

Contenido

Componentes de un ordenador	7
Ordenador	7
Despiece del hardware de un ordenador:	7
Placa base	7
Microprocesador	9
Random Access Memory	9
Discos duros, unidades de estado sólido	10
Conectividad y comunicaciones	11
Software	12
Software de sistema	13
Software de aplicación	14
Programas y algoritmos	14
Procesamiento de la información	14
Algoritmos	15
Programa y lenguaje	16
Python	16
¡Hola mundo!	17
¡Hola mundo!	17
Help	18
Secuencias de instrucciones	19
Comentarios	20
Entrada de datos, variables	21
Entrada de datos	21
Los nombres de las variables	22
Los valores de las variables	23
Asignación	24
Asignación múltiple	24
Entrada de valores no textuales	25
Tipos de datos, operadores	26
Tipos de datos	26
Números	26
Valores booleanos	27
Strings	28
Conocer el tipo de un dato	29

Type casting.....	30
Anotación de tipos	30
Operadores de comparación.....	31
Precedencia de operadores	31
Asociatividad de operadores	32
Ejecución alternativa.....	33
Sentencias de control.....	33
Condiciones	33
Sentencia if.....	34
Cláusula else.....	35
Anidamiento.....	36
Contracción elif	37
Funciones	38
Funciones	38
Funciones con parámetros.....	39
Funciones que devuelven un resultado	40
Funciones polimórficas	41
Anotación de los tipos de los parámetros de una función.....	42
Módulos y paquetes.....	42
Dónde escribir una función	42
Módulos	43
El módulo <code>__main__</code>	44
Paquetes.....	45
Tuplas	46
Tuplas	46
Acceso a los elementos individuales de una tupla.....	47
Operaciones con tuplas.....	47
Operaciones con tuplas (2)	48
Anotación de variables tupla.....	49
Iteración – while.....	49
Repetición	49
Repetición (2)	50
Esquema básico de iteración.....	50
Iteración – for	51
Sentencia for	51
Equivalencia entre for y while.....	53

La sentencia for y el tratamiento de secuencias.....	53
Acceso a los índices al iterar sobre una secuencia.....	54
Esquemas comunes de tratamiento de secuencias	55
Esquemas de recorrido y acumulación	55
Esquema de selección	56
Esquema de búsqueda	57
Alteración del flujo de un bucle	59
Depuración de programas.....	60
Depuración	60
Iniciando la depuración en Python.....	61
Breakpoints (puntos de parada).....	62
Avance	62
Inspección.....	63
Listas.....	64
Listas.....	64
Acceso a los elementos de una lista.....	64
Concatenación de listas.....	65
El operador *	65
Operador in	66
Anotación de variables lista	66
Operaciones de manejo de listas	66
Las listas son mutables	66
Borrado de elementos.....	67
Recorrido de una lista	67
Enumerate.....	68
List comprehension	68
List comprehension - más ejemplos.....	69
Funciones de manejo de listas	69
Función list().....	70
Métodos de manejo de listas sin modificación.....	70
Métodos que modifican listas	71
Slices.....	71
Asignación a slices	72
Comparación de listas	72
Listas y funciones	73
Alias	73

Shallow copy y deep copy	74
Paso de listas como parámetros	75
Listas como resultado de funciones	75
Funciones modificadoras y no modificadoras.....	76
Tratando múltiples listas a la vez	76
Procesando varias listas al mismo tiempo	76
Procesando varias listas al mismo tiempo con zip	77
Procesando simultáneamente listas de distintos tamaños con zip_longest	77
Procesando varias listas de diferentes tamaños a la vez usando índices	78
Estructuras 2D	79
Listas de tuplas	79
Tablas como listas de listas	80
Recorrido de una tabla	81
Mutabilidad	81
Usando funciones para crear estructuras 2D.....	82
Uso de comprehension para crear listas de listas.....	82
Listas de listas de distintas longitudes	83
Tratamiento estructurado de estructuras 2D	83
Diccionarios	84
Diccionarios	84
Otras formas de inicializar un diccionario.....	85
Operaciones en diccionarios	85
Eliminar una pareja clave:valor de un diccionario	86
El método <i>get()</i>	86
Iteración en diccionarios	87
Diccionarios como valores de diccionarios	88
Conjuntos	88
El tipo <i>set</i>	88
Modificación de conjuntos.....	89
Pertenencia e inclusión	90
Álgebra de conjuntos	91
Más sobre strings.....	92
Strings.....	92
Secuencias de escape	93
Mayúsculas y minúsculas	94
Categorías de caracteres	95

Acceso a los componentes de una string	96
Concatenación de strings	96
Localización de una substring	97
El método join()	98
El método split()	98
El método <i>replace()</i>	99
Recorte y relleno: los métodos strip(), lstrip(), rstrip(), ljust(), rjust(), center() y zfill()	99
Formateo de strings	100
exto con formato.....	100
Interpolación de strings: f-strings	101
Modificadores de formato	101
Modificadores de formato - ancho de campo	102
Modificadores de formato - Precisión.....	103
Modificadores de formato - Notación numérica	104
El método format().....	105
Expresiones regulares	106
Expresiones regulares	106
Metacaracteres	106
Metacaracteres - clases de caracteres	107
Metacaracteres - cuantificadores	108
Greedy vs no greedy.....	109
Metacaracteres - grupos	110
Metacaracteres - backreferences	110
Metacaracteres - conjuntos y rangos.....	111
Metacaracteres - delimitadores.....	112
El módulo re	112
El módulo re	112
La función search()	113
La función finditer()	113
La función findall().....	114
La función split().....	115
La función sub()	115
Flags.....	116
Funciones resursivas	117
Funciones recursivas (1)	117
Funciones recursivas (2).....	118

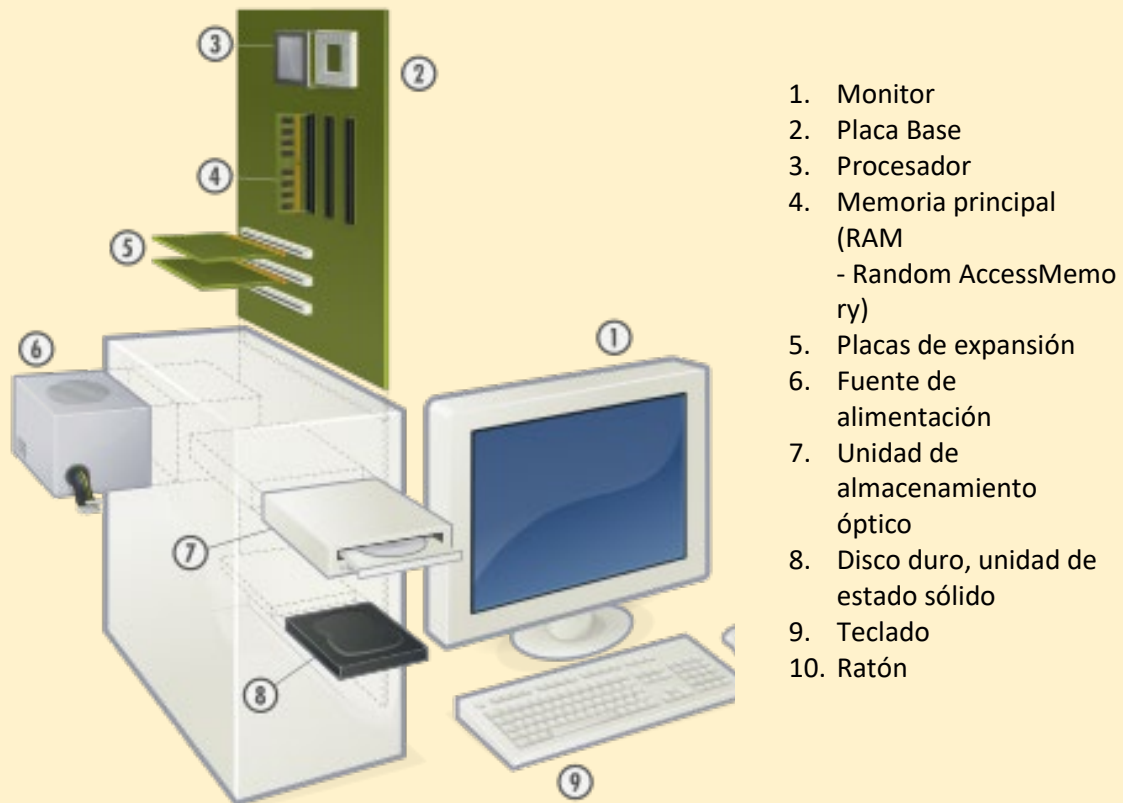
Búsqueda de una solución recursiva: paso 1 - el tamaño del problema	119
Búsqueda de una solución recursiva: paso 2 - el caso base.....	120
Búsqueda de una solución recursiva: paso 3 - descomposición del problema	121
Esquema de una solución recursiva	122
Ficheros de texto.....	122
Tratamiento de un archivo de texto	123
Tratamiento de un archivo usando la sentencia for	125
Leer un fichero en una string única.....	126
Escribir en un fichero de texto	126
Adición de contenido a un archivo existente.....	127
Terminología básica de BBDD	128
Datos	128
Atributos.....	128
Registros.....	128
Base de datos	129
SQL y Python	130
¿Qué es SQL?.....	130
Crear y eliminar BBDD.....	130
Creación de tablas	131
La instrucción INSERT INTO	132
La instrucción SELECT	133
La cláusula WHERE	135
La cláusula ORDER BY	136
Las instrucciones UPDATE y DELETE.....	136

Ordenador

Un ordenador es una máquina diseñada con el propósito de almacenar y procesar información con el fin de resolver problemas de diversa índole. Se compone básicamente de un **hardware**, que es el conjunto de dispositivos físicos (circuitos, cables, placas,...) que forman la máquina, y un **software**, que es el conjunto de programas que utilizan esos dispositivos para almacenar y procesar información.

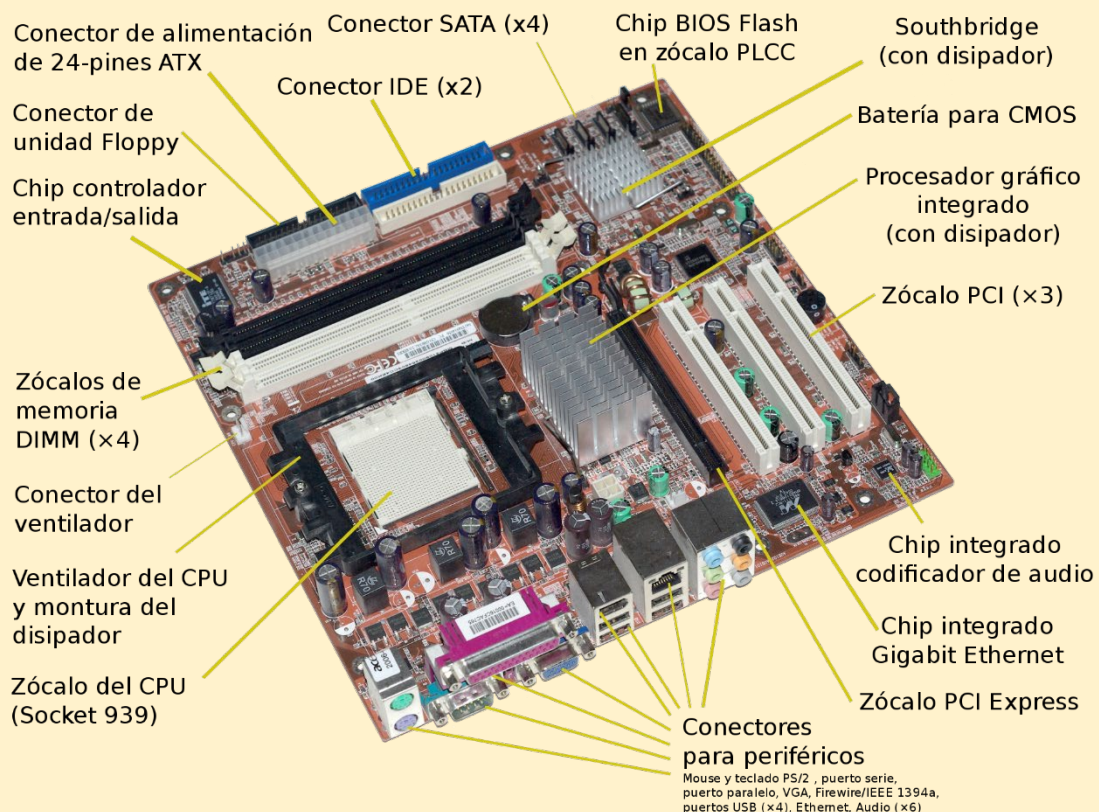
Un ordenador es una máquina programable, lo que le permite resolver problemas de diverso tipo, aplicando diferentes programas, que, básicamente, consisten en secuencias de instrucciones que definen las acciones a realizar para resolver un problema, proporcionando un resultado buscado a partir de unos datos de entrada.

Despiece del hardware de un ordenador:



Fuente: Wikipedia

Placa base



La placa base, también conocida como tarjeta madre, placa madre o placa principal (motherboard o mainboard en inglés), es una tarjeta de circuito impreso a la que se conectan los componentes que constituyen la computadora. Los componentes más destacados de una placa base son:

- Uno o varios **conectores de alimentación**, que reciben alimentación eléctrica de la fuente de alimentación de la computadora y la distribuyen a la CPU, el conjunto de chips, la memoria principal y las tarjetas de expansión.
- Uno o más **zócalos (socket) de CPU**, en los que se instala el microprocesador y lo conectan con el resto de componentes a través del **bus frontal** de la placa base.
- El **chipset**, que es un conjunto de circuitos electrónicos, que gestionan las transferencias de datos entre los diferentes componentes del ordenador (procesador, memoria, tarjeta gráfica, unidad de almacenamiento secundario, etcétera). El chipset, generalmente se divide en dos secciones:
 - El puente norte (northbridge), que gestiona la interconexión entre el microprocesador, la memoria RAM y la unidad de procesamiento gráfico
 - El puente sur (southbridge), que gestiona la interconexión entre los periféricos y los dispositivos de almacenamiento, como los discos duros o las unidades de disco óptico.
- **Ranuras** para insertar módulos de memoria RAM (Random Access Memory) del tipo adecuado, dependiendo de la velocidad, capacidad y fabricante requeridos según la compatibilidad de la placa base y la CPU.
- Chips de memoria no volátiles (generalmente Flash ROM en placas base modernas) que contienen el firmware o BIOS del sistema (un software que localiza y reconoce todos los dispositivos necesarios para cargar el sistema operativo en la memoria RAM).

- Las **ranuras de expansión**, que pueden acoger placas o tarjetas de expansión que se utilizan para agregar características o aumentar el rendimiento de la computadora (por ejemplo, una tarjeta gráfica).
- Un **generador de reloj** que produce la señal del reloj del sistema para sincronizar los distintos componentes.
- Conectores para discos duros, conectores para periféricos (puertos USB, conectores de video, audio, etc.), controladores integrados de periféricos ('controladores de discos, de ethernet, de usb, de gráficos,...)

Microprocesador

El microprocesador (o simplemente procesador) es el circuito integrado central más complejo de un sistema informático; a modo de ilustración, se le suele llamar por analogía el «cerebro» de un ordenador.

Es el encargado de ejecutar los programas, desde el sistema operativo hasta las aplicaciones de usuario; solo ejecuta instrucciones programadas en lenguaje de bajo nivel, realizando operaciones aritméticas y lógicas simples, tales como sumar, restar, multiplicar, dividir, las lógicas binarias y accesos a memoria.

Puede contener una o más unidades centrales de procesamiento (CPU) constituidas, esencialmente, por registros, una unidad de control, una unidad aritmético lógica (ALU) y una unidad de cálculo en coma flotante (conocida antiguamente como «coprocesador matemático»).

En las computadoras, el microprocesador se monta en la llamada placa base, sobre un zócalo conocido como zócalo de CPU, que permite las conexiones eléctricas entre los circuitos de la placa y el procesador. Sobre el procesador ajustado a la placa base se fija un disipador térmico de un material con elevada conductividad térmica, que por lo general es de aluminio, y en algunos casos de cobre. Éste es indispensable en los microprocesadores que consumen bastante energía, la cual, en gran parte, es emitida en forma de calor: en algunos casos pueden consumir tanta energía como una lámpara incandescente (de 40 a 130 vatios).

Random Access Memory

La memoria de acceso aleatorio (Random Access Memory, RAM) se utiliza como memoria de trabajo de computadoras y otros dispositivos para el sistema operativo, los programas y la mayor parte del software.

En la RAM se cargan todas las instrucciones que ejecuta la unidad central de procesamiento (procesador) y otras unidades del computador, además de contener los datos que manipulan los distintos programas.

Se denominan «de acceso aleatorio» porque se puede leer o escribir en una posición de memoria con un tiempo de espera igual para cualquier posición, no siendo necesario seguir un orden para acceder (acceso secuencial) a la información de la manera más rápida posible.

La memoria viene en módulos que se insertan en ranuras de la placa base.

Los ordenadores actuales usan módulos DIMM (Dual In-line Memory Module, ordenadores de sobremesa), o SO-DIMM (Small Outline DIMM, ordenadores portátiles).

La cantidad máxima de memoria que se le puede poner a un ordenador viene determinada por el número de ranuras multiplicado por la capacidad de los módulos que se le pueden poner.

El tipo de memoria se identifica con unas siglas, que indican la tecnología usada, separadas por un guión de un número que indica la velocidad efectiva del reloj, lo que determina la velocidad de transferencia de información. DDR3-1600 corresponde a Double Data Rate type three Synchronous Dynamic Random-Access Memory, con velocidad efectiva de 1600 MHz.

El modelo del módulo se identifica con unas siglas separadas por un guión de un número que indica el ancho de banda aproximado (máxima capacidad de transferencia). PC3-12800 indica un ancho de banda de 12800 MB/s.

El ancho de banda se obtiene multiplicando por 8 la capacidad efectiva del reloj indicada en el tipo de memoria.

Discos duros, unidades de estado sólido

Un disco duro o disco rígido (en inglés Hard Disk Drive, HDD) es un dispositivo de almacenamiento de datos no volátil que emplea un sistema de grabación magnética para almacenar datos digitales. Se compone de uno o más **platos**, unidos por un mismo eje que gira a gran velocidad dentro de una caja metálica sellada. Sobre cada plato, y en cada una de sus **caras**, se sitúa un **cabezal** de lectura/escritura que flota sobre una delgada lámina de aire generada por la rotación de los discos. (Wikipedia). Los platos se dividen en circunferencias concéntricas llamadas **pistas**. El conjunto de pistas que ocupan la misma posición en los distintos platos y caras forman un **cilindro**. Las pistas se dividen en sectores, que, según el estándar actual, tienen un tamaño de 512 bytes. Las pistas externas, al ser más largas, tienen más sectores que las internas. Un cluster es un conjunto contiguo de sectores que componen la unidad más pequeña de almacenamiento de un disco (un archivo puede ocupar uno o varios cluster, pero dos archivos no pueden compartir un cluster). Las características más importantes de los discos son:

- **Capacidad:** Cantidad de información que puede almacenar. Usualmente se mide en Gigabytes (miles de millones de bytes).
- **Tiempo medio de acceso:** Tiempo medio que tarda la cabeza en situarse en la pista y el sector deseado; se obtiene sumando:
 - **Tiempo medio de búsqueda:** Tiempo medio que tarda la cabeza en situarse en la pista deseada; es la mitad del tiempo empleado por la cabeza en ir desde la pista más periférica hasta la más central del disco.
 - **Latencia media:** Tiempo medio que tarda la cabeza en situarse en el sector deseado; es la mitad del tiempo empleado en una rotación completa del disco. Este tiempo es la inversa de la velocidad de rotación, medida como número de revoluciones por minuto (RPM).
- **Tasa de transferencia:** Velocidad a la que puede transferir la información a la computadora una vez que la cabeza está situada en la pista y sector correctos.

- **Interfaz de conexión:** existen diferentes formas de conectar el disco al ordenador, por ejemplo SCSI (Small Computer System Interface), que es muy popular en servidores, y SATA (Serial ATA), que es corriente en ordenadores de sobremesa.
- Otros aspectos a tener en cuenta son el consumo energético y la protección contra impactos (especialmente en el caso de portátiles)

Las unidades de estado sólido, SSD (acrónimo inglés de solid-state drive) son un tipo de dispositivo de almacenamiento de datos que utiliza memoria no volátil, como la memoria flash, para almacenar datos, en lugar de los platos o discos magnéticos de las unidades de discos duros (HDD) convencionales.

En comparación con los discos duros tradicionales, las unidades de estado sólido son menos sensibles a los golpes al no tener partes móviles, son prácticamente inaudibles, y poseen un menor tiempo de acceso y de latencia, lo que se traduce en una mejora del rendimiento exponencial en los tiempos de carga de los sistemas operativos. En contrapartida, su vida útil es muy inferior, ya que tienen un número limitado de ciclos de escritura, pudiendo producirse la pérdida absoluta de los datos de forma inesperada e irrecuperable. Las SSD hacen uso de la misma interfaz SATA que los discos duros, por lo que son fácilmente intercambiables sin tener que recurrir a adaptadores o tarjetas de expansión para compatibilizarlos con el equipo.

Conectividad y comunicaciones

DVI-D: Digital Visual Interface, conocida por las siglas también en inglés DVI, que significan “Interfaz Digital Visual”, es una interfaz de video diseñada para obtener la máxima calidad de visualización posible en pantallas digitales, tales como los monitores LCD de pantalla plana y los proyectores digitales. Fue desarrollada por el consorcio industrial Digital Display Working Group. Por extensión del lenguaje, al conector de dicha interfaz se le llama conector tipo DVI.

USB: El “Bus Universal en Serie” (BUS), en inglés: Universal Serial Bus más conocido por la sigla USB, es un bus estándar industrial que define los cables, conectores y protocolos usados en un bus para conectar, comunicar y proveer de alimentación eléctrica entre computadoras, periféricos y dispositivos electrónicos.

VGA: Video Graphics Array (VGA), Adaptador Gráfico de Video se utiliza para denominar: a una pantalla de computadora analógica estándar; a la resolución 640×480 píxeles; al conector VGA de 15 contactos D subminiatura; a la tarjeta gráfica que comercializó IBM por primera vez en 1988.

VGA fue el último estándar de gráficos introducido por IBM al que se atuvieron la mayoría de los fabricantes de computadoras compatibles IBM, convirtiéndolo en el mínimo que todo el hardware gráfico soporta antes de cargar un dispositivo específico. Por ejemplo, la pantalla de Microsoft Windows aparece mientras la máquina sigue funcionando en modo VGA, razón por la que esta pantalla aparecerá siempre con reducción de la resolución y profundidad de color.

La norma VGA fue oficialmente reemplazada por Extended Graphics Array de IBM, pero en realidad ha sido sustituida por numerosas extensiones clónicas ligeramente

distintas a VGA realizadas por los fabricantes y que llegaron a ser conocidas en conjunto como Super VGA.

LAN (Ethernet): Una red de área local (Local Area Network - LAN) define una red que interconecta ordenadores en un área limitada, tal como un domicilio, una escuela o una empresa. Sus características se definen en oposición a las redes de área amplia (Wide Area Network - WAN) que interconectan ordenadores en una zona geográfica amplia usando servicios de telecomunicaciones.

Ethernet (pronunciado /'i:θərnɛt/ en inglés) es un estándar de redes de área local para computadores con acceso al medio por detección de la onda portadora y con detección de colisiones (CSMA/CD). Su nombre viene del concepto físico de ether. Ethernet define las características de cableado y señalización de nivel físico y los formatos de tramas de datos del nivel de enlace de datos del modelo OSI.

Ethernet se tomó como base para la redacción del estándar internacional IEEE 802.3, siendo usualmente tomados como sinónimos. Se diferencian en uno de los campos de la trama de datos. Sin embargo, las tramas Ethernet e IEEE 802.3 pueden coexistir en la misma red.

WI-FI: Wifi —/'waɪfaɪ/; pronunciado en algunos países hispanohablantes /'wɪfi/, su nombre proviene de la marca comercial Wi-Fi—1 es un mecanismo de conexión de dispositivos electrónicos de forma inalámbrica. Los dispositivos habilitados con wifi, tales como un ordenador personal, una consola de videojuegos, un smartphone, o un reproductor de audio digital, pueden conectarse a Internet a través de un punto de acceso de red inalámbrica. Dicho punto de acceso tiene un alcance de unos 20 metros en interiores, una distancia que es mayor al aire libre.

«Wi-Fi» es una marca de la Wi-Fi Alliance —anteriormente la Wireless Ethernet Compatibility Alliance (WECA)—, la organización comercial que adopta, prueba y certifica que los equipos cumplen los estándares 802.11 relacionados a redes inalámbricas de área local.

Software

El software son los programas que hacen funcionar un ordenador; es la parte intangible del sistema, frente al hardware, que es la parte tangible, al estar formado por el conjunto de componentes físicos.

Al clasificar el software, se puede distinguir entre el sistema operativo y las aplicaciones, o, de modo más amplio, entre software de sistema y software de aplicación.

El software de sistema abarca aquellos programas que se ocupan del funcionamiento general del ordenador, facilitando la ejecución de las aplicaciones y la interacción con el usuario y los dispositivos externos.

El software de aplicación abarca aquellos programas diseñados para satisfacer una necesidad concreta del usuario.

Software de sistema

El software de sistema incluye los programas que hacen funcionar el ordenador, los que se usan para configurar, analizar y administrar el sistema, y los que se usan para desarrollar componentes software:

Rutinas de arranque del sistema

La BIOS (Basic Input-Output System) ejecuta, al encenderse el ordenador, una rutina que chequea el hardware y busca el sector de arranque (boot sector) del dispositivo de arranque predeterminado (HDD, SDD, DVD,...) para cederle el control. La rutina de arranque carga el sistema operativo en la memoria y le cede el control del ordenador.

Sistema operativo

Es el principal componente del software de sistema. Su propósito es tomar el control del hardware y actuar de intermediario entre éste y el resto de los programas, liberando a estos de tener que ocuparse de los detalles del hardware. Sus cometidos principales son: la ejecución de programas, la gestión de recursos (memoria, principal, almacenamiento secundario, CPU,...), la comunicación entre los componentes de la máquina y las aplicaciones y la seguridad.

Los sistemas operativos más populares en los ordenadores de sobremesa son: Windows, OSX y Linux; y en los dispositivos móviles Android e iOS.

Controladores de dispositivos (Device drivers)

En computación, un controlador de dispositivo es un programa de computadora que opera o controla un tipo particular de dispositivo que está conectado a una computadora. Un controlador proporciona una interfaz de software para dispositivos de hardware, permitiendo que los sistemas operativos y otros programas de computadora accedan a las [funciones](#) de hardware sin necesidad de conocer detalles precisos sobre el hardware que se está utilizando (ejemplos: drivers de disco, drivers de una tarjeta gráfica, drivers de un dispositivo de audio,...).

Servidores

Son programas que se ejecutan para proporcionar un servicio a otros programas (clientes), normalmente de forma remota. Por ejemplo Apache Tomcat o Internet Information Service (ISS, de Microsoft) son servidores http; sirven páginas web y documentos a requerimiento de un navegador (web browser) que, normalmente se ejecuta en una máquina remota. Un servidor FTP (File Transfer Protocol) proporciona servicios de carga y descarga de ficheros a una aplicación cliente. Un servidor de bases de datos (por ejemplo MySQL) proporciona acceso, local o remoto, a un Sistema de Bases de Datos, etc.

Utilidades

Las utilidades son programas diseñados para ayudar a analizar, configurar, optimizar o mantener un ordenador o sistema informático. Generalmente, se distribuye un conjunto

básico de programas de utilidad con el sistema operativo (SO), es frecuente que el usuario pueda instalar utilidades adicionales.

Ejemplos de programas de utilidad pueden ser: el Administrador de tareas de Windows, un desfragmentador de disco, un antivirus, la aplicación de configuración de iOS, un limpiador de registro, un monitor de actividad, una terminal de línea de comandos, etc.

Software de aplicación

El software de aplicación tiene objetivo satisfacer una necesidad del usuario, frente al software de sistema, que se centra en el funcionamiento del ordenador. Cabe distinguir entre software de aplicación de propósito general y software de aplicación de propósito específico.

El software de aplicación de propósito general satisface necesidades básicas comunes a la mayoría de los usuarios. Ejemplos de aplicaciones de propósito general son:

- Clientes de correo
- Navegadores (Web browsers)
- Procesadores de texto
- Hojas de cálculo
- Mensajería electrónica
- ...

El software de propósito específico satisface necesidades particulares de usuarios especializados, por ejemplo:

- Software específico para Ingeniería Civil
- Aplicaciones médicas
- Software para Ciencia de Datos
- Entornos Integrados de Desarrollo de Software
- ...

Programas y algoritmos

Procesamiento de la información

El diccionario de la RAE define la **informática** como el "conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información mediante ordenadores".

Wikipedia define la **información** como un "conjunto organizado de datos procesados, que constituyen un mensaje que cambia el estado de conocimiento del sujeto o sistema que recibe dicho mensaje", mientras que WordReference la define como "conjunto de datos sobre una materia determinada".

Wikipedia define un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable cuantitativa o cualitativa".

Por ejemplo, la información sobre una persona puede estar compuesta de datos como: nombre, apellidos, fecha de nacimiento, sexo, dirección, estado civil, ocupación,... Algunos datos, como la edad, pueden calcularse a partir de otros, como la fecha de nacimiento y la fecha actual.

El tratamiento de la información plantea **problemas** cuya proposición es hallar un dato desconocido a partir de otros datos conocidos. Los ordenadores se utilizan para solucionar problemas de tratamiento de información.

Ejemplos de problemas sencillos:

1. *Dadas la base y la altura de un triángulo, calcular su superficie.*
2. *A partir de los datos de partidos jugados, ganados y empatados por los equipos de una liga deportiva, listar la clasificación.*

Algoritmos

Los ordenadores se utilizan para solucionar problemas de tratamiento de información. Para que un ordenador solucione un problema, hay que darle un programa que le indique cómo solucionarlo.

La RAE define **algoritmo** como "un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema". Wikipedia da una definición parecida: "un conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permiten llevar a cabo una actividad mediante pasos sucesivos que no generen dudas a quien deba hacer dicha actividad".

En el contexto de los problemas de tratamiento de la información, un algoritmo describe la solución de un problema en función de los datos necesarios para representar un **caso** concreto del problema y de los pasos necesarios para obtener el resultado deseado.

Por ejemplo, supongamos que queremos explicar cómo multiplicar dos números (sin decimales) a una persona que solo sabe hacer sumas. Podríamos decirle algo como:

"Dados dos números, que llamaremos multiplicando y multiplicador, el resultado se obtiene al sumar el multiplicando tantas veces como indique el multiplicador. Por ejemplo, si queremos multiplicar 5 por 3, el resultado es: $5+5+5=15$ ".

La descripción de la multiplicación como una repetición de sumas sería el algoritmo. El ejemplo ilustra un caso concreto del problema, representado por dos datos: multiplicando = 5 y multiplicador = 3. La aplicación del algoritmo a este caso concreto conduce al resultado (15). Variando el multiplicando y/o el multiplicador tendríamos casos diferentes del problema de la multiplicación.

Nótese que los algoritmos, en sí mismos, no requieren ordenadores: un folleto para montar una estantería de Ikea es un algoritmo que se supone va ejecutar una persona. Incluso los algoritmos de tratamiento de información, como una operación matemática o el cálculo de una nómina pueden ser ejecutados por una persona, aunque los ordenadores suelen hacerlo más rápido.

Programa y lenguaje

Para que un ordenador solucione un problema, hay que darle un programa que le indique cómo solucionarlo. Wikipedia define un **programa informático** como "una secuencia de instrucciones, escritas para realizar una tarea específica en una computadora". Nótese el parecido con la definición de algoritmo: "conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permiten llevar a cabo una actividad mediante pasos sucesivos que no generen dudas a quien deba hacer dicha actividad".

Un algoritmo se puede expresar de muchas maneras, siempre que lo entienda quien deba ejecutarlo (realizar los pasos que indica). Una de estas maneras es un programa informático que es, básicamente, un algoritmo escrito en un lenguaje de programación.

Según Wikipedia, un **lenguaje de programación** es "un lenguaje formal que comprende un conjunto de instrucciones que producen diversos tipos de resultados (obtener datos del usuario u otra fuente externa, realizar operaciones matemáticas, comparar datos, mostrar resultados en una pantalla,...). Los lenguajes de programación se utilizan en la programación de computadoras para implementar algoritmos".

Los algoritmos describen la solución a un problema en términos de los datos necesarios para representar un caso del problema y el conjunto de pasos necesarios para producir el resultado deseado. Los lenguajes de programación deben proporcionar una manera de representar tanto el proceso como los datos; con este fin proporcionan instrucciones básicas, estructuras de control, tipos de datos y estructuras de datos.

Los lenguajes de programación se clasifican de diferentes formas; una de ellas es atendiendo a su nivel de abstracción, lo que diferencia entre lenguajes de alto nivel y lenguajes de bajo nivel, en función de su proximidad a la arquitectura de la máquina. El procesador entiende un lenguaje específico, formado por secuencias de ceros y unos, que se conoce como **código máquina**. La dificultad que supone para las personas un nivel de abstracción tan bajo llevó a la creación de los lenguajes ensamblador, que son un conjunto de mnemónicos (palabras cortas como "ADD", "SUB", "MOV", "JMP") que se corresponden directamente con instrucciones en lenguaje máquina, pero son más fáciles de entender y memorizar por las personas.

Los lenguajes de alto nivel usan una sintaxis próxima al lenguaje natural (generalmente inglés), pero muy simplificada, lo que supone un nivel de abstracción mucho más asequible para los programadores, permitiendo la realización de programas complejos. Los programas escritos en alto nivel deben traducirse a código máquina para su ejecución usando compiladores o intérpretes, que son, a su vez, programas escritos para tal fin. Un **compilador** traduce un programa completo generando un ejecutable que se puede usar tantas veces como se quiera. Un **intérprete** traduce las instrucciones de un programa una a una, a medida que se requiere su ejecución, no genera un código ejecutable de manera permanente, por lo que una nueva ejecución requiere una nueva traducción.

Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Python es utilizado en una gran variedad de ámbitos, desde el desarrollo web a devops (**DevOps** es un acrónimo inglés de development (desarrollo) y operations (operaciones), que se refiere a una metodología de desarrollo de software que se centra en la comunicación, colaboración e integración entre desarrolladores de software y los profesionales de sistemas en las tecnologías de la información), pero ha sido el aumento de su uso aplicado en machine learning y data science, lo que ha acelerado su crecimiento.

Python es un lenguaje de programación potente y fácil de aprender. Cuenta con estructuras de datos de alto nivel eficientes y un enfoque simple pero efectivo para la programación orientada a objetos. La elegante sintaxis y escritura dinámica de Python, junto con su naturaleza interpretada, lo convierten en un lenguaje ideal para la creación de scripts (programas simples que sirven para realizar diversas tareas interactuando con el sistema operativo y el usuario) y el rápido desarrollo de aplicaciones en muchas áreas en la mayoría de las plataformas.

El intérprete de Python y la extensa biblioteca estándar están disponibles gratuitamente en formato fuente o binario para todas las plataformas principales desde el sitio web de Python, <https://www.python.org/>, y pueden distribuirse libremente. El mismo sitio también contiene distribuciones y enlaces a muchos módulos, programas y herramientas de Python de terceros, y documentación adicional.

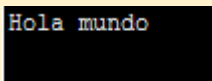
¡Hola mundo!

¡Hola mundo!

Normalmente, cuando se empieza a aprender un lenguaje de programación, el punto de arranque es un programa clásico, conocido como "¡Hola mundo!", que lo único que hace es mostrar un saludo en la pantalla del ordenador; se supone que es uno de los programas más simples que se pueden escribir en cualquier lenguaje. En Python es realmente simple:

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

En Python, la instrucción print muestra ("imprime") un mensaje. El resultado de la ejecución del programa anterior es:



```
Hola mundo
```

Nótese que, en el ejemplo, el mensaje a imprimir se encierra entre comillas (pueden ser dobles o simples), las cuales no forman parte del mensaje. La instrucción print admite varios valores, separados por comas, que imprimirá uno detrás de otro, separándolos, en principio, por espacios.

```
print("Hola", "mundo")
```

Resultado:

```
Hola mundo
```

Cuando se muestran múltiples valores, se puede cambiar el separador:

```
print("Hola", "mundo", sep="-")
```

```
Hola-mundo
```

La instrucción print escribe un salto de línea detrás del mensaje mostrado:

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

```
Hola  
mundo
```

Se puede evitar el salto de línea seleccionando un terminador con el parámetro end:

```
print("Hola", end = "/")  
print("mundo")
```

```
Hola/mundo
```

Help

Una instrucción como print admite diferentes opciones, como, por ejemplo, seleccionar el separador cuando se imprimen varios valores. Para saber qué opciones ofrece una instrucción cualquiera, se puede usar la instrucción help, pasándole entre paréntesis y comillas el nombre de la instrucción sobre la que se busca información:

```
help("print")
```

Resultado:

```
Help on built-in function print in module builtins:  
  
print(...)  
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
  
    Prints the values to a stream, or to sys.stdout by default.  
    Optional keyword arguments:  
    file:  a file-like object (stream); defaults to the current sys.stdout.  
    sep:   string inserted between values, default a space.  
    end:   string appended after the last value, default a newline.  
    flush: whether to forcibly flush the stream.  
  
(END)
```

Si se usa `help` sin nada entre paréntesis, se obtiene acceso a la ayuda general de Python, donde se puede interactuar para pedir ayuda sobre una instrucción concreta:

```
help()
```

Resultado:

```
Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> █
```

Para salir del entorno de ayuda hay que teclear `ctrl-c` (la tecla de control, `ctrl`, al mismo tiempo que la tecla `c`).

Secuencias de instrucciones

Wikipedia define un programa informático como "una secuencia de instrucciones, escritas para realizar una tarea específica en una computadora". Para escribir una secuencia de instrucciones en Python, solo hay que ponerlas una detrás de otra, una por línea. El siguiente programa imprime tres veces el mismo mensaje:

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

Resultado:

```
Hola mundo
Hola mundo
Hola mundo
```

Las instrucciones se ejecutan en el orden en que están escritas. El siguiente programa escribe "Hola mundo" y luego muestra la ayuda general de Python:

```
print("Hola mundo")
help()
```

Resultado:

```
Hola mundo

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> █
```

Este otro escribe los números del 1 al 5:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Resultado:

```
1
2
3
4
5
█
```

Nótese que los números, a diferencia de los mensajes de texto, no se encierran entre comillas.

Comentarios

Además de las instrucciones, en un programa se pueden incluir comentarios, que sirven para explicar el código a la **persona** que lo lea. En Python, el tipo básico de comentario es un texto que empieza con el símbolo #. El siguiente código incluye tres comentarios:

```
# Se muestra tres veces el texto Hola mundo
print("Hola mundo")
print("Hola mundo")
print("Hola mundo")
# Se muestran los números del 1 al 4
print(1)
print(2)
print(3)
print(4)

# Se muestra la ayuda del comando print
help("print")
```

El propósito de los comentarios no es dar información obvia, que puede deducirse fácilmente del código, como en el ejemplo anterior, sino explicar aspectos no evidentes o documentar el código para su futura reutilización o mantenimiento.

En una línea, se pueden incluir instrucciones antes del símbolo #, pero todo lo que esté después de este símbolo, hasta el final de la línea, es un comentario, independientemente de su contenido:

```
print("Hola, mundo") # muestra un saludo
```

Entrada de datos, variables

Entrada de datos

Un algoritmo describe la solución de un problema en función de los datos necesarios para representar un caso concreto del problema y de los pasos necesarios para obtener el resultado deseado. El tratamiento de la información plantea problemas cuyo propósito es hallar un resultado desconocido a partir de datos conocidos.

Para que un ordenador resuelva un caso de un problema, teniendo el programa para hacerlo, es necesario proporcionarle los datos del caso; esto se puede hacer mediante instrucciones de entrada. Véase el siguiente programa:

```
print("Hola ¿Cómo te llamas?")
name = input()
print("Hola", name)
```

El resultado de la ejecución de este programa, suponiendo que el usuario introdujera por teclado el nombre Pedro, sería el siguiente:

```
Hola ¿Cómo te llamas?
Pedro
Hola Pedro
```

El programa usa dos veces la instrucción print, que es una **instrucción de salida**, es decir, una instrucción para dar información, y aparece una instrucción nueva, input, que es una **instrucción de entrada**, una instrucción para recoger datos que van a ser usados por el programa.

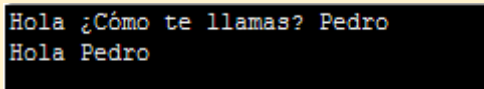
También aparece otro elemento, *name*, que no es una instrucción, sino una **variable**. Una variable se puede ver como una etiqueta que se usa para hacer referencia a un dato manejado por el programa.

El valor del dato referenciado por una variable puede ser distinto en diferentes ejecuciones del programa (supóngase que el programa anterior se ejecuta para otro usuario llamado Juan) y puede, incluso, variar durante la ejecución del programa (de ahí el nombre de variable). Esta capacidad de variación de los datos es lo que permite que un programa resuelva diferentes casos de un mismo problema.

El efecto de la ejecución de una instrucción de entrada como `input` es que el programa espera a que el usuario introduzca el valor del dato deseado. Una vez que lo hace, se recoge ese valor y se asigna a la variable que está a la izquierda del símbolo "=", que se conoce como **operador de asignación**. Como el programa se va a detener hasta que el usuario introduzca el valor de un dato, lo normal es que se le indique con un mensaje qué dato es el que se espera que introduzca; esto se hace en el ejemplo anterior con la primera instrucción `print`, pero se puede abreviar usando la propia instrucción `input`:

```
name = input("Hola ¿Cómo te llamas? ")
print("Hola", name)
```

El resultado de la ejecución de este programa es:



```
Hola ¿Cómo te llamas? Pedro
Hola Pedro
```

Nótese que en este caso la entrada del valor se espera en la misma línea donde se muestra el mensaje que lo solicita.

En ambas versiones del programa, la última instrucción muestra un saludo personalizado que incluye el valor del dato introducido por el usuario:

```
print("Hola", name)
```

name no se encierra entre comillas; las comillas sirven para indicar un valor literal tal cual (la palabra `Hola` en el ejemplo). En el caso de la variable `name`, lo que se quiere mostrar no es la palabra `name`, sino el dato referenciado por la variable (`Pedro` en el ejemplo).

Los nombres de las variables

Los programas usan **variables** para hacer referencia a los datos del problema que resuelven: datos de inicio, que varían de un caso del problema a otro; datos intermedios, generados durante la resolución del problema, y que pueden variar a medida que se ejecuta el programa; y datos finales, con los resultados que conforman la solución del problema.

Las variables tienen un **nombre** y hacen referencia al **valor** de un dato que se almacena en la memoria del ordenador.

Las **reglas sintácticas** de Python establecen que los nombres de las variables empiezan con una letra mayúscula (A - Z), o minúscula (a - z), o un guión bajo (`_`), y continúan con cualquier combinación de letras, guiones bajos y dígitos (0 - 9):

```
nombre
NOMBRE
_nombre_1
fecha_de_nacimiento
xASFasDasd24_d
```

Las **reglas de estilo** de Python establecen que los nombres de las variables deben escribirse en minúsculas, separando por guiones bajos sus componentes cuando se trate de nombres compuestos, y ser descriptivos respecto al dato que representan:

```
nombre
NOMBRE
_nombre_1
fecha_de_nacimiento
*ASFasDasd24_d
```

Los valores de las variables

Los programas manejan datos usando variables. Las variables tienen un **nombre** y hacen referencia al **valor** de un dato. Para asociar un valor con una variable, se usa una instrucción de asignación, representada por el símbolo `=`.

```
name = input()          # asigna a name un valor introducido por el
usuario                 # asigna a greeting un texto literal
greeting = "Hola, mundo"
```

Wikipedia define un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable cuantitativa o cualitativa". Existen muchos tipos de datos diferentes. En el ejemplo anterior, a las variables `name` y `greeting` se les asignan valores que representan un texto. Los valores que representan texto se conocen como strings. Una string es simplemente una secuencia de caracteres. Las strings pueden usarse para representar cosas como nombres, notas, observaciones, etc.

Otros posibles tipos de datos que se pueden asociar una variable son los números; por ejemplo, números enteros (sin parte decimal):

```
age = 21
```

números reales (con parte decimal):

```
temperature = 36.5
```

números complejos (con parte real y parte imaginaria):

```
complex_number = 34+5j
```

También se pueden usar valores booleanos, que pueden ser `True` (verdadero) o `False` (falso):

```
a_true_data = True
```

Los valores booleanos pueden ser el resultado de una comparación u otro tipo de operación:

```
a_false_data = 8 > 5 # la expresión 8 > 5 resulta ser False
```

Nótese que solo los valores de texto literales (strings) se representan encerrados entre comillas.

Las strings y los números, incluyendo los datos booleanos, que son una clase de números, son tipos básicos en Python. Python predefine también otros tipos de datos

([tuplas](#), [listas](#), diccionarios,...) o los incorpora a través de librerías externas, aparte de dar la posibilidad al programador de definir nuevos tipos según sus necesidades.

Asignación

Los programas manejan datos usando variables. Las variables tienen un **nombre** y hacen referencia al **valor** de un dato. Para asociar un valor con una variable, se usa una **instrucción de asignación**, representada por el símbolo =. A una variable se le puede asignar:

- El resultado de una operación de entrada

```
var1 = input()
```

- Un valor literal del tipo apropiado (nótese que los textos se encierran entre comillas y los números no)

```
var2 = 35
var3 = "Hola"
```

- El resultado de una expresión

```
var4 = 35 + 18 # a var4 se le asigna el valor 53
```

- Otra variable (en este caso, en Python, la variable asignada y la que se asigna hacen referencia al mismo valor)

```
var5 = var4
```

- Las expresiones pueden incluir variables

```
var6 = var4 + var5 + 10 # el valor asociado a var6 es 116
```

- La propia variable a la que se asigna el resultado de una expresión puede estar incluida en la expresión

```
var6 = var6 + 1 # el valor asociado a var6 es 117
```

- Si la variable forma parte de la expresión, muchas veces, se puede abreviar

```
var6 += 1 # equivale a var6 = var6 + 1
var6 *= 2 # equivale a var6 = var6 * 2
```

El **funcionamiento de la asignación** es: primero se evalúa la expresión que está a la derecha del símbolo de asignación (el término expresión, no solo se refiere a operaciones aritméticas; puede incluir: operaciones de otro tipo, [funciones](#) matemáticas, instrucciones de entrada, valores literales y variables) y el resultado de esa evaluación se asigna a la variable que está a la izquierda del símbolo de asignación.

Asignación múltiple

Los programas manejan datos usando variables. Las variables tienen un **nombre** y hacen referencia al **valor** de un dato. Para asociar un valor con una variable, se usa una **instrucción de asignación**, representada por el símbolo =. El funcionamiento de la asignación es: primero se evalúa la expresión que está a la derecha del símbolo de asignación y, luego, el resultado de esa evaluación se asigna a la variable que está a la izquierda del símbolo de asignación.

```
var1 = 35
```

Python permite asignar múltiples valores a múltiples variables en una sola instrucción de asignación:

```
a, b, c = 1, 2, 3
```

A la derecha de la asignación debe haber tantas expresiones, separadas por comas, como variables haya a la izquierda de la asignación. Los valores de las expresiones se asignan de izquierda a derecha a las variables correspondientes: el primer valor a la primera variable, el segundo valor a la segunda variable, etc. En el ejemplo anterior, el valor 1 se asigna a la variable a, el valor 2 a la variable b y el valor 3 a la variable c.

La asignación múltiple permite, entre otras cosas, intercambiar los valores de dos variables con una sola instrucción.

```
a, b = b, a
```

Esto mismo, en muchos otros lenguajes, requeriría tres instrucciones y una variable adicional:

```
aux = a
a = b
b = aux
```

La variable adicional (aux en el ejemplo) sirve para guardar el valor de la primera variable que se va a cambiar, de forma que no se pierda y pueda asignársele luego a la otra.

Entrada de valores no textuales

La instrucción input recoge los datos, sean del tipo que sean, como una secuencia de caracteres (string), tal como se introducirían por teclado.

```
name = input("¿Cómo te llamas? ")
```

Si la secuencia de caracteres introducidos se puede interpretar como un valor válido de otro tipo primitivo (número entero, número real, número complejo, valor booleano), hay que indicar que se quiere hacer esa interpretación:

```
text = input("Dame una string formada por una secuencia de dígitos: ")
integer_number = int(input("Dame un número entero: "))
real_number = float(input("Dame un número real: "))
complex_number = complex(input("Dame un número complejo: "))
boolean_value = bool(input("Dame un valor booleano: "))
```

Resultado de la ejecución (se resaltan los valores introducidos por el usuario):

```
Dame una string formada por una secuencia de dígitos: 23289
Dame un número entero: 23289
Dame un número real: 23289.0
Dame un número complejo: 14+23j
Dame un valor booleano: True
```

Nótese el uso de int, float, complex o bool para indicar la interpretación requerida. El caso de float, para los números reales, se debe a la representación en memoria de este tipo de valores, que usa un formato llamado "representación decimal en coma flotante" (de ahí lo de float=flotante). Cada tipo de datos usa una representación interna diferente, incluso en casos como los referenciados por las variables text e integer_number en el ejemplo, que responden a la misma secuencia de caracteres de entrada. Las strings (secuencias de caracteres) se identifican como str, pero no es necesario indicarlo en una instrucción de entrada, dado que ese es el tipo por omisión de los datos entrados por teclado.

En caso de que la secuencia de caracteres tecleada por el usuario no pueda interpretarse como un valor del tipo requerido, se produce un error.

Tipos de datos, operadores

Tipos de datos

Los programas manejan datos usando variables. Wikipedia define un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable cuantitativa o cualitativa".

Existen datos de diferentes tipos: texto, como el nombre de una persona, números, como la edad de una persona, fechas, como la fecha de nacimiento de una persona, etc.

Los tipos de datos estándar de Python incluyen números (enteros, reales, complejos y booleanos), strings (texto), [tuplas](#), [listas](#) y diccionarios. Las strings, las [tuplas](#) y las [listas](#) se agrupan en la categoría de secuencias.

Por ahora, nos centraremos en los números y las strings.

Números

Los números en Python pueden ser: enteros (int), reales (float) y complejos (complex). La siguiente tabla muestra ejemplos de valores literales de cada uno de estos tipos:

int	float	complex
10	0.0	3.14j
100	15.20	45.j

-786	-21.9	9.322e-36j
070	32.3+e18	.876j
-0490	-90.	-.6545+0j
-0x260	-32.54e100	3e+26j
0x69	70.2-E12	4.53e-7j

En la columna int, un 0 al principio del número indica que el resto de la secuencia representa un número en base 8 (octal), en lugar de la habitual base 10 (decimal). El prefijo 0x indica que el número está en base 16 (hexadecimal)

Los tipos numéricos, excepto complex, admiten las siguientes operaciones:

Operación Resultado

x + y	suma x más y (5 + 2 da 7)
x - y	resta x menos y (5 - 2 da 3)
x * y	producto de x por y (5 * 2 da 10)
x / y	división de x entre y (5 / 2 da 2.5)
x // y	división entera de x entre y (5 // 2 da 2)
x % y	resto de x // y (5 % 2 da 1)
-x	x negada (-5, si x vale 5)
+x	x sin cambio (+5 es igual a 5)
abs(x)	valor absoluto x (abs(-5) da 5)
divmod(x, y)	La pareja(x // y, x % y) (divmod(5, 2) da (2, 1))
pow(x, y)	x elevado a y (pow(5, 2) da 25)
x ** y	x elevado a y (5 ** 2 da 25)

Valores booleanos

Los valores booleanos son *True* y *False*, pero en realidad, el tipo bool es, en Python, un subtipo del tipo int, por lo que, también se consideran que tienen el valor *False*:

- El valor cero de cualquier tipo numérico (0 (int), 0.0 (float), 0j (complex))
- Una secuencia vacía (por ejemplo, la string "", que no tiene ningún carácter)
- Cualquier valor, x, de cualquier otro tipo, que al aplicársele la operación bool(x) dé como resultado un cero o False
- El valor None, un valor especial que significa "ningún valor"

A los valores booleanos se les aplican los operadores or, and y not:

Operación	Resultado
x or y	<i>True</i> si alguno de los dos es <i>True</i> :

Operación	Resultado
	Las variables se examinan de izquierda a derecha, de manera que si x es <i>True</i> , ese es el resultado, sin necesidad de mirar el valor de y; si x es <i>False</i> , el resultado será el valor de y
x and y	<i>True</i> si los dos son <i>True</i> : Las variables se examinan de izquierda a derecha, de manera que si x es <i>False</i> , ese es el resultado, sin necesidad de mirar el valor de y; si x es <i>True</i> , el resultado será el valor de y
not x	Negación: si x es <i>False</i> , el resultado es <i>True</i> , si no, es <i>False</i>

Strings

Las strings representan secuencias contiguas de caracteres. Los valores literales de tipo string (str), se encierran entre comillas dobles o simples:

```
var1 = "hola"
var2 = 'hola'
str_var = "Hello world"
print(str_var)          # Muestra "Hello world"
```

La longitud de una string (número de caracteres) se conoce programáticamente usando la función len():

```
str_var = "Hello world"
print(len(str_var))     # Muestra el valor entero 11
```

Se puede acceder a un carácter concreto de una string escribiendo la string, o el nombre de la variable que la referencia, seguida del índice de la posición que ocupa el carácter deseado entre corchetes. El índice de la primera posición es 0.

```
str_var = "Hello world"
print(str_var[0])       # Muestra el primer carácter de str_var: 'H'
print(str_var[4])       # Muestra el quinto carácter de str_var: 'o'
```

También se pueden usar índices relativos a la longitud de la string:

```
str_var = "Hello world"
print(str_var[-1])      # Muestra el último carácter de str_var: 'd'
print(str_var[-5])      # Muestra el quinto carácter, empezando por el final, de str_var: 'w'
```

Se puede obtener una substring (subsecuencia) de una string usando el operador de segmento, o slice, ([:]). El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final; si se omite el primero, toma desde el principio hasta la posición del índice anterior al indicado:

```
str_var = "Hello world"
print(str_var[2:5])     # Muestra los caracteres de 3º al 5º: "llo"
print(str_var[2:])      # Muestra los caracteres a partir del 3º: "llo world"
print(str_var[:5])      # Muestra los caracteres hasta la 5ª posición: "Hello"
```

El diccionario de la RAE define [concatenar](#) como "**unir dos o más cosas**"; en el caso de las strings, [concatenar](#) dos strings consiste en formar una nueva string juntando, en orden, dos preexistentes.

En Python existen dos operadores de concatenación: el signo más (+) es el operador de concatenación de strings y el asterisco (*) es el operador de repetición (equivalente a realizar una concatenación repetidas veces).

```
str_var = "Hello world"
print('a' + 'a')          # Muestra "aa"
print(str_var + "TEST")  # Muestra el valor de str_var concatenado con
"TEST": "Hello worldTEST"
print('ab' * 5)           # Muestra "ababababab"
print(str_var * 2)        # Muestra "Hello worldHello world"
```

Podríamos decir que la repetición de strings es a la concatenación de strings lo que la multiplicación de números enteros es a la [suma](#) de números enteros.

Nótese que la concatenación junta directamente las strings, sin añadir ningún carácter enmedio.

Conocer el tipo de un dato

Podemos conocer el tipo de un dato usando la operación *type()*.

```
a = 200
b = "200"
t1 = type(a)
t2 = type(b)
print(t1)
print(t2)
```

```
<type 'int'>
<type 'str'>
```

Nótese que el valor devuelto por *type()* es un valor de tipo *type*:

```
print(type(t1))
<type 'type'>
```

El valor, aunque se muestra con el formato de los ejemplos, es uno de los tipos de datos (*int*, *float*, *str*,...):

```
print(t1 == int)
True
```

Si queremos obtener una string con solo el nombre del tipo, podemos consultar el atributo `__name__` (un atributo es una propiedad asociada a un dato):

```
typeName = t1.__name__
print(typeName)
print(type(typeName))
int
<type 'str'>
```

Type casting

En programación, se conoce como "type casting" a la acción de obtener un valor de un tipo, T2, a partir de la interpretación de un valor de otro tipo, T1 (para ello es necesario que los valores de tipo T1 admitan una interpretación como valores de tipo T2).

int(): construye un número entero a partir de: un número entero, un número real (redondeando hacia abajo al número entero anterior), o una string (siempre que la string represente un número entero)

```
a = int(10)      # a toma el valor entero 10
b = int(3.5)     # b toma el valor entero 3
c = int("40")    # c toma el valor entero 40
s = "25"
d = int(s)       # d toma el valor entero 25
```

float(): construye un número real a partir de: un número entero, un número real, o una string (siempre que la string represente un número real o un número entero)

```
a = float(10)    # a toma el valor real 10.0
b = float(3.5)   # b toma el valor real 3.5
c = float("40")  # c toma el valor real 40.0
s = "25.0"
d = float(s)     # d toma el valor real 25.0
```

complex(): construye un número complejo a partir de: un número entero, un número real, un número complejo, o una string (siempre que la string represente un número real, un número entero o un número complejo)

```
a = complex(10)  # a toma el valor complejo 10+0j
b = complex(3.5) # b toma el valor complejo 3.5+0j
c = complex("40") # c toma el valor complejo 40+0j
s = "25"
d = complex(s)   # d toma el valor complejo 25+0j
e = complex(1+2j) # e toma el valor complejo 1+2j
```

str(): construye una string a partir de casi cualquier otro tipo de datos.

```
a = str(10)      # a toma el valor string "10"
b = str(3.5)     # b toma el valor string "3.5"
c = str("40")    # c toma el valor string "40"
s = "25"
d = str(s)       # d toma el valor string "25"
e = str(1+2j)    # e toma el valor string "1+2j"
```

Anotación de tipos

Python es un lenguaje de programación con tipado dinámico; eso significa que el tipo de una variable (tipo del dato asociado a la variable) se establece en el momento que se le asigna un valor y puede cambiar al asignarle un nuevo valor de un tipo diferente, a diferencia de los lenguajes con tipado estático, en los que, primero, se declara de qué tipo son los valores que pueden asignarse a una variable y, a partir de ahí, solo se le pueden asignar valores de ese tipo. Veamos las siguientes asignaciones en Python:

```
a = 0    # el tipo de a es int
b = 0.0  # el tipo de b es float
c = "0"  # el tipo de c es str
a = 0.0  # el tipo de a cambia a float
```

En Python, no obstante, es posible "anotar" el tipo de una variable, lo que en principio solo tiene un efecto informativo, equivalente a un comentario, acerca del tipo de valor que se espera que contenga:

```
a: int = 0    # el tipo de a es int
b: float = 0.0 # el tipo de b es float
c: str = "0"  # el tipo de c es str
```

A diferencia de los lenguajes con tipado estático, en Python, a una variable que haya sido anotada con un tipo se le puede asignar más tarde un valor de un tipo diferente; aunque hay herramientas que pueden analizar el código y señalar esas situaciones por si suponen un error potencial del programa al no ajustarse a la intención manifestada en la anotación.

Operadores de comparación

Estos operadores son aplicables a muchos tipos de datos; comparan los valores de sus dos operandos y devuelven un valor booleano en función de la relación entre ellos.

También se les llama **operadores relacionales**.

Supongamos, en el siguiente ejemplo, que la variable *a* tiene asociado el valor 10 y la variable *b* tiene asociado el valor 20:

operador	descripción	ejemplo
<code>==</code>	Si los valores de dos operandos son iguales, entonces la condición se vuelve verdadera.	<code>(a == b)</code> es falso
<code>!=</code>	Si los valores de dos operandos no son iguales, entonces la condición se vuelve verdadera.	<code>(a != b)</code> es verdad
<code>></code>	Si el valor del operando izquierdo es mayor que el valor del operando derecho, la condición se vuelve verdadera.	<code>(a > b)</code> es falso
<code><</code>	Si el valor del operando izquierdo es menor que el valor del operando derecho, la condición se vuelve verdadera.	<code>(a < b)</code> es verdad
<code>>=</code>	Si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, entonces la condición se vuelve verdadera.	<code>(a >= b)</code> es falso
<code><=</code>	Si el valor del operando izquierdo es menor o igual que el valor del operando derecho, entonces la condición se vuelve verdadera.	<code>(a <= b)</code> es verdad

Precedencia de operadores

Una expresión es una combinación de operandos y operadores. Por ejemplo:

3 + 8

La expresión anterior es una expresión simple, solo tiene un operador. En una expresión puede haber más de un operador; cuando esto ocurre, las reglas de precedencia determinan el orden de ejecución de los distintos operadores. Por ejemplo, la multiplicación tiene una precedencia mayor que la adición; como resultado, la expresión:

3 + 8 * 2

da 19, no 22, que sería lo que daría si los operadores se aplicaran en el orden en que están escritos (de izquierda a derecha). El orden de ejecución determinado por las reglas de precedencia se puede modificar usando paréntesis:

(3 + 8) * 2

El siguiente listado muestra el orden de precedencia predefinido entre los operadores de Python:

1. ()
2. **
3. +x, -x, ~x
4. *, /, //, %
5. +, -
6. <<, >>
7. &
8. ^
9. |
10. ==, !=, >, >=, <, <=, is, is not, in, not in
11. not
12. and
13. or

Asociatividad de operadores

La siguiente lista muestra la precedencia de los operadores en Python:

1. ()
2. **
3. +x, -x, ~x
4. *, /, //, %
5. +, -
6. <<, >>
7. &
8. ^
9. |
10. ==, !=, >, >=, <, <=, is, is not, in, not in
11. not
12. and
13. or

Como se puede observar, hay algunos operadores que tienen la misma precedencia. Cuando en una expresión se usan operadores con la misma precedencia, las reglas de asociatividad deciden el orden de ejecución. La mayoría de los operadores tienen asociatividad izquierda a derecha, es decir, se ejecutan en orden desde la izquierda:

$3 * 4 // 5$ es igual a $12 // 5$ que es igual a 2

Si la asociatividad de los operadores del grupo 4 de precedencia fuese derecha a izquierda, el resultado de la expresión anterior sería:

$3 * 4 // 5$ es igual a $3 * 0$ que es igual a 0

El operador de exponenciación **tiene** asociatividad de derecha a izquierda:

$2 ** 3 ** 2$ es igual a $2 ** 9$ que es igual a 512

Si tuviera asociatividad de izquierda a derecha el resultado sería:

$2 ** 3 ** 2$ es igual a $8 ** 2$ que es igual a 64

Los operadores de comparación no son asociativos en Python:

$a < b < c$

equivale a:

$a < b$ and $b < c$

Ejecución alternativa

Sentencias de control

Si un programa estuviese compuesto solo por una secuencia de instrucciones que se ejecutan, de la primera a la última, en el orden en que han sido escritas, no podría responder a situaciones en las se deben realizar distintas acciones dependiendo de condiciones ligadas a cada caso concreto del problema y que no se pueden conocer en el momento de escribir el programa.

Los lenguajes de programación proporcionan tanto instrucciones básicas como estructuras/**sentencias de control** que permiten variar la secuencia de instrucciones que se ejecuta para adaptarse a cada situación posible de un problema.

Existen sentencias de control que permiten elegir, en función de una condición, entre dos o más posibles "camino" por los que continuar la ejecución y, otras, que permiten repetir una subsecuencia de instrucciones un número determinado de veces, mientras se cumpla, o hasta que deje de cumplirse, una condición. En esta lección nos centraremos en las primeras.

Una **condición** es cualquier expresión que al ser evaluada devuelva un resultado booleano: Verdadero o Falso (**True** y **False** en Python).

Condiciones

Las sentencias de control suelen decidir las acciones a ejecutar en función de una condición, que es cualquier expresión que al ser evaluada devuelva un resultado booleano.

Para formar condiciones se usan con frecuencia los operadores relacionales, que son operadores que comparan los valores de sus dos operandos y devuelven un valor booleano en función de la relación entre ellos.

Supongamos, en el siguiente ejemplo, que la variable *a* tiene asociado el valor 10 y la variable *b* tiene asociado el valor 20:

operador	descripción	ejemplo
==	Si los valores de dos operandos son iguales, entonces la condición se vuelve verdadera.	(a == b) es falso
!=	Si los valores de dos operandos no son iguales, entonces la condición se vuelve verdadera.	(a != b) es verdad
>	Si el valor del operando izquierdo es mayor que el valor del operando derecho, la condición se vuelve verdadera.	(a > b) es falso
<	Si el valor del operando izquierdo es menor que el valor del operando derecho, la condición se vuelve verdadera.	(a < b) es verdad
>=	Si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, entonces la condición se vuelve verdadera.	(a >= b) es falso
<=	Si el valor del operando izquierdo es menor o igual que el valor del operando derecho, entonces la condición se vuelve verdadera.	(a <= b) es verdad

Se pueden formar condiciones más complejas combinando el resultado de condiciones simples usando operadores booleanos:

Operación Resultado

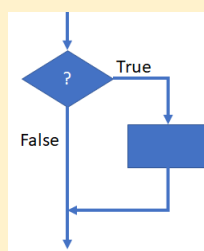
x or y	si x es False, y, si no, x
x and y	si x es True, y, si no, False
not x	si x es False, True, si no, False

Ejemplos de condiciones compuestas:

```
x > y and y < z
(x > y or x > z) and y > z
x % 2 == 0 and x > 2
```

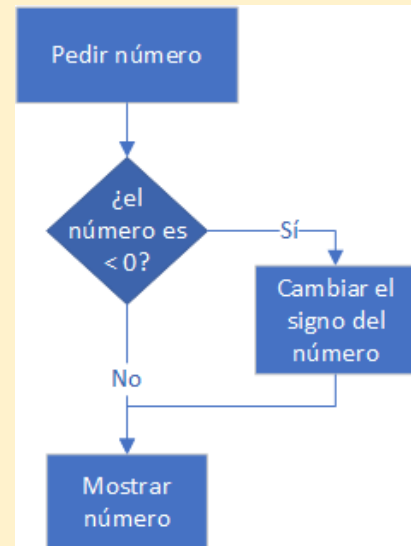
Sentencia if

En Python, la sentencia **if** permite decidir entre ejecutar, o no, una acción, o una secuencia de acciones, en función de una condición.



```
num = int(input("Dame un número entero: "))
if num < 0:
    num = -num

print("El valor absoluto del número es:",
num)
```



La sentencia `if` empieza con la palabra `if` seguida de la condición y dos puntos. Como sentencia de control, controla la ejecución de una o más instrucciones; en el ejemplo, únicamente de la instrucción `num = -num`, que se ejecutará, o no, en función de si la condición `num < 0` es verdadera o falsa.

La sentencia `if` se usa para decidir si se ejecuta, o no, un bloque de instrucciones. Si la decisión es negativa, se saltan y la ejecución continúa en la instrucción siguiente a la sentencia `if`; si se ejecutan, se alcanza ese mismo punto y se continúa.

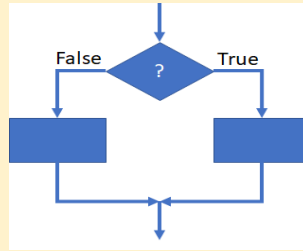
En todas las sentencias de control, las instrucciones que controla se empiezan a escribir a partir de la siguiente línea y aumentando en uno el nivel de sangrado respecto a la sentencia de control (con un salto de tabulación o, preferiblemente, cuatro espacios).

Nótese, que la instrucción `print` del final del ejemplo, no forma parte de las instrucciones controladas por el `if`, ya que está al mismo nivel de sangrado que éste, por lo que forma parte de la secuencia primaria del programa.

Cláusula `else`

La sentencia `if` se usa para decidir si se ejecuta, o no, un bloque de instrucciones. Si la decisión es negativa, se saltan y la ejecución continúa en la instrucción siguiente a la sentencia `if`; si se ejecutan, se alcanza ese mismo punto y se continúa.

Se puede añadir una cláusula `else` a una sentencia `if`. Una sentencia `if-else` se usa para decidir qué secuencia de instrucciones ejecutar entre dos posibles alternativas.

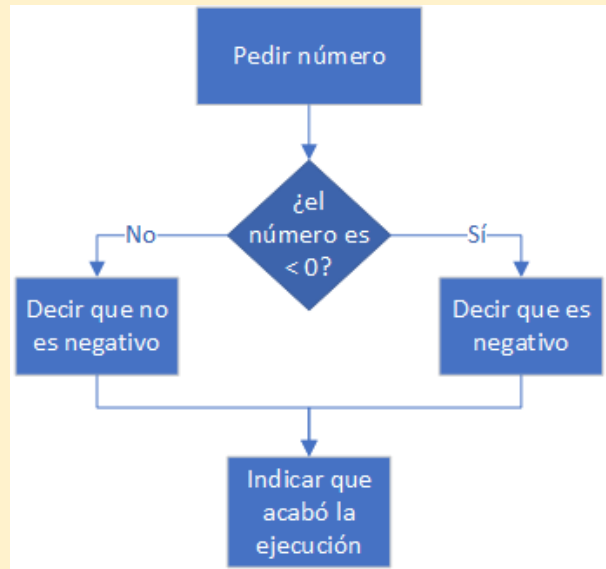


```

num = int(input("Dame un número entero: "))

if num < 0:
    print(num, "es un número negativo")
else:
    print(num, "es un número no negativo")

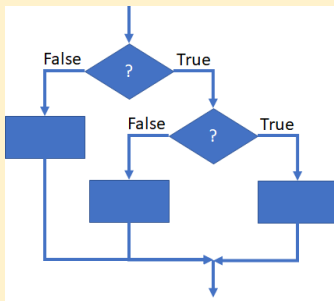
print("Fin de la ejecución")
  
```



En el ejemplo anterior, se muestra un mensaje diferente dependiendo de si num es un número negativo o no negativo.

Anidamiento

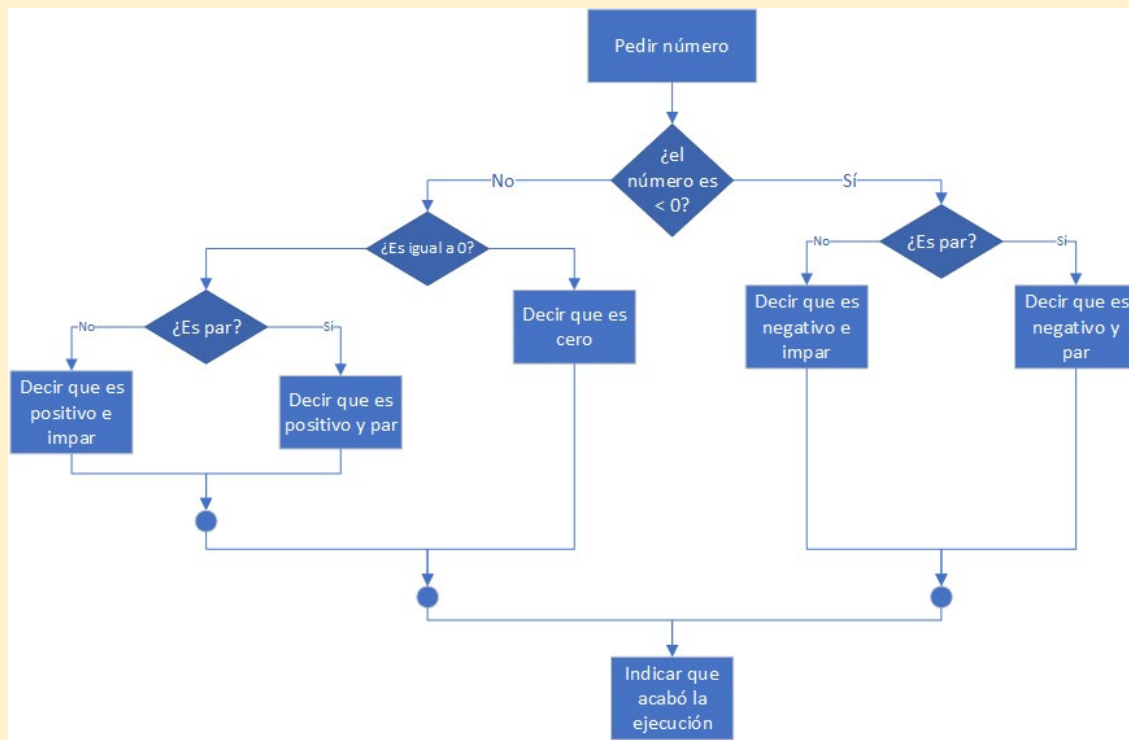
Las sentencias de control se pueden anidar, esto significa que una secuencia de instrucciones controlada por una sentencia de control puede contener a su vez una sentencia de control que controle una subsecuencia de instrucciones.



```

num = input("Dame un número entero: ")

if num < 0:
    if num % 2 == 0:
        print(num, "es un número negativo par")
    else:
        print(num, "es un número negativo impar")
else:
    if num == 0:
        print(num, "es cero")
    else:
        if num % 2 == 0:
            print(num, "es un número positivo par")
        else:
            print(num, "es un número positivo impar")
print("Fin de la ejecución")
  
```



En el ejemplo, se empieza evaluando si num es menor que cero; si lo es, se evalúa si es divisible por 2 (el resto de la división es cero), mostrando un mensaje adecuado en función de la respuesta.

Si el número no es menor que cero, se evalúa si es igual a cero o si es un número mayor que cero; en este último caso, se vuelve a evaluar si es par o impar.

Nótese, que al llegar al else, ya sólo queda una opción, ya que todas las otras opciones posibles han sido evaluadas y han resultado falsas, por lo que se puede realizar la opción correspondiente sin más preguntas.

Contracción elif

Cuando se encadenan varios else if.. anidados, se pueden contraer, facilitando la sintaxis para seleccionar entre múltiples casos:

```

num = int(input("Dame un número entero: "))
if num < 0:
    if num % 2 == 0:
        print(num, "es un número negativo par")
    else:
        print(num, "es un número negativo impar")
elif num == 0:
    print(num, "es cero")
elif num % 2 == 0:
    print(num, "es un número positivo par")
else:
    print(num, "es un número positivo impar")

print("Fin de la ejecución")
  
```

La cláusula `elif` equivale a la secuencia `else if...`

Nótese, que los `if` anidados dentro de la cláusula `if` de una sentencia `if-else` no se pueden abreviar de esta manera; sólo se pueden hacer con los anidados dentro de la cláusula `else`.

Funciones

Funciones

Una función es un trozo de código organizado para realizar una tarea determinada y con un nombre que permite llamarla (invocar su ejecución). Una función se ejecuta cuando se la llame, y puede llamársela las veces que haga falta.

```
def greetings_func():
    """Muestra un saludo"""
    print("Hola")
```

El ejemplo anterior muestra la definición de una función. La definición de una función consta de una cabecera y un cuerpo. La cabecera empieza con la palabra *def* seguida del nombre de la función. En el ejemplo, el nombre de la función es *greetings_func*. Al definir una función hay que poner paréntesis detrás del nombre. La cabecera de la función acaba con dos puntos.

El cuerpo de la función comienza en la línea siguiente a la cabecera y está desplazado un nivel de sangrado (un tabulador, o, preferiblemente, cuatro espacios) respecto a la misma. Está formado por cualquier combinación de instrucciones, incluyendo sentencias de control, con la única restricción de que deben realizar una tarea coherente.

Como primera línea del cuerpo de una función, justo debajo de la cabecera, es habitual poner un comentario breve en forma de una string encerrada entre comillas triples. Este comentario es una *docstring*, una string de documentación, que se usa para proporcionar información básica sobre la función. Esta información puede ser usada por las aplicaciones de programación para proporcionar ayuda automática y autocompletado de código al usar la función. Se puede acceder a la *docstring* de una función usando el atributo `__doc__`, como se muestra en el siguiente ejemplo:

```
print(greetings_func.__doc__)
```

El resultado del código anterior es:

```
Muestra un saludo
```

Las strings encerradas entre comillas triples pueden extenderse varias líneas:

```
"""Esto es un ejemplo de docstring multilínea.
   Las comillas de apertura y cierre
   deben estar alineadas.
   """
```

Para usar una función se la llama. La llamada a una función es como una instrucción más, que se forma poniendo el nombre de la función seguido de paréntesis:

```
def greetings_func():
    """Muestra un saludo"""
    print("Hola")
print("Antes de llamar a greetings_func")
greetings_func() # Llamada a greetings_func
print("después de llamar a greetings_func")
```

El resultado del código anterior es:

```
Antes de llamar a greetings_func
Hola
después de llamar a greetings_func
```

Funciones con parámetros

Las funciones pueden tener parámetros (variables) que funcionan como datos de entrada para la tarea a realizar. Por ejemplo:

```
def greetings_func(name):
    """Muestra un saludo
       Parameter name: str - nombre de la persona a saludar
    """
    print("Hola", name)
```

La función `greetings_func` tiene ahora un parámetro que se usa para incluir en el saludo el nombre de la persona a la que se saluda. Una función puede tener varios parámetros que se escriben dentro de los paréntesis que siguen al nombre de la función, formando la denominada lista de **parámetros formales**. La lista de parámetros formales indica qué datos espera recibir la función cuando se la llame para realizar su tarea.

```
def my_func(param1, param2, param3):
    ...
```

Al llamar a una función que tiene parámetros, hay que pasar valores para esos parámetros; estos valores se denominan **parámetros reales**, porque son aquellos sobre los que realmente se aplica la función, frente a los formales, que solo son indicadores de los parámetros que se necesitan:

```
name1 = "Pedro"
name2 = "Juan"
greetings_func(name1)
greetings_func(name2)
greetings_func("Elena")
```

Como muestra el ejemplo, los parámetros reales pueden ser variables, valores literales, o, en general, cualquier expresión que dé un resultado del tipo adecuado, como ocurre en el siguiente ejemplo con la instrucción `input`:

```
greetings_func(input("Hola ¿Cómo te llamas? "))
```

Si la función tiene varios parámetros, hay que pasar valores para todos ellos; en Python, esto se puede hacer por posición (un parámetro real se asocia con el parámetro formal que ocupa su misma posición en la lista de parámetros) o por nombre (un parámetro real se asocia explícitamente con el nombre de un parámetro formal). El paso por posición es el habitual en la mayoría de los lenguajes:

```
def my_func(param1, param2, param3):
    ...
my_func(1, 2, 3) # Paso por posición -> (param1 = 1, param2 = 2,
param3 = 3)
my_func(param2 = 1, param3 = 2, param1 = 3) # paso por nombre
```

Los parámetros formales pueden tener **valores por omisión**, que se usan si en la llamada se omite dar un valor para ese parámetro. Los parámetros que no tienen valores por omisión no pueden omitirse en la llamada.

```
def my_func(param1, param2, param3 = 0):
    ...
my_func(1, 2) # param1 = 1, param2 = 2; param3 toma el valor por
omisión (0)
```

Los parámetros que estén después de uno o más parámetros que se pasan por omisión tienen que pasarse obligatoriamente por nombre, ya que se pierde la relación posicional entre la lista de parámetros formales y la lista de parámetros reales.

Funciones que devuelven un resultado

Además de poder admitir datos como parámetros, las funciones pueden devolver un resultado al finalizar su tarea. Esto es coherente con que las funciones resuelven problemas, cuya proposición es hallar un dato desconocido a partir de otros datos conocidos, aplicando un algoritmo, que describe la solución de un problema en función de los datos necesarios para representar un caso concreto del problema y de los pasos necesarios para obtener el resultado deseado.

Para devolver un resultado, se utiliza la instrucción *return* seguida del dato a devolver:

```
def sum(a, b):
    """Suma los valores de sus parámetros"""
    result = a + b
    return result
```

El resultado devuelto por una función se puede usar directamente en una expresión:

```
print(sum(a, b))
...
if sum(a, b) > 5:
    ...
```

También se puede asignar a una variable para usarlo más tarde:

```
x = sum(a, b)
```


En realidad, en Python, todas las funciones devuelven un valor, solo que, las que no lo hacen explícitamente devuelven el valor *None*, que es el único valor del tipo de datos *NoneType* y se interpreta como "Ningún valor". Por ejemplo, el siguiente código:

```
def greetings_func():
    """Muestra un saludo"""
    print("Hola")

result = greetings_func();
print("Resultado devuelto por la función:", result);
```

produce el siguiente resultado:

```
Hola
Resultado devuelto por la función: None
```

La palabra "Hola" la escribe la función, que devuelve *None*. El valor *None* devuelto por la función es asignado a la variable *result* y mostrado por la instrucción *print* que se ejecuta en la siguiente línea.

Funciones polimórficas

Muchos lenguajes de programación exigen que se declaren los tipos de los datos que pueden asociarse a una variable o pasarse a un parámetro formal. Python tiene tipado dinámico, lo que significa que no requiere declaraciones de esta clase: el tipo de dato asociado a una variable se determina en el momento de crearla y un parámetro formal puede asociarse a un tipo de datos diferente en cada llamada. Esto, en el caso de las funciones, permite que sean polimórficas por definición. Una función como:

```
def sum(a, b):
    """Suma los valores de sus parámetros"""
    result = a + b
    return result
```

puede aplicarse a cualquier pareja de valores que admitan el uso del operador + entre ellos (una función polimórfica es aquella que se puede usar con distintos tipos de parámetros y/o distinto número de parámetros):

```
print(sum(1, 3))           # Números enteros
print(sum(3.5, 4.8))       # Números reales
print(sum(3.5, 4))         # Un número entero con un real
print(sum("Hola ", "mundo")) # Concatenación de strings
print(sum(3+2j, 5-1j))     # Números complejos
```

Resultado de la ejecución anterior:

```
4
8.3
7.5
Hola mundo
(8+1j)
```

Anotación de los tipos de los parámetros de una función

Aunque el polimorfismo proporcionado a Python por el tipado dinámico puede ser una potente herramienta (permite que una función se use en situaciones muy diferentes, sin tener que reescribir múltiples versiones), cuando no es nuestra intención escribir una función polimórfica puede ser útil explicitar qué tipo de dato se supone que debe recibir un parámetro formal, o qué tipo de resultado se supone que va a devolver una función:

```
def sum(a: int, b: int) -> int:
    """Suma los valores de sus parámetros"""
    result = a + b
    return result
```

Como se muestra en el ejemplo anterior, para los tipos más básicos (int, float, bool, str) basta con poner dos puntos detrás del parámetro y añadir el tipo esperado. Para otros tipos de datos, que se irán viendo en otras lecciones, puede ser necesario incluir alguna instrucción adicional. El tipo del resultado de la función se explicita después de una flecha añadida al final de la cabecera, antes de los dos puntos.

El primer efecto del tipado explícito es informativo. Normalmente, en la *docstring* de una función se pone información sobre sus parámetros, incluyendo de qué tipo se espera que sean, como en el siguiente ejemplo:

```
"""Suma los valores de sus parámetros
Parámetros:
a : (int) - Primer sumando
b : (int) - Segundo sumando
"""
```

Si se explicitan los tipos en la lista de parámetros formales y, además, se usan nombres adecuados, la *docstring* puede reducirse sustancialmente, ya que mucha información relevante está incluida en la propia cabecera de la función.

```
def sum(primer_sumando: int, segundo_sumando: int) -> int:
    """Suma los valores de sus parámetros"""
    result = primer_sumando + segundo_sumando
    return result
```

Además de esto, se pueden usar herramientas específicas que son capaces de, basándose en la información proporcionada por el tipado explícito, detectar por anticipado posibles errores que, de otro modo, sólo se manifestarían al ejecutar el programa.

Módulos y paquetes

Dónde escribir una función

Una función se puede escribir junto con el código que la usa, en el mismo archivo:

archivo `my_program.py`

```
def my_func(a: int, b: float) -> bool:
    return a > b
```

```
num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Pero un código puede llamar a una función escrita en otro archivo. Para ello, debe incluir previamente una cláusula de importación:

archivo my_program.py

```
import functions
num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(functions.my_func(num1, num2))
```

archivo functions.py

```
def my_func(a: int, b: float) -> bool:
    return a > b
```

Esta segunda aproximación independiza a la función del código que la usa, lo que permite que sea usada desde piezas de código escritas en diferentes archivos.

Módulos

La programación modular es una técnica de diseño de software que enfatiza la separación de la funcionalidad de un programa en módulos independientes e intercambiables, de manera que cada uno contiene todo lo necesario para ejecutar sólo un aspecto de la funcionalidad deseada. Por ejemplo, en un juego, podríamos tener, entre otros, un módulo para controlar la lógica del juego, otro para interactuar con el usuario y un tercero para mostrar el escenario de juego en la pantalla.

La programación modular permite gestionar la complejidad, dividiendo una funcionalidad compleja en otras más simples, y permite reusar el código, ya que los módulos deben ser independientes e intercambiables, es decir, un módulo desarrollado inicialmente para un programa podría usarse en otro y un programa que usa un módulo podría cambiarlo por otro mejor.

En Python, cada fichero con extensión .py alberga un módulo, que se llama como el archivo que lo contiene (sin la extensión). Para usar, desde un módulo distinto, las funcionalidades ofrecidas por un módulo, solo hay que poner su nombre en una cláusula de importación:

módulo my_program.py

```
import functions
num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(functions.my_func(num1, num2))
print(functions.pi)
```

módulo functions.py

```
pi = 3.14
```

```
def my_func(a: int, b: float) -> bool:
    return a > b
```

El módulo `functions` del ejemplo anterior ofrece dos elementos que se pueden utilizar: la variable `pi` y la función `my_func`. Ambos, se pueden utilizar, una vez importado el módulo, escribiendo su nombre precedido del nombre del módulo separado por un punto.

```
print(functions.my_func(num1, num2))
print(functions.pi)
```

Un módulo proporciona un *namespace*, un espacio de nombres en el que podemos declarar elementos sin entrar en conflicto con elementos de otros módulos que puedan tener el mismo nombre, ya que se diferenciarían al prefijarlos con el nombre del módulo (por ejemplo, podríamos usar en el mismo programa dos módulos que declarasen una función llamada *my_func* sin que se confundan).

En el caso de que un módulo ofrezca varios elementos pero sólo necesitemos usar algunos, los elementos que queramos se pueden importar individualmente, como muestra el siguiente ejemplo:

módulo `my_program.py`

```
from functions import my_func

num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Los elementos importados directamente no pueden ser prefijados con el nombre del módulo a la hora de usarlos. Para evitar conflictos si queremos usar elementos con el mismo nombre de distintos módulos, podemos, o importar los módulos, para poder prefijar los elementos que queremos usar, o usar alias:

```
from module1 import suma as suma1
from module2 import suma as suma2
```

También, se pueden usar alias al importar un módulo completo, generalmente, para usar un nombre más corto y manejable:

```
import functions as funcs
```

El módulo `__main__`

Un módulo puede incluir secuencias de instrucciones que no forman parte de ninguna función:

módulo `functions.py`

```
def my_func(a: int, b: float) -> bool:
    return a > b

print("Esto no forma parte de una función")
```

Estas instrucciones "libres" se ejecutan siempre que el módulo es cargado, a diferencia de las [funciones](#), que sólo se ejecutan cuando se las llama:

módulo my_program.py

```
from functions import my_func

radi = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Resultado:

```
Esto no forma parte de una función
Entra un número entero: 1
Entra un número entero: 2
False
```

Generalmente, no queremos que este código "libre" se ejecute cuando el módulo es importado por otro. Para evitarlo, añadimos una instrucción para ver si el módulo responde al nombre "`__main__`".

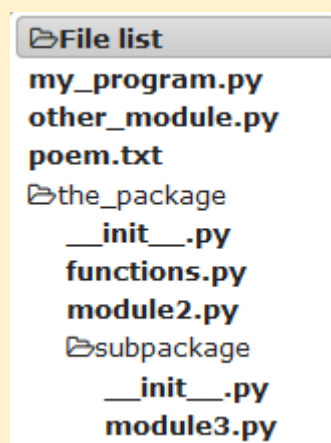
módulo functions.py

```
def my_func(a: int, b: float) -> bool:
    return a > b

if __name__ == "__main__":
    print("Esto no forma parte de una función")
```

El nombre "`__main__`" (nótese los dos guiones bajos a cada lado) es asignado automáticamente al módulo que inicia la ejecución de un programa: De esta manera, el código controlado por la sentencia "`if __name__ == '__main__':`" sólo se ejecuta si el módulo en que está es ejecutado directamente, pero no, cuando es importado por otro que se empezó a ejecutar primero.

Paquetes



En programación modular, el concepto "paquete" (en inglés, package) suele usarse para referirse a una agrupación lógica de módulos: distintas funcionalidades se implementan en diferentes módulos, que pueden incluir [funciones](#), variables, y otros elementos relacionados, y un conjunto de módulos con funcionalidades relacionadas se agrupan en un paquete.

En Python, al igual que un módulo se implementa en un archivo, un paquete, que es un conjunto de módulos, y por tanto implica un conjunto de archivos, se corresponde, físicamente, con una carpeta o directorio en el sistema de ficheros.

Para que un directorio sea un paquete debe incluir un "archivo de configuración" llamado `__init__.py`, que puede estar vacío. El archivo `__init__.py` sirve para controlar el acceso a los módulos contenidos en el paquete desde módulos externos.

La figura muestra una estructura formada por dos módulos (`my_program` y `other_module`) en el paquete que no forman parte de ningún paquete, un paquete (`the_package`) con dos módulos más (`functions` y `module2`) y un subpaquete (`subpackage`) con un único módulo (`module3`).

Al importar un módulo, hay que especificar el paquete al que pertenece, como se hace en el siguiente ejemplo:

módulo `my_program.py`:

```
from the_package.functions import my_func

num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Tuplas

Tuplas

Los tipos básicos de datos (`int`, `float`, `bool`) representan valores monolíticos, que no se pueden separar en componentes individuales con entidad propia. Por su parte las strings (`str`) representan valores compuestos, formados por una secuencia de caracteres, aunque el sentido de lo que representan suele venir dado por el conjunto, y no por los caracteres individuales. Los números complejos (`complex`) también son valores compuestos, formados por una pareja de números reales.

Hay muchas clases de información que se pueden entender como formadas por datos que son un agregado de datos más simples con entidad propia. Por ejemplo, un nombre español completo está formado habitualmente por tres strings: nombre de pila, primer apellido y segundo apellido; un punto en el espacio bidimensional se representa por dos coordenadas, que son números reales; una fecha está formada por día, mes y año; etc.

En Python, este tipo de información, compuesta por un agregado finito de datos, se puede representar de manera unitaria usando tuplas (tipo `tuple`). Podemos definir una *n*-tupla como una secuencia de *n* valores. En Python, una tupla se forma encerrando una secuencia de valores, separados por comas, entre paréntesis.

```
name = ("Pedro", "Martel", "Parrilla")
point2D = (3.5, 8.3)
vector3D = (3.5, 8.2, 4.6)
dni = "42387423H"
age = 16
address = "Calle del Pino, nº 2. Las Palmas."
person_id = (dni, name, age, address)
```

El ejemplo anterior define cuatro tuplas, referenciadas como: `name`, `point2D`, `vector3D` y `person_id`. Las variables `dni` y `address` referencian strings, no tuplas, y la variable `age` referencia un valor de tipo `int`. La variable `person_id` ilustra que:

1. Los elementos de una tupla pueden ser de tipos diferentes (`name` está formada solo por strings y `point2D` y `vector3D` solo por números reales, mientras que `person_id` está formada por dos strings (`dni` y `address`), un número (`age`) y una tupla (`name`)
2. Una tupla puede estar formada por elementos de cualquier tipo, incluidas tuplas

En el caso de que queramos formar una tupla con un solo elemento, debemos poner una coma detrás del elemento:

```
name = ("Pedro",)
```

Acceso a los elementos individuales de una tupla

Una tupla es una secuencia finita de valores:

```
name = ("Pedro", "Martel", "Parrilla")
```

Una secuencia establece un orden en el sentido de que cada elemento ocupa una posición concreta:

```
Posiciones:   1ª           2ª           3ª
name          = ("Pedro", "Martel", "Parrilla")
```

En Python, las posiciones de una secuencia se identifican mediante índices secuenciales, que son números enteros. El índice de la primera posición es el 0.

```
Posiciones:   1ª           2ª           3ª
name          = ("Pedro", "Martel", "Parrilla")
Índices      :    0           1           2
```

Para acceder a un elemento individual, basta con poner el nombre de la tupla seguido del índice de la posición a la que queremos acceder encerrado entre corchetes.

```
first  = name[0] # Pedro
second = name[1] # Martel
third  = name[2] # Parrilla
```

El índice de la última posición es igual a la longitud de la tupla (número de elementos) menos 1. Sabiendo esto, también se puede acceder a los elementos de una tupla usando índices negativos relativos a la longitud de la tupla.

```
first  = name[-3] # Pedro
second = name[-2] # Martel
third  = name[-1] # Parrilla
```

Operaciones con tuplas

De una tupla se puede conocer su longitud (número de elementos) usando la función predefinida *len*:

```
name = ("Pedro", "Martel", "Parrilla")
len_name = len(name) # a len_name se le asigna el valor 3
```

Se pueden [concatenar](#) ("sumar") dos tuplas:

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla3 = tupla1 + tupla2 # -> (1, 2, 3, 4, 5, 6)
```

Se puede "multiplicar" una tupla por un número positivo, lo que equivale a concatenarla tantas veces como indique el número:

```
tupla1 = (1, 2, 3)
tupla2 = tupla1 * 3 # -> (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Si se "multiplica" por un número negativo o 0, se obtiene la tupla vacía: ().

Se pueden comparar dos tuplas usando los operadores relacionales. La comparación se realiza comparando los elementos de principio a fin; es menor la que primero tenga un elemento menor, o, si todos los elementos son iguales hasta que se acaba una, la que tenga menos elementos.

```
tupla1 = (1, 2, 3)
tupla2 = (1, 3, 4)

iguales          = tupla1 == tupla2 # False
menor_la_1       = tupla1 < tupla2  # True
mayor_o_igual_la_1 = tupla1 >= tupla2 # False
```

Operaciones con tuplas (2)

Se puede formar una tupla a partir de un segmento de otra:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[3:6] # -> (52, 26, 17)
```

El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[3:] # -> (52, 26, 17, 81, 29)
```

Si se omite el primer valor, se toma el segmento desde el principio hasta la posición anterior a la indicada por el índice presente:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[:6] # -> (10, 21, 13, 52, 26, 17)
```

Si se omiten ambos índices, el segmento coincide con la tupla original completa:


```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[:] # -> (10, 21, 13, 52, 26, 17, 81, 29)
```

Se puede añadir un tercer valor para indicar un "paso" que permite saltarse elementos. En el siguiente elemento se toman los elementos desde el principio saltando de dos en dos hasta el final:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[::2] # -> (10, 13, 26, 81)
```

Nótese, que las operaciones de cálculo de la longitud (*len*) concatenación (+), repetición (*) y segmentación ([:]) son aplicaciones aplicables igualmente a las strings, porque, en realidad, **son operaciones de tratamiento de secuencias y tanto las strings como las tuplas son dos clases de secuencias**, en un caso, secuencias de caracteres y, en el otro, secuencias de elementos cualquiera.

Anotación de variables tupla

Para anotar que una variable o un parámetro es de un tipo no básico, hay que importarlo desde el módulo **typing**:

```
from typing import Tuple
tupla: Tuple # La variable Tupla se anota como una tupla (tipo tuple)
vector: Tuple[float] # La variable Vector se anota como una tupla de elementos de tipo float
```

A partir de la versión 3.9 de Python, se puede anotar usando el nombre real del tipo (en minúsculas):

```
tupla: tuple # La variable Tupla se anota como una tupla (tipo tuple)
vector: tuple[float] # La variable Vector se anota como una tupla de elementos de tipo float
```

Iteración – while

Repetición

A veces, un algoritmo requiere repetir una acción o conjunto de acciones varias veces seguidas. Por ejemplo, supongamos que queremos que un programa muestre diez líneas con la palabra "Hola". Es fácil:

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

La repetición, o iteración es bastante frecuente, por lo que los lenguajes ofrecen estructuras para expresarlas de un modo más conciso:

```
líneas_escritas = 0
líneas_a_escribir = 10
while líneas_escritas < líneas_a_escribir:
    print("Hola")
    líneas_escritas = líneas_escritas + 1
```

La sentencia *while* expresa la repetición de una secuencia de instrucciones mientras se cumpla una condición. Las instrucciones deben estar diseñadas de tal manera que busquen que la condición deje de cumplirse; si esto no ocurre, se entraría en un bucle infinito. En el ejemplo anterior esto se consigue incrementando la variable *líneas_escritas*, que actúa como contador de las líneas escritas; de esta forma, en algún momento llegará a igualar su valor con el de la variable *líneas_a_escribir*, que define el objetivo del problema; la condición dejará de cumplirse y cesará la repetición, continuando la ejecución en la línea siguiente del mismo nivel que el *while*, si la hubiera, que no es el caso.

Repetición (2)

Una sentencia de repetición, como *while*, permite expresar la repetición de una secuencia de instrucciones de forma concisa.

```
líneas_escritas = 0
líneas_a_escribir = 10
while líneas_escritas < líneas_a_escribir:
    print("Hola")
    líneas_escritas = líneas_escritas + 1
```

Sin embargo, la mayor virtud de una sentencia de repetición no es la concisión, sino la flexibilidad que proporciona. El siguiente ejemplo es una variante del anterior en la que el objetivo de líneas a escribir no depende de un valor fijo, sino que puede variar en cada ejecución, produciendo resultados diferentes en cada caso, sin tener que modificar el algoritmo:

```
líneas_escritas = 0
líneas_a_escribir = int(input("¿Líneas a escribir? "))
while líneas_escritas < líneas_a_escribir:
    print("Hola")
    líneas_escritas = líneas_escritas + 1
```

En el caso de que el usuario introduzca un valor igual o menor que cero, no se escribirá ninguna línea, ya que la condición no se cumple desde el principio; en cualquier otro caso, se mostrarán tantas líneas como haya indicado el usuario.

Esquema básico de iteración

La iteración expresada por la sentencia *while* responde a un esquema básico bastante simple:

```
inicializaciones
while condición:
    Hacer algo (Tratamiento)
    avanzar
```

```

líneas_escritas = 0 # Inicializaciones
líneas_a_escribir = int(input("¿Líneas a escribir? "))

while líneas_escritas < líneas_a_escribir:
    print("Hola") # Tratamiento
    líneas_escritas = líneas_escritas + 1 # Avance

```

Las inicializaciones, la condición, lo que se hace en cada iteración y cómo se avanza para preparar la siguiente pueden tener muchas variantes, pero siempre es posible reconocer el esquema.

Ejemplo 1:

```

current = 2
last = 15
sum_evens = 0
while current < last:
    sum_evens += current
    current += 2

```

Ejemplo 2:

```

penúltimo = 1
último = 1
actual = último
objetivo = 15
iteración = 1
while iteración <= objetivo:
    actual = último + penúltimo
    penúltimo = último
    último = actual
    iteración += 1

```

Ejemplo 3:

```

num1 = 128
num2 = 96
while num2 > 0:
    mod = num1 % num2
    num1 = num2
    num2 = mod

```

Iteración – for

Sentencia for

Un esquema de iteración consiste en repetir una secuencia de acciones mientras se cumpla una condición. En una variante común del esquema, la condición deja de cumplirse cuando se han realizado un número de iteraciones que puede calcularse a priori con facilidad. El siguiente ejemplo muestra, de menor a mayor, los números enteros positivos menores que uno dado (límite), para lo que el bucle debe iterar límite - 1 veces:

```

actual = 1
límite = int(input("Dame un valor entero positivo: "))

```

```
while actual < límite:
    print(actual)
    actual = actual + 1
```

Dada la frecuencia de esta variante, existe una sentencia de repetición específica, que permite expresarla de forma más concisa que la sentencia `while`:

```
límite = int(input("Dame un valor entero positivo: "))

for actual in range(1, límite):
    print(actual)

líneas_a_escribir = int(input("¿Líneas a escribir? "))

for líneas_escritas in range(líneas_a_escribir):
    print("Hola")
```

Básicamente, una variable de control (*actual*, *líneas_escritas*) va tomando secuencialmente valores de un rango y, cada vez que toma un nuevo valor, se ejecutan las instrucciones anidadas en la sentencia *for*. Nótese, que si se pone un solo valor, como en el ejemplo de *líneas_a_escribir*, el rango va desde cero hasta el valor previo al especificado.

```
range(10)      -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(1, 10)  -> 1, 2, 3, 4, 5, 6, 7, 8, 9
```

La función *range* define, en principio, una secuencia de números enteros consecutivos, pero admite un tercer parámetro que permite establecer un "paso" o distancia entre los valores sucesivos. El siguiente ejemplo muestra los números impares positivos menores que 15, por lo que la variable de control va saltando de 2 en 2 a partir del valor inicial:

```
for actual in range(1, 15, 2):
    print(actual, end = ', ')
```

Se muestra: 1, 3, 5, 7, 9, 11, 13,

El paso puede ser negativo, siempre que el rango esté invertido. En el siguiente ejemplo, se muestran, de mayor a menor, los números impares positivos menores o igual que uno dado (el valor inicial es *top*, o *top - 1*, si *top* es par, y la variable de control avanza de -2 en -2):

```
top = int(input("Dame un número entero positivo: "))
if top % 2 == 0: # Nos aseguramos de empezar con valor impar
    top -= 1
for actual in range(top, 0, -2):
    print(actual)
```

Si se introduce en la entrada el 8, se muestra:

```
7
5
3
1
```

Equivalencia entre for y while

La sentencia *for* es más específica que la *while*; es decir, todo lo que se puede hacer usando la sentencia *for* se puede hacer, aunque de forma menos concisa, usando la sentencia *while*. El siguiente ejemplo muestra la equivalencia entre la sentencia *for* y la sentencia *while*:

Sentencia for	Sentencia while
<pre>for value in range(first, last, step): hacer algo</pre>	<pre>value = first while value < last: hacer algo value += step</pre>

La principal diferencia entre ambas versiones es que en el *for* la inicialización de la variable de control y su incremento al final de la iteración son automáticos, mientras que en el *while* hay que indicarlos de forma explícita.

Todo lo que se puede hacer con un *for* se puede hacer con un *while*, pero la recíproca no es cierta. Por ejemplo, el bucle *while* del [siguiente](#) ejemplo no puede sustituirse por un *for* ya que no se conoce por adelantado el rango de valores que va a tomar la variable *num2* hasta llegar a cero (calcularlo costaría tanto como ejecutar el algoritmo):

```
num1 = int(input("Dame un número entero positivo: "))  
num2 = int(input("Dame un número entero positivo menor que el  
anterior: "))  
while num2 > 0:  
    mod = num1 % num2  
    num1 = num2  
    num2 = mod
```

La sentencia for y el tratamiento de secuencias

La función *range* del siguiente ejemplo define una secuencia de números enteros que debe ir tomando la variable de control del bucle *for*:

```
for actual in range(1, 15):  
    print(actual)
```

En realidad, la sentencia *for* admite cualquier clase de secuencia. En el siguiente ejemplo se muestran los días de la semana que han sido almacenados previamente en una tupla:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado",  
        "domingo")  
  
for day in days:  
    print(day)
```

El siguiente ejemplo cuenta el número de letras 'e' que hay en la string referenciada por la variable *text*:

```
text = "Dame un número entero positivo: "  
count = 0  
  
for char in text:  
    if char == 'e':  
        count += 1  
  
print(count)
```

El bucle *for* requiere que una variable de control recorra una secuencia de valores; da igual el tipo de secuencia de que se trate.

Acceso a los índices al iterar sobre una secuencia

Nótese, que al iterar sobre una secuencia se va accediendo a sus elementos pero no se tiene información sobre la posición que ocupan dichos elementos en la secuencia.

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado",  
        "domingo")  
  
for day in days:  
    print(day)
```

Se muestra:

```
lunes  
martes  
miércoles  
jueves  
viernes  
sábado  
domingo
```

Si fuese necesario conocer el índice de los elementos, se puede usar la función *enumerate*, e iterar usando una pareja de variables:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado",  
        "domingo")  
  
for num_day, day in enumerate(days):  
    print(num_day, "-", day)
```

Se muestra:

```
0 - lunes  
1 - martes  
2 - miércoles  
3 - jueves  
4 - viernes  
5 - sábado  
6 - domingo
```

Básicamente, lo que hace *enumerate* es tomar un objeto iterable (que se puede "recorrer") y añadirle un contador, devolviendo una nueva secuencia de elementos formados por parejas índice, valor.

Se puede obtener el mismo resultado iterando sobre el rango de índices de la secuencia a recorrer, como en el siguiente ejemplo:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")

for num_day in range(len(days)):
    print(num_day, "-", days[num_day])
```

Esquemas comunes de tratamiento de secuencias

Esquemas de recorrido y acumulación

Muchísimos problemas requieren el tratamiento de secuencias de valores aplicando una acción de forma repetida a cada uno de los valores de la secuencia, lo que se hace siguiendo los esquemas básicos determinados por un bucle *while* o un bucle *for*:

Bucle for	Bucle while
<pre>for item in range numbers: Hacer algo</pre>	<pre>Inicializaciones while condición: Hacer algo Avanzar</pre>

Sobre este esquema básico se pueden hacer múltiples variaciones. A modo de ilustración, el Ejemplo 1 muestra un **esquema de recorrido** y el Ejemplo 2 un **esquema acumulador**.

Ejemplo 1: esquema de recorrido

Bucle for	Bucle while
<pre>numbers = (21, 13, 24, 42, 12) for item in numbers: print(item)</pre>	<pre>numbers = (21, 13, 24, 42, 12) index = 0 while index < len(numbers): print(numbers[index]) index += 1</pre>

Ejemplo 2: esquema de acumulación

Bucle for	Bucle while
<pre>numbers = (21, 13, 24, 42, 12) addition = 0 for item in numbers: addition += item</pre>	<pre>numbers = (21, 13, 24, 42, 12) index = 0 addition = 0 while index < len(numbers): addition += numbers[index] index += 1</pre>

Bucle for

```
print (addition)
```

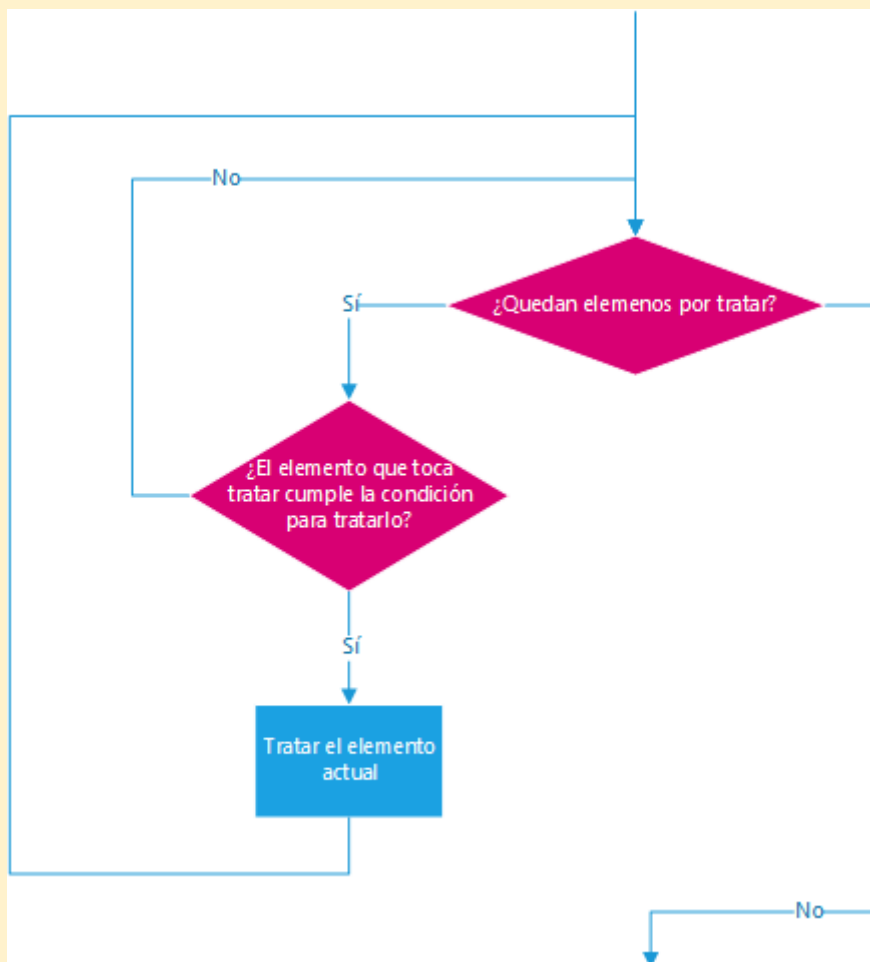
Bucle while

```
print (addition)
```

La diferencia entre el primer y segundo esquema es que el de recorrido trata una secuencia aplicando una acción a cada uno de sus elementos, acción que sólo depende del elemento en cuestión y no de ningún otro elemento de la secuencia, mientras que en el esquema de acumulación la acción va combinando los elementos para obtener un valor final, una vez acabado el bucle. Habitualmente, se asocia el esquema de acumulación con cualquier operación que tenga elemento neutro (para inicializar el proceso) y sea asociativa, como la [suma](#), el producto, o la concatenación de strings; pero podríamos generalizarlo para cubrir cualquier algoritmo en que se calcule un dato en función del conjunto de elementos de una secuencia, combinando los valores de los elementos de alguna forma.

Combinando la estructura de repetición con otras sentencias de control se pueden obtener esquemas más sofisticados.

Esquema de selección



Combinando un bucle con una sentencia de selección, se obtiene un **esquema de selección** como el del

siguiente ejemplo, que muestra sólo los valores impares de una tupla de números enteros.

Bucle for

```
numbers = (21, 13, 24, 42, 12)

for number in numbers:
    if number % 2 == 1:
        print(number)
```

Bucle while

```
numbers = (21, 13, 24, 42, 12)
index = 0
while index < len(numbers):
    number = numbers[index]
    if number % 2 == 1:
        print(number)

    index += 1
```

Un esquema de selección, como su nombre indica, aplica una acción sólo a los elementos de la secuencia que cumplen una condición determinada. También podría considerarse la variante de aplicar un tratamiento a los valores que cumplen una condición y otro distinto, a los que no la cumplen.

Bucle for

```
numbers = (21, 13, 24, 42, 12)

for number in numbers:
    if number % 2 == 1:
        print(number)

    else:
        print("*")
```

Bucle while

```
numbers = (21, 13, 24, 42, 12)
index = 0
while index < len(numbers):
    number = numbers[index]
    if number % 2 == 1:
        print(number)
    else:
        print("*")

    index += 1
```

Esquema de búsqueda

El siguiente ejemplo localiza el primer número par en una tupla de números enteros:

Bucle for

```
numbers = (19, 12, 35, 21, 12, 41, 37)

found = None

for number in numbers:

    if number % 2 == 0:

        found = number

        break
```

Bucle while

```
numbers = (19, 12, 35, 21, 12, 41, 37)
index = 0
found = None

while index < len(numbers):
    number = numbers[index]
    if number % 2 == 0:
        found = number
        break

    index += 1
if found != None:
    print("El primer número par es:", found)
else:
```

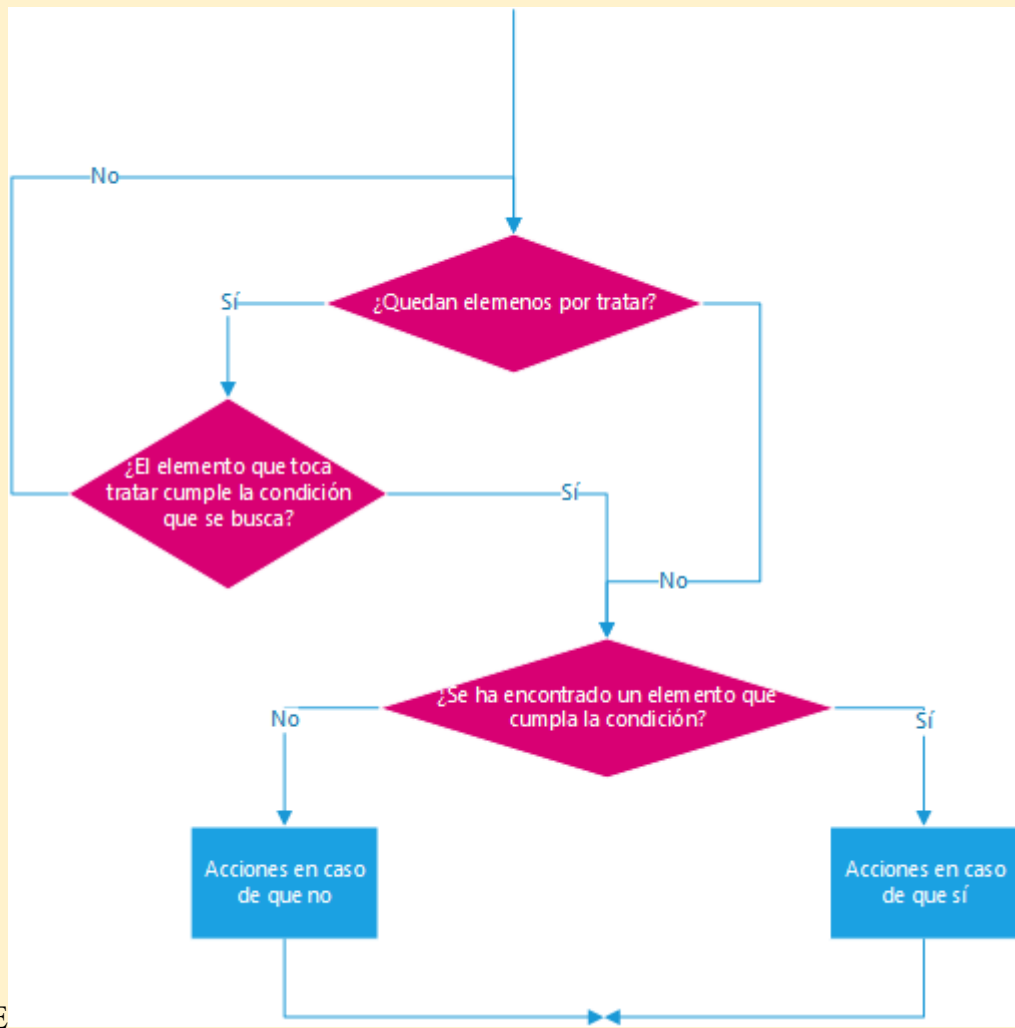
Bucle for

```
if found != None:
    print("El primer número
    par es:", found)

else:
    print("No hay números
    pares en la tupla")
```

Bucle while

```
print("No hay números pares
    en la tupla")
```



El ejemplo sigue un **esquema de búsqueda**, que es similar a un esquema de selección en que incluye una sentencia *if* para evaluar si el elemento actual de la secuencia cumple una condición pero se diferencia en la instrucción *break*, que interrumpe la condición del bucle cuando se cumple la ejecución.

Un esquema de búsqueda avanza mientras queden elementos que examinar y no se haya encontrado ninguno que cumpla las condiciones de la búsqueda. El bucle puede terminar, bien porque se acaba la secuencia sin encontrar ningún elemento que cumpla las condiciones de búsqueda, bien porque se ha encontrado un elemento que las cumpla. Si después del bucle es necesario saber si se ha encontrado o no, habrá que usar una sentencia *if* para discernirlo.

Cuando se usa un *while*, el esquema de búsqueda puede evitar el *if* y el *break* usando una condición doble en el *while*:

```
numbers = (19, 13, 35, 21, 13, 41, 37)
index = 0
while index < len(numbers) and numbers[index] % 2 != 0:
    index += 1
if index < len(numbers):
    print("El primer número par es:", numbers[index])
else:
    print("No hay números pares en la tupla")
```

Alteración del flujo de un bucle

Existen dos instrucciones que permiten alterar el flujo de ejecución de un bucle; ambas son aplicables a cualquier tipo de bucle, sea *while* o *for*. La instrucción *break* interrumpe la ejecución del bucle terminándolo de forma anticipada. El siguiente ejemplo muestra cada elemento de la tupla *numbers* emparejado con su sucesor en la secuencia de números enteros, hasta que se encuentre el primer elemento par, en cuyo caso se imprime la palabra *end* y se interrumpe el bucle, no realizándose el resto de la iteración actual (mostrar el sucesor) ni tratándose el resto de los elementos de la secuencia.

```
numbers = (19, 21, 35, 21, 12, 41, 28, 37)

for element in numbers:
    print(element, end = "-")

    if element % 2 == 0:
        print("end")
        break

    print(element + 1)
```

El resultado es:

```
19-20
21-22
35-36
21-22
12-end
```

La instrucción *continue*, por el contrario, termina la iteración actual, sin ejecutar las instrucciones que falten, pero no termina el bucle, sino que pasa a la iteración siguiente.

```
numbers = (19, 21, 35, 21, 12, 41, 37)

for element in numbers:
    print(element, end = "-")

    if element % 2 == 0:
        print()
        continue
```

```
print(element + 1)
```

El resultado del ejemplo anterior es:

19-20

21-22

35-36

21-22

12-

41-42

37-38

Depuración de programas

Depuración

Al escribir un programa se pueden cometer errores sintácticos, errores de escritura que impiden que el programa se ejecute. Un error sintáctico es, por ejemplo, no poner dos puntos al final de la cabecera de una función.

Los errores sintácticos son fáciles de arreglar, porque sólo hay que cumplir las reglas del lenguaje. Una vez que un programa está correctamente escrito de acuerdo con dichas reglas, se puede ejecutar, pero entonces pueden aparecer errores de funcionamiento.

Es habitual que un programa mínimamente complejo funcione bien la mayoría de las veces, pero que, para unos pocos casos, falle dando resultados erróneos, o incluso, aborte (se detenga bruscamente sin terminar su tarea).

El ciclo de desarrollo de un programa pasa por diferentes fases, las primeras son: análisis de los requerimientos y características del problema a resolver, diseño de la solución y codificación de la solución en un lenguaje de programación. Una vez que la solución está codificada, tenemos un programa y se pasa a la fase de prueba, en la que se somete al programa a una batería de casos de prueba con el fin de encontrar algún(os) caso(s) en que falle.

Si las pruebas detectan un caso para el que el programa falla, se pasa a la fase de depuración que consiste en analizar el código para buscar, normalmente con ayuda de herramientas específicas, cuál es la causa del fallo: la depuración es el proceso de buscar y solucionar defectos en el código de un programa que provocan que funcione incorrectamente.

Una herramienta de depuración (depurador o debugger) tiene que ofrecer funcionalidades para detener y reanudar la ejecución de un programa y, cuando está detenido, examinar el estado del proceso que lleva a cabo, inspeccionando los valores de las variables involucradas, la secuencia de llamadas a [funciones](#) realizada, el contexto de la instrucción actual, etc.

El depurador proporciona datos sobre el funcionamiento del programa que el programador tiene que analizar, razonando sobre ellos, para encontrar las causas del mal funcionamiento detectado por las pruebas.

Iniciando la depuración en Python

Python dispone de un módulo, llamado *pdb* (**p**ython **d**ebugger), que sirve como herramienta de depuración. Para iniciar la depuración de un programa, sólo hay que importar ese módulo y ejecutar la instrucción *set_trace()*, como se hace en las dos primeras líneas del siguiente ejemplo:

```
1. import pdb
2. pdb.set_trace()
3. import random
4. def find(numbers, searched):
5.     found = False
6.     for number in numbers:
7.         if number == searched:
8.             found = True
9.             break
10.    return found
11.
12. if __name__ == '__main__':
13.     numbers = (19, 12, 35, 21, 12, 41, 37)
14.     number = random.randint(10, 50)
15.
16.     if find(numbers, number):
17.         print("El número", number, "está en ", numbers)
```

Esta instrucción hace que se arranque el depurador y se quede esperando órdenes para continuar la ejecución del programa, que queda detenido justo en la línea siguiente:

```
> /home/p16251/main.py(3)<module>()
-> import random
(Pdb)
```

Obsérvese que, justo antes del prompt del depurador (Pdb) hay una flecha que indica que el programa está detenido en la línea cuyo contenido es "import random", la línea 3 de nuestro programa de ejemplo. Un comando muy útil en este momento, es *help*, que nos mostraría todos los comandos del depurador:

```
> /home/p13011/main.py(3)<module>()
-> import random
(Pdb) help
Documented commands (type help <topic>):
=====
EOF      c          d          h          list        q          rv         undis
play     a          cl         debug      help        ll         quit       s         unt
alias    clear      disable    ignore      longlist    r          source     until
args     commands  display    interact    n           restart    step       up
b        condition down       j          next        return     tbreak     w
```

```

break  cont      enable  jump    p      retval  u      whati
s
bt      continue  exit    l      pp     run     unalias where

Miscellaneous help topics:
=====
exec  pdb
(Pdb)

```

help seguido de un comando concreto, nos da ayuda sobre ese comando en particular.

Breakpoints (puntos de parada)

Una cosa que podemos hacer cuando iniciamos la depuración, es establecer **puntos de parada** (*breakpoints*); de esta manera, podemos hacer que la ejecución se vaya deteniendo en distintos puntos del programa, permitiéndonos inspeccionar el estado del proceso en cada momento. En el siguiente ejemplo se ponen dos puntos de parada en las líneas 6 y 14 de nuestro programa de ejemplo:

```

> /home/p17284/main.py(3)<module>()
-> import random
(Pdb) b 6
Breakpoint 1 at /home/p17284/main.py:6
(Pdb) b 14
Breakpoint 2 at /home/p17284/main.py:14
(Pdb)

```

Para poner un punto de parada se puede usar el comando *break*, o su forma abreviada, *b*, como en el ejemplo.

Avance

Cuando un programa está detenido en un punto de parada, o al inicio de la depuración, se puede **avanzar hasta el siguiente punto de parada**, si existe, usando el comando *continue*, o sus versiones abreviadas, *cont* o *c*:

```

> /home/p14373/main.py(3)<module>()
-> import random
(Pdb) b 14
Breakpoint 1 at /home/p14373/main.py:15
(Pdb) c
> /home/p14373/main.py(14)<module>()
-> numbers = (19, 12, 35, 21, 12, 41, 37)
(Pdb)

```

Nótese que, siempre, antes del prompt del depurador, una flecha nos muestra la línea en la que el programa está detenido, línea que aún no se ha ejecutado (justo antes, se muestra también el número de dicha línea).

Una opción alternativa es **avanzar solo una línea**, ejecutando la actual y parando en la siguiente, con el comando *next*, o *n*:

```

> /home/p16883/main.py(14)<module>()
-> numbers = (19, 12, 35, 21, 12, 41, 37)

```

```
(Pdb) n
> /home/p16883/main.py(15)<module>()
-> number = random.randint(10, 50)
(Pdb) n
> /home/p16883/main.py(17)<module>()
-> if find(numbers, number):
(Pdb)
```

Cuando la próxima instrucción a ejecutar sea una llamada a una función, cabe la posibilidad de usar el comando *next*, que ejecuta la llamada a la función y se para en la siguiente instrucción después de la llamada, o el comando *step*, abreviado *s*, que se para en la primera instrucción de la función, permitiéndonos continuar la depuración por dentro de la misma:

```
> /home/p17498/main.py(17)<module>()
-> if find(numbers, number):
(Pdb) s
--Call--
> /home/p17498/main.py(5) find()
-> def find(numbers, searched):
(Pdb)
```

Inspección

Cuando el programa está detenido, podemos inspeccionar el estado del proceso, para averiguar si va bien o ya se ha desviado de la situación esperada, lo que significa que el fallo del programa está antes del punto actual. Una de las cosas que podemos ver es el **contexto de la línea actual**, usando el comando *list* o *l*:

```
> /home/p12875/main.py(17)<module>()
-> if find(numbers, number):
(Pdb) l
12
13     if __name__ == '__main__':
14         numbers = (19, 12, 35, 21, 12, 41, 37)
15         number = random.randint(10, 50)
16
17 B->     if find(numbers, number):
18         print("El número", number, "está en ", numbers)
19     else:
20         print("El número", number, "no está en ", numbers)
(Pdb)
```

Para **inspeccionar el valor de una variable** usamos el comando *p* (*print*):

```
> /home/p12608/main.py(17)<module>()
-> if find(numbers, number):
(Pdb) p numbers
(19, 12, 35, 21, 12, 41, 37)
(Pdb) p number
40
(Pdb)
```

En el ejemplo, vemos que *numbers* referencia una lista con siete valores enteros y que *number* referencia el valor entero 40.

El comando *where* (abreviado *w*) permite conocer la **secuencia de llamadas** que nos han conducido al punto actual. Esto es útil cuando a una función se la puede llamar desde diferentes puntos de un programa, ya que el camino que se ha seguido para llegar al punto actual puede ser relevante para determinar cómo se produce el fallo:

```
> /home/p15071/main.py(6) find()
-> found = False
(Pdb) w
/home/p15071/main.py(17) <module>()
-> if find(numbers, number):
> /home/p15071/main.py(6) find()
-> found = False
(Pdb)
```

En el ejemplo, vemos que el programa está detenido en la función *find()*, en la línea 6 del módulo *main.py*, adonde se ha llegado ejecutando la llamada a *find()* que se encuentra en la línea 17 del mismo módulo.

Listas

Listas

Las listas son estructuras de datos diseñadas para almacenar secuencias de elementos.

Para crear una lista, ponemos los elementos separados por comas y encerrados entre corchetes:

```
temperatures = [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
students = ['Ada Lovelace', 'Alan Turing', 'Edsger W. Dijkstra',
'Donald Knuth', 'Niklaus Wirth', 'Ole-Johan Dahl', 'Kristen Nygaard']
years = [1945, 1969, 1971, 1978, 1980, 1984, 2012, 2015, 2019]
incoming_person = ['Susan', 'Slovakia', 2019, 52.18, True]
x = 17
y = 9
z = 1989
values = [x, y, z]
```

Como se ve en el ejemplo, una lista puede tener todos los elementos del mismo tipo, o mezclar elementos de tipos diferentes.

Una lista vacía se puede crear de dos formas.

```
a = []
o
a = list()
```

La función *type()* devuelve el tipo *list* cuando se le pasa una lista:

```
>>> type(a)
<class 'list'>
```

Acceso a los elementos de una lista

Para acceder a los elementos de una lista se usan índices. El índice del primer elemento es 0. Igual que en las strings o las [tuplas](#), se pueden usar índices negativos que referencian desde el final de la lista (el índice negativo del último elemento es -1).

```
a = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
     'Saturday', 'Sunday']
print(a[0])
print(a[2])
print(a[-1])
print(a[-2])
print(a[len(a)-1])
```

Resultado de la ejecución del código anterior:

```
Monday
Wednesday
Sunday
Saturday
Sunday
```

Concatenación de listas

Se pueden [concatenar](#) dos listas usando el operador +. Al unir dos listas, creamos una nueva, más larga, con todos los elementos de la primera lista seguidos de todos los elementos de la segunda lista, conservando el orden original.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = [6, 7, 8]
>>> a + b
[1, 2, 3, 4, 5, 6, 6, 7, 8]
>>> students = ['John', 'Martin']
>>> students = ['Joseph'] + students
>>> students
['Joseph', 'John', 'Martin']
>>> students + []
['Joseph', 'John', 'Martin']
```

El operador *

El operador * se usa para [concatenar](#) una lista repetidamente.

```
>>> languages = ['C', 'Java', 'Python'] + ['C', 'Java', 'Python'] +
['C', 'Java', 'Python']
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java', 'Python']
>>> languages = ['C', 'Java', 'Python'] * 3
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java', 'Python']
```

El operador * facilita la inicialización de una lista:

```
>>> counters = [0] * 10
>>> counters
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Operador in

Para ver si una lista contiene un elemento determinado usamos el operador *in*:

```
>>> languages = ['C', 'Java', 'Python']
>>> 'Pascal' in languages
False
>>> 'Python' in languages
True
>>> 'Java' not in languages
False
```

Anotación de variables lista

Para anotar que una variable o un parámetro es de un tipo no básico, hay que importarlo desde el módulo **typing**:

```
from typing import List
lista: List          # La variable Lista se anota como una lista (tipo
list)
vector: List[float] # La variable Vector se anota como una lista de
elementos de tipo float
```

A partir de la opción 3.9 de Python, se puede anotar usando el nombre real del tipo (en minúsculas):

```
from typing import List
lista: list          # La variable Lista se anota como una lista (tipo
list)
vector: list[float] # La variable Vector se anota como una lista de
elementos de tipo float
```

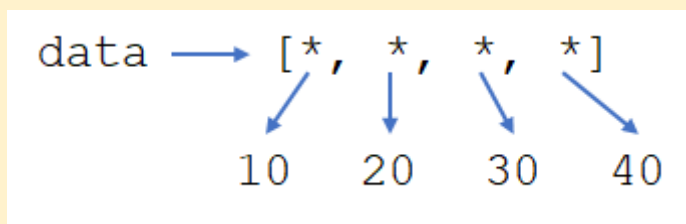
Operaciones de manejo de listas

Las listas son mutables

Considere la siguiente instrucción:

```
data = [10, 20, 30, 40]
```

Esta instrucción crea una lista referenciada por la variable *data*.

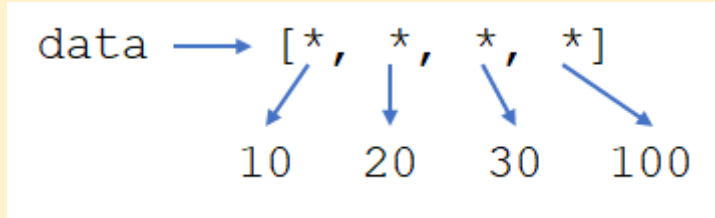


La lista consta de 4 variables `data[0]`, `data[1]`, `data[2]` y `data[3]` que hacen referencia a los cuatro valores *int*.

Las [listas](#) son estructuras mutables, esto significa que pueden ser modificadas.

```
data[3] = 100
```

La instrucción anterior cambia el último elemento de la lista asignándole a la variable correspondiente un nuevo valor.

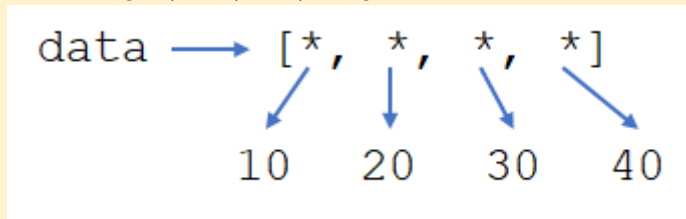


Nótese, que esto no se puede hacer con una tupla o con una string porque, a diferencia de las [listas](#), ambas, son estructuras de datos inmutables.

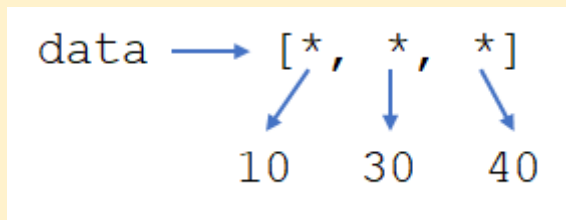
Borrado de elementos

Para borrar un elemento de una lista se puede usar la instrucción *del*.

```
data = [10, 20, 30, 40]
```



```
del(data[1])
```



Recorrido de una lista

Para recorrer una lista suele usarse un bucle *for*:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']  
for i in range(len(friends)):  
    print(i, "-", friends[i])
```

Resultado:

```
0 - Peter  
1 - Paul  
2 - Michael  
3 - George  
4 - John
```

Si sólo estamos interesados en los elementos y los índices no son relevantes, podemos usar una notación más concisa:

```
for element in friends:  
    print(element, end=' ')
```

En este último ejemplo, además, hemos añadido a la instrucción *print* el parámetro *end*, sustituyendo el salto de línea que se usa por omisión por un espacio; el resultado es:

Peter Paul Michael George John

Enumerate

Podemos hacer uso de la función *enumerate()* para acceder a ambos, el índice y el valor correspondiente al mismo tiempo:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']  
  
for i, element in enumerate(friends):  
    print(i, "-", element)
```

Resultado:

0 - Peter
1 - Paul
2 - Michael
3 - George
4 - John

La función *enumerate()* inicia el contador de la secuencia de posiciones en 0; podemos especificar el valor inicial usando el parámetro *start*:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']  
  
for i, element in enumerate(friends, start = 1):  
    print(i, "-", element)
```

Resultado:

1 - Peter
2 - Paul
3 - Michael
4 - George
5 - John

List comprehension

Es un mecanismo que proporciona una forma concisa de crear nuevas [listas](#). Consiste en una expresión seguida de una cláusula *for* y 0 o más cláusulas *for* o *if*. Por ejemplo:

[expression for element in list if condition]

[expression for element in list]

Con esta notación, obtenemos una nueva lista que resulta de evaluar la expresión en el contexto de las cláusulas *for* y *if*, que la siguen.

Las "comprehensiones" de [listas](#) pueden contener expresiones complejas y [funciones](#) anidadas.

Aquí hay algunos ejemplos; estudie atentamente los resultados:

```
list1 = [i for i in range(10)]
#list 1 vale: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list2 = [i ** 2 for i in range(5)]
#list2 vale: [0, 1, 4, 9, 16]
```

List comprehension - más ejemplos

```
import random
a = [random.randint(0, 9) for i in range(10)]
print(a)
```

Salida: [4, 4, 9, 7, 4, 1, 0, 0, 3, 6]

(La función *random.randint()* genera un número al azar en el rango indicado; la salida será diferente en diferentes ejecuciones)

```
words = ['this', 'is', 'a', 'list', 'of', 'words']
b = [word[0].upper() for word in words]
print(b)
```

Salida: ['T', 'I', 'A', 'L', 'O', 'W']

(El método *upper()* devuelve la conversión a mayúsculas de la string asociada)

```
things = ['pen', 'pencil', 'rubber', 'paper']
c = [thing for thing in things if len(thing) > 5]
print(c)
```

Salida: ['pencil', 'rubber']

```
from math import pi
d = [str(round(pi, i)) for i in range(1, 6)]
print(d)
```

Salida: ['3.1', '3.14', '3.142', '3.1416', '3.14159']

(La función *round()* redondea el *float* pasado como primer parámetro al número de decimales indicado por el segundo)

[Funciones](#) de manejo de [listas](#)

Python ofrece algunas [funciones](#) estándar útiles para manejar [listas](#); por ejemplo:

len(list) devuelve el número de elementos de una lista (su longitud)

sum(list) devuelve la [suma](#) de los elementos de una lista

max(list) devuelve el valor máximo de los elementos de una lista

min(list) devuelve el valor mínimo de los elementos de una lista

Obviamente, *sum* sólo se puede aplicar a [listas](#) con elementos numéricos y *max* y *min* a [listas](#) con elementos comparables y del mismo tipo.

Función `list()`

La función *list()* crea una lista con los elementos de una secuencia.

```
a = list(range(1, 10)) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
b = list('Python')    # ['P', 'y', 't', 'h', 'o', 'n']
c = list()             # []
d = ('milk', 'bread', 'butter', 'honey')
e = list(d)            # ['milk', 'bread', 'butter', 'honey']
f = list(a)
```

En el ejemplo anterior, *f* es una copia de la lista *a*.

Métodos de manejo de [listas](#) sin modificación

En Python, los objetos de tipo *list* también admiten algunos métodos ([funciones](#) especiales que se llaman asociadas al objeto sobre el que se aplican):

Sea *mylist* una lista. Los siguientes dos métodos proporcionan información sobre la misma sin cambiar su contenido:

```
a = mylist.count(x)          # Devuelve el número de veces que x
                             # aparece en mylist
b = mylist.index(x)          # Devuelve la posición de la primera
                             # ocurrencia de x en mylist
c = mylist.index(x, start, end) # Devuelve la posición de la primera
                             # ocurrencia de x en mylist
                             # en el rango de posiciones start y
end
```

Ejemplo:

```
fruits = ['banana', 'apple', 'pear', 'cherry', 'banana', 'banana']
print(fruits.count('banana')) # 3
print(fruits.index('banana')) # 0
print(fruits.index('banana', 3, len(fruits))) # 4
```

Debemos tener cuidado al usar el método *index*, ya que si el valor buscado no se encuentra en la lista devuelve un *ValueError*. Podemos prevenirlo usando el operador *in*:

```
if 'peach' in fruits:
    print(fruits.index('peach'))
```

```
else:
    print('There is no "peach" in this list.')
```

Métodos que modifican [listas](#)

Los siguientes métodos son mutables, modifican el contenido de la lista sobre la que se aplican.

Sea *mylist* una lista:

```
mylist.append(x)      # Añade el elemento x al final de mylist
mylist.insert(i, x)   # Añade el elemento x en la posición de mylist
                      # indicada por i
value = mylist.pop()  # Quita el último elemento de mylist y lo
                      # devuelve, se asigna a la variable value
value = mylist.pop(i) # Quita el elemento indicado por i de mylist y
                      # lo devuelve, se asigna a la variable value
mylist.remove(x)      # Quita el primer elemento de mylist cuyo valor
                      # sea igual a x.
                      # Si no hay ninguno se produce un ValueError

mylist.clear()        # Vacía mylist, quitando todos los elementos
```

Ejemplo:

```
fruits = ['banana', 'apple', 'pear', 'cherry', 'apple', 'banana']
fruits.append('peach')
print(fruits)          # ['banana', 'apple', 'pear', 'cherry',
                        # 'apple', 'banana', 'peach']
print(fruits.pop())    # peach
print(fruits.pop(0))   # banana
fruits.insert(0, 'coconut')
fruits.remove('apple')
print(fruits)          # ['coconut', 'pear', 'cherry', 'apple',
                        # 'banana']
fruits.clear()
print(fruits)          # []
```

En el siguiente ejemplo, se añaden elementos a una lista vacía para que contenga todos los números enteros no negativos menores que 1000 divisibles por 3 o 5:

```
numbers = []
for i in range(1000):
    if i % 3 == 0 or i % 5 == 0:
        numbers.append(i)
```

Slices

Igual que con las strings o las [tuplas](#), podemos obtener sublistas (slices, segmentos) a partir de una lista. La sintaxis completa es:

```
lista[inicio : fin : paso]
```

La operación de slice no modifica la lista original, sino que crea una nueva.

Ejemplo:

```
cities = ['Bratislava', 'Warsaw', 'Madrid', 'Praha']
a = cities[1:3] # ['Warsaw', 'Madrid']
b = cities[-3:] # ['Warsaw', 'Madrid', 'Praha']
c = cities[:-1] # ['Bratislava', 'Warsaw', 'Madrid']
d = cities[1:2] # ['Warsaw', 'Praha']
e = cities[::-1] # ['Praha', 'Madrid', 'Warsaw', 'Bratislava']
```

Asignación a slices

Cuando se toma un slice de una lista siempre se crea una lista nueva sin modificar la original.

La notación de slices puede usarse también a la izquierda de una asignación (como destino de la asignación); en ese caso, la parte derecha de la asignación (el origen de la asignación) debe ser una expresión que represente una secuencia (no obligatoriamente una lista). El funcionamiento de la asignación es como sigue:

1. Se evalúa la parte derecha de la asignación creando una lista nueva.
2. La nueva lista sustituye a la sublista representada por el slice, modificando la lista a la que el slice hace referencia.

Ejemplos:

1) Se sustituyen tres elementos por otros tres.

```
names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
names[1:4] = ['Paul', 'Peter', 'Thomas'] # ['Matthew', 'Paul', 'Peter',
'Thomas', 'Francis']
```

2) Tres elementos se sustituyen por dos.

```
names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
names[1:4] = ['Paul', 'Peter'] # ['Matthew', 'Paul', 'Peter',
'Francis']
```

3) Los dos últimos elementos son sustituidos por tres.

```
names[-2:] = ['Andrew', 'James', 'Philip'] # ['Matthew', 'Paul',
'Andrew', 'James', 'Philip']
```

4) Los dos primeros elementos son sustituidos por una lista vacía.

```
names[:2] = [] # ['Andrew', 'James', 'Philip']
```

5) El primer elemento se sustituye por seis elementos de una lista construida a partir de una string.

```
names[:1] = 'Python' # ['P', 'y', 't', 'h', 'o', 'n', 'James',
'Philip']
```

Comparación de [listas](#)

Se pueden comparar dos [listas](#) usando los operadores relacionales:

- Se van comparando de principio a fin los elementos que ocupan la mismas posiciones en las dos [listas](#) hasta que se encuentra una pareja diferente o se acaba alguna de las [listas](#).
- Si se encuentra una pareja de elementos diferentes, es menor la lista que tenga el elemento menor.
- Si se acaba una de las [listas](#), sin encontrar una pareja diferente, y la otra no, es menor la lista de menor longitud.
- Si se llega a la vez al final de las dos [listas](#) sin encontrar una pareja de elementos diferentes, las [listas](#) son iguales.

Ejemplos:

```
a = list(range(1,5))    # [1, 2, 3, 4]
b = list(range(1,5))    # [1, 2, 3, 4]
a == b                  # True
a != b                  # False
a[1] = 0
a == b                  # [1, 0, 3, 4] == [1, 2, 3, 4] False
a < b                    # [1, 0, 3, 4] < [1, 2, 3, 4] True
```

En la lista *b*, el elemento con índice 1 es mayor que el elemento correspondiente de la lista *a*.

```
a <= b                  # True
a > b                    # False
a >= b                   # False
[1,2] < b                 # [1, 2] < [1, 2, 3, 4] True
```

Los primeros dos elementos en las dos [listas](#) son iguales, pero la primera lista es más corta.

Listas y funciones

Alias

Una lista puede ser referenciada por varias variables:

```
a = [1, 2, 3]
b = a
```

The diagram illustrates that both variables 'a' and 'b' point to the same list object in memory, which contains the elements [1, 2, 3]. Two blue arrows originate from the variables 'a' and 'b' and converge on the list object [1, 2, 3].

Las variables *a* y *b* hacen referencia a la misma lista. Esto significa que si se modifica el objeto referenciado por una de las variables, el objeto referenciado por la otra queda modificado, ya que ambas variables son alias del mismo objeto:

```
b[1] = 0
```

The diagram shows the state after the modification. The list object now contains [1, 0, 3]. Both variables 'a' and 'b' still point to this same list object, as indicated by the two blue arrows converging on the [1, 0, 3] list.

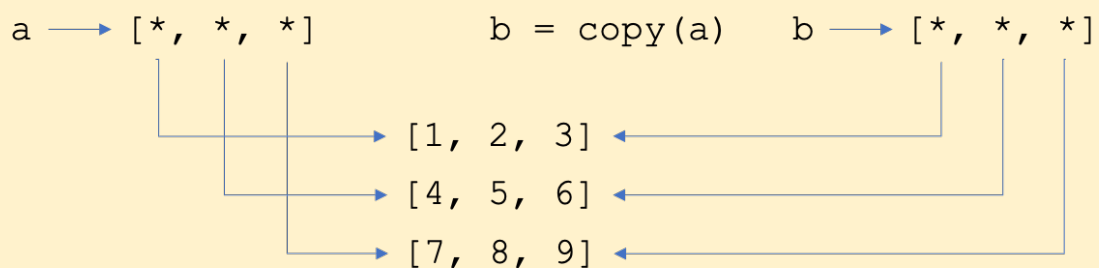
Si se necesita tener una copia independiente, se puede usar la función `copy()`, del módulo `copy`, que hay que importar.

`a = [1, 2, 3]` `a` \longrightarrow `[1, 2, 3]`

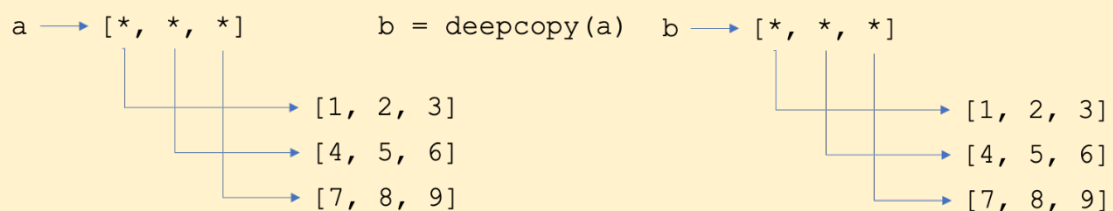
`b = copy(a)` `b` \longrightarrow `[1, 2, 3]`

Shallow copy y deep copy

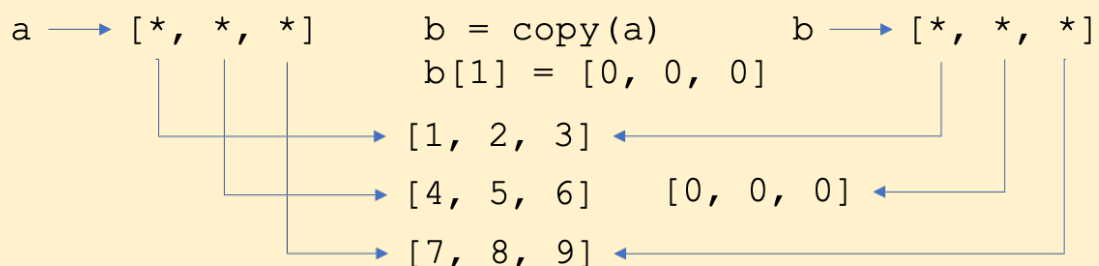
La función `copy()` del módulo `copy` realiza una copia "superficial"; esto significa que si los elementos de la lista, en vez de referenciar valores de un tipo simple, como números enteros, son referencias a objetos que, a su vez, referencian otros objetos (p.e., una lista de [listas](#), o una lista de [tuplas](#)), los elementos de 2º y sucesivos niveles quedan compartidos, ya que sólo se copian las referencias de primer nivel:



Si se quiere hacer una copia "profunda", hay que usar la función `deepcopy()` en vez de la `copy()`:



De todas formas, la compartición de elementos de segundo nivel no implica que al modificar una lista se modifique la otra, contrariamente a lo que pasa cuando se comparten elementos de primer nivel:



Paso de [listas](#) como parámetros

Cuando tenemos que procesar una lista, frecuentemente la pasamos como parámetro a una función. La función del siguiente ejemplo devuelve el valor máximo almacenado en una lista (en realidad, cualquier secuencia iterable):

```
def maximum(a):
    m = a[0]
    for i in range(1, len(a)):
        if a[i] > m:
            m = a[i]
    return m

data = [1, 3, 8, 5, 6]
print(maximum(data))      # 8
print(data)               # [1, 3, 8, 5, 6]
print(maximum('Python'))  # y
```

Nótese que la función `maximum` puede llamarse no sólo con [listas](#), sino con cualquier secuencia iterable, como, por ejemplo, una string.

Este otro ejemplo, intercambia dos elementos en una lista (se supone que tanto *i* como *j* son índices válidos para la lista *a*):

```
def exchange(a, i, j):
    a[i], a[j] = a[j], a[i]

data = [1, 3, 8, 5, 6]
exchange(data, 1, 3)
print(data)          # [1, 5, 8, 3, 6]
```

En ambos ejemplos, el parámetro formal, *a*, es una referencia al parámetro real que se pasa en la llamada (*data* en el segundo ejemplo), por lo que, en el segundo ejemplo, al modificar la lista usando *a* en la función, se está modificando la lista referenciada por *data*, fuera de la función.

[Listas](#) como resultado de [funciones](#)

Además de poder pasarse como parámetros, las [listas](#) también pueden devolverse como resultado de una función:

```
def create_list(n, value = 0):
    return [value] * n

print(create_list(10))      # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(create_list(5, -1))   # [-1, -1, -1, -1, -1]
print(create_list(8, None)) # [None, None, None, None, None, None, None, None]
print(create_list(5, 'Hi')) # ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

En tanto en cuanto no especifiquemos el tipo del resultado esperado y no usemos una herramienta para tenerlo en cuenta, la función `create_list()` es una plantilla general para crear [listas](#) de elementos con un valor determinado.

Para especificar que se requiere una lista como parámetro o se espera como resultado podemos usar, igual que para los tipos básicos, el nombre de tipo *list*. Si quisiéramos

especificar además de qué tipo deben ser, a su vez, los elementos de la lista, podemos importar del módulo *typing* el nombre *List*, como en el siguiente ejemplo:

```
from typing import List

def create_list(n: int, value: int = 0) -> List[int]:
    return [value] * n
```

Para las [tuplas](#) (tipos *tuple* y *typing.Tuple*) ocurre igual.

Funciones modificadoras y no modificadoras

Cuando escribimos [funciones](#) que manejan [listas](#), debemos tener presente si queremos que la lista original se modifique o no.

```
def add1(list, element):
    return list + [element]

def add2(list, element):
    list.append(element)
    return list

data = [1, 3, 8, 5, 6]
print(add1(data, 1000))      # [1, 3, 8, 5, 6, 1000]
print(data)                  # [1, 3, 8, 5, 6]
print(add2(data, 1000))      # [1, 3, 8, 5, 6, 1000]
print(data)                  # [1, 3, 8, 5, 6, 1000]
```

En el ejemplo anterior, la función *add1()* devuelve una nueva lista, sin modificar la original, mientras que la función *add2()* añade un elemento a la lista original. Decimos que la función *add1()* es no modificadora y la función *add2()* es modificadora.

Tratando múltiples listas a la vez

Procesando varias [listas](#) al mismo tiempo

En muchas ocasiones se necesita procesar dos o más [listas](#) simultáneamente. El siguiente ejemplo intercala los elementos de dos [listas](#), generando una nueva:

```
def interleave(list1, list2):
    result = []

    for i in range(len(list1)):
        result.append(list1[i])
        result.append(list2[i])

    return result;

numbers = [10, 20, 30]
chars= ['A', 'B', 'C']
print(interleave(numbers, chars)) # [10, 'A', 20, 'B', 30, 'C']
```

Como se observa en el ejemplo, no hace falta un bucle independiente para cada lista, ya que de lo que se trata es de recorrerlas al mismo tiempo, tomando un elemento de cada

una para pasarlo al resultado, esto es, 1 recorrido (de 1, 2, o más [listas](#) a la vez) == 1 bucle.

En la función del ejemplo se supone que ambas [listas](#) son del mismo tamaño, o que $\text{len}(\text{list2}) \geq \text{len}(\text{list1})$. De no ser así, se produciría un error (Index out of range).

Procesando varias [listas](#) al mismo tiempo con zip

Si no necesitamos tener acceso a los índices, una forma de procesar dos o más [listas](#) al mismo tiempo es usar la función `zip()`:

```
def interleave1(list1, list2):
    result = []

    for item1, item2 in zip(list1, list2):
        result.append(item1)
        result.append(item2)

    return result;

list1 = [10, 20, 30]
list2 = ['A', 'B', 'C']
print(interleave1(list1, list2)) # [10, 'A', 20, 'B', 30, 'C']
```

La función predefinida `zip()` permite iterar por una secuencia de [tuplas](#), formada de tal manera que el *i*-ésimo elemento de esa secuencia es una tupla formada por los *i*-ésimos elementos de cada una de las [listas](#) (en general, secuencias) que se le pasan como parámetros (en el ejemplo, el primer elemento devuelto por `zip` es una tupla formada por el primer elemento de `list1` y el primer elemento de `list2`).

En el caso de que las [listas](#) fuesen de diferentes longitudes, `zip()` procesaría sólo hasta la longitud de la más corta.

Procesando simultáneamente [listas](#) de distintos tamaños con `zip_longest`

La función `zip()` permite recorrer simultáneamente dos o más [listas](#), pero sólo hasta el tamaño de la más corta. Si se quiere recorrer hasta el tamaño de la más larga, se puede usar la función `zip_longest()` del módulo `itertools`:

```
import itertools
def interleave2(list1, list2):
    result = []

    for item1, item2 in itertools.zip_longest(list1, list2,
        fillvalue='-'):
        result.append(item1)
        result.append(item2)

    return result;

list1 = [10, 20, 30]
list2 = ['A', 'B', 'C', 'D']
print(interleave2(list1, list2)) # [10, 'A', 20, 'B', 30, 'C', '-', 'D']
```

Los elementos que faltan en las [listas](#) más cortas se sustituyen con el valor especificado en el parámetro *fillvalue* (el carácter '-' en el ejemplo). Si se omite el parámetro *fillvalue*, se usa el valor *None*.

Procesando varias [listas](#) de diferentes tamaños a la vez usando índices

Si se quiere recorrer dos [listas](#) de diferentes tamaños a la vez usando índices en vez de *zip_longest()*, es conveniente, como en el siguiente ejemplo, separar la iteración en tres bucles de los que sólo se ejecutan, como máximo, dos: el primero, mientras las dos [listas](#) tienen elementos, y uno de los otros dos, para terminar de procesar los elementos de la lista más larga.

```
def interleave3(list1, list2):
    result = []
    min_len = min(len(list1), len(list2))

    for i in range(min_len):
        result.append(list1[i])
        result.append(list2[i])
    for i in range(min_len, len(list1)):
        result.append(list1[i])
        result.append('-')
    for i in range(min_len, len(list2)):
        result.append('-')
        result.append(list2[i])
    return result;
```

Alternativamente, se pueden usar sentencias *while* en su lugar, aprovechando el valor del índice a la salida del primer bucle:

```
def interleave3b(list1, list2):
    result = []
    i = 0

    while i < min(len(list1), len(list2)):
        result.append(list1[i])
        result.append(list2[i])
        i += 1

    while i < len(list1):
        result.append(list1[i])
        result.append('-')
        i += 1

    while i < len(list2):
        result.append('-')
        result.append(list2[i])
        i += 1

    return result;
```

De hacer un solo bucle, hasta la longitud más larga, habría que incluir dentro del bucle preguntas para saber si se ha acabado alguna de las [listas](#), cuyo coste de ejecución se sumaría en cada ejecución, pero sería fácilmente extensible a cualquier número de [listas](#):

```
def interleave3c(list1, list2):
    result = []

    for i in range(max(len(list1), len(list2))):
        if i < len(list1):
            result.append(list1[i])
        else:
            result.append('-')
        if i < len(list2):
            result.append(list2[i])
        else:
            result.append('-')

    return result;
```

Una alternativa a esto, en Python, es usar *zip_longest()* junto con *enumerate()*, para dos o más [listas](#):

```
from itertools import zip_longest

def interleave3d(list1, list2):
    result = []

    for i, items in enumerate(zip_longest(list1, list2, fillvalue='-')):
        result.append(items[0])
        result.append(items[1])

    return result;
```

Estructuras 2D

[Listas](#) de [tuplas](#)

Las [listas](#) y las [tuplas](#) son tipos de datos estructurados que almacenan secuencias de elementos. Las [listas](#) son mutables y las [tuplas](#) son inmutables. Los elementos de una lista o de una tupla pueden ser de cualquier tipo, incluyendo tipos estructurados. Por ejemplo, podemos definir una lista de [tuplas](#) que representen coordenadas de puntos en el plano:

```
a = (0, 1)
b = (1, 1)
c = (2, -5)
d = (3, 7)
e = (4, 8)
points = [a, b, c, d, e]
print(points)           # [(0, 1), (1, 1), (2, -5), (3, 7), (4, 8)]
```

La variable *points* referencia a una lista que contiene 5 [tuplas](#).

Podemos añadir elementos a la lista *points*, actualizarlos, borrarlos o aplicar una operación de slice.

```
points.append((5, 0))
points.pop(0)
print(points)           # [(1, 1), (2, -5), (3, 7), (4, 8), (5, 0)]
points[2] = (1000, 1000)
```

```

print(points)          # [(1, 1), (2, -5), (1000, 1000), (4, 8),
(5, 0)]
print(len(points))     # 5
alias = points
points[:] = points[::-1]
print(alias)           # [(5, 0), (4, 8), (1000, 1000), (2, -5),
(1, 1)]

```

Podemos acceder a los elementos individuales de una tupla usando un segundo índice:

```

print(points[0])        # (5, 0) - la primera tupla de la lista
print(points[0][0])     # 5 - el primer elemento de la primera
tupla
print(points[0][1])     # 0 - el segundo elemento de la primera
tupla

```

Obviamente, dado que las [tuplas](#) son inmutables, la siguiente instrucción es errónea:

```

points[1][0] = 0        #... TypeError: 'tuple' object does not
support item assignment

```

ablas como [listas](#) de [listas](#)

A menudo necesitamos implementar una estructura bidimensional: una tabla con filas y columnas (una matriz de valores desde un punto de vista matemático). Dicha tabla de valores se puede crear utilizando [listas](#) de [listas](#) en Python:

```

m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

o, en una forma más legible:

```

m = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

```

Cada uno de los elementos `m[0]`, `m[1]`, `m[2]` referencia a una lista de 3 elementos. Otra forma de de crear la misma estructura:

```

row1 = [1, 2, 3]
row2 = [4, 5, 6]
row3 = [7, 8, 9]
m=[row1, row2, row3]

```

El número de filas o columnas se puede conocer usando la función `len()`.

```

print(len(m))          # 3 - número de filas

```

La longitud de cualquier elemento, por ejemplo `m[0]`, nos da el número de columnas.

```

print(len(m[0]))       # 3 - número de columnas

```

Los elementos de la tabla se pueden acceder mediante una pareja de índices. La expresión `m[i][j]` representa el valor del elemento que está en la columna `j` de la fila `i`. El siguiente ejemplo cambia el valor de la primera columna de la segunda fila de la tabla `m` y muestra la tabla:

```

m[1][0] = 0
print(m)              # [[1, 2, 3], [0, 5, 6], [7, 8, 9]]

```


Recorrido de una tabla

Consideremos de nuevo la tabla *m*:

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Para recorrer una tabla, generalmente usamos bucles for anidados:

```
for row in m:
    for x in row:
        print(x, end=' ')

    print()
```

El código anterior muestra el siguiente resultado:

```
1 2 3
4 5 6
7 8 9
```

A veces, es útil procesar los elementos de una tabla usando sus índices:

```
x = len(m) * len(m[0])

for i in range(len(m)):
    for j in range(len(m[0])):
        m[i][j] = x
        print(m[i][j], end=' ')
        x -= 1

    print()
```

El código anterior modifica la tabla y muestra el siguiente resultado:

```
9 8 7
6 5 4
3 2 1
```

Mutabilidad

Cuando se crea una lista de [listas](#) hay que tener en cuenta que las [listas](#) son mutables, luego los elementos de una lista de [listas](#) (o de una tupla de [listas](#)) son mutables.

Observe cómo se crea la tabla *m* en el siguiente ejemplo:

```
m = [[0, 0, 0]] * 3
```

Tiene tres filas con tres ceros cada una. Vamos a cambiar uno de los valores:

```
m[0][0] = -1
print(m) # [[-1, 0, 0], [-1, 0, 0], [-1, 0, 0]]
```

Los elementos *m*[1][0] y *m*[2][0] no deberían haber cambiado ¿Qué ha pasado? Realmente la signación:

```
m = [[0, 0, 0]] * 3
```

crea una lista con tres referencias a la misma lista [0, 0, 0], igual que el siguiente ejemplo:

```
row = [0, 0, 0]
m = [row, row, row]
```

Podemos chequear la identidad de las filas con el operador *is*:

```
m = [[0, 0, 0]] * 3
print(m[0] is m[1]) # True
```

Para crear una lista de [listas](#) de forma correcta debemos primero crear una lista vacía y luego añadirle cada fila:

```
m = []
for i in range(3):
    m.append([0,0,0])
```

Ahora todo está bien:

```
m[0][0] = -1
print(m) # [[-1, 0, 0], [0, 0, 0], [0, 0, 0]]
print(m[0] is m[1]) # False
```

Usando [funciones](#) para crear estructuras 2D

El uso de [funciones](#) puede ser muy útil para crear estructuras 2D, como [listas](#) de [listas](#). La función `create_table()` del siguiente ejemplo toma las dimensiones de la tabla a crear (filas y columnas) y un valor de inicialización:

```
def create_table(rows, columns, value=0):
    t = []
    for i in range(rows):
        t.append([value] * columns)
    return t

my_table = create_table(2, 3)
print(my_table) # [[0, 0, 0], [0, 0, 0]]
my_table = create_table(4, 2, '*')
print(my_table) # [['*', '*'], ['*', '*'],
['*', '*'], ['*', '*']]
```

La función `create_table2()` se crea una lista de tantos elementos como indique el parámetro `rows` inicializada con el valor `None`. Los elementos están inicializados pero aún no referencian ninguna fila. A continuación se crean las filas y se asigna su referencia a cada elemento de la lista inicial, sustituyendo al valor `None`.

```
def create_table2(rows, columns, value=0):
    t = [None] * rows
    for i in range(rows):
        t[i] = [value] * columns
    return t

my_table = create_table2(3, 5, 1)
print(my_table) # [ [1, 1, 1, 1, 1], [1, 1, 1, 1, 1],
[1, 1, 1, 1, 1]]
```

Uso de [comprehension](#) para crear [listas](#) de [listas](#)

Se puede crear estructuras 2D usando [comprehension](#):

```
t = [(i, j) for i in range(2) for j in range(3)]
print(t) # [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Nótese que, en el ejemplo anterior, cada elemento es una tupla.

```
t = [[0] * 3 for i in range(4)]
print(t) # [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

En el ejemplo anterior, cada fila se crea independientemente.

```
t = [list(str(2 ** i)) for i in range(10)]
print(t) # [['1'], ['2'], ['4'], ['8'], ['1', '6'], ['3', '2'], ['6', '4'], ['1', '2', '8'], ['2', '5', '6'], ['5', '1', '2']]
```

En el ejemplo anterior, las filas tienen diferentes longitudes, dado que 2^{**i} tiene más dígitos a mayor valor de i .

```
import random
t = [[random.randint(0, 9)] * 5 for i in range(3)]
print(t)
# [[8, 8, 8, 8, 8], [2, 2, 2, 2, 2], [7, 7, 7, 7, 7]]
```

En el ejemplo anterior todos los elementos de una misma fila tienen el mismo valor aleatorio.

Listas de listas de distintas longitudes

Hay muchas situaciones en las que necesitamos [listas](#) de [listas](#) de longitudes diferentes. La siguiente función toma como parámetros una secuencia con las longitudes deseadas para cada fila y un valor de inicialización:

```
def create_table(row_lengths, value=0):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [value] * row_lengths[i]
    return t
```

Primero, se crea una lista conteniendo un cierto número de None que luego se sustituirán por las referencias a las filas que se irán creando de forma independiente, atendiendo a las longitudes indicadas por el primer parámetro.

Veamos cómo se usa:

```
t = create_table((1, 2, 3, 4, 5))
print(t) # [[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]

t = create_table(range(4), 'x')
print(t) # [[], ['x'], ['x', 'x'], ['x', 'x', 'x']]
```

Nótese que `range(4)` representa la secuencia de números 0, 1, 2, 3, por lo que la primera fila resulta una lista vacía en el segundo ejemplo.

Tratamiento estructurado de estructuras 2D

Cuando se quiere procesar una secuencia de elementos tratando cada elemento, se sigue un esquema básico como el siguiente:

```
for elemento in lista:
    tratar elemento
```

Cuando tenemos una secuencia de secuencias (por ejemplo una lista de [listas](#)), cada elemento de dicha secuencia es a su vez una secuencia que típicamente se procesa siguiendo un mismo esquema, lo cual da lugar a un patrón de bucles anidados:

```
for lista in lista_de_listas:
    for elemento in lista:
        tratar elemento
```

Una alternativa es derivar el tratamiento de las secuencias anidadas a una función auxiliar:

```
def tratar(lista):
    for elemento in lista:
        tratar elemento
for lista in lista_de_listas:
    tratar(lista)
```

Este esquema se aplica en el siguiente ejemplo:

```
def longest_words(list_of_list_of_words):
    """Dada una lista de listas de palabras, devuelve
    una lista con las palabras mas largas de cada lista
    """
    words = []
    for list_of_words in list_of_list_of_words:
        words.append(longest_word(list_of_words))
    return words

def longest_word(list_of_words):
    """Devuelve la palabra más larga de una lista de palabras"""
    current_longest = ""
    for current_word in list_of_words:
        if len(current_word) > len(current_longest):
            current_longest = current_word
    return current_longest
```

La ventaja de separar cada tratamiento en [funciones](#) diferentes, en vez de tener una única función con bucles anidados, es que acabamos teniendo un conjunto de [funciones](#) muy simples, que siguen un esquema básico, y son más fáciles de desarrollar, entender y mantener que una función compleja, con bucles anidados. Nótese que esto se puede extender a estructuras multidimensionales ([listas](#) de [listas](#) de [listas](#) de [listas](#) de ...).

Diccionarios

Diccionarios

Los diccionarios, conocidos en algunos lenguajes como arrays asociativos, son colecciones de datos, como las secuencias, pero, a diferencia de las secuencias, no se indexan usando un rango de números (0, 1, 2,...) sino que usan una clave, que pueden ser de cualquier tipo inmutable (generalmente strings o números). Un diccionario (tipo dict) es una colección **no ordenada** de parejas clave-valor.

Un diccionario vacío se crea con una pareja de llaves:

```
d = {}
type(d)
```

```
<class 'dict'>
len(d)
0
```

También se puede crear un diccionario vacío usando el constructor *dict()*:

```
d = dict()
```

Un diccionario se puede crear no vacío inicializándolo con una lista de parejas clave:valor separadas por comas. El siguiente es un (mini) diccionario inglés-español:

```
d = {'mother': 'madre', 'father': 'padre', 'house': 'casa', 'car':
'coche'}
```

Otras formas de inicializar un diccionario

Al constructor *dict()* se le puede pasar una secuencia de pares clave:valor. El siguiente ejemplo inicializa un (mini) diccionario inglés-eslovaco:

```
my_list = [('mother', 'matka' ), ('father', 'otec'), ('house', 'dom'),
('car', 'auto')]
d = dict(my_list)
print(d) # {'mother': 'matka', 'father': 'otec', 'house': 'dom',
'car': 'auto'}
```

También se puede usar comprensión para crear diccionarios:

```
d = {x: x**2 for x in (2, 4, 6)}
print(d) # {2:4, 4:16, 6:36}
```

Cuando las claves son strings, se pueden usar como "parámetros por nombre" de *dict()*:

```
d = dict(Michael=77, Peter=66, Patrick=82)
print(d) # {'Michael':77, 'Peter':66, 'Patrick':82}
```

Operaciones en diccionarios

Las principales operaciones que se pueden realizar en un diccionario son:

- almacenar un valor asociado a una clave
- obtener el valor asociado a una clave
- borrar un par clave:valor

```
d = {'mother': 'madre', 'father': 'padre'}
d['child'] = 'hijo'
print(d) # {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
```

En el ejemplo anterior, la instrucción *d['child'] = 'hijo'* añade un nuevo par clave:valor al diccionario *d*. Si se usa una clave existente, el nuevo valor sustituye al almacenado previamente:

```
d['child'] = 'hija'
print(d) # {'mother': 'madre', 'father': 'padre', 'child': 'hija'}
```

Para acceder a un valor usamos su clave:

```
print(d['mother']) # madre
```

Es un error intentar acceder al diccionario usando una clave que no existe:

```
v = d['dog']
Traceback (most recent call last): File "<input>", line 1, in <module>
KeyError: 'dog'
Para saber si una clave existe en un diccionario, usamos el operador in:
if 'dog' in d:
    print(d['dog'])
else:
    print('La clave "dog" no existe en el diccionario')
```

Eliminar una pareja clave:valor de un diccionario

A veces necesitamos eliminar una pareja clave:valor de un diccionario. Podemos hacerlo con el método *pop()*.

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
value = d.pop('mother')
print(d) # {'father': 'padre', 'child': 'hijo'}
```

El método *pop()* elimina el par clave:valor asociado a la clave pasada como parámetro y devuelve el valor. Si no nos interesa quedarnos con el valor, podemos llamar al método *pop()* sin asignar su resultado a ninguna variable:

```
d.pop('mother')
La instrucción del también podemos usarla con diccionarios:
```

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
del d['mother']
print(d) # {'father': 'padre', 'child': 'hijo'}
```

En ambos casos, se produce un error (*KeyError*) si la clave pasada como parámetro no se encuentra en el diccionario.

Para vaciar un diccionario se usa el método *clear()*:

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
d.clear()
print(d) # {}
```

El método *get()*.

A veces, es mejor usar el método *get()* para recuperar el valor asociado a una clave:

```
capitals = {'Slovakia': 'Bratislava', 'Austria': 'Vienna'}
print(capitals['Slovakia']) # Bratislava
print(capitals.get('Slovakia')) # Bratislava
```

Supongamos que la clave no está en el diccionario:

```
capitals = {'Slovakia': 'Bratislava', 'Austria': 'Vienna'}
print(capitals.get('Hungary')) # None
print(capitals['Hungary'])
Traceback (most recent call last):
  File "my_program.py", line 2, in <module>
```

```
print(capitals['Hungary'])
KeyError: 'Hungary'
```

El método *get()* no produce un error cuando la clave no se encuentra en el diccionario; en su lugar se devuelve el valor *None*, a menos que se especifique un valor diferente añadiendo un segundo parámetro en la llamada:

```
print(capitals.get('Hungary'), '?') # ?
```

Iteración en diccionarios

Aparte de recuperar un valor usando una clave, podemos recuperar todas las claves, todos los valores, o todos los pares clave:valor de un diccionario:

```
t = {'A': 10, 'B': 1, 'C': 8, 'D': 9, 'E': 4, 'FX': 2}
print(t.keys())
print(t.values())
print(t.items())
```

Resultado:

```
dict_keys(['A', 'B', 'C', 'D', 'E', 'FX'])
dict_values([10, 1, 8, 9, 4, 2])
dict_items([('A', 10), ('B', 1), ('C', 8), ('D', 9), ('E', 4), ('FX', 2)])
```

Todos los objetos mostrados son iterables:

```
for k in t.keys():
    print(k, end=' ')      # A B C D E FX

for v in t.values():
    print(v, end=' ')      # 10 1 8 9 4 2

for k, v in t.items():
    print(k, v)            # A 10
                           # B 1
                           # C 8
                           # D 9
                           # E 4
                           # FX 2
```

El último resultado puede obtenerse también de la siguiente forma:

```
for k in t:
    print(k, t[k])
```

Los valores resultantes de llamar a los métodos *keys()*, *values()*, o *items()* pueden convertirse en [listas](#). La expresión:

```
list(d.keys())
```

devuelve una lista con todas las claves del diccionario *d*, en un **orden arbitrario** (recuérdese que un diccionario es una **colección no ordenada** de parejas clave:valor). Si se necesita tener las claves ordenadas, debe usarse la expresión:

`sorted(d.keys())`

Diccionarios como valores de diccionarios

Un diccionario puede ser un valor en otro diccionario:

```
s1 = {'name': 'Paul Carrot', 'address': {'Street': 'Long', 'Number': 13, 'City': 'Nitra'}}
s2 = {'name': 'Peter Pier', 'address': {'Street': 'Short', 'Number': 21, 'City': 'Nitra'}}
s3 = {'name': 'Patrick Nut', 'address': {'Street': 'Deep', 'Number': 77, 'City': 'Nitra'}}
students = [s1, s2, s3]
```

En el ejemplo anterior se crea una lista de 3 diccionarios.

El elemento `students[0]` es un diccionario con dos parejas clave.valor:

```
print(student[0]['name']) # Paul Carrot
print(student[0]['address']) # {'Street': 'Long', 'Number': 13, 'City': 'Nitra'}
```

El valor asociado a la clave 'address' es, a su vez, un diccionario:

```
print(student[0]['address']['Street']) # Long
print(student[0]['address']['Number']) # 13
print(student[0]['address']['City']) # Nitra
```

Conjuntos

El tipo *set*

El tipo `set` de Python se usa para representar colecciones **sin orden** de elementos **sin repetición**; básicamente, implementa el concepto matemático de conjunto. Un conjunto vacío se crea usando la función constructora `set()`:

```
s = set()
```

Un conjunto no vacío se puede crear escribiendo sus elementos encerrados entre llaves:

```
elementos = {"agua", "aire", "fuego", "tierra"}
print(elementos)
{'agua', 'tierra', 'aire', 'fuego'}
```

>>> Nótese que el orden de los elementos no está definido y no tiene que coincidir con el orden en que se escribieron al crear el conjunto.

Los elementos no pueden ser de un tipo mutable. En el siguiente ejemplo, el tercer elemento es una lista, lo que supone un error:

```
elementos = {"agua", "aire", ["fuego", "tierra"]}
Traceback (most recent call last):

  File "/home/p11049/p1.py", line 1, in <module>
```



```
elementos = {"agua", "aire", ["fuego", "tierra"]}

TypeError: unhashable type: 'list'
```

Otra forma de crear un conjunto es pasando una colección de elementos (tupla, lista, string, diccionario, otro conjunto) a la función constructora:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
print(elementos)
caracteres = set("esto es una string")
print(caracteres)
{'agua', 'fuego', 'aire', 'tierra'}
{'a', 's', 'r', 'e', 'g', ' ', 't', 'o', 'i', 'n', 'u'}
```

>>> *Nótese que en el conjunto caracteres no hay elementos repetidos, aunque sí los había en la string que se pasó para crearlo.*

En caso de que se pase un diccionario, se crea un conjunto con las claves del mismo (no las parejas clave/valor). Si se pasa un conjunto, se crea una copia del mismo.

Otra forma de copiar un conjunto es usando el método `.copy()`:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
copia = elementos.copy()
```

El cardinal (número de elementos de un conjunto) puede averiguarse usando la función `len()`, como en cualquier otra colección de Python:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
print(len(elementos))
4
```

Modificación de conjuntos

A un conjunto existente se le pueden añadir elementos individuales usando la operación `.add()`:

```
elementos = {"agua", "aire", "fuego", "tierra"}
elementos.add("eter")
print(elementos)
{'tierra', 'eter', 'agua', 'aire', 'fuego'}
```

También se le pueden añadir elementos de una colección usando la operación `.update()`:

```
elementos = {"agua", "aire"}
elementos.update(["fuego", "tierra"])
print(elementos)
{'fuego', 'aire', 'tierra', 'agua'}
```

Tanto `.add()` como `.update()` ignoran los elementos repetidos y no los añaden al conjunto.

Para eliminar un elemento concreto, se pueden usar las operaciones `.remove()` o `.discard()`:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.remove("agua")
print(elementos)
{'tierra', 'fuego', 'aire'}
elementos.discard("aire")
print(elementos)
{'tierra', 'fuego'}
```

La diferencia entre `.remove()` y `.discard()` es que el primero produce un error si el elemento a eliminar no está en el conjunto, mientras que el segundo lo ignora y no hace nada:

```
elementos.remove("eter")
Traceback (most recent call last):

  File "p1.py", line 6, in <module>

    elementos.remove("eter")
KeyError: 'eter'
```

El método `.pop()` elimina un elemento al azar:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.pop()
print(elementos)
{'agua', 'tierra', 'fuego'}
```

El método `.clear()` vacía el conjunto al eliminar todos sus elementos:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.clear()
print(elementos)
set()
```

Pertenencia e inclusión

Para saber si un valor pertenece a un conjunto (es uno de los elementos del conjunto) usamos el operador `in`:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
print('aire' in elementos)
True
print('eter' in elementos)
False
```

Los operadores relacionales `<`, `<=`, `>`, `>=` permiten saber si un conjunto está incluido en otro (es un subconjunto del otro):

```
alfabeto = set("abcdefghijklmnopqrstuvwxyz")
vocales = set("aeiou")
print(vocales < alfabeto)
True
```

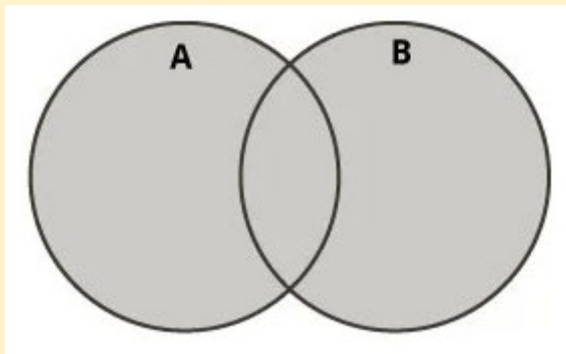
Los operadores $<$ y $>$ requieren que sea un subconjunto estricto, mientras que \leq y \geq admiten los subconjuntos propios (subconjuntos que son iguales al conjunto que los contiene).

Entre conjuntos también se pueden usar los operadores relacionales $==$ y $!=$, que determinan si dos conjuntos son iguales o distintos; dos conjuntos son iguales si tienen los mismos elementos (recuérdese que los elementos no tienen orden).

Álgebra de conjuntos

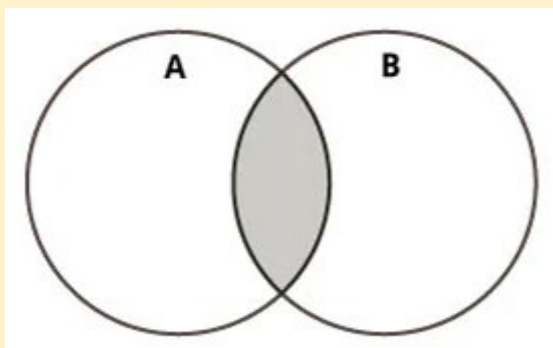
Los conjuntos disponen de cuatro operaciones básicas: unión, intersección, diferencia y diferencia simétrica.

La **unión** de dos conjuntos resulta en un conjunto que tiene todos los elementos de los conjuntos originales, sin repetición. En Python, la unión se hace con el operador $|$ (barra vertical):



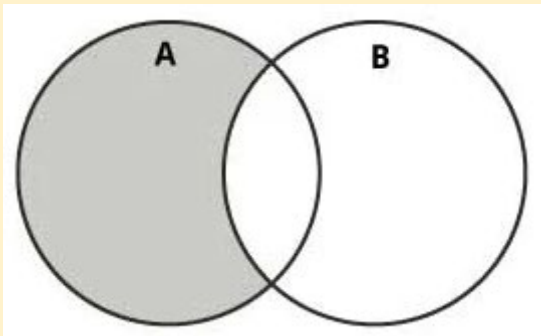
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
unión = set1 | set2
print(unión)
{1, 2, 3, 4, 5}
```

La **intersección** contiene los elementos comunes a ambos conjuntos y se hace con el operador $\&$:



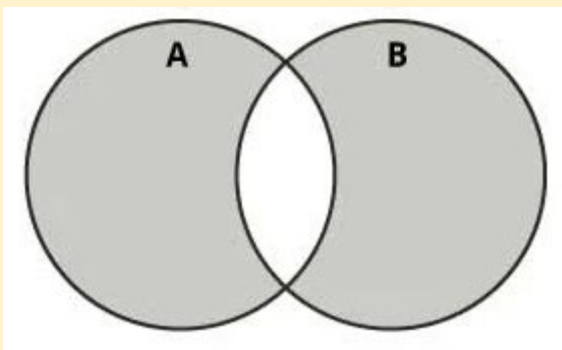
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersección = set1 & set2
print(intersección)
{3}
```

La **diferencia**, que se hace con el operador -, es el conjunto formado por todos los elementos del primer conjunto que no están en el segundo:



```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
diferencia = set1 - set2
print(diferencia)
{1, 2}
```

La **diferencia simétrica** comprende todos los elementos del primer conjunto que no están en el segundo y los del segundo que no están en el primero (los elementos no comunes), y se hace con el operador ^:



```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
diferencia = set1 ^ set2
print(diferencia)
{1, 2, 4, 5}
```

Más sobre strings

Strings

Las strings representan secuencias contiguas de caracteres. Los valores literales de tipo string (str), se encierran entre comillas dobles o simples:

```
var1 = "hola"
var2 = 'hola'
str_var = "Hello world"
print(str_var)          # Muestra "Hello world" (sin las comillas
delimitadoras)
```

Se puede crear una string multilínea usando triples comillas (simples o dobles):

```
var3 = '''Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.'''
```

La longitud de una string (número de caracteres) se conoce usando la función *len()*:

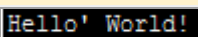
```
str_var = "Hello world"  
print(len(str_var))      # Muestra el valor entero 11
```

Secuencias de escape

Se pueden introducir caracteres especiales en las strings (tabuladores, saltos de línea, caracteres por código,...) usando secuencias de escapes, que son secuencias de caracteres que empiezan con el carácter `\` (llamado barra invertida o *backslash*).

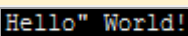
`\'` - Comilla simple

```
print("Hello\' World!")
```



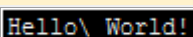
`\"` - Comilla doble

```
print("Hello\" World!")
```



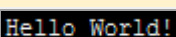
`\\` - backslash

```
print("Hello\\ World!")
```



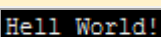
`\a` - Sonido de campana

```
print("Hello\a World!")
```



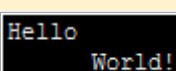
`\b` - Backspace

```
print("Hello\b World!")
```



`\f` - Formfeed

```
print("Hello\f World!")
```



`\xhh` - Carácter por valor hexadecimal

<code>print("Hello\x40 World!")</code>	<code>Hello@ World!</code>
--	----------------------------

\n - Linefeed (salto de línea)

<code>print("Hello\n World!")</code>	<code>Hello World!</code>
--------------------------------------	-------------------------------

\N{name} - Carácter Unicode por nombre

<code>print("Hello\N{BABY ANGEL} World!")</code>	<code>Hello👶 World!</code>
--	----------------------------

\ooo - Carácter por valor octal

<code>print("Hello\100 World!")</code>	<code>Hello@ World!</code>
--	----------------------------

\r - Carriage return

<code>print("Hello\r World!")</code>	<code>World!</code>
--------------------------------------	---------------------

\uxxxx - Carácter Unicode por valor hexadecimal (16 bits)

<code>print("Hello\u041b World!")</code>	<code>HelloЛ World!</code>
--	----------------------------

\Uxxxxxxxx - Carácter Unicode por valor hexadecimal (32 bits)

<code>print("Hello\U000001a9 World!")</code>	<code>HelloΣ World!</code>
--	----------------------------

Mayúsculas y minúsculas

Existen métodos para saber si las letras que contiene una string son mayúsculas o minúsculas y para hacer cambios de un caso al otro.

El método *isupper()* devuelve True si las letras de una string son mayúsculas; el método *islower()* hace lo propio si son minúsculas.

```
test_string = "en un lugar de La Mancha de cuyo nombre"
a = test_string.isupper()    # False
```

El método *upper()* transforma a mayúsculas las letras de una string; lo contrario hacen los métodos *lower()* y *casefold()* (ambos transforman a minúsculas).

```
b = test_string.upper()      # "EN UN LUGAR DE LA MANCHA DE CUYO  
NOMBRE"
```

El método *capitalize()* pone en mayúscula la letra inicial de una string y en minúscula el resto.

```
c = test_string.capitalize() # "En un lugar de la mancha de cuyo  
nombre"
```

El método *title()* pone en mayúscula la letra inicial de cada palabra de una string y en minúscula el resto. Existe un método llamado *istitle()* para verificar si una string sigue este formato.

```
d = test_string.title() # "En Un Lugar De La Mancha De Cuyo  
Nombre"
```

El método *swapcase()* invierte mayúsculas y minúsculas

```
f = d.swapcase() # "eN uN lUGAR dE lA mANCHA dE cUYO  
nOMBRE"
```

Categorías de caracteres

Existen una serie de métodos para conocer qué tipo de caracteres forman una string.

isalnum()

Devuelve True si todos los caracteres de la string son alfanuméricos.

isalpha()

Devuelve True si todos los caracteres de la string están en el alfabeto.

isdecimal()

Devuelve True si todos los caracteres de la string son decimales.

isdigit()

Devuelve True si todos los caracteres de la string son dígitos.

isidentifier()

Devuelve True si la string es un identificador. Una string se considera un identificador válido si sólo contiene letras alfanuméricas (a-z) y (0-9), o guiones bajos (_). Un identificador válido no puede comenzar con un número o contener espacios.

isnumeric()

Devuelve True si todos los caracteres de la string son numéricos

isprintable()

Devuelve True si todos los caracteres de la string son imprimibles.

isspace()

Devuelve True si todos los caracteres de la string son espaciadores (espacios, tabuladores, saltos de línea,...)

Acceso a los componentes de una string

Se puede acceder a un carácter concreto de una string escribiendo la string, o el nombre de la variable que la referencia, seguida del índice de la posición que ocupa el carácter deseado entre corchetes. El índice de la primera posición es 0.

```
str_var = "Hello world"
print (str_var[0])      # Muestra el primer carácter de str_var:'H'
print (str_var[4])      # Muestra el quinto carácter de str_var:'o'
```

También se pueden usar índices relativos a la longitud de la string:

```
str_var = "Hello world"
print (str_var[-1])     # Muestra el último carácter de str_var:'d'
print (str_var[-5])     # Muestra el quinto carácter, empezando por
el final, de str_var:'w'
```

Se puede obtener una substring (subsecuencia) de una string usando el operador de segmento, o slice, ([:]). El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final; si se omite el primero, toma desde el principio hasta la posición del índice anterior al indicado:

```
str_var = "Hello world"
print (str_var[2:5])    # Muestra los caracteres de 3º al 5º: "llo"
print (str_var[2:])     # Muestra los caracteres a partir del 3º:
"llo world"
print (str_var[:5])     # Muestra los caracteres hasta la 5ª
posición: "Hello"
```

Usando un paso de -1 se puede dar la vuelta a la string:

```
str_var = "Hello world"
print (str_var[::-1])   # Muestra "dlrow olleH"
```

Concatenación de strings

El diccionario de la RAE define [concatenar](#) como "unir dos o más cosas"; en el caso de las strings, [concatenar](#) dos strings consiste en formar una nueva string juntando, en orden, dos preexistentes.

En Python existen dos operadores de concatenación: el signo más (+) es el operador de concatenación de strings y el asterisco (*) es el operador de repetición (equivalente a realizar una concatenación repetidas veces).

```
str_var = "Hello world"
print('a' + 'a')        # Muestra "aa"
print(str_var + "TEST") # Muestra el valor de str_var concatenado con
```



```
"TEST": "Hello worldTEST"
print('ab' * 5)           # Muestra "ababababab"
print(str_var * 2)        # Muestra "Hello worldHelloworld"
```

Podríamos decir que la repetición de strings es a la concatenación de strings lo que la multiplicación de números enteros es a la [suma](#) de números enteros.

Nótese que la concatenación junta directamente las strings, sin añadir ningún caracter enmedio.

Localización de una substring

El método `find()` permite localizar una substring dentro de una string:

```
s1 = "Esto es una string de prueba"
print(s1.find("una")) # 8
```

El método `find()` devuelve el índice de la posición de la primera aparición de la string pasada como parámetro ("una" en el ejemplo) en la string sobre la que se aplica (la referenciada por `s1` en el ejemplo).

```
s1 = "Esto es una string, es una secuencia, pero no una lista"
p = s1.find("una") # 8
```

En caso de que la string a buscar no se encuentre en la string donde se busca, el método `find()` devuelve el valor -1.

También se puede usar el método `index()`, aplicable a cualquier tipo de secuencia (strings, [listas](#), [tuplas](#),...).

```
s1 = "Esto es una string de prueba"
print(s1.index("una")) # 8
```

La diferencia es que, si la string buscada no se encuentra, el método `index()` produce un error. Una forma de evitarlo es usar el operador `in` para comprobar primero si la string buscada está:

```
s1 = "Esto es una string de prueba"
if "una" in s1:
    print(s1.index("una"))
```

Si sólo nos interesa saber si la string buscada está, o no, en la string donde se busca, pero no necesitamos conocer su índice, basta con el operador `in`:

```
print("una" in s1) # True
```

El método `find()` admite dos parámetros adicionales opcionales:

- `start` .- índice de la posición donde comenzar la búsqueda (el valor por omisión es 0)
- `end` .- índice de la posición donde dejar de buscar (el valor por omisión es la longitud de la string donde se busca)

Por ejemplo:

```
print(s1.find("e ", 10, 22)) # 20
```

Otros métodos de localización

Los métodos `rfind()` y `rindex()` localizan la última ocurrencia de la string buscada, a diferencia de sus contrarios, `find()` e `index()`, que localizan la primera.

El método `startswith()` devuelve `True` si la string pasada como parámetro coincide con el comienzo de la string sobre la que se busca. Lo equivalente hace `endwith()` por el final.

El método `count()` devuelve el número de apariciones de la string buscada en la string de búsqueda.

El método `partition()` devuelve una tupla de tres elementos: el primero contiene el trozo de la string en que se busca desde el principio hasta la posición anterior a la primera ocurrencia de la string buscada, el segundo contiene la string buscada, y el tercero contiene el trozo de la string en que se busca que va desde la posición posterior a la primera ocurrencia de la string buscada hasta el final. El método `rpartition()` hace lo mismo, pero con la última ocurrencia de la string buscada.

El método `join()`

El método `join()` devuelve una string concatenando las strings contenidas en la secuencia pasada como parámetro, usando como separador la string con la que se llama al método.

Ejemplo:

```
words = ['sun', 'is', 'shining']
espacio = ' ' # un espacio
message = espacio.join(words)
print(message) # sun is shining
```

```
import random
numbers = [str(random.randint(1, 9)) for i in range(5)]
print(numbers) # e. g. ['9', '1', '9', '3', '2']
exercise = '+'.join(numbers)
print(exercise, '= ') # 9 + 1 + 9 + 3 + 2 =
```

En los ejemplos anteriores, las secuencias pasadas al método `join()` son [listas](#), pero pueden ser cualquier clase de secuencia, por ejemplo, otra string, en cuyo caso, el separador se intercalaría entre los caracteres de la misma:

```
s1 = "ABC"
print("-".join(s1)) # A-B-C
```

El método `split()`

El método `split()` crea una lista a partir de una string. Puede ser llamado sin parámetros, en cuyo caso devuelve una lista de strings formada a partir de la string original. Cada elemento de la lista es un trozo de la string original delimitado por *whitespaces* (un *whitespace* es cualquier separador estándar de texto, como un espacio en blanco o un tabulador).

```
a = 'esto es un ejemplo'
b = a.split() # b = ['esto', 'es', 'un', 'ejemplo']
```

Se puede especificar un separador para dividir la string; en el siguiente ejemplo es la coma:

```
a = 'esto es, un ejemplo, de split, con parámetros'
b = a.split(',') # b = ['esto es', ' un ejemplo', ' de split', ' con parámetros']
```

También se puede especificar, con un parámetro adicional, el número máximo de divisiones a realizar; el siguiente ejemplo devuelve una lista con tres elementos, dado que se pasa un 2 como segundo parámetro:

```
a = 'esto es, un ejemplo, de split, con parámetros'
b = a.split(',', 2) # ['esto es', ' un ejemplo', ' de split, con parámetros']
```

El método *replace()*

El método *replace*, devuelve una string que es igual a la string sobre la que se aplica, cambiando las ocurrencias de una substring por otro valor.

Supongamos la siguiente asignación:

```
a = "Susanita tiene un ratón"
```

Supongamos ahora que el ratón de Susanita se escapó:

```
b = a.replace("tiene", "tenía") # b = "Susanita tenía un ratón"
```

El método *replace()* sustituye cada una de las apariciones:

```
a = "Un globo, dos globos, tres globos"
b = a.replace("globo", "elefante") # b = "un elefante, dos elefantes, tres elefantes"
```

Recorte y relleno: los métodos *strip()*, *lstrip()*, *rstrip()*, *ljust()*, *rjust()*, *center()* y *zfill()*

A veces tenemos una string con espacios al principio o al final que no nos resultan útiles y queremos eliminar. El método *strip()* elimina los espaciadores de los extremos de una string:

```
a = "   string con espacios   "
b = a.strip() # "string con espacios"
```

Los métodos *lstrip()* y *rstrip()* hacen lo mismo, pero sólo en uno de los extremos (izquierdo - left o derecho -right):

```
left = a.lstrip() # "string con espacios   "
right = a.rstrip() # "   string con espacios"
```

También existen métodos para justificar una string en un tamaño determinado, ajustándola a la izquierda o a la derecha y rellenando el espacio sobrante con un carácter especificado:

```
a = "hola"
b = a.ljust(10, "_") # "hola_____"
c = a.rjust(10, "_") # "_____hola"
```

Combinando los métodos `ljust()` y `rjust()`, se puede lograr una justificación "centrada", aunque existe un método, `center()`, para hacer eso:

```
mid = 5 + len(a) // 2
d = a.ljust(mid, "_").rjust(10, "_") # "____hola____"
d1 = a.center(10, "_") # "____hola____"
```

Por su parte, el método `zfill()` rellena una string añadiéndole ceros por la izquierda hasta alcanzar un tamaño determinado:

```
num = "12"
num1 = num.zfill(5) # "00012"
```

Obviamente, siempre es posible rellenar con cualquier carácter usando operaciones más básicas:

```
num = "12"
num1 = 'X' * (5 - len(num)) + num # "XXX12"
```

Formateo de strings

texto con formato

Compare las dos capturas de pantalla que se muestran a continuación:

CAPTURA1

```
Aceite 9
Arroz 8
Azucar 10.5
Pollo 30.25
-----
Total 57.75
```

CAPTURA2

```
Aceite 9.00
Arroz 8.00
Azucar 10.50
Pollo 30.25
-----
Total 57.75
```

Ambas capturas muestran la misma información, pero la segunda muestra el texto formateado, alineando por un lado los productos y por el otro los precios, y unificando

el número de dígitos decimales de estos últimos. El texto con formato resulta más legible y, en consecuencia, es más fácil interpretar la información que suministra.

Interpolación de strings: f-strings

Desde la versión 3.6 de Python, el método preferente para formatear texto son los llamados "literales de string formateados" o f-strings, que usan como técnica la interpolación de expresiones:

```
name = "John"
age = 24
print(f'{name} tiene { age } años') #los espacios entre las llaves no
tienen efecto
```

El código previo muestra el mensaje:

```
John tiene 24 años
```

Al anteponer una f, o una F, a un literal de tipo string, indicamos que se trata de una f-string. Una f-string es un literal string en el que se pueden intercalar expresiones de cualquier tipo, encerradas entre llaves, que se evaluarán, se convertirán a string y se insertarán en la f-string en la posición correspondiente.

```
num1 = 12
num2 = 10
print(f"sumar {num1} y {num2} da {num1 + num2}")
sumar 12 y 10 da 22
```

Si queremos incluir los caracteres de llaves en una f-string, habrá que doblarlas:

```
print(f"La palabra {{llaves}} se muestra entre llaves")
La palabra {llaves} se muestra entre llaves
```

Las f-strings se pueden asignar a variables, igual que un literal de string normal:

```
name = "John"
age = 24
message = f'{name} tiene {age} años'
print(message)
John tiene 24 años
```

Modificadores de formato

la captura de pantalla que se muestra a continuación:

```
Aceite  9.00
Arroz   8.00
Azucar 10.50
Pollo   30.25
-----
Total   57.75
```

ha sido generada con el siguiente código (*prices* es un diccionario cuyas claves son nombres de productos, y los valores son sus correspondientes precios):

```
total = 0
for product, price in prices.items():
    print(f'{product:7}{price:5.2f}')
    total += price
print("-----")
print(f"Total    {total:5.2f}")
```

Como se puede observar, se usa una f-string en la que se interpolan las variables cuyos valores se quiere mostrar:

```
f'{product:7}{price:5.2f}'
```

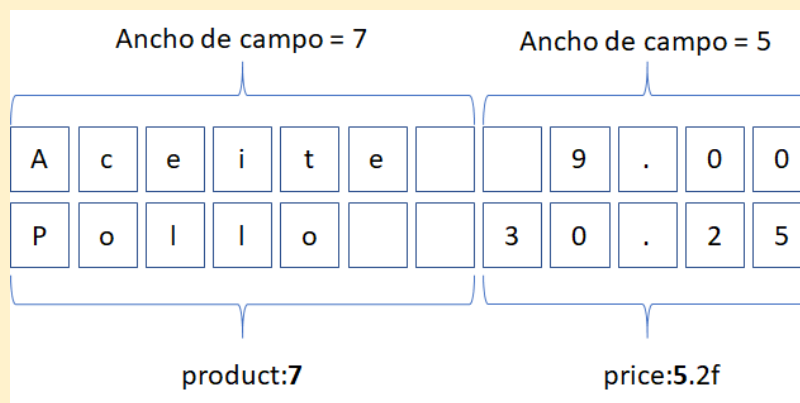
pero, en lugar de estar solo los nombres de las variables, se han añadido modificadores de formato, que son los textos que están entre los dos puntos y el cierre llaves. Los modificadores de formato controlan cómo se va a mostrar la expresión sobre la que se aplican (la que está a la izquierda de los dos puntos que empiezan el modificador).

Modificadores de formato - ancho de campo

Los modificadores de formato controlan como se va a mostrar la expresión sobre la que se aplican (la que está a la izquierda de los dos puntos que empiezan el modificador).

```
f'{product:7}{price:5.2f}'
```

El primer número del modificador de formato indica el "ancho del campo" (número de caracteres) en el que se va a mostrar el valor



Por omisión, el texto se alinea a la izquierda del campo especificado si es una string y a derecha si es un número, pero puede especificarse qué tipo de justificación se quiere, anteponiendo al ancho de campo los símbolos < para alinear a la izquierda, ^ para centrar y > para alinear a la derecha:

```
value = "align left"
print(f"[{value:<20}]")
[align left          ]
value = "align center"
print(f"[{value:^20}]")
[   align center    ]
```

```
value = "align right"
print(f"[{value:>20}]")
[align right]
```

Hay que tener en cuenta que el modificador de ancho de campo define un mínimo; si el valor a mostrar requiere un ancho mayor, se usará el que sea necesario.

Cuando el tamaño del valor a mostrar es menor que el ancho de campo especificado, el espacio sobrante se rellena, por omisión, con espacios, pero puede especificarse otro valor de relleno. Si el valor a mostrar es un número, un cero a la izquierda del ancho de campo indica que se rellene con ceros:

```
for num in range(5, 11):
    print(f"{num:03}")
005
006
007
008
009
010
```

En cualquier otro caso, el carácter de relleno tiene que ir seguido de un código de alineación(<^>), antes del ancho de campo:

```
for num in range(5, 11):
    print(f"{num:*>3}") # > indica alineación derecha
**5
**6
**7
**8
**9
*10
```

Modificadores de formato - Precisión

Un punto seguido de un número indica la precisión con la que se quiere mostrar el valor formateado. Véanse los siguientes ejemplos:

```
import math
print(math.pi) # 3.141592653589793
```

El valor del número pi se muestra sin formatear, de acuerdo con la precisión con la que está representado en memoria.

```
print(f"{math.pi:.2}") # 3.1
```

Se indica una precisión de 2; se muestran los dos primeros dígitos del número π . También se muestra el punto decimal puesto que va en medio de los dígitos que hay que mostrar.

```
print(f"{math.pi:.2f}") # 3.14
```

Se indica una precisión de 2 y también que el valor debe mostrarse como float (la 'f' al final del modificador de formato). Al ser un float, la precisión se refiere al número de dígitos decimales a mostrar.

```
print(f"{math.pi:5.2f}")      #  3.14
```

Lo mismo que el caso anterior, pero en un campo de ancho 5.

```
print(f"{str(math.pi):.2}")    # 3.
```

El número pi se convierte primero a string; la precisión de 2 referida a una string indica que se muestren los dos primeros caracteres de la misma (nótese la diferencia con respecto al caso en que se muestra como un número pero sin especificar que es float).

Modificadores de formato - Notación numérica

Al final del modificador de formato se puede añadir una letra para especificar cómo queremos interpretar el tipo de datos. Véanse los siguientes ejemplos:

```
print(f"{12:5o}")
```

```
14
```

El valor entero 12 (por omisión, en la usual base diez) se imprime en octal (base ocho, que solo usa los dígitos del 0 al 7), en un campo de tamaño 5.

```
print(f"{12:5x}")
```

```
c
```

```
print(f"{12:5X}")
```

```
C
```

El valor entero se imprime en hexadecimal (base dieciséis, que usa los dígitos 0-9 y a-f). Las letras 'a', 'b', 'c', 'd', 'e' y 'f' de la representación hexadecimal se muestran en mayúscula si el modificador 'X' está en mayúscula.

```
print(f"{12:5.1f}")
```

```
12.0
```

Un valor, entero o flotante, se imprime como float (si el valor es entero, se añaden ceros como parte fraccionaria)

```
print(f"{-140:5.1e}")
```

```
-1.4e+02
```

```
print(f"{0.014:5.1E}")
```

```
1.4E-02
```

Un valor, entero o flotante, se imprime en notación científica (coma flotante) indicando el exponente de la [potencia](#) de diez: $-140 = -1.4 \times 10^2$, $0.014 = 1.4 \times 10^{-2}$. Obsérvese que se muestra "e" o "E" según el modificador.

```
print(f"{12:5d}")
```

```
12
```

Un valor *entero* se imprime en formato decimal (la usual base diez, que usa los dígitos del 0 a 9).

Carácter de signo

Podemos controlar qué ocurre al mostrar el signo cuando el número no es negativo. Ejemplos:

```
print(f"{-12:5d}")  
-12
```

Por omisión, el carácter para el signo (– o +) solo se muestra si el número es negativo.

```
print(f"{-12:+5d}")  
-12  
print(f"{12:+5d}")  
+12
```

Si al principio de los modificadores de formato añadimos el carácter '+' se mostrará tanto el signo '-' como el '+', en función de si el valor numérico es negativo o no.

```
print(f"{-12:-2d}")  
-12  
print(f"{12:-2d}")  
12  
print(f"{-12: 2d}")  
-12  
print(f"{12: 2d}")  
 12
```

Añadir un carácter '-' no afecta al resultado (solo se muestra el signo cuando el valor es negativo), pero añadir un espacio en su lugar muestra un espacio (que aquí hemos representado por) en lugar del signo cuando el valor no es negativo.

El método `format()`

Una alternativa a la interpolación de strings que, aunque propia de versiones de Python anteriores a la introducción de este mecanismo, sigue disponible es el método `format()`:

```
num1 = 12 / 7  
print("Primero = {:.4.2f}, segundo = {:2X}".format(num1, 10))  
Primero = 1.71, segundo = A
```

Como se ve en el ejemplo, es muy similar a la interpolación de strings, salvo que las expresiones a mostrar no se introducen en su sitio dentro de la string, sino que se pasan como parámetros del método `format()`, ocupando el lugar que tienen reservado en la string, en principio por orden de aparición.

Existe la posibilidad de dar un nombre a cada expresión, lo que independiza su orden en la string del orden en que se pasan al método `format()`.

```
print("Primero = {num1:4.2f}, segundo = {num2:2X}".format(num1=num1,  
num2=10))  
print("Segundo = {num2:2X}, primero = {num1:4.2f}".format(num1=num1,  
num2=10))
```

Expresiones regulares

Numerosos problemas de tratamiento de rstras requieren la localización en un texto de subrstras que cumplen un determinado patrón: fechas, números de DNI, números de la seguridad social, e-mails, direcciones web, etc. La búsqueda exacta que proporcionan operaciones como el método `find()` de las strings resulta insuficiente para resolver este tipo de problemas.

Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda en una string. Por ejemplo, la secuencia:

`^r..o$`

define un patrón para una string de cuatro letras, empezando por la letra `r` (`^r`), terminando por la letra `o` (`o$`) y con dos letras cualquiera en medio (`..`). Strings que encajan en este patrón de búsqueda son, por ejemplo: `'ramo'`, `'rabo'`, `'rato'`, `'ralo'`,...

Python tiene un módulo llamado **re** que proporciona herramientas para trabajar con expresiones regulares:

```
import re
pattern = '^r..o$'
test_string = 'ramo'
result = re.match(pattern, test_string)
if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

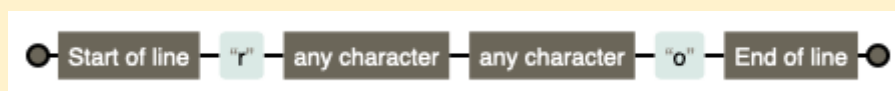
Nótese que las expresiones regulares, al ser secuencias de caracteres, se representan en un programa usando variables de tipo string (la variable `pattern` en el ejemplo). La función `match()` del módulo `re` devuelve un objeto de tipo `match` (del que hablaremos más adelante), si encuentra una coincidencia o `None`, si no la encuentra.

Metacaracteres

Las expresiones regulares son una combinación de caracteres normales y metacaracteres, caracteres que tienen un significado especial, distinto de su valor nominal. En la expresión:

`^r..o$`

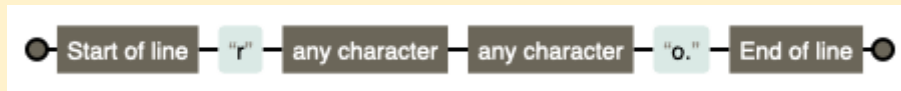
los caracteres `^`, `.` y `$` son metacaracteres; `^` significa "principio de línea", `.` significa "cualquier caracter" y `$` significa "final de línea".



Si necesitamos que un metacaracter se busque por su valor como carácter, tenemos que "escaparlo" con el carácter '\':

```
^r..o\.$
```

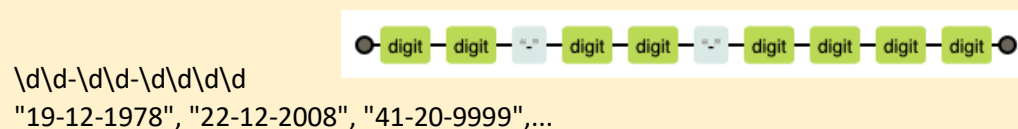
La expresión significa "una string que ocupa una línea, empieza por la letra 'r', seguida de dos caracteres cualquiera y termina con la secuencia 'o.'" ("ramo.", "reto.", "ralo.",...):



Metacaracteres - clases de caracteres

El carácter '.' es un metacarácter que se usa como comodín para indicar que el patrón admite cualquier carácter en esa posición. Varias secuencias de caracteres concretas, que empiezan con el carácter '\' se usan para indicar distintas clases de caracteres.

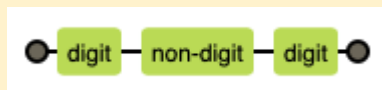
La secuencia '\d' significa "cualquier dígito decimal":



```
\d\d-\d\d-\d\d\d\d
```

"19-12-1978", "22-12-2008", "41-20-9999",...

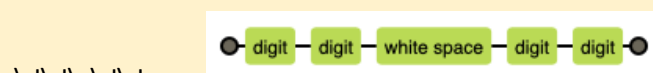
La secuencia '\D' significa "cualquier carácter que no sea un dígito decimal":



```
\d\D\d
```

"1A2", "3-5", "1 5",...

La secuencia '\s' significa "un carácter espaciador" ('\t' - tabulador, '\n' - salto de línea, '\r' - carriage return, '\f' -line feed, ' ').



```
\d\d\s\d\d
```

"12 32", "12\t21", "33\n35",...

La secuencia '\S' significa "cualquier carácter que no sea un espaciador".

La secuencia '\w' significa "un carácter de palabra (la 'w' es de word)". Se consideran "caracteres de palabra" las letras mayúsculas y minúsculas, los dígitos decimales, del '0' al '9', y el guión bajo o underscore '_'.

```
\w\w\w  
"abc", "123", "A_1",...
```

La secuencia '\W' significa "cualquier carácter que no sea de palabra.

Metacaracteres - cuantificadores

Algunos metacaracteres sirven como cuantificadores, indicando repeticiones en el patrón.

El carácter asterisco '*' indica cero o más ocurrencias del patrón que le precede:

```
muu*cho  
"mucho", "muucho", "muuuucho", ...
```

El carácter '+' indica 1 o más ocurrencias del patrón que le precede:

```
mu+cho  
"mucho", "muucho", "muuuucho", ...
```

El carácter '?' indica 0 o 1 ocurrencia del patrón que le precede:

```
estudiantes?  
"estudiante", "estudiantes"
```

Unas llaves con dos números en medio separados por comas, '{m, n}' se usan para indicar un rango de ocurrencias del patrón que las precede; significa "como mínimo m ocurrencias y como máximo n"):

```
a{3,5}  
"aaa", "aaaa", "aaaaa"
```

Atención a cómo se muestra en el diagrama, que especifica "2...4 times". Con ello se pretende indicar que ese camino de "vuelta atrás" se ha de recorrer, como mínimo, 2 veces y, como máximo, 4 veces, por lo que el número de veces que hemos pasar por la "a" cuando recorremos el diagrama desde su extremo izquierdo al derecho será entre 3 y 5 veces, que son los números que indicamos en la expresión regular. Por tanto, las strings mostradas son las únicas que encajan con ese patrón, ya que cualquier otra string no permitiría hacer un recorrido como se ha descrito.

Unas llaves con un número en medio seguido de una coma, '{m,}' indica un número mínimo de apariciones del patrón precedente:

```
a{3,}
"aaa", "aaaa", "aaaaa", "aaaaaaaaaaa", ...
```

Unas llaves con un número en medio, '{m}' se usa para indicar un número exacto de apariciones:

```
\d{2}-\d{2}-\d{4}
"12-10-1978", "15-03-2020", "43-29-0098", ...
```

Atención de nuevo, por ejemplo "once": se ha de recorrer el camino de vuelta atrás una vez (y no más) por lo que deberán haber exactamente dos dígitos seguidos.

Greedy vs no greedy

Los cuantificadores repetitivos, como + o *, son "avariciosos" (greedy), esto quiere decir que intentan tomar el máximo número de repeticiones posible, lo cual puede ser problemático en ciertas circunstancias. En el siguiente ejemplo, se intenta obtener, separadas, todas las etiquetas de un documento html; una etiqueta del lenguaje html empieza con el carácter '<', seguido de cualquier número de caracteres y finaliza con el carácter '>':

```
import re
pattern = '<.*>'
test_string = '<html><head><title>Titulo</title></head><body>Cuerpo
del documento</body></html>'
print(re.findall(pattern, test_string))
```

Aunque el patrón usado parece correcto, debido al carácter "avaricioso" del cuantificador * hace que todo el texto se considere como una única etiqueta html:

```
['<html><head><title>Titulo</title></head><body>Cuerpo del
documento</body></html>']
```

Para solucionarlo, se puede incluir una interrogación detrás del *, para indicarle que se extienda lo menos posible, en vez de lo más posible:

```
import re
pattern = '<.*?>'
test_string =
```

```
'<html><head><title>Título</title></head><body>Cuerpo</body></html>'
print(re.findall(pattern, test_string))
```

Ahora se obtiene:

```
['<html>', '<head>', '<title>', '</title>', '</head>', '<body>',
 '</body>', '</html>']
```

Metacaracteres - grupos

Se pueden agrupar porciones de una expresión regular usando paréntesis:

```
(\d\d)-(\d\d)-(\d{4})
```

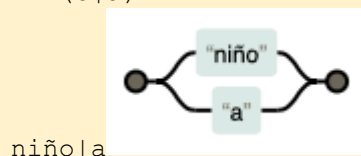
El agrupamiento tiene varias ventajas. La primera es que permite identificar subexpresiones (grupos) a las que se puede acceder por separado en las operaciones de tratamiento de strings que se usen. Suponiendo que el ejemplo anterior se usara para localizar posibles fechas en un texto, al usar paréntesis el día queda identificado como grupo 1, el mes como grupo 2 y el año como grupo 3; la fecha completa sería el grupo 0.

El agrupamiento también permite compactar las expresiones. Las ocurrencias encontradas por el patrón del ejemplo anterior también pueden ser encontradas por el siguiente patrón:

```
(\d\d-){2}(\d{4})
```

El agrupamiento ayuda a delimitar un grupo de alternativas. Se usa una barra vertical '|' para indicar que dos trozos del patrón son alternativas; pero, si no existiese la posibilidad de agrupar, sería difícil expresar determinados patrones de alternativas. Obsérvense los dos siguientes patrones:

```
niñ(o|a)
```



Al no usarse agrupamiento en el segundo caso, todo lo que está a la izquierda de la barra es alternativa de lo que está a la derecha. Para encontrar la misma string que con el primer patrón sin usar agrupamiento, habría que usar un patrón más largo:

```
niño|niña
```

Metacaracteres - backreferences

Otra ventaja de los agrupamientos, es que permiten hacer "backreferences". Supongamos que queremos usar el patrón del primer ejemplo para buscar posibles fechas en un texto:

```
(\d\d)-(\d\d)-(\d{4})
```

Este patrón permite encontrar fechas que usen guiones como separadores ("30-06-1979"), pero no fechas que usen barras ("30/06/1970"). Podemos usar alternativas para contemplar ambos separadores:

```
(\d\d) (-|/) (\d\d) (-|/) (\d{4})
```

Este patrón acepta como separadores guiones o barras, pero permite mezclarlos en una misma fecha ("30-06/1970", "30/06-1970"), lo que no parece muy correcto.

Una "backreference" consiste en insertar en una expresión regular una referencia a un grupo anterior, usando su número precedido de una barra invertida '\' (*backslash*):

```
(\d\d) (-|/) (\d\d) (\2) (\d{4})
```

En este caso, la especificación para el separador de la derecha hace referencia con \2 a que debe ser igual a la sub-string que encajó con el grupo 2, esto es, si el separador de la izquierda es un guión, el otro deberá serlo también, y si es una barra, el otro deberá ser también una barra.

Como sabemos, hemos de duplicar la '\' cuando en un literal de string queremos evitar que se interpreten como secuencias de escape determinadas secuencias comenzadas por dicho carácter. Así haremos pues, para que una backreference no pueda ser interpretada como secuencia de escape octal:

```
import re
pattern = '(\d\d) (-|/) (\d\d) (\\2) (\d{4}) '
test_string = '30/06/1970'
result = re.match(pattern, test_string)

if result:
    print("Búsqueda exitosa")
else:
    print("Búsqueda fallida")
```

Metacaracteres - conjuntos y rangos

Los corchetes se pueden usar en una expresión regular para expresar "cualquier carácter de un conjunto". Por ejemplo, el siguiente patrón permite que el último carácter sea una 'a' o una 'o':

```
niñ[oa]
```

Se puede usar un guión '-' dentro de los corchetes para indicar un rango. El siguiente patrón admite una letra mayúscula (rango A-Z), una letra minúscula (rango a-z), un dígito, o un guión bajo:

```
[A-Za-z\d_]
```

Un carácter '^' (acento circunflejo) al principio de los corchetes sirve para negar su contenido, es decir indica "cualquier carácter que no esté en el conjunto listado a continuación". Nótese la expresión "None of" en la imagen adjunta al siguiente patrón:

```
[^A-Za-z\d_]
```

Metacaracteres - delimitadores

Algunos metacaracteres sirven como delimitadores, esto es, no representan carácter en sí sino requisito que debe cumplir la string en ese punto del patrón.

El acento circunflejo '^', cuando no está dentro de corchetes, indica que ese punto del patrón debe corresponder con principio de línea. Dualmente, el signo de dólar '\$' indica fin de línea. El siguiente patrón abarca una línea completa, de principio a fin:

```
^r..o$
```

También se puede usar la secuencia '\A' que indica principio de string. Nótese que esto es distinto de principio de línea, sólo en el caso de strings multilínea.

La secuencia '\b' indica extremo de "palabra" (en el sentido '\w' visto anteriormente):

```
\bser                p.e. en "al servir"
ser\b p.e. en "al coser"
\bser\b p.e. en "el ser o la nada"
```

La secuencia '\B', por el contrario, indica que ese punto no debe ser extremo de palabra:

```
\Bcos p.e. en "descoser", "cascos"
cos\b p.e. en "descoser", "coser"
\Bcos\b p.e. en "descoser"
```

El módulo re

El módulo re

Python tiene un módulo llamado **re** para trabajar con [expresiones regulares](#). Este módulo ofrece diferentes [funciones](#) para buscar un patrón expresado por una expresión regular en una string, e incluso para sustituir las apariciones de un patrón determinado por otro valor.

Muchas de estas [funciones](#) devuelven un objeto de coincidencia que contiene diversa información sobre la ocurrencia encontrada. Por ejemplo, la función `match()` busca la ocurrencia de un patrón al principio de una string y devuelve un objeto de tipo `Match`, si la hay, o `None`, si no la hay:

```
import re
pattern = '(\d\d)(-|/)(\d\d)(\2)(\d{4})'
test_string = '30/06/1970 es una fecha muy popular'
result = re.match(pattern, test_string)
if result:
    print(result)          # <_sre.SRE_Match object; span=(0, 10),
match='30/06/1970'>
    print(result.start())  # 0
    print(result.end())    # 10
    print(result.span())   # (0, 10)
    print(result.group(0)) # 30/06/1970
```



```
print(result.group(5)) # 1970
```

Los objetos de coincidencia ofrecen diferentes métodos para extraer la información de un objeto de ese tipo:

<code>start()</code>	índice de la posición de comienzo de la ocurrencia encontrada (para el método <code>match()</code> será siempre 0)
<code>end()</code>	índice de la posición de finalización de la ocurrencia
<code>span()</code>	una tupla con los valores de <code>start()</code> y <code>end()</code>
<code>group()</code>	la substring que ha encajado con el grupo cuyo número se pasa por parámetro, además <code>group(0)</code> devuelve la ocurrencia completa

Hay que tener cuidado con el uso del método `group()`; el grupo 0 siempre existe pero, dependiendo de la expresión, otros grupos podrían no existir (se retorna `None`), por lo que habría que preguntar si existen, antes de intentar acceder a ellos. En el siguiente ejemplo, los grupos 2 y 3 son excluyentes, de manera que si en una ocurrencia aparece uno de ellos, no puede aparecer el otro. Por ejemplo, dado el siguiente patrón:

```
\d((\d\d)|(\D\D))_\d
```

El grupo 2 está en la string "123_4", donde no está el grupo 3, y el grupo 3 está en la string "1AB_4", donde no está el grupo 2.

La función `search()`

Si la función `match()` devuelve un objeto `match` que representa una ocurrencia del patrón al principio de la string de búsqueda, la función `search()` devuelve un objeto `match` que representa la primera ocurrencia del patrón en la string de búsqueda, independientemente de la posición donde se encuentre. Si no hay ninguna ocurrencia del patrón en la string de búsqueda, devuelve `None`, igual que hace la función `match()`:

```
import re
pattern = '\d\d(-|/)\d\d\\1\d{4}'
test_string = 'palabra1 30/06/1970 PALABRA2 12-12-2008 palabra3'
result = re.search(pattern, test_string)
print(result)
```

El ejemplo anterior muestra:

```
<_sre.SRE_Match object; span=(9, 19), match='30/06/1970'>
```

La función `finditer()`

La función `finditer()` devuelve una secuencia iterable de objetos `match` con todas las ocurrencias del patrón en la string donde se busca:

```
import re
```

```

pattern = '\d\d(-|/)\d\d\\1\d{4}'
test_string = 'palabra 1 30/06/1970 PALABRA2 12-12-2008 palabra3'
for match in re.finditer(pattern, test_string):
    print(match)
    print(match.start())
    print(match.end())
    print(match.span())
    print(match.group(0))

```

En el ejemplo anterior se muestra:

```

<re.Match object; span=(10, 20), match='30/06/1970'>
10
20
(10, 20)
30/06/1970
<re.Match object; span=(30, 40), match='12-12-2008'>
30
40
(30, 40)
12-12-2008

```

La función findall()

La función findall devuelve una lista con todas las ocurrencias de todos los grupos del patrón en la lista de búsqueda:

```

import re
pattern = '\d\d-\d\d-\d{4}'
test_string = 'palabra1 30-06-1970 PALABRA2 12-12-2008 palabra3'
for occurrence in re.findall(pattern, test_string):
    print(occurrence)

```

El bucle del ejemplo anterior itera, mostrando una ocurrencia cada vez, sobre la lista devuelta por la llamada a findall(), que es la siguiente:

```
['30-06-1970', '12-12-2008']
```

Hay que resaltar que la función findall() devuelve los grupos encontrados en cada ocurrencia. En el ejemplo anterior devuelve una lista de strings, cada una de las cuales es una ocurrencia completa, pero si cambiamos el patrón por:

```
'((\d\d)-(\d\d)-(\d{4}))'
```

Lo que obtenemos es:

```
[('30-06-1970', '30', '06', '1970'), ('12-12-2008', '12', '12', '2008')]
```

una lista de [tuplas](#) en la que cada tupla representa una ocurrencia y cada elemento de la tupla es un grupo de esa ocurrencia. Además, el primer elemento de cada tupla, que muestra la ocurrencia completa, es el grupo 1, dado que hemos puesto paréntesis encerrando todo el patrón; el grupo 0 no se muestra, a menos que no haya otros grupos definidos; si quitamos los paréntesis más externos del patrón lo que tendríamos sería:

```
[('30', '06', '1970'), ('12', '12', '2008')]
```

La función split()

La función `split()` divide la string de búsqueda en una lista de substrings usando las ocurrencias del patrón como separadores:

```
import re
pattern = '\d\d-\d\d-\d{4}'
test_string = 'palabra1 30-06-1970 PALABRA2 12-12-2008 palabra3'
print(re.split(pattern, test_string))
```

El ejemplo anterior muestra la lista:

```
['palabra1 ', ' PALABRA2 ', ' palabra3']
```

Igual que con `findall()`, hay que tener cuidado con la definición de grupos, ya que de haberlos se devuelven también. Si en el ejemplo anterior cambiamos el patrón por:

```
'(\d\d)-\d\d-(\d{4})'
```

obtenemos:

```
['palabra1 ', '30', '1970', ' PALABRA2 ', '12', '2008', ' palabra3']
```

Nótese que no aparece el mes de la fecha porque no se ha definido grupo para él.

La función sub()

La función `sub()` sirve para sustituir las ocurrencias del patrón en la string de búsqueda por un nuevo valor:

```
import re
pattern = '(\d\d)-(\d\d)-(\d{4})'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
new_string = re.sub(pattern, '<DATE>', test_string)
print(new_string)
```

El resultado del ejemplo anterior es:

```
palabra1 <DATE> PALABRA2 <DATE> palabra3
```

El nuevo valor puede especificarse como una expresión regular que modifique la ocurrencia haciendo referencia a los grupos que la componen:

```
import re
pattern = '(\d\d)-(\d\d)-(\d{4})'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
new_string = re.sub(pattern, '\\2/\\1/\\3', test_string)
print(new_string)
```

El resultado del ejemplo es:

```
palabra1 06/30/1970 PALABRA2 10/12/2008 palabra3
```

Se le han intercambiado los grupos 1 y 2 de cada fecha y se han sustituido los guiones por barras.

Existe una función gemela llamada `subn()` que, en lugar de devolver la string modificada, devuelve una tupla en la que el primer elemento es la string modificada y el segundo el número de ocurrencias del patrón que se han encontrado y sustituido.

Flags

A todas las [funciones](#) de manejo de [expresiones regulares](#) se les pueden añadir parámetros flags que modifican su comportamiento. Por ejemplo, el siguiente código:

```
import re
pattern = 'bra\d'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
print(re.findall(pattern, test_string))
```

da como resultado:

```
['bra1', 'bra3']
```

Pero si añadimos un parámetro con el flag `re.IGNORECASE`:

```
import re
pattern = 'bra\d'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
print(re.findall(pattern, test_string, re.IGNORECASE))
```

se obtiene:

```
['bra1', 'BRA2', 'bra3']
```

al realizar la búsqueda ignorando la diferencia entre mayúsculas y minúsculas.

Existen 6 posibles flags, cada uno con una versión larga y una abreviada, de una sola letra:

ASCII, A

Hace que las secuencias de escape `\w`, `\b`, `\s` y `\d` busquen sólo caracteres ASCII

DOTALL, S

Hace que el metacarácter `.` incluya los caracteres de salto de línea (`'\n'`). Por omisión el `.` significa "cualquier carácter que no sea un salto de línea"

IGNORECASE, I

Ignora la diferencia entre mayúsculas y minúsculas

LOCALE, L

Tiene en cuenta las convenciones locales del lenguaje.

MULTILINE, M

Búsqueda en strings multilinea. Hace que los metacaracteres `^` y `$` coincidan, respectivamente, con el comienzo y fin de cada línea en una string multilinea. Por defecto coinciden con el comienzo y fin de la string (suponiendo que hay un carácter de fin de línea al final de la string). Por ejemplo, el siguiente código:

```
import re
pattern = "^a\\w+"
test_string = """amor
agua
casa
azúcar
"""
print(re.findall(pattern, test_string))
```

da como resultado ['amor'], pero si añadimos el flag multiline:

```
print(re.findall(pattern, test_string, re.M))
obtenemos: ['amor', 'agua', 'azúcar']
```

VERBOSE, X (por 'extended')

Facilita usar expresiones más legibles, permitiendo usar espacios para darles formato e introducir comentarios para explicarlas. Por ejemplo:

```
pattern = """
    &[#]                # Comienzo de un referencia numérica
    (
        0[0-7]+         # Formato octal
        | [0-9]+         # Formato decimal
        | x[0-9a-fA-F]+  # Formato Hexadecimal
    )
    ;                    # Punto y coma final
    """
```

en vez de:

```
pattern = "&[#] (0[0-7]+|[0-9]+|x[0-9a-fA-F]+) ;"
```

Si se quiere usar varios flags, se pueden combinar con el operador `|`:

```
print(re.findall(pattern, test_string, re.IGNORECASE | re.DOTALL |
re.VERBOSE))
```

Funciones resursivas

Funciones recursivas (1)

Supongamos que queremos resolver el siguiente problema:

Desarrolle una función llamada **sum_ints** que tome como parámetro una lista con valores de diferentes tipos y devuelva la [suma](#) de los elementos de tipo *int* que haya en dicha lista.

Por ejemplo:

Entra: [10, "Pedro", 42, "Margarita", 18.5, 8]

Devuelve: $10 + 42 + 8 = 60$

Una posible solución es:

```
def sum_ints(the_list):
    """Suma los números enteros de una lista que
       contiene elementos de diferentes tipos
    """
    result = 0

    for item in the_list:
        if type(item) == int:
            result += item

    return result
```

[Funciones](#) recursivas (2)

Supongamos que, en el problema de sumar todos los números enteros de una lista que contiene elementos de diferentes tipos, algunos de esos elementos son a, a su vez, [listas](#):

```
lista = [10, [25, 10, "Pedro"], 42, "Margarita", [12, "casa", 10], 8]
```

y que queremos incluir en la [suma](#) los números enteros que pudiera haber en las [listas](#) que forman parte de la lista original. En nuestra versión inicial de la solución, tratábamos los elementos que eran de tipo *int* e ignorábamos el resto. Sólo tenemos que modificarla para tratar las [listas](#) que nos encontremos, sumando los números enteros que contengan:

```
def sum_ints(the_list):
    """Suma los números enteros de una estructura que
       contiene elementos de diferentes tipos
    """
    result = 0

    for item in the_list:
        if type(item) == int:
            result += item
```

```

        elif type(item) == list:
            result += sum_ints(item)

    return result

```

Básicamente, lo que tenemos es que, si el elemento es de tipo `int`, lo sumamos al resultado, y, si el elemento es una lista, sumamos al resultado la [suma](#) de los números enteros que contenga. ¿Y cómo hacemos para calcular esa [suma](#)? Simplemente aplicamos a la lista contenida el mismo algoritmo que estamos aplicando a la lista contenedora (el de la función `sum_ints`).

Cuando resolvemos una parte de un problema aplicando el mismo algoritmo que estamos usando para resolver el problema completo, decimos que estamos aplicando una **solución recursiva**.

Nótese que, tal como está planteada, la solución funciona también cuando alguna de las [listas](#) contenidas en la inicial tiene elementos que son, a su vez, [listas](#), hasta cualquier nivel de anidamiento.

Búsqueda de una solución recursiva: paso 1 - el tamaño del problema

Muchas veces, que un problema tenga o no una solución recursiva depende de cómo se mire. Por ejemplo, supongamos que queremos desarrollar una función llamada *contains* a la que se le pase una lista de números y un número y devuelva *True* si el número está contenido en la lista y *False* si no lo está.

Si abordamos el problema con una "mentalidad iterativa", la solución es recorrer la lista comprando cada elemento con el valor buscado hasta que lo encontremos o se nos acaben los valores con los que comparar:

```

def contains(container, value):
    """Determina si el valor value está en la lista container"""
    # Recorremos la lista comparando los elementos con el valor
    buscado
    for item in container:
        if item == value:
            return True # Hemos encontrado el valor buscado
    # Hemos comparado con todos los elementos de la lista sin
    encontrar
    # el valor buscado
    return False

```

Si queremos buscar una solución recursiva, tenemos que buscar una forma de descomponer el problema consiguiendo que aparezcan subproblemas de la misma naturaleza.

Hay que tener en cuenta que, los subproblemas incluidos en un problema tienen, necesariamente que ser "más pequeños" que el problema que los incluye.

El primer paso para buscar una solución recursiva a un problema es averiguar qué factores determinan el "tamaño" del problema.

En el caso de nuestro ejemplo, los datos del problema son una lista y un número. Para encontrar si el número está en la lista habrá que irlo comparando con cada elemento de la lista. En el peor caso habrá que compararlo con todos los elementos, y en promedio

con la mitad. La probabilidad de realizar más comparaciones aumenta cuanto más larga sea la lista. Eso es cierto independientemente de cuál sea el valor a buscar; un valor menor no implica que hagamos menos comparaciones, ni uno mayor que hagamos más (a menos que la lista esté ordenada y lo aprovechemos deteniéndonos cuando encontramos un valor mayor que el buscado).

En resumen, el valor buscado no influye en el trabajo a realizar, el tamaño de la lista sí lo hace; el tamaño del problema viene determinado, en este caso, por el número de elementos de la lista.

Siempre que estamos trabajando con secuencias (strings, [tuplas](#), [listas](#)), en general podemos suponer que el tamaño del problema viene determinado por el número de elementos de la secuencia a tratar: normalmente tardaremos 10 veces más en recorrer una lista de 100 elementos que una de 10. Cuando el problema viene definido exclusivamente por datos numéricos, generalmente números mas grandes implican problemas más grandes; por ejemplo, la conocida función [factorial](#) se define como:

```
si n = 0 n! = 1
si n > 0 n! = n * (n - 1)!
```

6! > 5! > 4! > 3! > 2! > 1! > 0! en cuanto al trabajo a realizar.

Búsqueda de una solución recursiva: paso 2 - el caso base

Cuando conocemos el factor que determina el tamaño del problema a resolver, podemos conocer cuál es el caso más pequeño del problema ¿Por qué es esto importante? Podemos suponer que cuesta más solucionar un problema cuanto mayor es, por lo tanto, el caso más pequeño es el más fácil de solucionar: no depende de otros subproblemas recursivos de menor tamaño y, normalmente, tiene una solución directa. Por ejemplo, en el cálculo del [factorial](#), cuando $n = 0$ sabemos que su [factorial](#) es 1 sin necesidad de ningún cálculo; 0 es el número más pequeño del dominio del problema del [factorial](#), que solo está definido para números enteros no negativos.

En el problema de buscar un valor en una lista, dado que hemos determinado que el tamaño del problema depende de la longitud de la lista, está claro que el caso más pequeño es cuando la lista tiene longitud 0, es decir, está vacía ¿Y cuál es el resultado de la búsqueda cuando la lista está vacía? Evidentemente *False*, dado que ningún elemento puede encontrarse en una lista vacía:

```
def contains(container, value):
    """Determina si el valor value está en la lista container"""
    if len(container) == 0:
        result = False
    else:
        # falta resolver el problema para el caso de len(container) >
0
        return result
```

El caso más pequeño de un problema recursivo se conoce como **caso base**, ya que es el que pone fin a la expansión de la recursividad y empieza a retornar una solución parcial. Sin el caso base, la recursión se convertiría en infinita.

Búsqueda de una solución recursiva: paso 3 - descomposición del problema

Una vez resuelto el caso base, hay que estudiar como descomponer el problema general, de modo que obtengamos problemas más pequeños, pero buscando que algunos de estos problemas sean de la misma naturaleza que el original y podamos operar recursivamente sobre ellos (si no tenemos esto en mente, estamos buscando una solución iterativa, no recursiva).

El objetivo, en última instancia, es alcanzar el caso base, que nos proporcionará una solución parcial de partida para construir la solución general del problema.

Podemos plantearnos que lo que buscamos es quitar uno o más "trozos" del problema, que podamos resolver más o menos directamente, de tal forma que lo que quede una vez quitado ese trozo sean subproblemas de la misma naturaleza que el original, pero más pequeños.

En el ejemplo que estamos desarrollando ¿qué le podemos quitar a una lista para quedarnos con una lista mas pequeña? Normalmente definimos una lista como una "secuencia de elementos" que es una forma iterativa de verla (los problemas de secuencias se resuelven iterando por los elementos de las mismas). Necesitamos una forma distinta de definir una lista. Una posibilidad (entre otras) es decir que una lista:

- está vacía, o, si no,
- está formada por un elemento seguido por una lista

Si partimos de esta definición de lista, podemos cuando no está vacía, simplemente, separar el primer elemento y lo que nos queda es una lista con el resto de los elementos:

```
first = list[0]
tail = list[1:]
```

El primer elemento podemos compararlo con el valor a buscar; si coinciden, ya hemos resuelto el problema, si no, solo tenemos que continuar la búsqueda en la lista que hemos separado (resolver recursivamente el problema más pequeño) y devolver como resultado lo que nos depare esa búsqueda:

```
if first == value:
    result = True
else:
    result = contains(tail, value)
```

La solución completa quedaría (tras eliminar las variables *first* y *tail*, que no usamos mas que una vez):

```
def contains(container, value):
    """Determina si el valor value está en la lista container"""
    if len(container) == 0:
        result = False
    else:
        if list[0] == value:
            result = True
```

```

    else:
        result = contains(list[1:], value)
    return result

```

Esquema de una solución recursiva

Ahora tenemos dos formas de resolver el problema de encontrar un valor en una lista:

Solución iterativa

```

def contains(container,
value):
    for item in container:
        if item == value:
            return True

    return False

```

Solución recursiva

```

def contains2(container, value):
    if len(container) == 0:
        result = False
    else:
        if list[0] == value:
            result = True
        else:
            result =
contains2(list[1:], value)
    return result

```

La solución iterativa sigue el conocido esquema de recorrido de una secuencia. La solución recursiva muestra el esquema habitual de una solución de esta clase. La recursividad siempre divide el dominio del problema al menos dos partes: caso base/caso general, lo que, automáticamente implica que en el algoritmo de la solución destaque la presencia de una estructura de selección (if) para discriminar entre ambas partes. Podemos observar como esto ocurre igualmente en otros problemas clásicos:

Factorial de un número natural

```

si n = 0, n! = 1
si n > 0, n! = n * (n - 1)!

```

Solución recursiva

```

def factorial(n):
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1)
    return result

```

n-esimo número de Fibonacci

```

si n = 1, fibo(n) = 1
si n = 2, fibo(n) = 1
si n > 2, fibo(n) = fibo(n - 1) +
fibo(n - 2)

```

Solución recursiva

```

def fibo(n):
    if n <= 2:
        result = 1
    else:
        result = fibo(n - 1) +
fibo(n - 2)
    return result

```

Ficheros de texto

Archivos de texto

Las aplicaciones del mundo real procesan (leen y producen) diversos datos que se guardan en archivos externos. Muchos de ellos son textos. Ejemplos de archivos que tienen un formato de texto incluyen aquellos con extensión .txt, .csv, .html.

Un archivo de texto es una secuencia de caracteres, incluido el carácter de final de línea '\n'.

O podemos pensar en un archivo de texto como una secuencia de líneas. Entonces, cada línea es una secuencia de caracteres que termina con el carácter de fin de línea '\n'. Un archivo de texto puede contener líneas vacías (sólo tienen el carácter '\n').

En general, trabajar con archivos comprende 3 pasos:

- abrir el archivo (establece una conexión entre el programa y el archivo físico en el sistema de ficheros del ordenador)
- leer o escribir en el archivo (transferir información desde el archivo al programa o desde el programa al archivo)
- cerrar el archivo (libera la conexión entre el programa y el archivo físico en el sistema de ficheros)

Apertura de un archivo

Para abrir un archivo de texto, llamamos a la función estándar `open()`:

```
f = open('data.txt', 'r')
```

La función `open()` tiene, generalmente, 2 parámetros de tipo string:

- El nombre del archivo que se quiere abrir y
- La 'r' en caso de abrir archivo para su lectura.

Podemos especificar una ruta absoluta o relativa en la primera string. Pero ahora hemos asumido que el archivo de entrada y la secuencia de comandos de Python (el programa) están en el mismo directorio.

La función `open()` devuelve una referencia al archivo abierto.

O, en otras palabras: el flujo de entrada se ha abierto y la variable de archivo, `f`, proporciona la conexión necesaria.

Si no hay en el disco un archivo con el nombre especificado, se genera un error (`FileNotFoundError`).

En el tercer argumento opcional, podemos especificar la codificación del archivo, por ejemplo.:

```
f = open('data.txt', 'r', encoding = 'utf-8')
```

Tratamiento de un archivo de texto

Supongamos que el archivo 'poem.txt' contiene el texto mostrado a continuación y que queremos leerlo y mostrarlo en pantalla:

*Twinkle, twinkle, little star,
how I wonder what you are.
Up above the world so high,
like a diamond in the sky.*

```
f = open('poem.txt', 'r')  
print(f.readline())      # 'Twinkle, twinkle, little star,\n'
```

Con el método `readline()`, leemos la primera línea del archivo. Este método lee y devuelve una línea, incluyendo el carácter de final de línea.

La variable de archivo, `f`, actualiza la posición real en un archivo después de cada lectura, por lo que la siguiente llamada devolverá la segunda línea:

```
print (f.readline())      # 'how I wonder what you are.\n'
```

Después de leer todas las líneas, la posición real llega al final del archivo y el método `readline()` devuelve "" (una string vacía).

Para leer todas las líneas, a menudo, usamos la instrucción `while`:

```
f = open('poem.txt', 'r')  
line = f.readline ()  
while line != '':  
    print(line, end = '') # la línea ya contiene un '\n'  
    line = f.readline ()
```

También es posible la siguiente versión (una línea vacía equivale a `false`):

```
f = open('poem.txt', 'r')  
line = f.readline ()  
while line:  
    print(line, end = '') # la línea ya contiene un '\n'  
    line = f.readline ()
```

Después de leer todos los datos necesarios, el archivo debe cerrarse inmediatamente:

```
f.close()
```

Después del cierre, se liberan todos los recursos del programa relacionados con el archivo y el archivo puede ser usado por otros programas.

Al leer líneas de un archivo, puede ser interesante ver su contenido real (todos los caracteres, incluido el final de las líneas). La función `repr()` es útil para este propósito:

```
f = open('poem.txt','r')  
line = f.readline()  
while line != '':  
    print(repr(line))  
    line = f.readline()  
  
f.close()
```

La salida:

```
'Twinkle, twinkle, little star,\n'
'How I wonder what you are.\n'
'Up above the world so high,   \t\n'
'Like a diamond in the sky.\n'
```

Observe que la tercera línea contiene más espacios y un tabulador antes del '\n' final.

En Python, caracteres como el espacio, el tabulador y el final de línea se consideran espaciadores (*whitespaces*). A veces, debemos eliminar todos los espaciadores [iniciales](#) y finales antes de procesar una línea. Podemos usar el método `strip()` para esto. El método `strip()` elimina los espaciadores al principio y al final de una string:

```
f = open('poem.txt', 'r')
line = f.readline()
while line != '':
    print(repr(line.strip()))
    line = f.readline()
f.close()
```

La salida:

```
'Twinkle, twinkle, little star,'
'How I wonder what you are.'
'Up above the world so high,'
'Like a diamond in the sky.'
```

Obsérvese que en la instrucción `print` no se usa en esta ocasión el parámetro `sep = "`, sino que se usa el terminador por omisión (`'\n'`), ya que el carácter `'\n'` ha sido eliminado por la operación `strip()`.

El método `strip()` puede tener un parámetro opcional: string indicando el repertorio de caracteres que se quiere eliminar (no necesariamente espaciadores). También están disponibles las variantes `lstrip()`, que elimina los espaciadores, o caracteres especificados del principio de una string y `rstrip()` que hace lo mismo por el final (repasar la semana 9 "Más sobre strings").

Tratamiento de un archivo usando la sentencia `for`

La sentencia `for` puede ser muy útil al procesar el archivo de texto.

En caso de que sepamos la cantidad de líneas, podemos repetir la llamada `readline()`, por ejemplo, 4 veces para imprimir las 4 líneas del poema:

```
f = open('poem.txt', 'r')
for i in range(4):
    print(f.readline(), end='')
f.close()
```

Pero también puede usarse, y facilita el ciclo de lectura, en caso de un número desconocido de líneas:

```
f = open('poem.txt','r')
for line in f:
    print(line, end='')
f.close()
```

Leer un fichero en una string única

A veces necesitamos leer el contenido de un archivo de entrada como una única string:

```
f = open('poem.txt','r')
text = f.read()
print(len(text))          # total number of characters
print(text)               # contents of the file
f.close()
```

El método `read()` puede tener un parámetro, el número de caracteres a leer:

```
f = open('poem.txt','r')
print(f.read(1))          # first character
print(f.read(5))          # next 5 characters
print(f.read())           # the rest of the file's contents
f.close()
```

Escribir en un fichero de texto

El modo de escritura debe especificarse utilizando el carácter 'w' como segundo parámetro de la función `open()`.

```
f = open('output.txt','w')
f.write('We are learning\n')
f.write('about using text files\n')
f.write('in Python\n ')
f.close()
```

Si el archivo no existe, se crea vacío. Si el archivo ya existe, su contenido será sobrescrito:

Cuando se utiliza el método `write()`, todos los caracteres de fin de línea deben escribirse explícitamente.

Las tres líneas de texto también se pueden enviar a un archivo llamando al método `write()` una sola vez:

```
f = open('output.txt', 'w')
f.write('We are learning\nabout using text files\ninPython\n ')
f.close()
```

Hay que **acordarse de cerrar siempre los archivos abiertos**. En caso de no cerrarse, no se puede estar seguro de si la salida realmente se guarda en el disco.

También se puede escribir en un fichero de texto usando la función *print()*. Podemos redireccionar la salida usando un parámetro opcional para escribir en un fichero, en vez de, en la salida estándar:

```
f = open('primes.txt', 'w')

for number in (2, 3, 5, 7, 11, 13, 17, 19):
    print(number, end = ' ', file = f)

f.close()
```

El archivo `primes.txt` contendrá una línea con números separados por un espacio:

```
2 3 5 7 11 13 17 19
```

Si no se hubiese especificado el parámetro `end = ' '`, cada número estaría en una línea diferente.

Cuando se trabaja con archivos, se recomienda usar la cláusula `with`:

```
with open('poem.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line, end='')
```

Se hace referencia al archivo abierto con la variable `f`. Después de ejecutar el bloque `with`, **el archivo se cierra automáticamente**.

Copiemos los contenidos de un archivo. El primer archivo debe abrirse para leer, el segundo para escribir:

```
with open('original.txt', 'r') as fr:
    with open('copy.txt', 'w') as fw:
        fw.write(fr.read())
```

Podríamos enumerar los archivos que se utilizarán en una única línea:

```
with open('original.txt', 'r') as fr, open('copy.txt', 'w') as fw:
    fw.write(fr.read())
```

Adición de contenido a un archivo existente

Además de los modos de lectura y escritura, también es posible agregar contenido a un archivo.

Supongamos que ya hay 3 líneas guardadas en el archivo *names.txt*:

```
John
Paul
George
```

Abriremos este archivo en el modo `append` (añadir) usando `'a'` como segundo parámetro:

```
file = open('names.txt', 'a')
file.write('Ringo\n')
file.close()
```

La posición real en el archivo se establece al final del archivo inmediatamente después de abrirlo. Luego, se anexa la cuarta línea y se cierra el archivo.

Los contenidos resultantes del archivo *names.txt* son:

John
Paul
George
Ringo

Terminología básica de BBDD

Datos

Los **datos** son propiedades de los objetos/entidades, normalmente obtenidos por medición u observación. Para que los datos sean procesados, deben ser expresados. Los datos se pueden expresar por texto, voz, imagen (gráficamente), electrónicamente, etc. Cuando lo simplificamos, podemos decir que los datos se expresan mediante [signos](#) o señales.

Para trabajar con los datos, estos se agrupan en unidades lógicas superiores denominadas **registros** (oraciones). Un registro es una unidad de datos lógica. Sin embargo, un registro no es el elemento de datos más pequeño, un registro (oración) puede ser descompuesto. Al igual que en la vida cotidiana, la oración consta de palabras, y los datos dividen el registro en **atributos**. El atributo es la parte más pequeña direccionable de la oración. Igual que la palabra se puede descomponer en letras individuales, en algunos casos es posible dividir un atributo. Sin embargo, igual que las palabras, perdería su significado.

Atributos

Los atributos pueden ser atómicos o estructurados. Un ejemplo de un atributo estructurado puede ser una dirección (calle, número de casa, ciudad, código postal, ...). Es bueno evitar tales atributos y dividirlos en atributos atómicos. Los atributos tienen una cierta posición en la oración (registro), y tienen su significado. Para que un atributo desempeñe su función correctamente, los valores que adquirirá deben ser significativos. El conjunto de valores permitidos que un atributo puede adquirir se denomina **dominio**.

El dominio no sólo especifica que el atributo NOMBRE DE UNA PERSONA es una cadena de caracteres. Esta es la determinación del tipo de datos (se explicará más adelante con más detalle). El dominio es más específico. Representa todos los valores significativos para el atributo dado. En el dominio del atributo NOMBRE DE UNA PERSONA se puede incluir el valor "Juan", pero el valor "x7br_15" no cae en el dominio de este atributo, incluso siendo una cadena de texto.

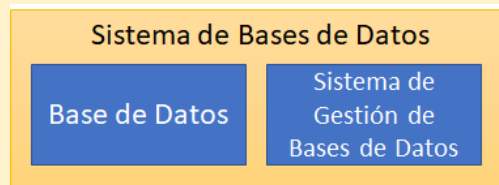
Registros

El tipo de registro determina qué atributos (incluidos los dominios) tiene el registro. El orden de los atributos individuales también juega un papel importante. El tipo de registro (oración) se especifica por sus atributos, por ejemplo, NOMBRE DE UNA PERSONA, ACTIVIDAD, OBJETO. Una frase de este tipo puede ser: "Juan conduce un coche".

La información es intangible. La información son datos que tienen significado. La información es, por lo tanto, un subconjunto de datos. La información puede responder a preguntas, reduciendo así la ignorancia (incertidumbre). La información puede estar contenida tanto en los [signos](#) (o señales) como en su disposición. La apariencia física puede variar (texto, imagen, señales, etc.).

Los datos proporcionan información sólo a aquellos que los entienden, que pueden reconocer su sintaxis y entender la semántica. Lo que es información para alguien, puede ser sólo datos para otro.

Los registros del mismo tipo se agrupan en **archivos de datos**. Para trabajar con ellos (por ejemplo, buscar, eliminar, editar), cada registro debe distinguirse claramente de los demás. Por lo tanto, cada registro debe ser identificado por la clave de archivo. Una clave es un conjunto de atributos que identifican de forma única un registro. El número de atributos que pertenecen a la clave se llama k . El número total de atributos de la oración se denota como n . Siempre se cumple que $k \leq n$. El objetivo es mantenerlo lo más pequeño posible. Todas las claves mínimas crean un espacio K^* . Una de las claves está marcada como la **clave principal**.



Como puede verse en la imagen, un **sistema de base de datos** consta de dos componentes básicos:

- La base de datos,
- Un sistema de gestión de bases de datos

Base de datos

Una **base de datos** es un conjunto de archivos homogéneos y estructurados que almacenan datos para su posterior procesamiento. La distribución de los datos en los archivos individuales responde a un significado global. Todos los datos dentro de un mismo archivo tienen la misma estructura.

Para acceder a la base de datos, existe una herramienta llamada **sistema de gestión de base de datos**. A menudo, también se utiliza la abreviatura DBMS (por Data Base Management System). Un sistema de administración de base de datos es una herramienta de software integrada que permite definir, crear y administrar el acceso a la base de datos y trabajar con ella. Normalmente, es una colección de programas que conforman la interfaz entre los programas de aplicación y los datos almacenados.

El sistema de gestión de bases de datos tiene muchas [funciones](#): en primer lugar, permite manipular los datos; asegura que sólo los usuarios autorizados puedan acceder a los datos; también permite el acceso concurrente de múltiples usuarios; proporciona gestión de transacciones; dependiendo del modelo utilizado, crea una base de datos y

define su esquema (estructura); en caso de fallo, permite la recuperación; y comprueba la integridad de los datos, entre otras muchas tareas.

Los beneficios que aporta el DBMS al acceso directo a los datos son indiscutibles. Los principales incluyen:

- **Abstracción de datos:** el usuario no trabaja directamente con los archivos de origen, sino con estructuras formalizadas en el nivel lógico superior de abstracción.
- **Independencia de los datos:** si los datos físicos cambian, no afecta el trabajo de los programas de aplicación. La interfaz de datos seguirá siendo la misma, cara al exterior.
- **Administración de datos centralizada:** todos los datos están en un solo lugar. Todo es tratado de manera similar. Es posible mostrar una descripción de la estructura de datos.
- **La capacidad de formular consultas ad hoc fuera de los programas de aplicación:** los usuarios pueden hacer consultas aleatoriamente en una base de datos a través del DBMS. No necesitan otros programas para hacerlo.

SQL y Python

¿Qué es SQL?

SQL es un lenguaje estándar para acceder y manipular bases de datos. SQL se convirtió en un estándar del American National Standards Institute (ANSI) en 1986 y de la Organización Internacional de Normalización (ISO) en 1987.

Las siglas SQL significan Lenguaje de Consulta Estructurado (Structured Query Language).

Usando el lenguaje SQL se pueden crear nuevas bases de datos, [crear tablas](#), ejecutar consultas, recuperar datos, insertar, actualizar y eliminar registros, etc.

Existen diferentes versiones del lenguaje SQL pero, para cumplir con el estándar, todos admiten al menos los comandos principales.

Python puede usarse para desarrollar programas que manipulen bases de datos. Estos programas se conectan con un Sistema de Gestión de Bases de Datos, usando un conector, e interactúan con las bases de datos por medio de comandos SQL.

En este tutorial vamos a usar SQLite 3 como Sistema de Gestión de Bases de Datos. Para interactuar con SQLite 3 desde un programa escrito en Python, debemos importar el módulo *sqlite3*:

```
import sqlite3
```

Crear y eliminar BBDD

La manipulación de bases de datos usando el lenguaje SQL se realiza mediante sentencias que se componen de una o varias instrucciones.

Para crear una base de datos se usa el comando "CREATE DATABASE *nombre*";. Por ejemplo, la siguiente sentencia crea una base de datos llamada *FormulaUno*:

```
CREATE DATABASE FormulaUno;
```

Aunque, por convención, en este tutorial pondremos en mayúscula las instrucciones SQL, el lenguaje no hace distinción entre mayúsculas y minúsculas. Obsérvese que la sentencia acaba en punto y coma, esto es obligatorio en algunos SGBD y no en otros, pero es el estándar para separar una secuencia de sentencias SQL.

Desde Python conectamos con una base de datos sqlite usando la función connect:

```
db = sqlite3.connect("FormulaUno")
```

Si la base de datos existe, el comando connect abre una conexión con la misma y la devuelve, quedando asignada a la variable *db* para su uso posterior. Si la base de datos no existe, la crea primero y luego abre la conexión. Si no puede crearla se produce un error.

Cuando terminemos de trabajar con una base de datos debemos cerrar la conexión correspondiente:

```
db.close()
```

No existe un comando para eliminar una base de datos en *sqlite3*. Como la base de datos se almacena en un fichero con la extensión 'db' ('FormulaUno.db'), basta con eliminar ese fichero. En el estándar SQL se usa el comando "DROP DATABASE *nombre*";. Por ejemplo:

```
DROP DATABASE FormulaUno;
```

Creación de tablas

Para crear una tabla en la base de datos activa, se usa el comando:

"CREATE TABLE *table_name* (*columnal* tipo, *columna2* tipo, *columna3* tipo, ...);".

Por ejemplo:

```
CREATE TABLE Pilotos (  
    id INTEGER,  
    Piloto TEXT(20),  
    Escudería TEXT(20),  
    País TEXT(20),  
    Puntos INTEGER,  
    Victorias INTEGER  
)
```

Crea una tabla cuyos registros tienen la estructura:

id	Piloto	Escudería	País	Puntos	Victorias
----	--------	-----------	------	--------	-----------

En un entorno interactivo proporcionado por un SGBD, los comando SQL se escriben tal cual. Para usarlos desde Python, se debe crear un cursor asociado a la base de datos:

```
db = sqlite3.connect("FormulaUno")
cursor = db.cursor()
```

Una vez creado el cursor, se usa para ejecutar los comandos SQL, que se le pasan como parámetros de tipo string:

```
sql = """
CREATE TABLE Pilotos (
    id INTEGER, Piloto TEXT(20),
    Escudería TEXT(20),
    País TEXT(20),
    Puntos INTEGER,
    Victorias INTEGER
);
"""
cursor.execute(sql)
```

En la creación de la tabla se usan los tipo de datos INTEGER y TEXT. SQLite usa los tipos de datos: NULL, INTEGER, REAL, NUMERIC Y BLOB; otros SGBD pueden ofertar una variedad de tipos más amplia. La siguiente tabla muestra la equivalencia entre los tipos de Python y los de SQLite:

Python type	SQLite type
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-encoded)	TEXT
unicode	TEXT
buffer	BLOB

Para borrar una tabla se usa el comando "DROP TABLE *table_name*";. Por ejemplo:

```
DROP TABLE Pilotos;
```

La instrucción INSERT INTO

Para insertar nuevos registros en una base de datos se usa la instrucción INSERT INTO *Tabla (columna1, columna2, columna3, ...)* VALUES (*valor1, valor2, valor3, ...*);. Por ejemplo:

```
INSERT INTO Pilotos VALUES (1, 'Hamilton', 'Mercedes', 'Inglaterra',
250, 8)
```

En Python:

```
sql = "INSERT INTO Pilotos VALUES (?, ?, ?, ?, ?, ?)"
values = (1, 'Hamilton', 'Mercedes', 'Inglaterra', 250, 8)
cursor.execute(sql, values)
```

Nótese que, cuando se usa SQL desde un programa (en cualquier lenguaje, no sólo Python), no se deben incluir los valores de los campos en la string del comando SQL, sino que deben proporcionarse como una tupla separada, sustituyéndolos por interrogaciones en la sentencia SQL; esto es por razones de seguridad, ya que si no, se podría hackear la base de datos usando "inyección de código".

Existe la opción de no proporcionar valores para todas las columnas, en cuyo caso hay que especificar en el comando INSERT TO las columnas para las que se proporciona valores:

```
sql = "INSERT INTO Pilotos (id, Piloto, Escudería, Puntos) VALUES (?, ?, ?, ?)"
values = (2, 'Bottas', 'Mercedes', 188)
cursor.execute(sql, values)
```

En el ejemplo anterior se omiten las columnas País y Victorias, que quedan almacenadas en la base de datos con el valor *NULL*, equivalente al valor *None* de Python.

La instrucción SELECT

Para recuperar información de una base de datos se usa la instrucción "SELECT *columna1*, *columna2*, ... FROM *Tabla*";. Por ejemplo:

```
SELECT Piloto, Puntos FROM Pilotos
```

El resultado de una instrucción SELECT se almacena en una tabla temporal, el *result_set*, que, en un entorno interactivo, normalmente se muestra al usuario, o bien, se usa como parte de otro comando. Para ejecutar la instrucción SELECT desde Python hay que recurrir al consabido cursor:

```
sql = "SELECT Piloto, Puntos FROM Pilotos"
cursor.execute(sql)
```

Para acceder luego, desde Python, al resultado de la sentencia SELECT, se debe ejecutar el método *fetchall()* del cursor, lo que proporciona una lista de [tuplas](#), cada una de las cuales representa un registro del *result_set*:

```
result_set = cursor.fetchall()
for item in result_set:
    print(item)
```

Suponiendo la siguiente tabla *Pilotos*:

id	Piloto	Escudería	País	Puntos	Victorias
----	--------	-----------	------	--------	-----------

1	L. Hamilton	Mercedes	Inglaterra	250	8
2	V. Bottas	Mercedes	Finlandia	188	2
3	M Verstappen	Red Bull	Holanda	181	2
4	S. Vettel	Ferrari	Alemania	156	0
5	C. Leclerc	Ferrari	Mónaco	132	0
6	P. Gasly	Red Bull	Francia	63	0
7	C. Sainz Jr.	McLaren	España	58	0
8	K. Raikkonen	Alfa Romeo	Finlandia	31	0
9	D. Kvyat	Toro Rosso	Rusia	27	0
10	L. Norris	McLaren	Inglaterra	24	0

El resultado del ejemplo anterior sería:

```
( 'Hamilton', 250)
( 'Bottas', 188)
( 'Verstappen', 181)
( 'Vettel', 156)
( 'Leclerc', 132)
( 'Gasly', 63)
( 'Sainz Jr.', 58)
( 'Raikkonen', 31)
( 'Kvyat', 27)
( 'Norris', 24)
```

Nótese que cada registro del *result_set*, es siempre una tupla, aunque tenga un solo elemento; si en ejemplo anterior cambiamos la sentencia SELECT, para obtener sólo los pilotos:

```
sql = "SELECT Piloto FROM Pilotos"
cursor.execute(sql)
result_set = cursor.fetchall()
```

Se obtiene el siguiente valor para *result_set*:

```
[('Hamilton',), ('Bottas',), ('Verstappen',), ('Vettel',),
('Leclerc',), ('Gasly',), ('Sainz Jr.',), ('Raikkonen',), ('Kvyat',),
('Norris',)]
```

El primer piloto no es *result_set[0]*, sino *result_set[0][0]*

Si se quieren recuperar todas las columnas, basta con poner un asterisco en vez de enumerarlas:

```
SELECT * FROM Pilotos;
```

A veces, el resultado de una instrucción SELECT puede tener registros repetidos; por ejemplo:

```
SELECT Escudería FROM Pilotos;
```

da como resultado :

```
[('Mercedes',), ('Mercedes',), ('Red Bull',), ('Red Bull',),
('McLaren',), ('Alfa Romeo',), ('Toro Rosso',), ('McLaren',)]
```

Si queremos evitar las repeticiones, podemos añadir a la instrucción SELECT la cláusula DISTINCT:

```
SELECT DISTINCT Escudería FROM Pilotos;
```

y obtenemos:

```
[('Mercedes',), ('Red Bull',), ('McLaren',), ('Alfa Romeo',), ('Toro Rosso',)]
```

La cláusula WHERE

La cláusula WHERE se usa para establecer condiciones para filtrar los datos de una búsqueda. Por ejemplo, dada la siguiente tabla *Pilotos*:

id	Piloto	Escudería	País	Puntos	Victorias
1	Hamilton	Mercedes	Inglaterra	250	8
2	Bottas	Mercedes	Finlandia	188	2
3	Verstappen	Red Bull	Holanda	181	2
4	Vettel	Ferrari	Alemania	156	0
5	Leclerc	Ferrari	Mónaco	132	0
6	Gasly	Red Bull	Francia	63	0
7	Sainz Jr.	McLaren	España	58	0
8	Raikkonen	Alfa Romeo	Finlandia	31	0
9	Kvyat	Toro Rosso	Rusia	27	0
10	Norris	McLaren	Inglaterra	24	0

El código:

```
sql = "SELECT Piloto, Puntos FROM Pilotos WHERE Puntos > 150"
cursor.execute(sql)
select_result = cursor.fetchall()
for item in select_result:
    print(item)
```

Da como resultado:

```
('Hamilton', 250)
('Bottas', 188)
('Verstappen', 181)
('Vettel', 156)
```

La condición de la cláusula WHERE puede ser compuesta:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Puntos > 150 AND Victorias > 2
```

En las condiciones de una cláusula WHERE se pueden usar:

- Los operadores booleanos (AND, OR, NOT)
- Los operadores relacionales (<, <=, =, <>, !=, >= >)
- Otros operadores (BETWEEN, LIKE, IN)

El operador BETWEEN sirve para establecer un rango:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Puntos BETWEEN 150 AND 200
```

El operador LIKE sirve para establecer un patrón con comodines. En SQLite se pueden usar dos comodines, el carácter '%', que significa "cero o más caracteres" y el carácter '_', que significa "un carácter". Por ejemplo:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Piloto LIKE '%n'
```

encuentra todos los registros cuyos pilotos tienen un nombre termina con la letra 'n', precedida de cualquier número de caracteres (Hamilton, Verstappen y Raikkonen).

Mientras que:

```
SELECT Piloto, Puntos FROM Pilotos WHERE País LIKE '%l_n%'
```

Encuentra los registros cuyo campo país contiene una 'l' separada de una 'n' por un único carácter y con cualquier número de caracteres por delante y por detrás de ese trío de letras (Finlandia y Holanda).

Nótese, de este último ejemplo, que los campos que se incluyen en una cláusula WHERE no tienen que ser los mismos que se incluyen en el resultado de la búsqueda.

La cláusula ORDER BY

Los resultados de una consulta se pueden ordenar usando la cláusula "...ORDER BY *columna1, columna2, columna3*,... ASC|DESC". Por ejemplo:

```
sql = "SELECT * FROM Pilotos WHERE Puntos > 150 ORDER BY Escudería,  
Piloto ASC"  
cursor.execute(sql)  
select_result = cursor.fetchall()  
print(select_result)  
[(2, 'Bottas', 'Mercedes', 'Finlandia', 188, 2), (1, 'Hamilton',  
'Mercedes', 'Inglaterra', 250, 8), (3, 'Verstappen', 'Red Bull',  
'Holanda', 181, 2)]
```

Las opciones ASC y DESC al final significan, respectivamente, "ascendente" y "descendente". Si no se pone nada, los resultados se ordenan en orden ascendente.

Las instrucciones UPDATE y DELETE

Se puede actualizar un registro usando la instrucción "UPDATE *tabla* SET *columna1* = *valor1*, *column2* = *valor2*, ... WHERE *condición*";. Por ejemplo:

```
UPDATE Pilotos SET Puntos = 300 WHERE Piloto = 'Hamilton';
```

La cláusula WHERE en la instrucción UPDATE es casi obligatoria; se puede omitir, pero, en ese caso, la actualización afectará a todos los registros al no haber ningún filtro.

Lo mismo sucede en el caso de querer borrar un registro, lo que se hace usando la instrucción "DELETE FROM *tabla* WHERE *condición*";". Por ejemplo:

```
DELETE FROM Pilotos WHERE Escudería = 'Ferrari'
```

Olvidar la cláusula WHERE en una instrucción *delete* **vaciaría la tabla en cuestión.**