCICIOS

Ejercicio #1

Escriba una función que tome una lista de números y devuelva la suma acumulada, es decir, una nueva lista donde el primer elemento es el mismo, el segundo elemento es la suma del primero con el segundo, el tercer elemento es la suma del resultado anterior con el siguiente elemento y así sucesivamente. Por ejemplo, la suma acumulada de [1,2,3] es [1, 3, 6]

```
● ● ●  
def sumaacumulada(lista):  
    s = 0  
    n_l = []  
    for e in lista:  
        s += e  
        n_l.append(s)  
    return n_l  
lista_numero = [2,2,3,4]  
r = sumaacumulada(lista_numero)  
print(r)
```



Ejercicio #2

Escribe una función llamada "esordenada" que tome una lista como parámetro y devuelva True si la lista está ordenada en orden ascendente y devuelva False en caso contrario.

Por ejemplo, esordenada([1, 2, 3]) retorna True y esordenada([b, a]) retorna False.

```
def esordenada(lista):
    for i in range(len(lista)):
        if i < len(lista) -1:
            if lista[i] <= lista[i+1]:
                r = True
            else:
                r = False
    return r

print(esordenada( ["a", "b", "c","c" ] ))
```

Ejercicio #3

Escribe una función llamada "duplicado" que tome una lista y devuelva True si tiene algún elemento duplicado. La función no debe modificar la lista.

```
# ["a","b","d","b"]

def duplicado(lista):
    r = False
    for i in range(len(lista)): # Principal
        #print(f"Vamos a comparar {lista[i]} con {lista[i+1:len(lista)]}")
        for j in range(i+1,len(lista)):
            #print(f"{lista[i]} es igual {lista[j]} >> {lista[i] == lista[j]}")
            if lista[i] == lista[j]:
                r = True
    return r
duplicado([2,3,4,6])
```



Ejercicio #4

Crear una función que genere una lista de 23 números aleatorios del 1 al 100.

```
● ● ●

import random

def generar():
    l = []
    for i in range(23):
        n = random.randint(1,100)
        l.append(n)
    return l

print(generar())
```

Ejercicio #5

Escribe una función llamada "elimina_duplicados" que tome una lista y devuelva una nueva lista con los elementos únicos de la lista original. No tienen porque estar en el mismo orden.

[1,2,3,4,1,2] >> [3,4]

```
● ● ●

def elimina_duplicado(lista):
    nueva_lista = []
    for e in lista:
        if e in nueva_lista:
            nueva_lista.remove(e)
        else:
            nueva_lista.append(e)
    return nueva_lista
lista = generar()
#print("Lista original")
#print(lista)
print(elimina_duplicado([1,2,3,4,1,2]))
```



Ejercicio #6

Escribe una función llamada "busqueda_palindromo" que busque todas las palabras palíndromas de una lista.

Ejemplo de palabras inversas: radar, oro, rajar, rallar, salas, somos, etc...

```
● ● ●

def invertir(frase):
    l_n = -len(frase)
    i = -1
    invertida = ""
    while i >= l_n:
        invertida += frase[i].lower()
        i-=1
    return invertida

def busqueda_palindromo(lista):
    n_l = []
    for e in lista:
        if e.lower() == invertir(e):
            n_l.append(e)
    return n_l

print(busqueda_palindromo(["Hola","Oso","Luna","rallar"]))
```

Ejercicio #7

Queremos guardar los nombres y la edades de los alumnos de un curso. Realiza un programa que solicite el nombre y la edad de cada alumno. El proceso de lectura de datos terminará cuando se introduzca como nombre un asterisco (*) Al finalizar se mostrará los siguientes datos:

Todos los alumnos mayores de edad (≥ 18)



```
● ● ●

nombres = []
edades = []
while True:
    nombre = input("Ingresa nombre: ")
    if nombre != "*":
        edad = int(input("Ingresa edad: "))
        nombres.append(nombre)
        edades.append(edad)
    else:
        break

for i in range(len(edades)):
    if edades[i] >= 18:
        print(nombres[i],edades[i])
```

Ejercicio #8

Realizar una función llamada "ordenar" que reciba una lista desordenada de números (flotantes, enteros) y devuelva la lista ordenada de manera ascendente. Utilizar ordenamiento por selección.

```
● ● ●

def ordenar(lista):
    for i in range(len(lista)):
        for j in range(i+1,len(lista)):
            if lista[j] > lista[i]:
                aux = lista[i]
                lista[i] = lista[j]
                lista[j] = aux
    return lista

print(ordenar([5,7,9,1,2,8,7,1]))
```



MATRICES

Matriz

En programación, se le denomina vector, formación, matriz (en inglés array, del cual surge la mala traducción arreglo), a una zona de almacenamiento contiguo que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

En principio, se puede considerar que todas las matrices son de una dimensión, la dimensión principal, pero los elementos de dicha fila pueden ser a su vez matrices (un proceso que puede ser recursivo), lo que nos permite hablar de la existencia de matrices multidimensionales, aunque las más fáciles de imaginar son las de una, dos y tres dimensiones.

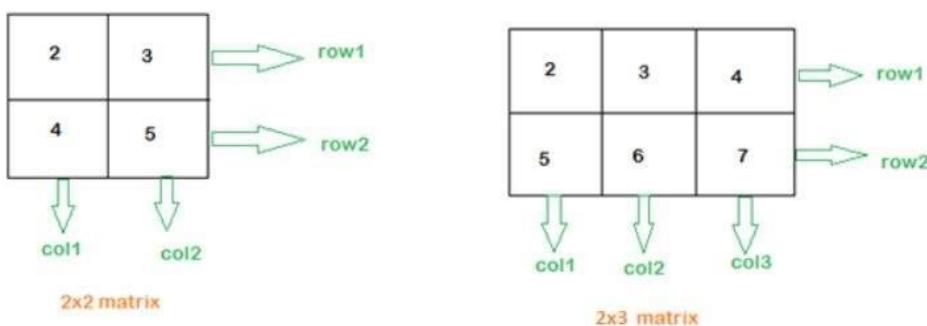
Matrices - Python

Cuando hablamos de matrices en Python, nos estamos refiriendo a una matriz rectangular bidimensional especializada de datos, los cuales están almacenados



en filas y columnas. Dentro de esta matriz puede haber datos en forma de números, cadenas, símbolos, expresiones, etc. La matriz es una de las estructuras de datos importantes que se pueden utilizar en cálculos matemáticos y científicos.

Los datos dentro de la matriz tienen un aspecto como el mostrado en esta gráfica:



Matriz izquierda: Acá se muestra una matriz de 2x2, donde se observan dos filas y dos columnas. Los datos dentro de esta matriz están presentados en forma de números, donde se observa en la fila uno los valores 2, 3 y en la fila dos los valores 4, 5. En las columnas, está la columna uno con los valores 3 y 4 y la columna dos con los valores 3 y 5.

Matriz derecha: Se muestra una matriz ligeramente diferente, de 2 x 3, con dos filas y tres columnas. Los datos dentro de la primera fila tienen valores 2, 3, 4 y en la segunda fila tienen valores 5, 6, 7. La primera columna tiene valores 2,5, la segunda columna tiene valores 3,6 y la tercera columna tiene valores 4,7.

Crea una matriz de Python usando un tipo de datos de lista anidada

En Python, las matrices se representan mediante el tipo de datos de lista. Vamos a ver cómo crear una matriz de 3x3 haciendo uso de la lista.

La matriz está compuesta por tres filas y tres columnas.

- La fila número uno dentro del formato de lista tendrá los siguientes datos: [8,14, -6]
- La fila número dos será: [12,7,4]
- La fila número tres será: [-11,3,21]



Ejemplo:

```
#Creando matriz
M1 = [[8, 14, -6],
      [12, 7, 4],
      [-11, 3, 21]]
#Imprimiendo matriz
print(M1)
Salida: [[8, 14, -6], [12, 7, 4], [-11, 3, 21]]]
```

Leer el último elemento de cada fila

```
#Creando matriz
M1 = [[8, 14, -6],
      [12, 7, 4],
      [-11, 3, 21]]

matrix_length = len(M1)

#Leyendo último elemento de cada fila
for i in range(matrix_length):
    print(M1[i][-1])
Salida:
-6
4
21
```

imprimir las filas de una matriz

```
#Creando matriz
M1 = [[8, 14, -6],
      [12, 7, 4],
      [-11, 3, 21]]

matrix_length = len(M1)

#Imprimiendo las filas de la matriz
for i in range(matrix_length):
    print(M1[i])
Salida:
[8, 14, -6]
[12, 7, 4]
[-11, 3, 21]
```



MATRICES

EJERCICIOS

Ejercicio #1

Realizar la suma de las matrices A y B.

```
● ● ●
A = [
    [2,4,6],
    [7,4,5],
    [4,5,6]
]
B = [
    [4,2,3],
    [2,5,7],
    [9,8,2]
]

R = [[],[],[]]

for f in range(len(A)):
    fila = R[f]
    for c in range(len(A[f])):
        r = A[f][c] + B[f][c]
        fila.append(r)

print(R)
```



Ejercicio #2

Obtener la Matriz transpuesta de la matriz A.

$$\mathbf{Z}_{3 \times 3} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \mathbf{d} & \mathbf{e} & \mathbf{f} \\ \mathbf{g} & \mathbf{h} & \mathbf{i} \end{pmatrix} \rightarrow \text{Trasposición} \rightarrow \mathbf{Z}^T_{3 \times 3} = \begin{pmatrix} \mathbf{a} & \mathbf{d} & \mathbf{g} \\ \mathbf{b} & \mathbf{e} & \mathbf{h} \\ \mathbf{c} & \mathbf{f} & \mathbf{i} \end{pmatrix}$$



```
● ● ●
A = [
    [1, 5, 9],
    [7, 5, 3],
    [4, 8, 6]
]

T = []
for f in range(len(A)):
    fila = []
    for c in range(len(A[f])):
        t = A[c][f]
        fila.append(t)
    T.append(fila)

print(T)
```

Ejercicio #3

Generar una matriz cuadrada de dimensión 5 y rellenarla con números aleatorios.



```
● ● ●

import random
MC5 = []
d = 5
for f in range(d):
    fila = []
    for c in range(d):
        e = random.randint(1,10)
        fila.append(e)
    MC5.append(fila)

for f in range(len(MC5)):
    print("|",end="")
    for c in range(len(MC5[f])):
        print("{:^4}".format(MC5[f][c]),end="")
    print("|",end="")
    print("")
```

Ejercicio #4

Definir una función que permita generar matrices cuadradas con el siguiente patrón, la función recibe como parámetro la dimensión.

```
● ● ●

import random

def generar_matrices(d):
    MG = []
    for f in range(d):
        fila = []
        for c in range(d):
            if c <= f:
                e = random.randint(1,10)
            else:
                e = 0
            fila.append(e)
        MG.append(fila)
        imp_matriz(MG)
        print("-----")
    return MG
imp_matriz(generar_matrices(7))
```



Ejercicio #5

Definir una función que permita generar matrices cuadradas con el siguiente patrón, la función recibe como parámetro la dimensión. Debe ser generada por columnas.

```
● ● ● ● ●  
  
def generar_matrices5C(d):  
    MG = []  
    for i in range(d):  
        MG.append([])  
    e = 1  
    for c in range(d):  
        for f in range(d):  
            fila = MG[f]  
            if c <= f:  
                fila.append(e)  
                e += 1  
            else:  
                fila.append(0)  
    #imp_matriz(MG)  
    return MG  
  
def generar_matrices5F(d):  
    MG = []  
    for f in range(d):  
        fila = [f+1]  
        s = 4  
        for c in range(1,d):  
            if f >= c:  
                fila.append(fila[c-1]+s)  
            else:  
                fila.append(0)  
            s -= 1  
        MG.append(fila)  
    #imp_matriz(MG)  
    #print("-----")  
    return MG  
imp_matriz(generar_matrices5C(5))
```



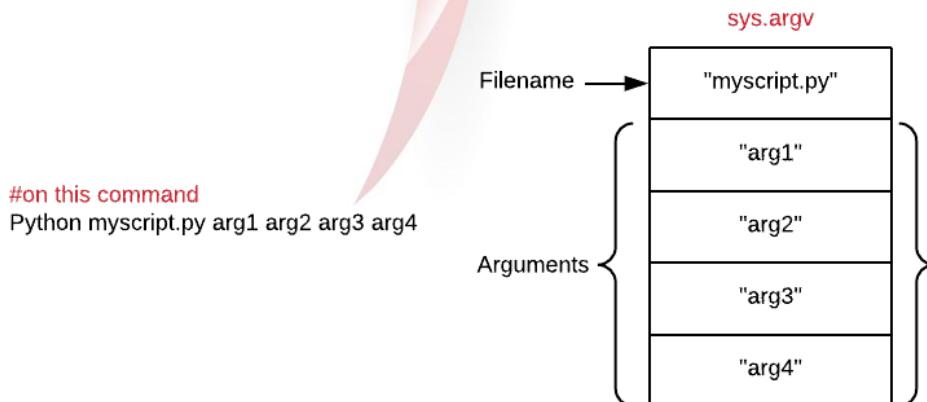
ARGUMENTOS POR LINEA DE COMANDOS

Argumentos por línea de comando

Al invocar un programa desde la línea de comando, frecuentemente podemos querer pasarle argumentos que controlen algún aspecto de su comportamiento. Por ejemplo, si el programa procesa información almacenada en un archivo, se lo puede invocar pasándole el nombre del archivo a procesar. De esa manera, se puede utilizar el mismo programa para procesar cualquier archivo, sin necesidad de modificar el código.

La forma más básica y sencilla de procesar los argumentos de la línea de comando es mediante la función argv, provista por el módulo sys.

Sys.argv



Para que esta función esté accesible, al escribir el programa en Python primero se debe importar el módulo mediante la instrucción import sys. Al invocar el



programa, se puede agregar en la línea de comando una serie de argumentos, que son cadenas de caracteres separados por espacio en blanco:



```
python programa.py argumento_1 argumento_2 [.... argumento_N]
```

Si un argumento debe incluir espacios en blanco, la cadena debe encerrarse entre comillas:



```
python programa.py "argumento con espacios"
```

La función `sys.argv` devuelve una lista de cadenas de caracteres, correspondientes a cada uno de los argumentos escritos al invocar el programa.

Los argumentos se ordenan en la lista tal como aparecen en la línea de comando, y por tanto se los llama argumentos posicionales. El primer elemento de la lista (`sys.argv[0]`) es el nombre del programa. La cantidad de argumentos pasados al invocar el programa se obtiene con la función `len(sys.argv)`.

Ejecución

Desde la terminal

Para ejecutar un programa desde la línea de comando, deben utilizar la instrucción `python` (o `python3`, según corresponda) desde una terminal, en el mismo directorio en el que se encuentran los programas de ejemplo, tal como se ven en la parte de arriba.



```
python programa.py argumento_1 argumento_2 [.... argumento_N]
```

Desde Jupyter

También puedes ejecutar un script ".py" desde jupyter, para ello solo debes escribir !python seguido del nombre del archivo y los argumentos.



```
!python programa.py 1 2 3
```





TUPLAS

Tuples in Python

```
t = (1, 2, 'Python', tuple(), (42, 'hi'))  
     \ |    \ |    |  
t[0] t[1] t[2] t[3] t[4]
```

Tuplas

A veces, desea crear una lista de elementos que no se pueden cambiar en todo el programa. Las tuplas te permiten hacer eso.

Una tupla es una colección de objetos ordenados e inmutables. Las tuplas son secuencias, al igual que las listas. Las diferencias entre tuplas y listas son que las tuplas no se pueden cambiar a diferencia de las listas y las tuplas usan paréntesis, mientras que las listas usan corchetes.

Características:

1. Las tuplas son colecciones de datos ordenadas e indexadas. De forma similar a los índices de cadena, el primer valor de la tupla tendrá el índice [0], el segundo valor [1] y así sucesivamente.
2. Las tuplas pueden almacenar valores duplicados.
3. Una vez que los datos se asignan a una tupla, los valores no se pueden cambiar.



4. Las tuplas le permiten almacenar varios elementos de datos en una variable. Puede optar por almacenar solo un tipo de datos en una tupla o mezclarlos según sea necesario.

Crear un tupla

Una tupla se crea colocando todos los elementos (elementos) entre paréntesis (), separados por comas. La documentación oficial de Python establece que colocar elementos entre paréntesis es opcional y que los programadores pueden declarar tuplas sin usarlas. Sin embargo, se considera una buena práctica usar corchetes al declarar una tupla, ya que hace que el código sea más fácil de entender.

Una tupla puede tener cualquier número de elementos y pueden ser de diferentes tipos (entero, flotante, lista, cadena , etc.).

Ejemplo:

```
#Creando tuplas
tup1 = ('física', 'química', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"

#Imprimiendo tuplas
print(tup1)
print(tup2)
print(tup3)
```

Acceso a valores en tuplas

En Python, puede acceder a las tuplas de varias formas. Lo que es vital que recuerde es que los índices de tupla de Python son como los índices de cadena de Python: ambos están indexados y comienzan en 0.

Por lo tanto, al igual que los índices de cadena, las tuplas se pueden concatenar, dividir, etc. Las tres formas principales de acceder a las tuplas en Python son la indexación, la indexación negativa y el corte.

Indexación

El operador de índice resulta útil al acceder a las tuplas. Para acceder a una tupla específica en una tupla, puede usar los operadores "[]". Tenga en cuenta que la indexación comienza desde 0 y no desde 1.



En otras palabras, una tupla que tiene cinco valores tendrá índices del 0 al 4. Si intenta acceder a un índice fuera del rango existente de la tupla, generará un "IndexError".

Además, el uso de un tipo flotante u otro tipo dentro del operador de índice para acceder a los datos en una tupla generará un "TypeError".

Ejemplo:

```
tupla1 = (1, 3, 5, 7, 9);  
print(tupla1 [0])
```

También puede poner tuplas dentro de tuplas. Esto se llama anidar tuplas. Para acceder a un valor de una tupla que está dentro de otra tupla, debe encadenar los operadores de índice.

Ejemplo:

```
tupla1 = ((1, 3), (5, 7));  
print(tupla1 [0] [1])  
print(tupla1 [1] [0])
```

Indexación negativa

Algunos idiomas no permiten la indexación negativa, pero Python no es uno de esos idiomas.

En otras palabras, el índice "-1" en tuplas se refiere al último elemento de la lista, el índice "-2" se refiere al penúltimo elemento, y así sucesivamente.

Ejemplo:

```
tupla1 = (1, 3, 5, 7)  
print(tupla1[-1])  
print(tupla1 [-2])
```

Slicing

Acceder a valores de tupla a través de cortes significa acceder a los elementos usando el operador de corte, que son los dos puntos ("::").



Ejemplo:

```
tupla1 = ('p','y','t','h','o','n')
print(tuple1[1:4])
print(tuple1[:-4 ])
print(tuple1[3:])
print(tuple1[:])
```





MÉTODOS BÁSICOS - TUPLAS

Método count()

El método count() de Tuple devuelve el número de veces que aparece el elemento dado en la tupla.

Sintaxis:

```
tupla.cuenta(elemento)
```

Ejemplo:

```
# Creando tuplas
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
Tuple2 = ('python', 'dato', 'python',
          'for', 'java', 'python')

# contando las apariciones de 3
res = Tuple1.count(3)
print('La cantidad del elemento 3 en Tuple1 es:', res)

# contando las apariciones de python
res = Tuple2.count('python')
print('La cantidad del elemento python en Tuple2 es:', res)
```



Salida:

La cantidad del elemento 3 en Tuple1 es: 3
La cantidad del elemento python en Tuple2 es: 3

Método index()

El método Index() devuelve la primera aparición del elemento dado de la tupla.

Sintaxis:

```
tuple.index(elemento, inicio, final)
```

Parámetros:

- elemento: El elemento a buscar.
- inicio (Opcional): El índice de inicio desde donde se inicia la búsqueda
- fin (Opcional): El índice final hasta donde se realiza la búsqueda

Nota: este método genera un ValueError si el elemento no se encuentra en la tupla.

Ejemplo #1:

```
# Creando tupla
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)

# obteniendo indice de 3
res = Tuple.index(3)
print('La primera aparición de 3 es ', res)

# obteniendo indice de 3 después de la posición 4
res = Tuple.index(3, 4)
print('La primera aparición de 3 después del 4º índice es:', res)
```

Salida:

```
La primera aparición de 3 es 3
La primera aparición de 3 después del 4º índice es: 5
```



Ejemplo #2:

```
# Creando tupla
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)

# obteniendo índice de 4
res = Tuple.index(4)
```

Salida: ValueError: tuple.index(x): x not in tuple





DESEMPAQUETANDO TUPLAS



Desempaquetando Tuplas

Las tuplas se utilizan para almacenar objetos inmutables. Las tuplas de Python son muy similares a las listas, excepto en algunas situaciones. Las tuplas de Python son inmutables, lo que significa que no se pueden modificar en todo el programa.

En Python, hay una función de asignación de tupla muy poderosa que asigna el lado derecho de los valores al lado izquierdo. De otra forma, se llama desempaquetar una tupla de valores en una variable. Al empaquetar, ponemos valores en una nueva tupla mientras que al desempaquetar extraemos esos valores en una sola variable.



Ejemplo #1:

```
# Creamos una tupla
a = ("Intecssa", 5000, "Programación")

# desempaquetamos en otras variables
(instituto, estudiante, curso) = a

# imprimiendo el primero
print(instituto)

# imprimiendo el segundo
print(estudiante)

# Imprimiendo el tercero
print(curso)
```

Salida:

Intecssa
5000
Ingenieria

NOTA: Al desempaquetar la tupla, el número de variables en el lado izquierdo debe ser igual al número de valores en la tupla a dada.

Ejemplo #2:

Python usa una sintaxis especial para pasar argumentosopcionales (*args) para el desempaquetado de tuplas. Esto significa que puede haber muchos argumentos en lugar de (*args) en python. Todos los valores se asignarán a cada variable en el lado izquierdo y todos los valores restantes se asignarán a *args.

```
# el primero y el último se asignarán a x y z
# al restante se asignará a y
x, *y, z = (10, "Curso", "de", "Python", 50)
print(x)
print(y)
print(z)

# primero y segundo serán asignados a x e y
# al restante se asignará a z
x, y, *z = (10, "Curso", "de", "Python", 50)
print(x)
print(y)
print(z)
```

Salida:

```
10
['Curso', 'de', 'Python']
50
10
Curso
[' de', 'Python', 50]
```

Ejemplo #3:

En python, las tuplas se pueden desempaquetar usando una función, en la función se pasa la tupla y en la función, los valores se desempaquetan en una variable normal. Considere el siguiente código para una mejor comprensión.

```
# función toma argumentos normales
# y los multiplica
def result(x, y):
    return x * y
# función con variable normales
print (result(10, 100))

# Se crea una tupla
z = (10, 100)

# Se pasa la tupla desempaquetada
print (result(*z))
```

Salida:

```
1000
1000
```



CONJUNTOS

Conjuntos

Un conjunto es una colección no ordenada de objetos únicos. Python provee este tipo de datos «por defecto» al igual que otras colecciones más convencionales como las listas, tuplas y diccionarios.

Los conjuntos son ampliamente utilizados en lógica y matemática, y desde el lenguaje podemos sacar provecho de sus propiedades para crear código más eficiente y legible en menos tiempo.

Creación de un conjunto

Para crear un conjunto especificamos sus elementos entre llaves. Al igual que otras colecciones, sus miembros pueden ser de diversos tipos:

Ejemplo:

```
>>> s = {1, 2, 3, 4}  
>>> s = {True, 3.14, None, False, "Hola mundo", (1, 2)}
```



No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

```
>>> s = {[1, 2]}\nTraceback (most recent call last):\n...\nTypeError: unhashable type: 'list'
```

Cómo acceder a los elementos de un conjunto

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos list o tuple. Es por ello que no se puede acceder a los elementos a través de un índice.

Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

Ejemplo:

```
mi_conjunto = {1, 3, 2, 9, 3, 1}\nfor e in mi_conjunto:\n    print(e)
```

Salida:

```
1\n2\n3\n9
```

Número de elementos de un conjunto

Como con cualquier otra colección, puedes usar la función len() para obtener el número de elementos contenidos en un conjunto.

Ejemplo:

```
>>> mi_conjunto = set([1, 2, 5, 3, 1, 5])\n>>> len(mi_conjunto)\n4
```

Cómo saber si un elemento está en un conjunto

Con los conjuntos también se puede usar el operador de pertenencia in para comprobar si un elemento está contenido, o no, en un conjunto.



Ejemplo:

```
>>> mi_conjunto = set([1, 2, 5, 3, 1, 5])
>>> print(1 esta en mi_conjunto)
True
>>> print(6 esta en mi_conjunto)
False
>>> print(2 esta en mi_conjunto)
False
```





AÑADIENDO ELEMENTOS

Añadiendo elementos

Agregar elementos a un conjunto en Python básicamente significa actualizar el conjunto con uno o varios elementos. Sabemos que los valores establecidos son inmutables, lo que significa que los valores de un conjunto no se pueden cambiar después de su creación. Sin embargo, el conjunto en sí es mutable, lo que significa que podemos realizar operaciones de agregar, leer y eliminar en él.

Usando la función update()

Esta es una función incorporada que se usa para agregar elementos a un conjunto en Python. Esta función utiliza una sola línea para realizar la adición de elementos. Es más rápido y práctico que otros. Este método es útil si el usuario desea agregar varios elementos de una sola vez.

Sintaxis:

```
set.update( iterable )
```



Ejemplo #1:

```
#Creando set
set1 = {1, 2, 3, 4, 5}

# lista de números a agregar
list_to_add = [6, 7, 8]

# agregando lista de número al set
set1.update(list_to_add)

print('Conjunto actualizado después de agregar elementos: ', set1)
Salida: Conjunto actualizado después de agregar elementos: {1, 2, 3, 4, 5, 6, 7, 8}
```

Ejemplo #2:

El siguiente ejemplo toma un conjunto de elementos de entrada. Se definen tres nuevas listas de elementos que contienen elementos para agregar al conjunto original . Pasamos las listas como argumento a la función actualizar(). Agrega todos los elementos de las tres listas al conjunto. El conjunto contiene solo elementos únicos, por lo que los elementos que no estaban presentes en el conjunto se agregarán y los elementos duplicados se omitirán.

```
# Creando set
set1 = {11, 12, 13, 14}

# 3 listas de números
list1 = [15, 16, 17]
list2 = [18, 19]
list3 = [30, 31, 19, 17]

# agregando múltiples listas
set1.update(list1, list2, list3)

print('Conjunto actualizado: ', set1)
Salida: Conjunto actualizado: {11, 12, 13, 14, 15, 16, 17, 18, 19, 30, 31}
```

Usando la función add()

Esta es una función incorporada de set que se usa para agregar un elemento al conjunto. Esta función agrega solo un elemento a la vez al conjunto. El conjunto contiene solo elementos únicos, por lo que si intentamos agregar un elemento que ya existe en el conjunto, no agrega ese elemento y ejecuta el programa con el conjunto original.



Sintaxis:

```
set.add( elemento )
```

Ejemplo #1:

```
#Creando set
set1 = {1, 2, 3, 4, 5}

# agregando 1 elemento al set
set1.add(6)

print('Conjunto actualizado después de agregar el elemento: ', set1)
Salida: Conjunto actualizado después de agregar el elemento: {1, 2, 3, 4, 5, 6}
```

Ejemplo #2:

También podemos agregar una lista de elementos en el conjunto. Se define una nueva lista de elementos que contiene elementos para agregar al conjunto original . Pasamos la lista como argumento a la función add(). En el caso del método add(), si pasamos una lista a la función add(), entonces el programa dará TypeError porque la lista tiene un objeto que no se puede modificar y add() requiere un objeto que se puede modificar como cadena, tupla, etc.

```
#Creando set
set1 = {1, 2, 3, 4, 5}

#lista de numeros a agregar
list1 = [6,7]

# agregando lista al set
set1.add(list1)

print('Conjunto actualizado después de agregar elementos: ', set1)
Salida: TypeError: tipo no modificable: 'lista'
```

Ejemplo #3:

En este ejemplo, se define una tupla de elementos que contiene elementos para agregar al conjunto original . Pasamos la tupla como argumento a la función add(). Agrega todos los elementos de la tupla al conjunto. En este caso, si intentamos agregar una tupla ya existente a un conjunto, no la agregará y no generará un error.

```
#Creando set
set1 = {1, 2, 4, 5}

# tupla para agregar
```



```
tuple1 = (6, 7)
set1.add(tuple1)

print("Conjunto actualizado después de agregar la tupla: ', set1)
```

Salida: Conjunto actualizado después de agregar la tupla: {1, 2, 4, 5, (6, 7)}





REMOVIENDO ELEMENTOS



Aprenderemos sobre 4 métodos para eliminar elementos del conjunto de Python, todos son métodos integrados de Python Set para eliminar elementos (eliminar, descartar, abrir, borrar). Estas son las funciones incorporadas del conjunto para eliminar elementos del conjunto.

- `remove()` : Elimina un elemento de Set.
- `discard()` : Elimina un elemento de Set
- `pop ()` : elimina un elemento arbitrario (aleatorio)
- `clear ()` : eliminar todos los elementos

Usando la función `remove()`

Elimina el elemento dado del conjunto. Si un elemento pasado no existe en Set, genera un KeyError.

Parámetro:



Elemento: Toma un solo parámetro requerido, indica el elemento que queremos eliminar.

Valor de retorno:

Elimina el elemento dado del conjunto y actualiza el conjunto. Devuelve None (nada).

Sintaxis:

```
Set.remove(elemento)
```

Ejemplo:

```
#Creando set
first_set = {'noche', 'Hola', 5, 6, 7, 'Bueno'}

#eliminando elemento
first_set.remove(5)

print('Conjunto después de eliminar elemento :',first_set)
```

Salida: Conjunto después de eliminar elemento: {6, 7, 'noche', 'Bueno', 'Hola'}

En el siguiente ejemplo de código, estamos eliminando un elemento que no existe en Set . Como podemos ver, arroja KeyError .

```
#Creando set
first_set = {'noche', 'Hola', 5, 6, 7, 'bueno'}

#eliminando elemento que no existe
first_set.remove(10)

print('Conjunto después de eliminar elemento :',first_set)
```

Salida:

```
Traceback (most recent call in <module>):
  File "main.py", line 5, in 
    first_set.remove(10)
KeyError: 10
```



Usando la función discard()

Elimina el elemento dado del conjunto. Si un elemento no existe, no generará un error. En lugar de remove() , podemos usar esto para evitar un error.

Parámetro:

Elemento : Toma un solo parámetro requerido, indica el elemento que queremos eliminar.

Valor de retorno:

Elimina el elemento dado del conjunto y actualiza el conjunto. Devuelve None (nada).

Sintaxis:

```
Set.discard(elemento)
```

Ejemplo:

```
#Creando set
fruit_Set = {'manzana','kiwi',5,6,7,'higo'}

#eliminando elemento
fruit_Set.discard(6)

print('Conjunto después de eliminar elemento :',fruit_Set)
Salida: Conjunto después de eliminar elemento : {5, 'kiwi', 7, 'manzana',
'higo'}
```

En el siguiente ejemplo de código, estamos eliminando un elemento que no existe en Set . Como podemos ver, arroja KeyError .

```
#Creando set
fruit_Set = {'manzana','kiwi',5,6,7,'higo'}

#eliminando elemento
fruit_Set.discard(9)

print('Conjunto después de eliminar elemento :',fruit_Set)
```

Salida:



None

Conjunto después de eliminar elemento : {5, 'kiwi', 6, 7, 'manzana', 'higo'}

Usando la función pop()

Elimina un elemento arbitrario (aleatorio) del conjunto y devuelve el elemento eliminado.

Parámetro:

El método pop no acepta ningún parámetro.

Valor de retorno:

Devolvió un elemento eliminado arbitrariamente del conjunto y actualizó el conjunto. El elemento ya no existe en SET.

Nota: si el conjunto está vacío, genera una excepción `TypeError`.

Sintaxis:

```
set.pop()
```

Ejemplo:

Nota : la salida puede ser diferente porque Pop() elimina un elemento arbitrario (aleatorio).

```
#Creando set
fruit_Set = {'manzana', 'kiwi', 5, 6, 7, 'higo'}

#eliminando elemento con pop()

print('Elemento eliminado: ', fruit_Set.pop())

print('Conjunto después de eliminar elemento :', fruit_Set)
```

Salida:

Elemento	eliminado:	5
----------	------------	---

Conjunto después de eliminar elemento : {'kiwi', 6, 7, 'manzana', 'higo'}



Usando la función clear()

El método integrado clear() de set elimina todos los elementos del Set.

Parámetro:

El método pop no acepta ningún parámetro.

Valor de retorno:

No devuelve ninguno (nada)

Sintaxis:

```
set.clear()
```

Ejemplo:

```
#Creando set
fruit_Set = {'manzana','kiwi',5,6,7,'higo'}

#eliminando todos los elementos con clear()

print('Valor devuelto con el método clear(): ',fruit_Set.clear())

print('Conjunto después de eliminar elementos :',fruit_Set)
```

Salida:

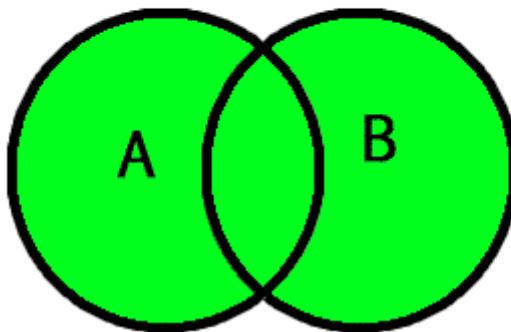
Valor devuelto con el método clear(): 5

Conjunto después de eliminar elementos : set()



UNIENDO CONJUNTOS

$$\begin{aligned} A &= \{1,2,3\} \\ B &= \{3,4,5\} \\ A \cup B &= \{1,2,3,4,5\} \end{aligned}$$



La unión de dos conjuntos dados es el conjunto más pequeño que contiene todos los elementos de ambos conjuntos. La unión de dos conjuntos A y B dados es un conjunto que consta de todos los elementos de A y todos los elementos de B, de manera que ningún elemento se repite.

Usando el método union()

El método Python set union() devuelve un nuevo conjunto con distintos elementos de todos los conjuntos.

Parámetro:

Toma como parámetros los conjuntos que se quieren unir.



Valor de retorno:

Devuelve un nuevo conjunto con elementos del conjunto y todos los demás conjuntos (pasados como argumento).

Sintaxis:

```
A.union(*other_sets)
```

Ejemplo:

```
#Creando sets
A = {'a', 'c', 'd'}
B = {'c', 'd', 2 }
C = {1, 2, 3}

#Uniendo
print('A U B =', A.union(B))
print('B U C =', B.union(C))
print('A U B U C =', A.union(B, C))

print('A.union() =', A.union())
```

Salida:

```
A      U      B      =      {2,          'a',          'd',          'c'}
B      U      C      =      {1,          2,          3,          'd',          'c'}
A      U      B      U      C      =      {1,          2,          3,          'a',          'd',          'c'}
A.union() = {'a', 'd', 'c'}
```

Usando el operador |

Python le proporciona el operador de unión conjunto | que te permite unir dos conjuntos.

Sintaxis:

```
new_set = set1 | set2
```

Ejemplo:

```
#Creando sets
s1 = {'Python', 'Java'}
```



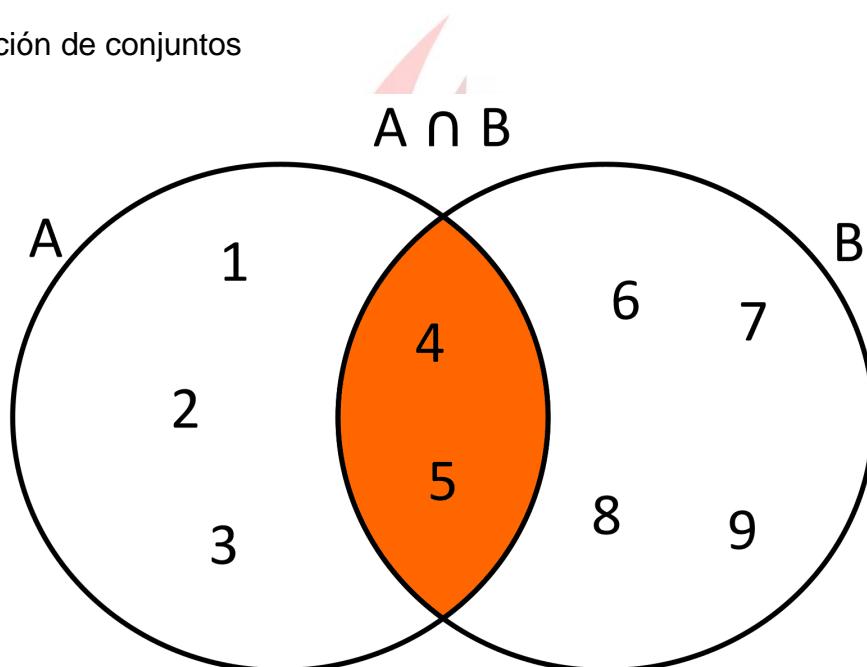
```
s2 = {'C#', 'Java'}  
  
#uniendo sets  
s = s1 | s2  
  
print(s)  
Salida: {'Java', 'C#', 'Python'}
```





INTERSECCIÓN DE CONJUNTOS

Intersección de conjuntos



En teoría de conjuntos, la intersección de dos (o más) conjuntos es una operación que resulta en otro conjunto que contiene los elementos comunes a los conjuntos de partida.

Usando el método intersection()

El `intersection()` método devuelve un nuevo conjunto con elementos que son comunes a todos los conjuntos.

Parámetro:

Permite un número arbitrario de argumentos (conjuntos).



Valor de retorno:

- Devuelve la intersección de conjunto A con todos los conjuntos (pasado como argumento).
- Si el argumento no se pasa a intersection(), devuelve una copia superficial del conjunto (A).

Sintaxis:



```
A.intersection(*otros_conjuntos)
```

Ejemplo:

```
● ● ●
```

```
#Creando sets
A = {2, 3, 5, 4}
B = {2, 5, 100}
C = {2, 3, 8, 9, 10}
```

```
#interceptando sets
print(B.intersection(A))
print(B.intersection(C))
print(A.intersection(C))

print(C.intersection(A, B))
```



Salida:

```
{2, 5}  
{2}  
{2, 3}  
{2}
```

Usando el operador &

Python le proporciona el operador de intersección de conjuntos (&) que le permite interceptar dos o más conjuntos.

Sintaxis:



```
new_set = s1 & s2 & s3 & ...
```

Ejemplo:

```
● ● ●  
  
#Creando sets  
A = {100, 7, 8}  
B = {200, 4, 5}  
C = {300, 2, 3, 7}  
D = {100, 200, 300}  
  
#interceptando sets  
print(A & C)  
print(A & D)  
  
print(A & C & D)  
print(A & B & C & D)
```



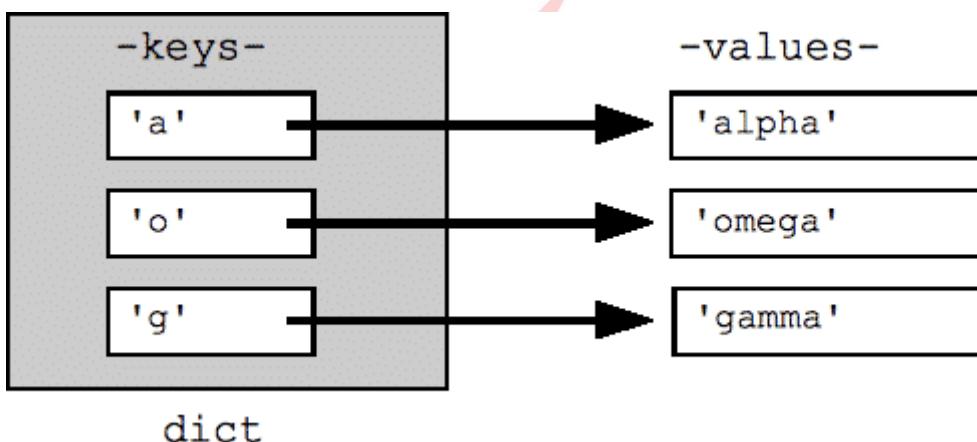
Salida:

```
{7}  
{100}  
set()  
set()
```





DICCIONARIOS



Los diccionarios son la implementación de Python de una estructura de datos que se conoce más generalmente como una matriz asociativa. Un diccionario consta de una colección de pares clave-valor. Cada par clave-valor asigna la clave a su valor asociado.

Propiedades

Algunas propiedades de los diccionario en Python son las siguientes:

- Son dinámicos, pueden crecer o decrecer, se pueden añadir o eliminar elementos.
- Son indexados, los elementos del diccionario son accesibles a través del key.
- Y son anidados, un diccionario puede contener a otro diccionario en su campo value.



Diccionarios y Listas

Similitudes

Los diccionarios y listas comparten las siguientes características:

- Ambos son mutables.
- Ambos son dinámicos. Pueden crecer y encogerse según sea necesario.
- Ambos se pueden anidar. Una lista puede contener otra lista. Un diccionario puede contener otro diccionario. Un diccionario también puede contener una lista y viceversa.

Diferencias

Los diccionarios se diferencian de las listas principalmente en cómo se accede a los elementos:

- Se accede a los elementos de la lista por su posición en la lista, a través de la indexación.
- Se accede a los elementos del diccionario mediante teclas.

Creación de un diccionario

Puede definir un diccionario encerrando una lista separada por comas de pares clave-valor entre llaves ({}). Dos puntos (:) separan cada clave de su valor asociado:

Sintaxis:



```
d = {  
    <key>: <value>,  
    <key>: <value>,  
    .  
    .  
    .  
    <key>: <value>  
}
```



Ejemplo #1:

```
colores = {  
    'uno' : 'blanco',  
    'dos' : 'rojo',  
    'tres': 'verde',  
    'cuatro': 'azul',  
    'cinco' : 'amarillo'  
}
```

También puede construir un diccionario con la dict() función integrada. El argumento dict() debe ser una secuencia de pares clave-valor. Una lista de tuplas funciona bien para esto:

```
d = dict([  
    (<key>, <value>),  
    (<key>, <value>),  
    .  
    .  
    .  
    (<key>, <value>)  
])
```



Ejemplo #2:

```
colores = dict( [  
    ('uno' , 'blanco'),  
    ('dos' , 'rojo'),  
    ('tres', 'verde'),  
    ('cuatro', 'azul'),  
    ('cinco' , 'amarillo')  
])
```

Si los valores clave son cadenas simples, se pueden especificar como argumentos de palabras clave. Así que aquí hay otra forma de definir:

Ejemplo #3:

```
colores = dict(  
    uno = 'blanco',  
    dos = 'rojo',  
    tres = 'verde',  
    cuatro = 'azul',  
    cinco = 'amarillo'  
)
```



Mostrar diccionario

Una vez que haya definido un diccionario, puede mostrar su contenido, lo mismo que puede hacer para una lista. Las tres definiciones que se muestran arriba aparecen de la siguiente manera cuando se muestran:

Ejemplo:



```
#Creamos diccionario
colores = dict( [
    ('uno' , 'blanco'),
    ('dos' , 'rojo'),
    ('tres', 'verde'),
    ('cuatro', 'azul'),
    ('cinco' , 'amarillo')
] )

print(colores)

#recorremos el diccionario
for key in colores:
    print key, ":", colores[key]
```



Salida:

```
{'uno': 'blanco', 'dos': 'rojo', 'tres': 'verde', 'cuatro': 'azul', 'cinco': 'amarillo'}  
{'uno': 'blanco'}  
{'dos': 'rojo'}  
{'tres': 'verde'}  
{'cuatro': 'azul'}  
{'cinco': 'amarillo'}
```





DICCIONARIOS

MÉTODOS BÁSICOS

Python proporciona varias funciones integradas para manejar diccionarios.

- `get()`: esta función devuelve el valor de la clave dada
- `keys()`: esta función devuelve un objeto de vista que muestra una lista de todas las claves en el diccionario en orden de inserción
- `values()`: esta función devuelve una lista de todos los valores disponibles en un diccionario dado
- `items()`: esta función devuelve la lista con todas las claves del diccionario con valores

Método `get()`

El método `get()` de Python devuelve el valor de la clave dada si está presente en el diccionario. De lo contrario, devolverá Ninguno (si se usa `get()` con un solo argumento).

Parámetros:

- Clave: el nombre clave del elemento del que desea devolver el valor
- Valor: (Opcional) Valor que se devolverá si no se encuentra la clave. El valor predeterminado es `None`.

Devoluciones: Devuelve el valor del elemento con la clave especificada.



Sintaxis:



```
Dict.get(clave, default=None)
```

Ejemplo:



```
#Creando diccionario
dic = {"A": 1, "B": 2}

#buscando por clave
print(dic.get("A"))
print(dic.get("C"))
print(dic.get("C", "No encontrado !"))
```

Salida:

None
No encontrado !

Método keys()

El método keys() en Python Dictionary, devuelve un objeto de vista que muestra una lista de todas las claves en el diccionario en orden de inserción.

Parámetros:

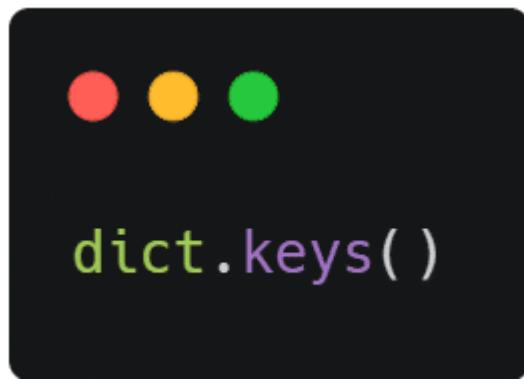
- clave: el nombre clave del elemento del que desea devolver el valor



- **Valor:** (Opcional) Valor que se devolverá si no se encuentra la clave. El valor predeterminado es None.

Devoluciones: Devuelve el valor del elemento con la clave especificada.

Sintaxis:



Ejemplo:

```
● ● ●

# Creando diccionario
Dictionary1 = {'A': 'uno', 'B': 'dos', 'C': 'tres'}

print(Dictionary1.keys())

# Creando diccionario vacío
empty_Dict1 = {}

print(empty_Dict1.keys())
```

Salida:

```
dict_keys(['A', 'B', 'C'])
dict_keys([])
```



Método values()

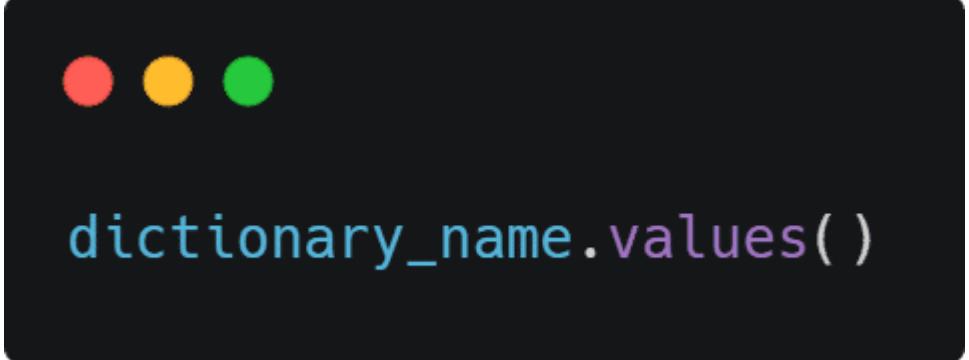
Es un método incorporado en el lenguaje de programación Python que devuelve un objeto de vista. El objeto de vista contiene los valores del diccionario, como una lista. Si usa el método type () en el valor de retorno, obtiene "objeto dict_values". Debe emitirse para obtener la lista real.

Parámetros: No hay parámetros.

Devoluciones: Devuelve una lista de todos los valores disponibles en un diccionario dado.

Los valores se han almacenado de forma inversa .

Sintaxis:



```
● ● ●  
dictionary_name.values( )
```

Ejemplo:



```
● ● ●  
  
#Creando diccionario  
dictionary = {"uno": 2, "dos": 3, "tres": 4}  
print(dictionary.values())  
  
# Valores strings  
dictionary = {"curso": "5", "de": "3", "scala": "5"}  
print(dictionary.values())
```



Salida:

```
dict_values([2, 3, 4])  
dict_values(['5', '3', '5'])  
Método items()
```

El método items() se usa para devolver la lista con todas las claves del diccionario con valores.

Parámetros: No hay parámetros.

Devoluciones: Un objeto de vista que muestra una lista del par de tuplas (clave, valor) de un diccionario dado.

Sintaxis:



```
dictionary.items( )
```

Ejemplo:



```
# Creando diccionario  
Dictionary1 = { 'A': 'Geeks', 'B': 4, 'C': 'Geeks' }  
  
print("Elementos del diccionario:")  
print(Dictionary1.items())
```



Salida:

Elementos del diccionario:

```
dict_items([('A', 'Geeks'), ('B', 4), ('C', 'Geeks')])
```





DICCIONARIOS

ACTUALIZANDO DATOS

El método `update()` actualiza el diccionario con los elementos de otro objeto de diccionario o de una iteración de pares clave/valor.

Parámetros:

- El método `update()` toma un diccionario o un objeto iterable de pares clave/valor (generalmente tuplas).
- Si `update()` se llama sin pasar parámetros, el diccionario permanece sin cambios.

Valor de retorno:

- El método actualiza el diccionario con elementos de un objeto de diccionario o un objeto iterable de pares clave/valor.
- No devuelve ningún valor (returns None).

Sintaxis:



```
dict.update( [other] )
```



Ejemplo #1:

```
● ● ●

#Creando diccionarios
d = {1: "uno", 2: "tres"}
d1 = {2: "dos"}

# actualizando el valor de la clave 2
d.update(d1)

print(d)

d1 = {3: "tres"}

# agregando elementos con clave 3
d.update(d1)

print(d)
```

Salida:

```
{1: 'uno', 2: 'dos'}
{1: 'uno', 2: 'dos', 3: 'tres'}
```



Ejemplo #2:

```
● ● ●

# Creando diccionario
Dictionary1 = {'A': 'uno'}

# Antes de la actualización
print("Diccionario original:")
print(Dictionary1)

# Actualizando con iterable
Dictionary1.update(B='dos', C='tres')
print("Diccionario después de actualizar:")
print(Dictionary1)
```

Salida:

Diccionario original:
{'A': 'uno'}

Diccionario después de actualizar:
{'C': 'uno', 'B': 'dos', 'A': 'tres'}



Ejemplo #3:



```
d = {'x': 2}
```

```
d.update(y = 3, z = 0)
```

```
print(d)
```

Salida: {'x': 2, 'y': 3, 'z': 0}



DICCIONARIOS

AÑADIENDO NUEVOS DATOS

Para agregar un elemento a un diccionario de Python, debe asignar un valor a una nueva clave de índice en su diccionario.

A diferencia de las listas y las tuplas , no hay ningún método insert(), append() que pueda usar para agregar elementos a su estructura de datos. En su lugar, debe crear una nueva clave de índice, que luego se utilizará para almacenar el valor que desea almacenar en su diccionario.

Sintaxis:



```
dictionary_name[clave] = valor
```



Ejemplo:

```
● ● ●

comida= {
    "Fruta" : 22,
    "Simple" : 14,
    "Canela" : 4,
    "Queso" : 21
}

comida[ "Cereza" ] = 10

print (comida)
```

Salida:

```
{'Fruta': 22, 'Simple': 14, 'Canela': 4, 'Queso': 21, 'Cereza': 10}
```

En nuestro código, primero declaramos un diccionario llamado "comida". A continuación, agregamos la clave "Cereza" a nuestro diccionario y le asignamos el valor 10 usando el código para agregar.

Método update(), usando la misma notación

La misma notación se puede usar para actualizar un valor en un diccionario. Podríamos actualizar nuestro diccionario para reflejar esto usando la siguiente línea de código:



Ejemplo:

```
● ● ●  
d = {1: "one", 2: "two"}  
  
d1 = {3: "three"}  
  
d.update(d1)  
  
print(d)
```





DICCIONARIOS

ELIMINAR DATOS

Método del

La palabra `del` clave se puede usar para eliminar en su lugar la clave que está presente en el diccionario. Un inconveniente que se puede pensar al usar esto es que genera una excepción si no se encuentra la clave y, por lo tanto, se debe manejar la inexistencia de la clave.

Sintaxis:



```
del nombre_diccionario[clave]
```

Ejemplo:

```
● ● ●

# Creando diccionario
test_dict = {"rojo" : 22, "amarillo" : 21, "azul" : 21, "naranja" : 21}

# mostrando antes de eliminar elemento
print ("El diccionario antes de realizar la eliminación es : " + str(test_dict))

# eliminando elemento
del test_dict['azul']

# mostrando después de eliminar elemento
print ("El diccionario después de eliminar es : " + str(test_dict))
```



Salida:

El diccionario antes de realizar la eliminación es: {'amarillo': 21, 'naranja': 21, 'rojo': 22, 'azul': 21}

El diccionario después de eliminar es: {'amarillo': 21, 'naranja': 21, 'rojo': 22}

Método popitem()

El método `popitem()` se puede usar para eliminar una clave y su valor en su lugar. La ventaja sobre el uso del `del` es que proporciona el mecanismo para imprimir el valor deseado si se intenta eliminar un dict que no existe. En segundo lugar, también devuelve el valor de la clave que se está eliminando además de realizar una operación de eliminación simple.

Sintaxis:

```
● ● ●  
nombre_diccionario.pop(clave,valor)
```

Ejemplo:

```
● ● ●

# Creando diccionario
test_dict = {"rojo" : 22, "amarillo" : 21, "azul" : 21, "naranja" : 21}

# mostrando antes de la eliminación
print ("El diccionario antes de realizar la eliminación es : " + str(test_dict))

# eliminando valor por clave
removed_value = test_dict.pop('azul')

# mostrando después de la eliminación
print ("El diccionario después de eliminar es : " + str(test_dict))
print ("El valor de la clave eliminada es : " + str(removed_value))

print ('\r')

# Usando pop() para eliminar
# no genera una excepción
# asigna 'No se encontró ninguna clave' a valor_eliminado
removed_value = test_dict.pop('verde', 'No se encontró ninguna clave')

# mostrando después de la eliminación
print ("El diccionario después de eliminar es: " + str(test_dict))
print ("El valor de la clave eliminada es : " + str(removed_value))
```

Salida:

El diccionario antes de realizar la eliminación es: {'rojo': 22, 'amarillo': 21, 'azul': 21, 'naranja': 21}

El diccionario después de eliminar es: {'rojo': 22, 'amarillo': 21, 'naranja': 21}

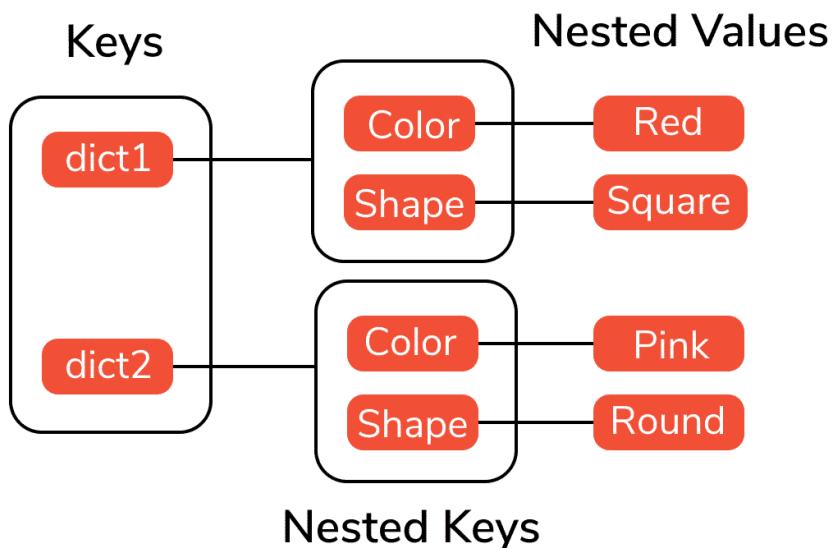
El valor de la clave eliminada es: 21

El diccionario después de eliminar es: {'rojo': 22, 'amarillo': 21, 'naranja': 21}

El valor de la clave eliminada es: No se encontró ninguna clave



DICCIONARIOS ANIDADOS



En Python, un diccionario anidado es un diccionario dentro de otro diccionario. Es una colección de diccionarios en un solo diccionario.

Puntos clave para recordar:

1. Diccionario anidado es una colección desordenada de diccionario
2. No es posible dividir el diccionario anidado.
3. Podemos reducir o aumentar el diccionario anidado según sea necesario.
4. Al igual que Dictionarios, también tiene clave y valor.
5. Se accede al diccionario usando la tecla



Sintaxis:



```
anidado_dict = { 'dictA': {'clave_1': 'valor_2'},  
                 'dictB': {'clave_2': 'valor_2'}}
```

Crear un diccionario anidado

Vamos a crear un diccionario anidado ,personas. El diccionario interno 1 y 2 está asignado a personas. Aquí, ambos diccionarios tienen clave nombre, edad, sexo con diferentes valores. Ahora imprimimos el resultado de personas.

Ejemplo:



```
personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},  
           2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'}}  
  
print(personas)
```

Salida:

```
{1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'}, 2: {'nombre': 'Marie',  
'edad': '22', 'sexo ': 'Femenino'}}
```

Acceder a elementos de un diccionario anidado

Para acceder a elementos de un diccionario anidado, usamos la sintaxis [] de indexación en Python.



```
personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},  
           2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'}}  
  
print(personas[1]['nombre'])  
print(personas[1]['edad'])  
print(personas[1]['sexo'])
```



Salida:

John
27
Masculino

Agregar elemento a un diccionario anidado

Para agregar un elemento a un diccionario anidado, podemos usar la misma sintaxis que usamos para modificar un elemento. No hay ningún método que deba usar para agregar un elemento a un diccionario anidado. Necesitas usar el operador de asignación.

Sintaxis:



```
dictionary_name[nueva_clave] = nuevo_valor
```

Ejemplo:



```
personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},  
          2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'}}  
  
personas[3] = {}  
  
personas[3]['nombre'] = 'Luna'  
personas[3]['edad'] = '24'  
personas[3]['sexo'] = 'Femenino'  
personas[3]['casado'] = 'No'  
  
print(personas[3])
```

**Salida:**

```
{'nombre': 'Luna', 'edad': '24', 'sexo': 'Femenino', 'casado': 'No'}
```

En el programa anterior, creamos un diccionario vacío 3 dentro del diccionario personas. Luego, agregamos el key:value par, es decir , persona[3]['nombre'] = 'Luna' dentro del diccionario 3. Del mismo modo, hacemos esto para key edad, sexo y casado uno por uno. Cuando imprimimos el personas[3], obtenemos key:value pares de diccionario 3.

Eliminar elementos de un diccionario anidado

En Python, usamos la declaración "del" para eliminar elementos del diccionario anidado.

Ejemplo:

```
personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},
            2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'},
            3: {'nombre': 'Luna', 'edad': '24', 'sexo': 'Femenino', 'casado': 'No'},
            4: {'nombre': 'Peter', 'edad': '29', 'sexo': 'Masculino', 'casado': 'Si'}}

del personas[3]['casado']
del personas[4]['casado']

print(personas[3])
print(personas[4])
```

Salida:

```
{'nombre': 'Luna', 'edad': '24', 'sexo': 'Femenino'}
{'nombre': 'Peter', 'edad': '29', 'sexo': 'Masculino'}
```



Eliminar un diccionario de un diccionario anidado

```
● ● ●

personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},
           2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'},
           3: {'nombre': 'Luna', 'edad': '24', 'sexo': 'Femenino'},
           4: {'nombre': 'Peter', 'edad': '29', 'sexo': 'Masculino'}}

del personas[3], personas[4]

print(personas)
```

Salida:

{1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'}, 2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'}}

Iterando a través de un diccionario anidado

Usando los bucles for, podemos iterar a través de cada elemento en un diccionario anidado.

Ejemplo:

```
● ● ●

personas= {1: {'nombre': 'John', 'edad': '27', 'sexo': 'Masculino'},
           2: {'nombre': 'Marie', 'edad': '22', 'sexo': 'Femenino'}}

for p_id, p_info in personas.items():
    print("\nPersona ID:", p_id)

    for key in p_info:
        print(key + ':', p_info[key])
```

Salida:

Persona ID: 1
Nombre: John
Edad: 27
Sexo: Masculino



Persona ID: 2

Nombre: Marie

Edad: 22

Sexo: Femenino

En el programa anterior, el primer ciclo devuelve todas las claves en el diccionario anidado personas. Consiste en las ID p_id de cada persona. Usamos estas identificaciones para descomprimir la información p_info de cada persona.

El segundo bucle pasa por la información de cada persona. Luego, devuelve todas las claves nombre, edad, sexo del diccionario de cada persona.

Ahora, imprimimos la clave de la información de la persona y el valor de esa clave.



DICCIONARIOS

EJERCICIOS

Ejercicio #1

Escribir un programa que guarde en una variable el diccionario {'Euro':'€', 'Dollar':'\$', 'Yen':'¥'}, pregunte al usuario por una divisa y muestre su símbolo o un mensaje de aviso si la divisa no está en el diccionario.

```
● ● ●  
divisas = {'euro':'€', 'dollar':'$', 'yen':'¥'}  
  
consulta = input("Ingrese la divisa: ")  
  
respuesta = divisas.get(consulta.lower(),"La divisa no se encuentra registrada.")  
  
print(respuesta)
```

Ejercicio #2

Escribir un programa que pregunte al usuario su nombre, edad, dirección y teléfono y lo guarde en un diccionario. Después debe mostrar por pantalla el mensaje < nombre > tiene < edad > años, vive en <dirección> y su número de teléfono es <teléfono>.



```
● ● ●

nombre = input("Ingresa tu nombre: ")
edad = input("Ingresa tu edad: ")
direccion = input("Ingresa tu direccion: ")
telefono = input("Ingresa tu telefono: ")

datos_personales = {"nombre": nombre, "edad": edad, "direccion": direccion, "telefono": telefono}

print(f" {nombre} tiene {edad} años, vive en {direccion} y su número de teléfono es {telefono}")
```

Ejercicio #3

Escribir un programa que guarde en un diccionario los precios de las frutas de la tabla, pregunte al usuario por una fruta, un número de kilos y muestre por pantalla el precio de ese número de kilos de fruta. Si la fruta no está en el diccionario debe mostrar un mensaje informando de ello. Tras cada consulta el programa nos preguntará si queremos hacer otra consulta.

Fruta Precio

Plátano 1.35

Manzana 0.80

Pera 0.85

Naranja 0.70



```
frutas = {"Plátano":1.35,"Manzana":0.80,"Pera":0.85,"Naranja":0.70}

f_u = input("Que fruta desea comprar: ")
c_u = input("Que cantidad desea: ")

while True:
    f = frutas.get(f_u)
    if f == None:
        print(f"No contamos con {f_u}")
    else:
        p_t = f * int(c_u)
        print(f"El precio total por {c_u} {f_u} es {p_t}.")
    consulta = input("Desea consultar nuevamente s/n: ")

    if consulta == "s":
        f_u = input("Que fruta desea comprar: ")
        c_u = input("Que cantidad desea: ")
    else:
        print("Adios!!")
        break
```

Ejercicio #4

Escribir un programa que pregunte una fecha en formato dd/mm/aaaa y muestre por pantalla la misma fecha en formato dd de < mes > de aaaa donde < mes > es el nombre del mes. Para obtener el nombre del mes debe tener un diccionario con los datos.



```
meses = {  
    1:"Enero",  
    2:"Febrero",  
    3:"Marzo",  
    4:"Abril",  
    5:"Mayo",  
    6:"Junio",  
    7:"Julio",  
    8:"Agosto",  
    9:"Septiembre",  
    10:"Octubre",  
    11:"Noviembre",  
    12:"Diciembre"  
}  
  
fecha = input("Ingresa una fecha en formato dd/mm/aaaa: ")  
d,m,a = fecha.split("/")  
mes = meses.get(int(m))  
  
print(f"{d} de {mes} de {a}.")s!!"  
break
```

Ejercicio #5

Escribir una función que reciba una cadena y devuelva un diccionario con la cantidad de apariciones de cada palabra en la cadena. Por ejemplo, si recibe "Qué lindo día que hace hoy" debe devolver: 'que': 2, 'lindo': 1, 'día': 1, 'hace': 1, 'hoy': 1



```
● ● ●

c_a = int( input("Ingresa la cantidad de alumnos a guardar: "))
alumnos = {}
c = 0
while c < c_a:
    nombre = input("Ingresa el nombre: ")
    lista_notas = []
    nota = float(input("Ingresa la nota: "))
    while nota >= 0:
        lista_notas.append(nota)
        nota = float(input("Ingresa la nota: "))
    if alumnos.get(nombre) != None:
        print("Error,el alumno ya se encuentra registrado..")
    else:
        alumnos[nombre] = lista_notas
        c += 1

for k,v in alumnos.items():
    c_n = len(v)
    s = 0
    for n in v:
        s += n
    media = s / c_n
    print(f"{k}= {media}")
```

Ejercicio #6

Codifica un programa en python que nos permita guardar los nombres de los alumnos de una clase y las notas que han obtenido. Cada alumno puede tener distinta cantidad de notas. Guarda la información en un diccionario cuya claves serán los nombres de los alumnos y los valores serán listas con las notas de cada alumno.

El programa pedirá el número de alumnos que vamos a introducir, pedirá su nombre e irá pidiendo sus notas hasta que introduzcamos un número negativo. Al final el programa nos mostrará la lista de alumnos y la nota media obtenida por cada uno de ellos. Nota: si se introduce el nombre de un alumno que ya existe el programa nos dará un error.



```
● ● ●

def normalizar(c):
    remplazos = (("á","a"),("é","e"),("í","i"),("ó","o"),("ú","u"))
    for a,b in remplazos:
        c = c.replace(a,b)
    return c

def contar_apariciones(frase):
    frase = frase.lower()
    frase = normalizar(frase)
    frase_separada = frase.split(" ")
    r = {}
    for p in frase_separada:
        c_a = frase.count(p)
        r[p] = c_a
    return r

print(contar_apariciones("Qué lindo día que hace hoy"))
```

Ejercicio #7

Escribir una función que cuente la cantidad de apariciones de cada carácter en una cadena de texto, y los devuelva en un diccionario.

```
● ● ●

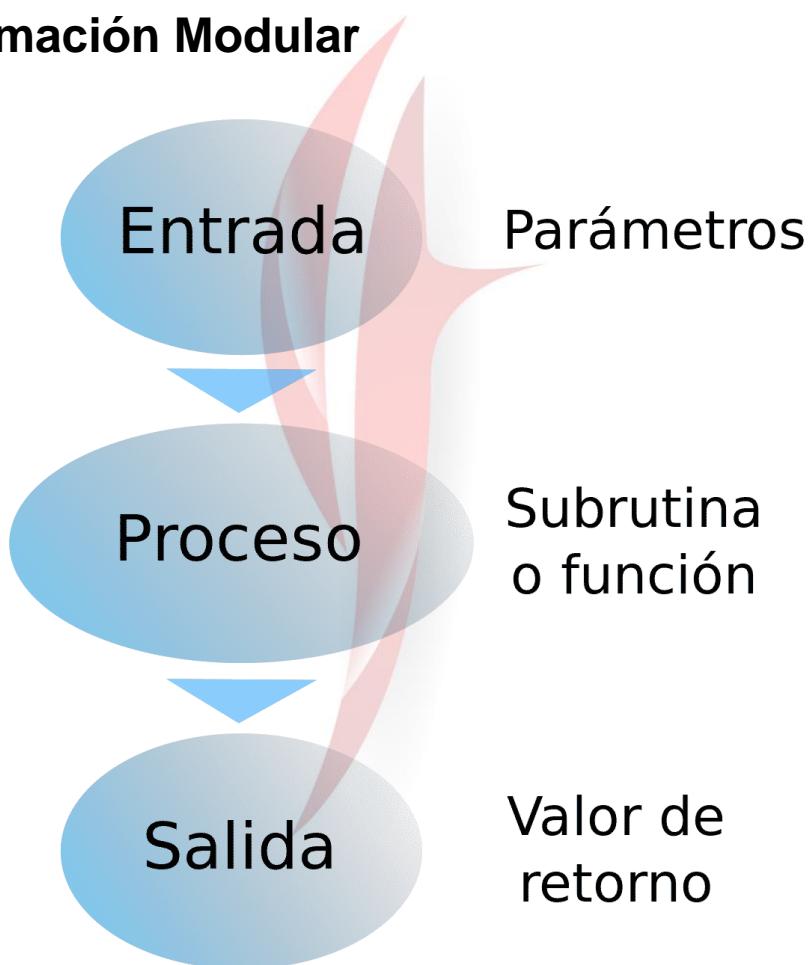
def contar_caracter(frase):
    r = {}
    for c in frase:
        c_a = frase.count(c)
        r[c] = c_a
    return r

print(contar_caracter("Escribir una función que cuente "))
```



MÓDULOS

Programación Modular



La programación modular se refiere al proceso de dividir una tarea de programación grande y difícil de manejar en subtareas o módulos separados, más pequeños y más manejables. Luego, los módulos individuales se pueden improvisar como bloques de construcción para crear una aplicación más grande.



Ventajas de usar una programación modular:

Hay varias ventajas de modularizar el código en una aplicación grande:

- **Simplicidad:** En lugar de enfocarse en todo el problema en cuestión, un módulo generalmente se enfoca en una porción relativamente pequeña del problema. Si está trabajando en un solo módulo, tendrá un dominio de problema más pequeño para entender. Esto hace que el desarrollo sea más fácil y menos propenso a errores.
- **Capacidad de mantenimiento:** Los módulos suelen diseñarse de manera que imponen límites lógicos entre diferentes dominios de problemas. Si los módulos se escriben de manera que se minimice la interdependencia, hay menos probabilidad de que las modificaciones a un solo módulo tengan un impacto en otras partes del programa. (Es posible que incluso pueda realizar cambios en un módulo sin tener ningún conocimiento de la aplicación fuera de ese módulo). Esto hace que sea más viable para un equipo de muchos programadores trabajar en colaboración en una aplicación grande.
- **Reutilización:** La funcionalidad definida en un solo módulo puede reutilizarse fácilmente (a través de una interfaz definida apropiadamente) por otras partes de la aplicación. Esto elimina la necesidad de duplicar el código.
- **Alcance:** Los módulos suelen definir un espacio de nombres separado, lo que ayuda a evitar colisiones entre identificadores en diferentes áreas de un programa.

Módulos de Python

Los módulos hacen referencia a un archivo que contiene sentencias y definiciones de Python.

Un archivo que contiene código de Python, por ejemplo: example.py, se denomina módulo y su nombre de módulo sería example.

Usamos módulos para dividir programas grandes en pequeños archivos manejables y organizados. Además, los módulos proporcionan la reutilización del código.

Podemos definir nuestras funciones más utilizadas en un módulo e importarlo, en lugar de copiar sus definiciones en diferentes programas.



Definir módulos

En realidad, hay tres formas diferentes de definir un módulo en Python:

- Un módulo se puede escribir en Python mismo.
- Un módulo puede escribirse en C y cargarse dinámicamente en tiempo de ejecución, como el módulo re(expresión regular).
- Un módulo incorporado está intrínsecamente contenido en el intérprete, como el módulo itertools.

Se accede al contenido de un módulo de la misma manera en los tres casos: con la declaración import.

Ejemplo:

Vamos a crear un módulo. Escriba lo siguiente y guárdelo como example.py.

```
● ● ●

# Python Módulo ejemplo

def add(a, b):
    """Este programa suma
       dos números y devuelve el resultado"""

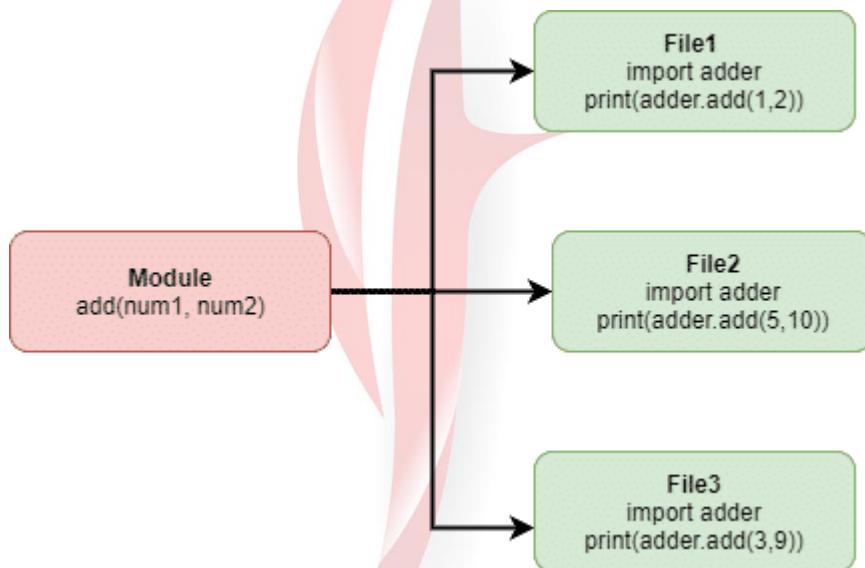
    result = a + b
    return result
```

Aquí, hemos definido una función add() dentro de un módulo llamado example. La función toma dos números y devuelve su suma.



IMPORTAR MÓDULO

Importando módulos



Un archivo se considera como un módulo en python. Para usar el módulo, debe importarlo usando la palabra clave de importación. La función o las variables presentes dentro del archivo se pueden usar en otro archivo importando el módulo.

Podemos importar las definiciones dentro de un módulo a otro módulo o al intérprete interactivo en Python.

Sentencia import

En Python, la sentencia `import` se utiliza para importar módulos o elementos específicos (como funciones o clases) desde módulos externos. Esta es una



parte fundamental del lenguaje que permite a los programadores acceder a la funcionalidad proporcionada por otros módulos o paquetes.

Hay varias formas de utilizar la sentencia import:

Importar un módulo completo:

import modulo

Esto importará todo el módulo modulo. Para acceder a los elementos dentro de este módulo, debes utilizar la notación de punto, por ejemplo: modulo.funcion() o modulo.clase.

Importar un módulo con un alias:

import modulo as mi_alias

Esto importará el módulo modulo, pero le asignará un alias mi_alias. Esto puede ser útil si el nombre del módulo es largo o propenso a conflictos de nombres.

Importar elementos específicos de un módulo:

from modulo import funcion, clase

Esto importará solo las funciones o clases específicas (funcion y clase) desde el módulo modulo. No necesitas usar la notación de punto para acceder a estos elementos después de la importación.

Importar todos los elementos de un módulo:

*from modulo import **

Esto importará todos los elementos del módulo modulo directamente en el espacio de nombres actual. Sin embargo, este enfoque no es recomendado, ya que puede causar conflictos de nombres y hacer que sea más difícil rastrear de dónde vienen las funciones y clases.

Importar un módulo dentro de un paquete:

from paquete import modulo

Esto importará un módulo modulo desde un paquete paquete. Puedes seguir utilizando la notación de punto para acceder a los elementos dentro del módulo importado.



Namespaces

Para acceder (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el namespace, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo:



```
print(modulo.CONSTANTE_1)
print(paquete.modulo1.CONSTANTE_1)
print(paquete.subpaquete.modulo1.CONSTANTE_1)
```

Alias

Es posible también, abbreviar los namespaces mediante un alias. Para ello, durante la importación, se asigna la palabra clave as seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:



```
import modulo as m
import paquete.modulo1 as pm
import paquete.subpaquete.modulo1 as psm
```

Luego, para acceder a cualquier elemento de los módulos importados, el namespace utilizado será el alias indicado durante la importación:



```
print(m.CONSTANTE _1)
print(pm.CONSTANTE _1)
print(psm.CONSTANTE_1)
```

Ejemplo:

example.py



```
# Python Módulo ejemplo

def add(a, b):
    """Este programa suma
       dos números y devuelve el resultado"""

    result = a + b
    return result
```

main.py



```
import example

example.add(5,2)
```



MAIN - NAME

Main - name

Antes de ejecutar el código, el intérprete de Python lee el archivo fuente y define algunas variables especiales/variables globales.

Si el intérprete de python está ejecutando ese módulo (el archivo fuente) como el programa principal, establece la variable especial `__name__` para que tenga un valor “`__main__`” . Si este archivo se importa desde otro módulo, `__name__` se establecerá en el nombre del módulo. El nombre del módulo está disponible como valor para la variable global `__name__`.

Un módulo es un archivo que contiene definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` adjunto.

Cuando ejecutamos el archivo como comando para el intérprete de python:





Ejemplo:

```
● ● ●

print ("Siempre ejecutado")

if __name__ == "__main__":
    print ("Se ejecuta cuando se invoca directamente")
else:
    print ("Ejecutado cuando se importa")
```

- Todo el código que está en el nivel de sangría 0 [Bloque 1] se ejecuta. Las funciones y clases que están definidas están definidas, pero ninguno de sus códigos se ejecuta.
- Aquí, como ejecutamos script.py directamente, la variable `__name__` será `__main__`. Entonces, el código en este bloque si [Bloque 2] solo se ejecutará si ese módulo es el punto de entrada a su programa.
- Por lo tanto, puede probar si su script se está ejecutando directamente o si lo está importando otra cosa probando la variable `__name__`.
- Si algún otro módulo está importando el script en ese momento , `__name__` será el nombre del módulo.

¿Por qué lo necesitamos?

Por ejemplo, estamos desarrollando un script que está diseñado para usarse como módulo:

```
● ● ●

# Programa para ejercutar
# directamente una función
def my_function():
    print ("Estoy dentro de la función")

# Podemos probar la función llamándola
my_function()
```



Ahora, si queremos usar ese módulo importando, tenemos que comentar nuestra llamada. En lugar de ese enfoque, el mejor enfoque es usar el siguiente código:

```
# Programa python pra usar
# main para llamada de función
if __name__ == "__main__":
    my_function()

import myscript

myscript.my_function()
```

Ventajas:

- Cada módulo de Python tiene su `__name__` definido y si es '`__main__`', implica que el usuario está ejecutando el módulo de forma independiente y podemos realizar las acciones apropiadas correspondientes.
- Si importa este script como un módulo en otro script, el `__name__` se establece en el nombre del script/módulo.
- Los archivos de Python pueden actuar como módulos reutilizables o como programas independientes.
- `if __name__ == "main":` se usa para ejecutar algún código solo si el archivo se ejecutó directamente y no se importó.



CREANDO MÓDULO

Creando módulo

Como ya sabemos un módulo es un archivo Python, ahora vamos a implementar un módulo que vamos a poder utilizarlo si lo importamos o también si lo ejecutamos como programa principal.

El módulo tendrá el nombre de `convertsystem` donde vamos a implementar cuatro funciones que nos van a permitir convertir entre el sistema binario decimal y quinario.

Las funciones serán las siguientes:

- `decToInt`: Función que permite convertir un número decimal a binario.
- `decToQuin`: Función que permite convertir un número decimal a quinario.
- `binToInt`: Función que permite convertir un número binario a decimal.
- `intToDec`: Función que permite convertir un número quinario a decimal.



Definiendo funciones

```
● ● ●

def dectobin(n):
    binario = ""
    while n > 0:
        b = str(n % 2)
        binario = b + binario
        n //= 2
    return binario

def dectoquin(n):
    quinario = ""
    while n > 0:
        q = str(n % 5)
        quinario = q + quinario
        n //= 5
    return quinario

def bintodec(n):
    n = str(n)
    decimal = 0
    l = len(n)-1
    for b in n:
        decimal += int(b) *(2**l)
        l -= 1
    return decimal

def quintodec(n):
    n = str(n)
    decimal = 0
    l = len(n) - 1
    for q in n:
        decimal += int(q)* (5**l)
        l -= 1
    return decimal
```



Si name es igual a main

Ahora que ya tenemos los métodos necesarios vamos a hacer uso de la variable name para que el módulo pueda ocuparse tanto en ejecución como archivo principal así como módulo importado.

```
# $ python convertsystem.py dtb 12
# > 1100
if __name__ == "__main__":
    if len(sys.argv) == 3:
        if sys.argv[1] == "dtb":
            print(dectobin(int(sys.argv[2])))
        elif sys.argv[1] == "dtq":
            print(dectoquin(int(sys.argv[2])))
        elif sys.argv[1] == "btd":
            print(bintodec(sys.argv[2]))
        elif sys.argv[1] == "qtd":
            print(quintodec(sys.argv[2]))
    else:
        print("Los datos no son correctos.")
```



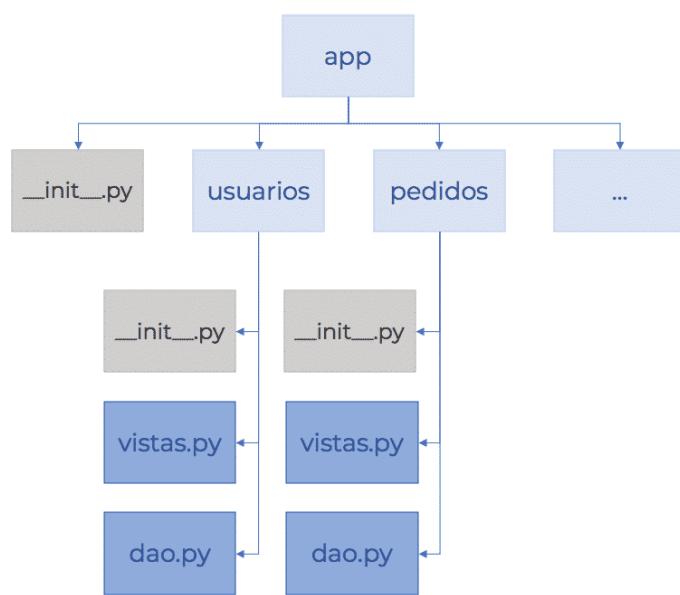
PAQUETES

Del mismo modo en que agrupamos las funciones y demás definiciones en módulos, los paquetes en Python permiten organizar y estructurar de forma jerárquica los diferentes módulos que componen un programa. Además, los paquetes hacen posible que existan varios módulos con el mismo nombre y que no se produzcan errores.

Paquetes

Un paquete es simplemente un directorio que contiene otros paquetes y módulos. En Python 2, para que un directorio sea considerado un paquete, este debía incluir un módulo llamado `__init__.py`, hoy en Python 3 ya no es necesario para que se considere un paquete.

Para que lo veas todo de forma gráfica, aquí tienes los conceptos en una imagen. Imagina que estás haciendo una aplicación para gestionar pedidos. Una forma de organizar los diferentes módulos podría ser la siguiente:





Al igual que sucede con los módulos, cuando se importa un paquete, Python busca a través de los directorios definidos en **sys.path** el directorio perteneciente a dicho paquete.

Importar paquetes

Para importar módulos y definiciones de módulos que están contenidos en paquetes, se usa el operador “.”. Las referencias se hacen indicando el nombre completo del módulo, es decir, especificando los paquetes hasta llegar al módulo en cuestión separándolos con puntos.

Teniendo en cuenta el diagrama de la sección anterior, si en el módulo app_pedidos_vistas se quiere importar el módulo app_usuarios_dao, simplemente hay que añadir la siguiente sentencia:

```
# Módulo app_pedidos_vistas  
  
import app.usuarios.dao
```

El único problema de hacerlo así, es que si, por ejemplo, dicho módulo define una función llamada guardar(), hay que especificar toda la jerarquía para invocar a esta función:

```
app.usuarios.dao.guardar(usuario)
```

Una forma mejor es importar el módulo. Esto se consigue de la siguiente manera:



```
# Módulo app.pedidos.vistas
from app.usuarios import dao
dao.guardar(usuario)
```

Incluso, se puede importar una definición de un módulo del siguiente modo:



```
# Módulo app.pedidos.vistas
from app.usuarios.dao import guardar
guardar(usuario)
```

Imagina ahora que desde el módulo app.pedidos.vistas quieres importar los módulos app.pedidos.dao y app.usuarios.vistas. Se podría hacer como hemos visto hasta ahora:



```
# Módulo app.pedidos.vistas
from app.pedidos import dao
from app.usuarios import vistas
```

O también se podría hacer así:



```
# Módulo app_pedidos_vistas

from . import dao # Un punto referencia al paquete actual
from ..usuarios import vistas # Dos puntos referencian al paquete padre
```





PAQUETE DISTRIBUIBLE

La distribución de código Python, le permite hacer portable de forma amigable usando herramienta de gestión de paquetes Python como la herramienta pip. Esta labor se hace mediante el módulo distutils, y más reciente incorporando el módulo setuptools.

Módulo distutils

El módulo distutils en Python es una biblioteca estándar que proporciona funcionalidades para construir, distribuir e instalar paquetes Python. Es una parte integral del ecosistema de distribución de paquetes en Python y es ampliamente utilizado en conjunto con setuptools para manejar tareas relacionadas con la distribución de paquetes.

Actualmente a partir de la versión 3.12 de Python, distutils no está disponible.

Módulo setuptools

El módulo setuptools en Python es una biblioteca que amplía las funcionalidades proporcionadas por el módulo distutils estándar. Es la herramienta más comúnmente utilizada para construir, distribuir e instalar paquetes Python, y es ampliamente aceptada como el estándar de facto para la distribución de paquetes en el ecosistema de Python.

Para construir un paquete distribuible se debe seguir la siguiente estructura:

```
DIRECTORIO-DEL-PROYECTO
├── LICENSE
├── MANIFEST.in
├── README.txt
└── setup.py
└── NOMBRE-DEL-PAQUETE
    ├── __init__.py
    ├── ARCHIVO1.py
    ├── ARCHIVO2.py
    └── MODULO (OPCIONAL)
        └── __init__.py
            └── MAS_ARCHIVOS.py
```



- **DIRECTORIO-DEL-PROYECTO**

Puede ser cualquiera, no afecta en absoluto, lo que cuenta es lo que hay dentro.

- **NOMBRE-DEL-PAQUETE**

Tiene que ser el nombre del paquete, si el nombre es tostadas_pipo, este directorio tiene que llamarse también tostadas_pipo. Y esto es así. Dentro estarán todos los archivos que forman la librería.

- **LICENSE**

Es el archivo donde se define los términos de licencia usado en su proyecto. Es muy importante que cada paquete cargado a PyPI incluirle una copia de los términos de licencia. Esto le dice a los usuario quien instala el paquete los términos bajos los cuales pueden usarlo en su paquete. Para ayuda a seleccionar una licencia, consulte <https://choosealicense.com/>.

- **MANIFEST.in**

Es el archivo donde se define los criterios de inclusión y exclusión de archivos a su distribución de código fuente de su proyecto. Este archivo incluye la configuración del paquete como se indica a continuación:

```
include LICENSE
include *.txt *.in
include *.py
recursive-include tostadas_pipo *
global-exclude *.pyc *.pyo *~
prune build
prune dist
README.txt
```

Es el archivo donde se define la documentación general del paquete, este archivo es importante debido a que no solo es usado localmente en una copia descargada, sino como información usada el en sitio de PyPI. Entonces abra el archivo README.txt e ingrese el siguiente contenido. Usted puede personalizarlo como quiera:

```
=====
NOMBRE-DEL-PAQUETE
=====
```

Este es un ejemplo simple de un paquete Python.

Usted puede usar para escribir este contenido la guía

`Restructured Text (reST) and Sphinx CheatSheet
<http://openalea.gforge.inria.fr/doc/openalea/doc/_build/html/source/sphinx/rest_syntax.html>`_.



Setup.py

Es el archivo donde se define el paquete, el formato es el mismo para el módulo setuptools y para el módulo distutils. Lo puede ver a continuación. Este archivo incluye la configuración del paquete como se indica a continuación:

```
"""Instalador para el paquete "tostadas_pipo"."""
from setuptools import setup

long_description = (
    open('README.txt').read()
    + '\n' +
    open('LICENSE').read()
    + '\n')

setup(
    name="tostadas_pipo",
    version="0.1",
    description="Sistema Administrativo de Tostadas Pipo C.A.",
    long_description=long_description,
    # Get more https://pypi.org/pypi?%3Aaction=list_classifiers
    classifiers=[
        # ¿Cuan maduro esta este proyecto? Valores comunes son
        # 3 - Alpha
        # 4 - Beta
        # 5 - Production/Stable
        "Development Status :: 3 - Alpha",
        # Indique a quien va dirigido su proyecto
        "Environment :: Console",
        "Intended Audience :: Developers",
        "Topic :: Software Development :: Libraries",
        # Indique licencia usada (debe coincidir con el "license")
        "License :: OSI Approved :: GNU General Public License",
        # Indique versiones soportadas, Python 2, Python 3 o ambos.
        "Programming Language :: Python",
        "Programming Language :: Python :: 2.7",
        "Operating System :: OS Independent",
    ],
    keywords="ejemplo instalador paquete tostadas_pipo",
    author="Leonardo J. Caballero G.",
    author_email="leonardocaballero@gmail.com",
    url="https://twitter/macagua",
    download_url="https://github.com/macagua/tostadas_pipo",
    license="GPL",
    platforms="Unix",
    packages=["tostadas_pipo", "tostadas_pipo/utilidades/"],
    include_package_data=True,
)
```

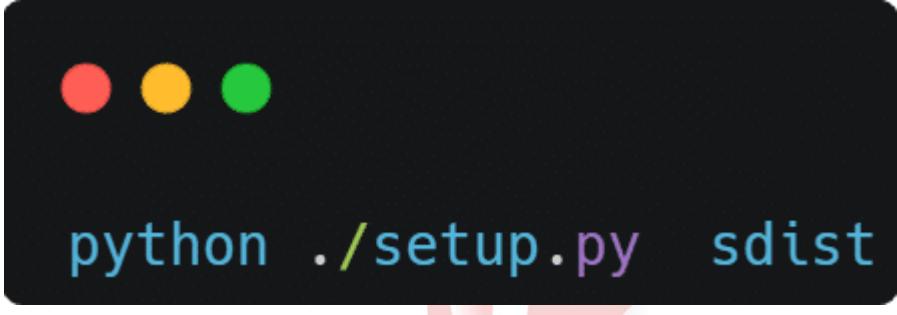


Crear paquetes

Usted puede crear diversos tipos de formatos de instalación y distribución de sus paquetes Python, a continuación se describen los mas usados:

Distribución de código fuente

Tanto el módulo setuptools y distutils le permiten crear una distribución de código fuente o source distribution (sdist) de su paquete en formatos como tarball, archivo zip, etc. Para crear una paquete sdist, ejecute el siguiente comando:



```
● ● ●  
python ./setup.py sdist
```



PIP

El administrador de paquetes estándar para Python es pip. Le permite instalar y administrar paquetes que no forman parte de la biblioteca estándar de Python.

PIP

Pip es un administrador de paquetes para Python. Eso significa que es una herramienta que le permite instalar y administrar bibliotecas y dependencias que no se distribuyen como parte de la biblioteca estándar. El nombre pip fue introducido por Ian Bicking en 2008:

- Terminó de cambiar el nombre de pyinstall a su nuevo nombre: pip. El nombre pip es [un] acrónimo y declaración: pip instala paquetes.

La gestión de paquetes es tan importante que los instaladores de Python lo han incluido pipdesde las versiones 3.4 y 2.7.9, para Python 3 y Python 2, respectivamente. Muchos proyectos de Python usan pip, lo que la convierte en una herramienta esencial para todos los Pythonistas.

El concepto de un administrador de paquetes puede resultarle familiar si proviene de otro lenguaje de programación. JavaScript usa npm para la administración de paquetes, Ruby usa gem y la plataforma .NET usa NuGet . En Python, pipse ha convertido en el administrador de paquetes estándar.

Encontrar pip en tu sistema

Puede verificar que pip esté disponible buscando el ejecutable pip en su sistema. Seleccione su sistema operativo a continuación y use el comando específico de su plataforma en consecuencia.

WINDOWS

C:\> where pip3 o pip

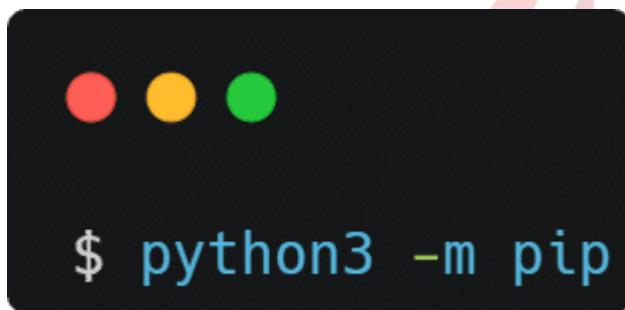


LINUX

\$ which pip3 o pip

Ejecutar pip como módulo

Cuando ejecuta su sistema pip directamente, el comando en sí no revela a qué versión de pip Python pertenece. Desafortunadamente, esto significa que podría usar pip para instalar un paquete en los paquetes del sitio de una versión anterior de Python sin darse cuenta. Para evitar que esto suceda, puede ejecutar pip como un módulo de Python:



Tenga en cuenta que utiliza python3 -m para ejecutar pip. El interruptor -m le dice a Python que ejecute un módulo como un ejecutable del intérprete python3. De esta manera, puede asegurarse de que la versión predeterminada de Python 3 de su sistema ejecute el comando pip.



INSTALACIÓN DE PAQUETES

Instalaciones automáticas desde PyPI

Python se considera un lenguaje con pilas incluidas . Esto significa que la biblioteca estándar de Python contiene un amplio conjunto de paquetes y módulos para ayudar a los desarrolladores con sus proyectos de codificación.

Al mismo tiempo, Python tiene una comunidad activa que contribuye con un conjunto aún más amplio de paquetes que pueden ayudarlo con sus necesidades de desarrollo. Estos paquetes se publican en Python Package Index , también conocido como PyPI (pronunciado Pie Pea Eye).

PyPI

Para instalar paquetes, pip proporciona un comando install. Puede ejecutarlo para instalar el paquete django por ejemplo:

```
● ● ●  
python3 -m pip install django  
o  
python -m pip install django  
o  
pip install django
```



Instalación manual

Para instalar paquetes manualmente con pip, debemos tener el paquete con extensión .zip, tar.gz u otro, luego deberás utilizar el mismo comando install seguido del paquete.

```
● ● ●  
pip install paquete.zip  
o  
pip install paquete.tar.gz
```

Desinstalación

El comando uninstall le muestra los archivos que se eliminarán y solicita confirmación. Si está seguro de que desea eliminar el paquete porque verificó sus dependencias y sabe que nada más lo está usando, puede pasar la opción -y para suprimir la lista de archivos y el cuadro de diálogo de confirmación:

```
● ● ●  
$ python -m pip uninstall certifi  
o  
$ python -m pip uninstall urllib3 -y
```

Puede usar el comando list para mostrar los paquetes instalados en su entorno, junto con sus números de versión:



```
$ python -m pip list
```

Package	Version
certifi	x.y.z
charset-normalizer	x.y.z
idna	x.y.z
pip	x.y.z
requests	x.y.z
setuptools	x.y.z
urllib3	x.y.z

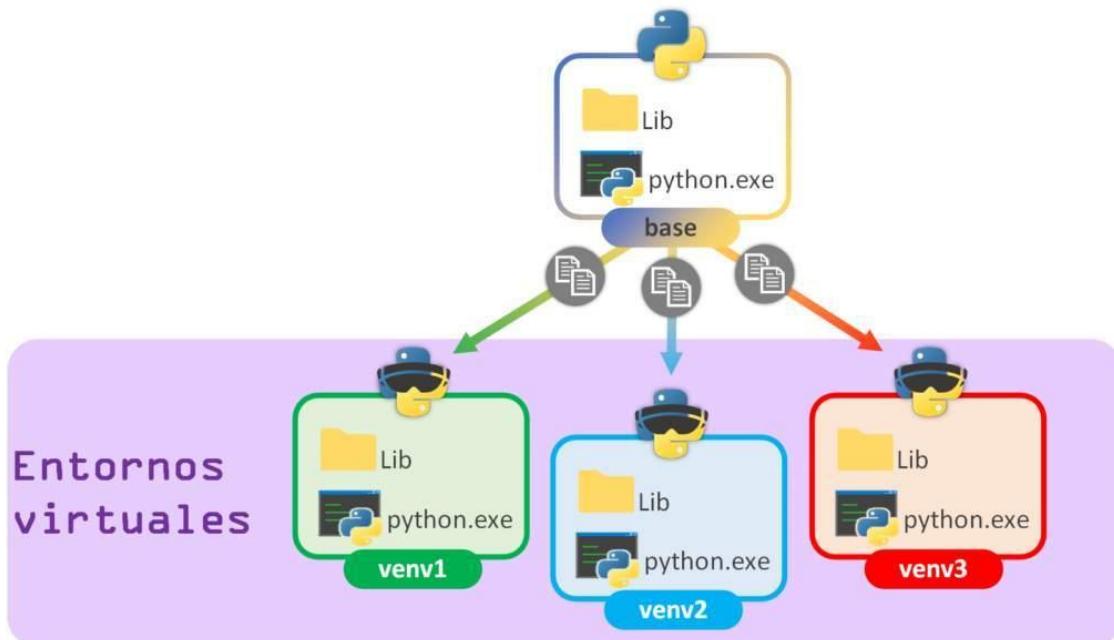


ENTORNO VIRTUAL

Un entorno de desarrollo virtual python o simplemente entorno virtual python es un mecanismo que me permite gestionar programas y paquetes python sin tener permisos de administración, es decir, cualquier usuario sin privilegios puede tener uno o más "espacios aislados" donde poder instalar distintas versiones de programas y paquetes python.

Para crear los entornos virtuales podemos usar el programa virtualenv o el módulo venv y para instalar paquetes python se usa el programa pip .

Entornos Virtuales



Puedes tener varios entornos, con varios conjuntos de paquetes, sin conflictos entre ellos. De esta manera, los requisitos de diferentes proyectos se pueden satisfacer al mismo tiempo.

Puedes lanzar fácilmente tu proyecto con sus propios módulos dependientes.



Podemos utilizar la herramienta virtualenv para crear nuestros entornos virtuales. Sin embargo, desde la versión 3.3 de python tenemos a nuestra disposición un módulo del sistema venv que podemos utilizar para crear nuestro entorno virtual. Por lo tanto debemos diferenciar los distintos paquetes que podemos utilizar:

Virtualenv

Es un software que podemos encontrar en el Python Package Index o PyPI , que es el repositorio de paquetes de software oficial para aplicaciones de terceros en el lenguaje de programación Python.

```
$ pip install virtualenv
```

Venv

Es un módulo oficial del lenguaje que a partir de la versión 3.3 nos permite crear entornos virtuales.

Crear entornos virtuales

Para crear entornos virtuales lo hacemos mediante la terminal utilizando cualquiera de las dos herramientas de la siguiente manera:

```
● ● ●  
$ python -m venv nombreentorno  
o  
$ virtualenv nombreentorno
```

Activar entorno virtual

Una vez que tenemos creado el entorno virtual debemos activarlo para poder trabajar en él, para ello hacemos lo siguiente:



POWER SHELL

```
$ .\entornovirtual\Scripts\Activate.ps1
```

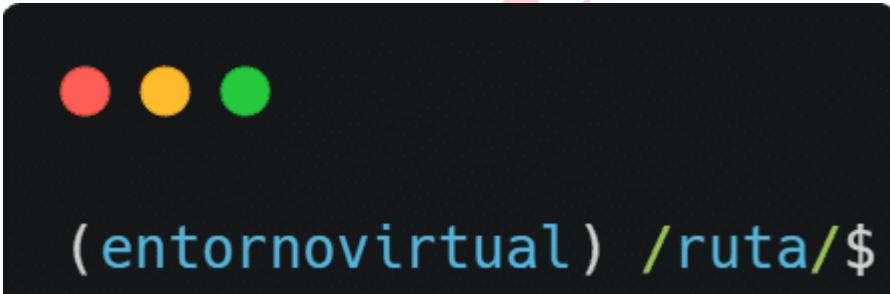
CMD O COMMAND PROMT

```
$ .\entornovirtual\Scripts\Activate.bat
```

LINUX O MACOS

```
$ source entornovirtual/bin/activate
```

Si todo ha salido bien, podremos ver que al lado izquierdo de la terminal entre paréntesis el nombre del entorno virtual.



```
(entornovirtual) /ruta/$
```

Para desactivar debemos ejecutar el comando deactivate.



```
(entornovirtual) /ruta/$ deactivate
```



REQUIREMENTS

Requirements

Administrar los paquetes instalados (dependencias) en un entorno virtual de Python puede ser bastante desafiante, especialmente para personas nuevas en el lenguaje. Al desarrollar un nuevo paquete o proyecto de Python, es probable que también necesite utilizar otros paquetes que eventualmente lo ayudarán a escribir menos código (en menos tiempo) para que no tenga que reinventar la rueda. Además, su paquete de Python también puede usarse como dependencia en proyectos futuros.

- En términos muy simples, las dependencias son paquetes externos de Python en los que se basa su propio proyecto para realizar el trabajo previsto. En el contexto de Python, estas dependencias generalmente se encuentran en el Índice de paquetes de Python (PyPI) o en otras herramientas de administración de repositorios, como Nexus.
- Cada dependencia, que a su vez es un paquete de Python por sí mismo, también puede tener otras dependencias. Por lo tanto, la administración de dependencias a veces puede ser bastante complicada o desafiante y debe manejarse adecuadamente para evitar problemas al instalar o incluso mejorar el paquete.

Requirements.txt

Es requirements.txt un archivo que enumera todas las dependencias de un proyecto de Python específico. También puede contener dependencias de dependencias, como se explicó anteriormente. Las entradas enumeradas pueden estar ancladas o no ancladas. Si se usa un pin, puede especificar una versión de paquete específica (usando ==), un límite superior o inferior o incluso ambos.

Ejemplo:



```
● ● ●  
matplotlib>=2.2  
numpy>=1.15.0, <1.21.0  
pandas  
pytest==4.0.1
```

Puedes copiar todos los paquetes instalados en un entorno virtual dentro de un archivo requirements.txt mediante freeze de pip de la siguiente manera:

```
● ● ●  
(venv) $ python -m pip freeze > requirements.txt  
o  
pip freeze > requirements.txt
```

Finalmente, puede instalar estas dependencias (normalmente en un entorno virtual o para otro proyecto) utilizando pip con el siguiente comando:

```
● ● ●  
(venv) $ python -m pip install -r requirements.txt  
o  
pip install -r requirements.txt
```