HERRAMIENTAS DE PROGRAMACIÓN

Dr. José Miguel Acosta Martín

viu Universidad Internacional de Valencia

Universidad Internacional de Valencia

Edita
Universidad Internacional de Valencia

Herramientas de programación

6 ECTS

Dr. José Miguel Acosta Martín

Leyendas

Enlace de interés

Ejemplo

Importante

abc Los términos resaltados a lo largo del contenido en color **naranja** se recogen en el apartado **GLOSARIO.**

Índice

Capítulo 1

Herramientas de programación

Objetivos

- Aprender los principales fundamentos de la programación, así como las diferentes estructuras y tipos de datos que la conforman, además de conocer las distintas variables y tipos de operadores disponibles en el lenguaje de Python.
- Comprender el concepto y el enfoque de la programación orientada a objetos en el lenguaje Python.
- Manejar los diferentes tipos de datos y estructuras del lenguaje de programación Python, así como los diccionarios y *dataframes* para el tratamiento de los datos.
- Adquirir los conocimientos necesarios para el uso de las diferentes librerías de integración, exploración, tratamiento y modelización de datos que permiten el acceso a la base de datos.
- Distinguir los procesos *multiprocessing* y *threading* aplicados al lenguaje de programación Python y las diferencias existentes entre el procesamiento simple y el procesamiento distribuido.

Introducción

En este capítulo se van a tratar los principales conceptos de la programación orientada a objetos enfocados al lenguaje de programación Python, así como los fundamentos necesarios para la formación de este lenguaje. Además, se explicará cómo aplicar las diferentes técnicas y hacer uso de la sintaxis de programación, se expondrán las diferentes estructuras de datos disponibles y se describirá cómo hacer uso de los procesos e hilos en las diferentes aplicaciones que lo requieran.

1.1. Fundamentos de programación

Para empezar a escribir código es preciso tener los conocimientos necesarios en las diferentes áreas de la programación, además de un conocimiento avanzando del lenguaje concreto a utilizar, incluidos sus algoritmos y la lógica de su funcionamiento.

Solo a partir de estos conocimientos básicos será posible comenzar el proceso de desarrollo del código, además de analizar y depurar cualquier programa y llevar a cabo el mantenimiento para su correcto funcionamiento.

Los principales conceptos que se han de tener en cuenta son los siguientes:

Programación

Consiste en el necesario proceso de diseño, codificación y depuración, así como en la realización del código fuente de los programas.

La programación tiene asociadas una serie de reglas de obligado cumplimiento, y está compuesta por un conjunto de órdenes, expresiones, instrucciones y comandos que son similares a los del lenguaje natural y que permiten evitar la ambigüedad en sus usos.

Algoritmos

Son un conjunto de operaciones que se encuentran ordenadas de forma lógica y que permiten realizar determinadas acciones preestablecidas para resolver problemas.

Así pues, están formados por una serie de funciones y reglas ya establecidas en el lenguaje de programación que, tras una serie de pasos, son capaces de obtener una solución al problema.

Las diferentes características de los algoritmos son:

- Precisión. Deben resolver problemas determinados e indicar el orden de realización de los pasos.
- Definición. Cada vez que se ejecute el algoritmo, debe proveer el mismo resultado; a no ser que se cambien los parámetros de entrada, los resultados deben ser idénticos.
- Finitud. Un algoritmo siempre debe tener un resultado final, por lo que deberá constar de una serie finita de pasos para llegar al resultado de su ejecución.

Flujograma

Para la representación gráfica de los procesos y los diferentes algoritmos se hace uso del diagrama de flujo o diagrama de actividades, que utiliza una serie de símbolos ya definidos, cada uno de los cuales representa un paso del proceso de la ejecución.

La representación de los procesos de ejecución se define mediante flechas, entre las que se indican y representan mediante su símbolo correspondiente los diferentes pasos del proceso representado.

La principal característica de los diagramas de flujo es que tienen un único punto de entrada o inicio y también un único punto final o fin de proceso.

Las características que debe tener un diagrama de flujo son las siguientes:

- Deben posibilitar que sea sencillo seguir el sentido de las operaciones representadas en los diagramas.
- Deben permitir la realización de correcciones de una forma sencilla y fácil.
- Al no utilizar algoritmos y usar simbología para la representación de los flujos, se pueden interpretar en cualquier lenguaje de programación, ya que no van ligados a ninguna sintaxis en concreto.

A continuación se especifica la simbología del diagrama de flujos:

- Terminal: indica el inicio y la terminación del flujo.
- Disparador: indica el inicio de un procesamiento; además, en su interior se introduce el nombre que lo identifica.
- Operación: representa el inicio de una actividad u operación.
- Decisión: se introduce en el punto donde el flujo de ejecución tiene que establecer los caminos alternativos que puede tomar en función del resultado.
- Notas: se utiliza cuando se quieren añadir comentarios para aclarar, por ejemplo, el funcionamiento de la operación. No forma parte del diagrama de flujo, pero aporta información importante en la documentación.
- Documento: representa documentos que se utilizan en el flujo de la ejecución, tanto la entrada como la salida de ellos.
- Archivo: se refiere a los archivos que comúnmente podemos encontrar en una oficina.
- Conector: permite conectar diferentes partes del diagrama de flujo. Se utiliza cuando existe cierta lejanía o distancia entre ellas.
- Conector de página: permite conectar una hoja con otra diferente en la que continúa el diagrama de flujo.
- Línea de comunicación: permite representar a transmisión de información de un lugar a otro.

Pseudocódigo

Es un falso lenguaje que permite realizar una descripción del algoritmo informático de programación en un alto nivel, de forma muy compacta y utilizando un lenguaje informal, pero que se basa en la manera de desarrollar la estructuras en un lenguaje de programación.

Está diseñado para que sea de fácil comprensión a través de una lectura humana, en lugar de utilizar un lenguaje más complejo como el de programación, que resulta de más fácil lectura para las máquinas que lo van a ejecutar.

Por tanto, permite realizar la descripción del funcionamiento del flujo con total independencia del lenguaje de programación, además de omitir los detalles que complican la comprensión directa cuando se hace una lectura humana.

Así, se suelen omitir las variables y el código que sea muy específico de funcionamiento del flujo, así como las subrutinas, para aportar más simplicidad al pseudocódigo.

A continuación se muestra un ejemplo de pseudocódigo que realiza diferentes operaciones matemáticas:

INICO DE EJECUCIÓN

Leer variables (a, b, c, d)

SumaTotal ← Realizar Suma (a + b + c + d)

RestaTotal ← Realizar Resta (a – b – c – d)

ProductoTotal ← Realizar Producto (a * b *c * d)

Mostrar Resultado (SumaTotal, RestaTotal, ProductoTotal)

FIN DE EJECUCIÓN

Lenguaje de programación

El lenguaje de programación permite expresar los procesos que deben ejecutar las máquinas en un lenguaje formal.

Se usa para crear los diferentes programas que deben controlar el funcionamiento, tanto físico como lógico, de las máquinas que los ejecutan.

Además, permite expresar con mucha precisión los algoritmos que se deben usar, así como la comunicación entre el lenguaje natural humano y el de las máquinas.

El lenguaje de programación está formado por un conjunto de reglas sintácticas y semánticas, junto con una serie de símbolos, que permiten definir la estructura y el significado de cada una de las expresiones y elementos que lo forman.

La programación es el proceso de escritura, prueba y depuración, así como de compilación en caso de que sea necesario, es decir, la formación del código fuente de un programa.

El lenguaje de programación surgió de la necesidad de los programadores de interactuar con las máquinas de una forma sencilla, utilizando un lenguaje que permitiese una comunicación más natural. Las máquinas son

capaces de interpretar los signos que los programadores escriben mediante el uso del lenguaje de programación, y lo reinterpretan de forma interna en código ensamblador, que es el lenguaje que realmente entienden.

A través de múltiples métodos, la programación intenta resolver los diferentes tipos de problemas que puedan surgir, y puesto que para cada método debe haber un código que lo defina, puede haber infinidad de ellos que sea necesario definir.

Por este motivo se han creado los paradigmas de programación, que permiten resolver problemas usando diferentes tipos de modelos para delimitarlos.

A continuación se especifican los diferentes tipos de paradigmas de programación.

Paradigma lineal

Este paradigma es una evolución natural de la programación lineal. Dada la gran cantidad de código que puede haber a la hora de realizar operaciones muy complejas, se introdujo la idea de los saltos.

Los saltos consisten en el desplazamiento de la ejecución del programa a determinados puntos dentro del código, lo cual permite ejecutar diferentes instrucciones que pueden estar ubicadas a lo largo de este, de manera que es posible controlar el flujo de la ejecución.

Las estructuras de control permiten controlar el flujo de la programación al crear ciclos de ejecución de instrucciones, es decir, diferentes bifurcaciones entre los tipos de saltos que el programador indique.

Programación modulada

Incluso con la programación estructurada, si el problema a resolver es excesivamente complejo, la organización del código se vuelve más complicada, por lo que se introdujo la idea de los segmentos principales que ejecutan determinados tipos de procedimientos.

Este tipo de programación permite ejecutar los diferentes segmentos tantas veces como sea necesario sin tener que mover el código dentro del programa, ya que el fragmento encargado de la ejecución puede ser llamado desde cualquier lugar.

Programación orientada a objetos (POO)

Este es el paradigma de programación más aceptado por los programadores, ya que permite procesar un gran volumen de información, al clasificar en forma de plantillas todas las entidades que forman parte del problema.

Las plantillas contienen todas las propiedades necesarias y los procedimientos que puede realizar la entidad que representan, por lo que se llaman clases que pretenden representar un objeto.

1.1.1. Tipos de datos simples: enteros, reales, booleanos en Python y R

Tipos de datos

La información que es necesario procesar en los sistemas informáticos se representa mediante los distintos tipos de datos, que pueden ser simples o compuestos.

Por ejemplo, si tratamos información numérica, el tipo de datos que usaremos serán valores o datos simples, normalmente de tipo real o entero; en cambio, si se quiere trabajar con expresiones lógicas que den un resultado verdadero o falso, se utilizan para ello los datos lógicos.

En el caso de que sea necesario manipular texto se hace uso de los datos de tipo cadena de caracteres o de tipo carácter (si se trata de solo una letra).

En cambio, si lo que se necesita es representar información numérica, lógica o de texto agrupada en tablas, por ejemplo, será necesario usar datos de tipo compuesto.

Los datos simples son los llamados datos elementales o escalares, por ser objetos que no se pueden dividir. Además, se caracterizan por tener asociado un único valor, y los tipos posibles son: entero, real, flotante, booleano y carácter.

En Python en concreto no existe el dato simple de tipo carácter, pero este tipo de datos que solo contienen una letra se pueden representar mediante una cadena de caracteres.

Enteros

En matemáticas, los números naturales son los números enteros (*integer*), incluidos los números negativos y el cero. Ejemplo: 1,2 –50, 0.

Los números que tienen decimales no se consideran números enteros; estos números pueden ser de tipo real o flotante.

En la mayoría de los lenguajes de programación y en Python, los números enteros están representados por la palabra *int*; además, existe la función *type*, que permite conocer el tipo de datos.

En Python, los datos enteros se pueden guardar con una precisión arbitraria, lo cual permite utilizar justamente el número de bytes precisos para guardar el número entero.

Para saber el binario equivalente al número entero guardado se usa la función *bin*(N), que permite convertir el número entero a un **string** con el binario que le corresponde.

Reales

En matemáticas, los números reales son los valores continuos del conjunto de estos números; en programación tienen el mismo nombre.

Para representar los números reales o de coma flotante en el lenguaje de programación se utiliza *float*. No todos los números reales se pueden representar de forma exacta en informática, sobre todo cuando sus cifras decimales en matemáticas son infinitas.

Para resolver el problema de representación de los números reales con decimales infinitos se establece un nivel de precisión que permite que se puedan aproximar lo suficiente al nivel deseado para su utilización.

La representación de los números reales viene marcada por la norma IEEE 754, que permite usar la notificación científica con una mantisa y un exponente separados por la letra; por ejemplo, el número 1000 se puede representar de la siguiente forma: 1e3, que quiere decir diez elevado a tres.

La representación de los números reales en las máquinas se hace con base binaria; en Python se usa la norma IEEE 754 en doble precisión, usando *binary64* con 8 bytes, es decir, 64 bits.

Booleanos

Este tipo de datos se utilizan para representar valores booleanos o lógicos. En Python se utiliza *bool* para su definición.

Los datos booleanos pueden tomar dos valores:

- Verdadero: true o 1.
- Falso: false o 0.

A continuación se muestra un ejemplo del uso de los valores booleanos:

```
>>> a = 8 > 1
>>> a
True
>>> type(a)
<class 'bool'>
>>> 10 > 5
False
```

Carácter

Este tipo de elemento es escalar e indivisible, ya que representa solo a un carácter; un conjunto de caracteres es lo que forma una cadena de texto, los llamados *string*.

Los caracteres se ordenan conforme a la tabla ACSII:

! " # $ % & ' ( ) * +, -. / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

El orden en el que los caracteres están representados se usa para evaluar cuál de ellos es mayor que los otros de acuerdo con el valor numérico asignado en la tabla ASCII.

En Python, el tipo carácter no está definido. En este lenguaje, los caracteres se definen como si fueran de tipo texto, pero en este caso solo contienen una letra; es decir, para su representación se utilizará la cadena de caracteres *string*.

Ejemplo de declaración de una cadena de *string*:

```
>>> type('b')
<class 'str'>
```

1.1.2. Variables, asignación y operadores aritméticos, lógicos y de comparación en Python y R

Variables

Las variables de asignación en matemáticas se usan para representar valores numéricos; se puede utilizar un texto o carácter para su representación.

Normalmente se requiere guardar los valores numéricos, de texto o booleanos que tengan que ser usados más de una vez en el lenguaje de programación para poder acceder a los valores almacenados en las diferentes partes del código.

Una variable almacena su valor en un contenedor, que es un lugar reservado en la memora RAM de la máquina con un identificador asociado, el cual permite identificar dónde se ha guardado dicho valor.

Sin embargo, en Python las variables se guardan de forma que se indica dónde está almacenado su valor, pero en realidad no se reserva ningún lugar en la memoria RAM por cada valor declarado: solamente las propias variables contienen el lugar de referencia de dicho valor en la memoria RAM.

Para darle a la variable un valor determinado se usan las asignaciones. En Python se asocia el valor con la variable en cuestión, para lo cual la variable se sitúa a la izquierda y el valor a la derecha.

```
>>> x = 10
```

Para la asignación se utiliza el símbolo =, que permite indicar que el valor 10 se le está asignando a la variable x.

En este caso, lo que ocurre en Python es que al ejecutar la sentencia anterior se debe crear primero un objeto con el valor 10, en concreto de tipo entero, por ser un número natural, y después se ubica dentro de la memoria; este lugar de la memoria es la identidad del objeto en Python.

A la hora de darle el valor 10 a la variable x, el identificador hará referencia al lugar donde se ha guardado el valor 10 en la memoria, es decir, tanto x como el valor 10 harán referencia al mismo emplazamiento de la memoria.

Si se quiere comprobar que las referencias de identificador, tanto de x como de 10, son las mismas, se puede usar la función *id*(), que devuelve la identidad donde se encuentra el objeto guardado en la memoria, es decir, su lugar en esta.

Los datos simples en Python cumplen siempre tres premisas: valor, tipo e identidad.

Además, Python permite las asignaciones múltiples; a continuación se muestra un ejemplo:

```
>>> x, y, z = 8, 1.1, 5
```

Operadores aritméticos

En el lenguaje de programación, los símbolos que representan las acciones de cálculo son los operadores, los cuales pueden ser aritméticos, booleanos o lógicos y relacionales.

Operadores aritméticos

En Python, los operadores aritméticos disponibles para el cálculo matemático son los siguientes:

Operación	Operador	Expresión	Resultado
Suma	+	a + b	Entero si a y b son enteros; real si alguno es real
Resta	-	a - b	Entero si a y b son enteros; real si alguno es real
Multiplicación	*	a * b	Entero si a y b son enteros; real si alguno es real

Operación	Operador	Expresión	Resultado
División real	/	a / b	Siempre es real
División entera	//	a // b	Devuelve la parte entera del cociente a÷b
Módulo resto	%	a % b	Devuelve el resto de la división a÷b
Exponenciación	**	a ** b	Entero si a y b son enteros; real si alguno es real

Operadores aritméticos con asignaciones

Cuando se necesita incrementar el valor de una variable se requieren los operadores de asignación.

A continuación se describen los operadores de asignación que puede tener el lenguaje de programación:

Operación	Operador	Expresión	Equivalente
Suma y asigna	+=	a += b	a = a+b
Resta y asigna	-=	a -= b	a = a-b
Multiplica y asigna	*=	a *= b	a = a*b
Divide y asigna	/=	a /= b	a = a/b
Divide y asigna parte entera	//=	a //= b	a = a//b
Módulo y asigna	%=	a %= b	a = a%b
Potencia y asigna	**=	a **= b	a = a**b

Operadores lógicos

Son aquellos que realizan acciones sobre datos de tipo booleano. Estas operaciones pueden ser de tipo Y (lógico) o conjunción O (lógico); además, es posible usar la negación de los operadores unitarios, así como la disyunción sobre dos operandos.

En Python existen palabras reservadas para este tipo de operaciones lógicas: and, or y not; y los valores lógicos pueden ser verdadero (true - 1 - cierto) o falso (false - 0 - falso).

A continuación se muestra una tabla de equivalencia de las diferentes operaciones lógicas:

A	B	A or B	A and B	not A	A	B	A or B	A and B	not A
FALSE	FALSE	FALSE	FALSE	TRUE	0	0	0	0	1
FALSE	TRUE	TRUE	FALSE	TRUE	0	1	1	0	1
TRUE	FALSE	TRUE	FALSE	FALSE	1	0	1	0	1
TRUE	TRUE	TRUE	TRUE	FALSE	1	1	1	1	0

Python incluye también operadores lógicos sobre números binarios, que se realizan bit a bit. Efectúan las operaciones con el número decimal equivalente introducido.

Operadores relacionales (de comparación)

Estos operadores son los utilizados para comparar dos valores o expresiones, y el resultado siempre debe ser booleano, es decir, que puede tomar como valores verdadero o falso.

A continuación se muestra una tabla con los diferentes operadores relacionales que se pueden utilizar en el lenguaje de programación:

Matemáticas	Python	Significado	Ejemplo	Resultado
=	==	Igual a	'a' == 'b'	FALSE
≠	!=	Distinto a	'b' != 'B'	TRUE
<	<	Menor que	3 < 2	FALSE
>	>	Mayor que	2 > 3	TRUE
≤	<=	Menor o igual que	3 <= 3	TRUE
≥	>=	Mayor o igual que	3 >= 2	TRUE

Para las expresiones que tienen múltiples operadores se debe seguir un orden de prioridad en las operaciones.

En caso de querer dar prioridad a una operación concreta o hacer un uso más claro del lenguaje, dicha operación se puede cerrar entre paréntesis.

El orden de prioridad con el que se ejecutan las expresiones es el siguiente:

1. Operaciones que se encuentran encapsuladas entre paréntesis.
2. Exponenciaciones.
3. Operaciones unitarias.
4. Operaciones aritméticas.
    a. Multiplicación.
    b. División.
    c. Suma y resta.
5. Operaciones relacionales.
6. Operaciones booleanas.
    a. Negación.
    b. AND.
    c. OR.

Las operaciones con los textos de strings no pueden llevarse a cabo matemáticamente, pero en cambio el operador + realiza la acción de concatenar dos strings, es decir, permite añadir una cadena de texto a otra.

1.1.3. Estructuras algorítmicas de control: secuencial, condicional e iterativa

La programación estructurada genera código y programas más fáciles de depurar; además, este código es mucho más sencillo de leer y actualizar en caso necesario, permite el uso de módulos y funciones que pueden ser reutilizados y hace que el programa sea mucho más fácil de usar.

Estructura secuencial

La composición o estructura secuencial es la forma más simple en programación, ya que consiste en una serie de acciones, sentencias e instrucciones que se pueden procesar secuencialmente una tras otra en forma de bloques.

Las instrucciones incluyen llamadas a otras funciones. Por otra parte, en Python cada instrucción se ejecuta por el intérprete mediante la traducción del lenguaje de programación al código máquina.

A continuación se indica un ejemplo de estructura secuencial:

```
# Permite convertir los segundos a días, horas, minutos, y segundos
s = int(input("Segundos ejecutados: "))
print(s, 'segundos:')
m = s//60
s = s%60
h = m//60
m = m%60
d = h // 24
h = h % 24
print(d, "días,", h, "horas,", m, "minutos y", s, "segundos")
```

Estructura condicional

Este tipo de estructura permite ejecutar un conjunto de instrucciones de manera que otro objeto es la alternativa condicional resultante.

En el caso de que se cumpla con parte de la expresión lógica booleana es cuando se realiza la acción necesaria y se ejecuta el bloque correspondiente de código.

En el caso de que la condición resultante sea falsa, se procede a la ejecución del código programado para cuando la expresión booleana tiene como consecuencia este resultado.

Existen tres tipos de estructuras condicionales:

- Estructura condicional tradicional o simple:

```
>>> x, y, z = 8, 11, 5
```

- Estructura condicional doble (if-else):

```
if condición:
secuencia de instrucciones para condición verdadera
else:
secuencia de instrucciones para condición falsa
```

- Estructura condicional múltiple. Este tipo de estructuras están compuestas de otras estructuras alternativas, es decir, se dan cuando algunas de las secuencias de instrucciones de la condición son verdaderas o falsas y a su vez alguna de ellas tiene estructuras alternativas, de forma que están anidadas. Estas estructuras son muy útiles cuando se tiene que escoger entre múltiples casos.

Estructura iterativa

Este tipo de estructura tiene la cualidad de permitir repetir una instrucción de manera automática. Junto con las estructuras alternativas, dan lugar a los algoritmos y permiten cumplir las reglas del paradigma de la programación estructurada. Las estructuras iterativas son los llamados bucles (*loops*).

Existen diferentes tipos de estructuras iterativas:

- Secuencia de datos. Este tipo de secuencia permite acceder a los elementos de uno en uno, es decir, primero se accede al primer elemento de la secuencia y luego a los sucesivos, sin realizar ningún salto entre ellos.
- Esquemas iterativos. Este tipo de esquemas permiten realizar cálculos y utilizar expresiones aritméticas, booleanas o relacionales sobre los datos:
  - Esquema de recorrido. Consiste en recorrer todos los elementos de la secuencia de datos para obtener un resultado.
  - Esquema de búsqueda. En este caso no es necesario recorrer todos los elementos de la secuencia, sino que se puede acceder de forma aleatoria a cada uno de ellos hasta obtener el resultado esperado.
- Estructura iterativa *while*. Consiste en que el bloque de la secuencia se ejecuta siempre que la condición dada en la sentencia *while* se cumpla.
- Bucles para efectuar sumatorias. Se trata de realizar las sumas de los números que hay en una secuencia. Este tipo de operaciones son muy frecuentes en los problemas matemáticos.
- Sentencias break y *continue*. En Python existen sentencias que permiten interrumpir la estructura iterativa o saltar iteraciones del bucle:
  - Sentencia break. Permite salir de forma inmediata del bucle cuando es llamada. Las sentencias que se encuentren más abajo de la instrucción no serán ya ejecutadas.
  - Sentencia *continue*. Permite saltar el código del bloque, pero solo en la iteración que se está produciendo; es decir, si en la siguiente iteración no se encuentra la sentencia *continue*, el código será ejecutado en su totalidad.

1.1.4. Uso y diseño de funciones Python

Los programas informáticos más complejos necesitan realizar subprogramas o módulos que permitan su reutilización, así como la mejora de la forma y el diseño de los propios programas.

El paradigma de la programación modular sirve en este caso para permitir que el programa sea más sencillo, claro y legible, a la vez que menos complejo, y para que posibilite la reutilización de los subprogramas.

Al reutilizar los subprogramas (también en otros programas, con el consiguiente ahorro de tiempo y esfuerzo) se puede corregir el código por separado, realizando las modificaciones de una forma más simple.

Python permite almacenar los grupos de funciones en módulos, los cuales se pueden interpretar como una biblioteca que contiene las herramientas y funciones necesarias para resolver problemas concretos o para satisfaces algunas necesidades.

Una función es la instrucción o bloque de código que permite realizar una tarea o los cálculos programados. A su vez, tiene un identificador, su nombre, el cual sirve para poder llamar a la función desde cualquier parte del código.

Existen dos maneras establecidas de hacer uso de las funciones de los módulos:

- Importando la función necesaria y aplicando directamente las instrucciones:

```
from math import sqrt,
log10 x = sqrt(10)
dB = log10(x/2)
```

- Importando el módulo. Esta opción nos permite acceder y utilizar todas las funcionalidades del módulo:

```
import math
x = math.sqrt(10)
dB = math.log10(x/2)
```

En Python se pueden diseñar funciones para hacer cálculos y que devuelvan un resultado. Normalmente, este tipo de funciones se utilizan para operaciones matemáticas, aunque también las hay que directamente no devuelven ningún resultado, sino que su cometido es modificar el objeto, como ocurre con la función *random*().

Pero al usar funciones, Python siempre devuelve un resultado: en las funciones de tipo productivo devuelve un valor, y en las de tipo estéril el valor devuelto es *None*.

A la hora de elaborar el diseño de las funciones propias, es decir, aquellas que el programador quiera aparte de las preestablecidas en el lenguaje, se debe proceder de la siguiente forma:

```
def nombre_funcion(parametros):
código de la función
```

Para ello se utiliza la palabra reservada *def*, que permite señalizar que en ese punto empieza a definirse una función. A continuación se indica el nombre de la función, que es el identificador de esta y permitirá llamarla desde distintas partes del código, y entre paréntesis los parámetros de entrada que se quieran incluir.

Para usar la función, Python requiere que esté definida de antemano, lo cual es aconsejable hacer de todas formas al principio del programa.

Cuando se quiere hacer uso de variables globales dentro de las funciones, deben declararse usando la palabra reservada global.

Esto significa que si otra función hace uso de la misma variable global, su valor quedará cambiado para todas las funciones que la usen; es decir, una variable global puede ser cambiada por cualquier función.

```
global a
```

Si se quieren definir parámetros para las funciones Python, es necesario hacerlo mediante un identificador único que sea válido. El tipo de parámetro es el mismo que el tipo de argumento que se envía en el programa.

Los parámetros no permiten ninguna acción que pueda modificar sus valores, por lo que son inmutables. El valor solo puede cambiar en el caso de que la variable sea asignada de nuevo.

El paso de los parámetros se realiza exclusivamente por valor, es decir, se copia el parámetro dentro de la función, pero si este parámetro es cambiado fuera de la función, el cambio no se ve reflejado en el propio parámetro dentro de la función.

```
def funcion_con_parametros(x, y):
```

También es posible indicar en los parámetros de entrada de las funciones el valor por omisión que tendrán si a la hora de declarar la función no se indica su valor.

```
def funcion_con_parametros (x, y=150):
```

En el caso de que en una función haya varios parámetros con valores por omisión, estos se pueden llamar con cualquier orden siempre que se indique el nombre del parámetro y el argumento asignado.

```
def funcion(x, y=1, z=2, w=0):
return (x + y)*(z + w)
print(funcion (3)) # x=3, y=1, z=2, w=0
print(funcion (8, w=2)) # x=8, y=1, z=2, w=2
```

A la hora de depurar el código se puede hacer uso del módulo *doctest*, que permite llamar a la función *testmod*, la cual busca los fragmentos de texto en el *docstring* y todas las sesiones interactivas de Python; esto permite ejecutar todas las sesiones y comprobar que funcionan correctamente.

También existe un tipo de funciones recursivas que pueden ejecutarse a sí mismas hasta que obtienen un resultado básico, es decir, se llaman hasta que llega un punto en que encuentran el valor adecuado y paran la ejecución.

Todas las funciones de Python que se hayan programado se pueden guardar en un módulo para reutilizar en otros programas; esto tiene sentido sobre todo cuando disponemos de muchas funciones de un mismo tema.

Para integrar un módulo personalizado es necesario crear un fichero con extensión .py, que es el que contiene todas las funciones y los demás valores necesarios para el correcto funcionamiento de funciones y módulo.

Resumen

La programación requiere del diseño, codificación y depuración del código según determinadas reglas, y está compuesta por un conjunto de órdenes, comandos, etc. que son similares al lenguaje natural.

Se puede programar utilizando diferentes tipos de paradigmas que especifican la forma en la que se debe estructurar la programación.

Uno de los paradigmas más usados en la actualidad es la programación orientada a objetos.

Capítulo 2

Programación orientada a objetos (OOP). Enfoque de OOP en Python

Desde los inicios de la programación, siempre se ha pretendido que la tarea de realizar programas de *software* sea cada vez más sencilla, flexible y portable.

La OOP es un nuevo paso hacia ese fin, el paso más importante que se ha dado hasta el momento, impulsado por los avances tecnológicos, conceptuales y en cuanto al enfoque de la programación.

Un ordenador es un conjunto de microinterruptores que se pueden apagar o encender. En su origen, estos interruptores eran relés, simples electroimanes que se podían cerrar o abrir; en el caso de que estuvieran apagados, su estado era cero, y si estaban abiertos, su estado era uno.

Desde hace ya muchos años, los microprocesadores son lo que hace que los ordenadores puedan funcionar de forma correcta. Se fabrican para ser programados, en este caso no en binario, sino en hexadecimal.

El primer paso importante que se dio fue la aparición de los lenguajes ensambladores, a bajo nivel, ya que se encuentran fuertemente ligados a la forma de trabajo de cada máquina con la que se programa y opera.

Aunque el lenguaje ensamblador no supusiera una gran diferencia, ya que su única aportación consistía en no tener que escribir en binario, sino en hexadecimal, la revolución se produjo por el hecho de que, por primera vez, lo que se escribía para programar no era entendible por la máquina directamente y necesitaba traducción previa para que esta pudiera entenderlo y ejecutarlo de forma correcta.

Así, el lenguaje ensamblador transcribe las palabras fáciles de escribir (nemónicos) a secuencias de ceros y unos que sí son entendibles por la máquina.

A continuación se muestra un ejemplo de lenguaje hexadecimal y lenguaje ensamblador para que se puedan apreciar las diferencias:

- Hexadecimal: 1A 01 04
- Ensamblador: MOV AX, 04

Como se observa, es bastante más fácil recordar el código del lenguaje ensamblador que el del hexadecimal.

Aun así, el problema que presenta el lenguaje ensamblador es que depende totalmente de la máquina en la que se ejecuta, de manera que solo puede usarse en la máquina en la que se ha programado.

Por ello, el siguiente paso lo constituyeron los lenguajes de alto nivel, que permiten un desarrollo mucho más rápido y simple, además de que resulta mucho más fácil comprenderlos y detectar posibles errores.

Esto se debe a dos factores principales:

- Cada instrucción escrita en el lenguaje de alto nivel puede equivaler a una cantidad ingente de código ensamblador.
- La sintaxis de las instrucciones y los nemónicos que se usan presentan muchas similitudes con el lenguaje normal.

Por tanto, crear un programa en un lenguaje de alto nivel puede llevarnos muy poco tiempo, mientras que si lo hiciéramos en código ensamblador, podríamos tardar meses.

No obstante, es preciso tener en cuenta que los programas elaborados con lenguaje de alto nivel realizan los mismos procesos, pero de forma más lenta que si se hubieran escrito con un lenguaje de bajo nivel. Aunque esto no es demasiado importante, dado que actualmente la potencia de las máquinas se ha incrementado sobremanera; solo en algunos casos muy particulares podría llegar a representar un problema.

Como el lenguaje de alto nivel es independiente de la máquina en la que deba ejecutarse, el mismo código puede servir para varias de ellas, es decir, puede ser traducido al lenguaje hexadecimal en las máquinas destino, siempre que se disponga de traductor.

Si nos centramos en la evolución de la conceptualización, el primer avance de la metodología de programación se produjo con la programación estructurada y la programación con funciones, que están estrechamente relacionadas.

La programación con ensamblador es totalmente lineal, lo que quiere decir que las instrucciones se ejecutan en el mismo orden en el que se escriben; en el caso de que se requiera alterar el orden de ejecución, se pueden realizar saltos en el código.

Conforme crecía el código, seguir el flujo de la programación lineal y estructurada se hizo bastante complejo, por lo cual se creó el concepto de función.

La idea de las funciones es bastante simple: a menudo es necesario realizar procesos que se repiten de forma constante, quizás cambiando algún valor o alguna variable; en este caso se trata ese proceso repetitivo como un subprograma que se puede llamar cada vez que se necesite.

Esto permite reducir el margen de error, al reutilizar el código y disminuir el número de líneas de este, ya que no es necesario repetirlo cada vez que se quiera llevar a cabo esa acción determinada, sino que es más que suficiente con llamar al subprograma (función).

En caso de darse un error en una función, también se puede corregir directamente, sin que sea necesario corregirlo en más lugares.

Las funciones en sí son como cajas negras en las que se introducen valores y devuelven otros como resultado, solo se tienen que programar una vez y se pueden probar por separado para comprobar su correcto funcionamiento. Pueden usarse cada vez que se desee.

Además del concepto de función, existe el de variable de ámbito reducido. En un lenguaje no estructurado, cualquier variable que se usase en el programa era conocida en la totalidad de este; en cambio, en el lenguaje estructurado las variables solo son conocidas por las partes del programa en las que se requiere su uso.

Tras todos estos enfoques se produjo el siguiente avance, la programación orientada a objetos (OOP), que nos permite tener aún mayor dominio sobre el programa.

Antes de la programación orientada a objetos, el control recaía sobre el programador, que, conforme se iba ejecutando el programa, debía tener en consideración los procesos que se realizaban, sus efectos colaterales y las posibles colisiones que pudieran surgir en cualquier momento.

En cambio, en la programación orientada a objetos el programa se autocontrola, por lo que el programador ya no debe tener en cuenta todos los procesos; esto permite que se puedan crear programas muchísimo más complejos.

Los objetos en OOP son entidades autónomas que se pueden controlar a sí mismas, lo cual impide que se mezclen con otros tipos de datos.

Aunque en la programación estructurada ya era posible evitar que se modificasen los datos entre funciones, era el propio desarrollador quien tenía que controlarlo; en cambio, con la OOP es el propio sistema el que controla que no se mezclen los datos.

La forma de reutilizar el código también es bastante superior que con el uso de las funciones, además de permitir una mayor portabilidad.

El enfoque también se entiende como una evolución conceptual de la programación. Existen dos tipos de enfoques diferentes:

- Programación procedural. La mayoría de los lenguajes trabajan de forma procedural, para lo cual se les debe indicar cómo han de alcanzarse los objetivos.
- Programación declarativa. Como se ha indicado, casi todos los lenguajes son procedurales, además de los cuales existe el lenguaje PROLOG, que es declarativo.

El lenguaje declarativo se basa en manipulaciones lógicas: para realizar sus deducciones utiliza la lógica proposicional o lógica de predicados.

En PROLOG no se programa nada, sino que se declaran hechos, y es la máquina la que se encarga de obtener las conclusiones que se han declarado.

El motor principal de este lenguaje es el motor de inferencias. Este tipo de programas se utilizan en el desarrollo de la inteligencia artificial.

2.1. Clasificación, clases e instancias u objetos

Las clases nos permiten empaquetar los datos y las funcionalidades de forma conjunta; cuando se crea una nueva clase, se crea un tipo de objeto.

A su vez, esto permite crear nuevas instancias de este tipo de objeto en cualquier punto del código.

Cada instancia de clase puede tener diferentes atributos que le permiten mantener su estado. Además, las instancias de clases tienen métodos que se encuentran definidos en dichas clases y que permiten modificar su estado.

En comparación con otros lenguajes de programación, Python permite crear nuevas clases con el mínimo indispensable de sintaxis y semánticas.

En Python, las clases tienen todas las características necesarias para la programación orientada a objetos, debido al mecanismo de herencia de clases.

Los objetos pueden tener una gran cantidad de datos de cualquier tipo; al igual que sucede con los módulos, las clases también son dinámicas.

Por otra parte, los objetos tienen individualidad, pero al mismo objeto pueden vincularse múltiples nombres; esto se conoce como *aliasing*, que se usa normalmente para beneficio del programa, ya que los renombres funcionan como punteros.

Hay que tener en cuenta el espacio de nombre: en Python, en una relación de nombres a objetos, los espacios de nombres suelen estar implementados como diccionarios, aunque esta forma de funcionar no es óptima para el rendimiento y se está barajando cambiarla en un futuro.

No hay relación entre los distintos espacios de nombres; se pueden crear en diferentes momentos y con diferentes tiempos de vida, se originan cuando se inicia el intérprete y nunca se borran.

El lugar desde el que puede accederse a un espacio de nombre en Python se llama ámbito. Si un nombre se declara como global, entonces todas las referencias y asignaciones se dirigen de forma directa al ámbito intermedio que contiene todos los nombres globales del módulo.

Cuando se quieren modificar las variables globales se utiliza la declaración *nonlocal*; en el caso de que se quieran solo leer, es tan simple como declarar la variable sin usar esa función.

Normalmente, el ámbito local hace referencia a los nombres locales de la propia función actual; en cambio, fuera de una función el ámbito local hace referencia al mismo espacio de nombre.

Así, el ámbito global de una función definida en un módulo lo es solamente dentro del espacio de nombres de ese módulo.

Unas de las principales particularidades de Python es que si no se declaran las asignaciones de nombre con global o *nonlocal*, siempre van en el ámbito interno.

Las asignaciones no copian los datos, solamente asocian los nombres a los objetos; lo mismo ocurre cuando se borra: solamente se elimina la asociación en el espacio de nombre local.

Cuando se quiere indicar que las variables se encuentran en el ámbito global se usa la declaración global; en cambio, la declaración *nonlocal* permite indicar que las variables se encuentran dentro de un ámbito cerrado.

A continuación se muestra un ejemplo de uso de los espacios de nombre:

```
def ejecutar_test():
    def do_local():
        spam = "local test"
    def do_nonlocal():
        nonlocal test
        spam = "nonlocal test"
    def do_global():
        global test
        test = "global test"
    test = "prueba test"
    do_local()
    print("Declaración Local:", test)
    do_nonlocal()
    print("Declaración No Local:", test)
    do_global()
    print("Declaración Global:", spam)
scope_test()
print("Global:", test)
```

```
Declaración Local: local test
Declaración No Local: nonlocal test
Declaración Global: nonlocal test
Global: global test
```

2.2. Instanciación, métodos y atributos

Sintaxis de definición de clases

La forma más simple de definir una clase es utilizar la palabra reservada *class*, seguida del nombre de la clase y luego del código.

Del mismo modo que sucede con las definiciones de las funciones, es necesario que las definiciones de las clases se ejecuten antes de que tengan algún efecto; por ello se puede situar una definición de clase dentro de un *if* o dentro de una función.

Las declaraciones dentro de una clase son las definiciones de las funciones, aunque también están permitidos otros tipos de declaraciones.

Cuando se requiere introducir una definición de clase, se debe crear un nuevo espacio de nombre, el cual sirve como ámbito local, por lo que todas las asignaciones de variables se encontrarán dentro del espacio de nombre creado. De forma general, las definiciones asocian el nombre de las funciones nuevas en el mismo sitio.

Al finalizar la definición de clase es cuando se crea el objeto clase, que básicamente es el que envuelve los contenidos del espacio de nombre que ha sido creado por la definición de la clase. El ámbito local original es restablecido y el objeto clase se asocia al nombre que se le puso a la clase en el encabezado cuando se realizó su definición.

Objeto clase

Los objetos clase permiten dos tipos de operaciones:

- Hacer referencia a atributos.
- Hacer referencia a instanciación.

En Python se utiliza la sintaxis estándar de las referencias a atributos (objeto.nombre). Para ser válidos, los nombres tienen que estar en el espacio de nombres desde que se creó la clase.

Ejemplo de declaración de objeto de clase:

```
class MiClase:
  “Mi clase de prueba”
  i = 12345
  def f(self):
    return ‘hola mundo’
```

Para la instanciación de una clase se usa la notación de funciones, pero sin poner parámetros:

```
x = MiClase()
```

Para poder instanciar la clase es necesario llamar al objeto; pero esto simplemente crearía un objeto vacío, por lo que para crear un objeto con instancias en un estado inicial es necesario definir el método especial ¬¬¬¬__init__().

Ahora, cuando la clase sea invocada, se llamará al código que se encuentra dentro del método especial __ init__(). Además, este método puede tener parámetros de entrada, por los que permite una mayor flexibilidad.

Objetos de instancia

La operación que entienden los objetos de instancia es la referencia de atributos. Hay dos tipos de nombres de atributos:

- Atributos de datos. Son las variables de instancia o variables de miembro. Los atributos de datos no tienen que ser declarados, por lo que son creados la primera vez que se les asigna algo.
- Atributos de métodos. Un método es la función que pertenece a un objeto. En Python, el método no está limitado a instancias de clase, por lo que otros tipos de datos también pueden tener métodos.

Los nombres válidos de métodos de un objeto de instancia vienen determinados por su clase, por lo que todos los atributos de clase que son objetos son los que definen los métodos correspondientes de cada una de sus instancias.

Objetos método

Normalmente, para que un método sea llamado, primero ha de haber sido vinculado. Para llevar a cabo la vinculación se puede proceder de la siguiente forma:

```
a.metodo()
```

La característica principal de los métodos es que el objeto es pasado como primer argumento de la función. Por lo general, llamar a un método con una lista de n argumentos es igual que llamar a la función correspondiente con una lista de argumentos que se crea insertando el objeto del método al principio del primer argumento.

Cuando un atributo sin datos de una instancia se referencia, se puede acceder a la clase de la instancia. Si el nombre hace referencia a un atributo válido de clase que además es una función del objeto, se crea un objeto que hace referencia a la instancia y al objeto función, que se unen en un objeto abstracto; este objeto abstracto sería el objeto método.

En cambio, cuando el objeto método se llama con una lista de argumentos, se debe crear un nuevo listado de argumentos a partir del objeto instancia y dicha lista de argumentos pasados, por lo que finalmente el objeto función puede ser llamado con esta nueva lista de argumentos.

Variables de clase y de instancia

Las variables de instancia se usan por norma general como datos únicos de cada instancia, y las variables de clase se usan para los atributos y métodos compartidos por todas las instancias de clase.

Cuando se comparten datos, estos pueden tener efectos adversos e inesperados que pueden involucrar a objetos mutables, como pueden ser las listas o los diccionarios.

2.3. Encapsulamiento, herencia y polimorfismo

Encapsulamiento

Cuando se define un objeto como un conjunto de datos y métodos, se debe tener en cuenta que los métodos son procedimientos que trabajan con los datos del objeto.

Por ello, aunque existe la posibilidad de cambiar los valores de los atributos dentro de un objeto, en muchas ocasiones es necesario que esas modificaciones no se hagan de forma directa, sino a través de los métodos que controlan los parámetros de entrada y la operación de la asignación de valor.

Para hacerlo es necesario definir los atributos como privados; las variables que contiene un guion bajo (__) se consideran atributos privados.

Para implementar la encapsulación y no permitir el acceso directo a los atributos, se pueden poner atributos ocultos y declarar los diferentes métodos necesarios para poder acceder a ellos y modificarlos.

En Python, las propiedades *getters* son las que permiten implementar la funcionalidad de encapsulamiento, lo cual posibilita el acceso a estos métodos como atributos.

En cambio, los métodos *setters* son aquellos que permiten la modificación de los atributos a través de métodos.

A continuación se muestra un ejemplo de uso de encapsulamiento usando los métodos *getters* y *setters*:

```
class cuadrado():
        def __init__(self, diagonal):
                self. diagonal = diagonal
        @property
        def cuadrado (self):
                print("Estoy dando la diagonal")
                return self.__ diagonal
        @ cuadrado.setter
        def cuadrado (self, diagonal):
                if diagonal >=0:
                        self.__ diagonal = diagonal
                else:
                        print("La diagonal debe ser positiva")
                        self.__ diagonal =0
```

Herencia

Una de las principales características de las clases es que pueden soportar la herencia. A continuación se muestra un ejemplo de la sintaxis para hacer referencia a una clase derivada de otra clase:

```
class ClaseDerivada(ClaseBase):
  <statement-1>
  <statement-N>
```

El nombre de la clase base está definido por el ámbito que contiene la definición de la clase derivada, y se permiten otras expresiones arbitrarias. Esto se utiliza sobre todo cuando la clase base está definida como otro módulo. A continuación se muestra un ejemplo:

```
class ClaseDerivada(nombreModulo.ClaseBase):
```

Así, la ejecución de una definición de clase derivada se efectúa de la misma forma que la de una clase base.

Esto permite resolver las referencias a atributos. Si un atributo solicitado no se encuentra en la clase, la búsqueda continúa por la clase base; esta regla se puede aplicar a modo de recurso si la clase base también deriva de otras clases.

Por todo ello, cuando se instancian las clases derivadas lo que se hace es buscar el atributo de clase correspondiente; se desciende por toda la cadena de clases base si es necesario y en cuanto se entrega un objeto de función válida es cuando la referencia al método también se valida.

A su vez, las clases derivadas pueden redefinir los métodos de su clase base, ya que los métodos no tienen privilegios especiales cuando llaman a otros métodos del mismo objeto.

Esto quiere decir que un método de la clase base puede llamar a otro método definido en la misma clase base y que igualmente puede terminar llamando a un método de la clase base derivada que lo haya redefinido.

Los métodos redefinidos en una clase derivada pueden extender el funcionamiento de la clase base usando el mismo nombre; para hacer esto es necesario llamar a la clase base de la siguiente forma:

```
ClaseBase.metodo(self, argumentos)
```

Las funciones integradas que funcionan como herencia dentro de Python son:

- *Isinstance*(): permite verificar el tipo de una instancia.
- *Issubclass*(): permite verificar la herencia de las clases.

Polimorfismo

El concepto de polimorfismo quiere decir que si una porción de código llama a un determinado método de un objeto, podrán obtenerse distintos resultados según la clase objeto desde la cual se llame.

Esto es posible porque distintos objetos pueden tener un método con el mismo nombre, pero que a su vez realice distintas operaciones.

Para ello existe el concepto de interfaz, que tiene el conjunto de funciones, métodos o atributos con los diferentes nombres específicos.

Una interfaz es como una especie de contrato entre el programador que crea la clase y el que la utiliza para definir los diferentes elementos que debe tener la clase.

Por ello, la idea de polimorfismo se basa en que se pueden utilizar distintos tipos de datos a través de una interfaz común.

También será necesario definir un método con el mismo nombre en distintas clases de la forma que esté determinada por una interfaz; a esto se le llama redefinición.

Para que un bloque de código se pueda considerar polimórfico es preciso que dentro de ese código se realicen llamadas a los métodos que permitan su redefinición en las distintas clases necesarias.

Como en Python no es necesario especificar el tipo de parámetros que recibe una función, de forma natural son polimórficas; en cambio, en otros lenguajes de programación solo algunas funciones en concreto son polimórficas, por lo que se hace más complicado este comportamiento en este tipo de lenguajes.

Se deben proveer diferentes implementaciones de los métodos que tienen el mismo nombre pero que han de comportarse polimórficamente, como pueden ser las redefiniciones de los métodos __str__ o __cmp__.

Python permite la sobrecarga de operadores, un proceso que se realiza mediante la redefinición de algunos de los métodos especiales del lenguaje.

La sobrecarga suele estar presente en todos los lenguajes orientados a objetos y permite la posibilidad de tener, dentro de una misma clase, dos métodos que se llamen igual pero que puedan recibir varios parámetros diferentes. El tipo de método al que hay que llamar se termina decidiendo por los parámetros y no por el tipo del objeto que lo contiene, al coincidir en su nombre, pero no en los parámetros de entrada.

En Python no existe sobrecarga de métodos, ya que, al no tener que definir los tipos de los parámetros, no es posible diferenciar a qué método se debe llamar. En cambio, sí que existe la sobrecarga de operadores, ya que cuando se encuentra un operador, se llama a los distintos métodos según el tipo de variables que se quieran realizar.

A continuación se muestra un ejemplo de polimorfismo:

```
def frecuencias(secuencia):
  "Calcula las frecuencias de aparición de los elementos de la secuencia recibida y devuelve un diccionario con elementos: {valor: frecuencia}"
  # se crea el diccionario sin datos
  frec = dict()
  # ejecuta la secuencia
  for elemento in secuencia:
    frec[elemento] = frec.get(elemento, 0) + 1
  return frec
```

Resumen

La programación orientada a objetos es un paradigma que permite programar de una forma específica. El código se organiza en clases, las cuales, a su vez, permiten crear objetos que se pueden relacionar entre sí.

La OOP permite reutilizar el código, evitar la duplicación y crear programas de manera eficiente, ya que son mucho más fáciles de depurar.

También impide el acceso indebido a los datos, y posibilita encapsular el código y abstraerlo.

Capítulo 3

Tipos y estructuras de datos. Árboles, colas y pilas

Una estructura de datos es una colección de valores y la relación que existe entre los valores y las operaciones que se puedan realizar sobre ellos. Se refiere a la forma en cómo los datos se organizan y cómo se pueden administrar, por lo que una estructura de datos describe el formato en el que los valores tienen que ser almacenados, además de cómo tienen que ser accedidos o modificados.

Lo que hace única una estructura de datos es el tipo de problema que permite resolver: algunas veces nos podrá servir una estructura de datos muy simple, que solo pueda almacenar un número muy limitado de datos, sin gestión de índices, entre otras opciones posibles, porque nuestro problema no necesite de más funcionalidades. Si necesitamos almacenar una mayor cantidad de datos y tener más control sobre ellos, seguramente se necesitará una estructura de datos que nos permita tener una mayor flexibilidad con los datos almacenados.

Las estructuras de datos permiten manipular los datos de una forma organizada, por lo que la tarea de desarrollo y programación utilizando este tipo de estructuras será mucho más sencilla, al tener parte ya del trabajo realizado.

En cualquier caso, cuando hablamos de estructura de datos, lo primero es entender cómo los datos se representan en la memoria, con lo que, a su vez, se puede determinar cómo se van a guardar en la estructura de datos:

- **Estructuras contiguamente asignadas**: se componen de bloques de memoria únicos e incluyen a los *arrays*, matrices, *heaps, hash* y *tables*.
- **Estructuras enlazadas**: se componen de distintos fragmentos de memoria que se encuentra unidos por *pointers* o puntero; estos incluyen a las *lists*, tres y *graphs*.

- **Contenedores**: son estructuras que nos posibilita almacenar y recuperar los datos en un determinado orden, sin que importe el contenido; en este tipo de estructuras se encuentra los *stacks* y *queues*.

Como introducción aclaratoria se hace referencia a las principales estructuras de datos que se usan comúnmente en el desarrollo de *software*:

Array

Este tipo de estructura es la más fundamental. Es del tipo contiguo y de tamaño fijo, de modo que cada elemento se puede encontrar de una forma muy eficiente, ya que se encuentra ubicado por su índice correspondiente.

A continuación se muestra une ejemplo de cómo se puede guardar el abecedario en un *array*:

índice	1	2	3	4	5	6	7	8	9	10
elemento	A	B	C	D	E	F	G	H	I	J

Los *arrays* tienen ventajas respecto a otras estructuras de datos, entre las que se han de considerar:

- **Acceso de tiempo constante:** cada *index* tiene un espacio contiguo en la memoria de cada elemento, por lo que el *array* apuntan directamente a una dirección de memoria.

  De esta forma es posible acceder de forma arbitraria a los datos de una forma muy eficiente, ya que es una operación instantánea, al saber la dirección de memoria exacta donde queda alojado el dato.

- **Eficiencia de espacio:** los *arrays* son simples datos, por lo que no es necesario asignar más espacio necesario a información extra para poder localizar los elementos almacenados en el *array*.

- **Localidad de memoria:** normalmente, en las estructuras los datos tienen que ser iterados, por lo que los arrays representan una buena forma de realizar los recorridos de los datos, al tenerlos perfectamente localizados en memoria. Así, se puede aprovechar al máximo la memoria caché del sistema.

También tienen desventajas. Por ejemplo, no se puede ajustar su tamaño una vez se está ejecutando el código, por lo que no podemos manipular su tamaño cuando necesitemos aumentarlo, si es necesario. Aquí es donde entra en juego el concepto de *dynamic arrays,* que permite crear un nuevo *array* doblando el tamaño del mismo cada vez que se necesite crecer y volver a copiar los datos del *array* anterior al nuevo *array*.

Los *arrays* son la base de los diferentes tipos de estructura que de datos. Este es el caso de las matrices o tablas, que son *arrays* de múltiples dimensiones, es decir, un *array* bidimensional, en el caso de que fuera de dos dimensiones, y así sucesivamente.

Estructuras enlazadas

Son las estructuras que apuntan a una dirección de memoria donde se tiene que encontrar el valor. Los *pointers*, en este caso, son los encargados de mantener los diferentes enlaces entre valores, de forma que es posible tener una secuencia de valores todos enlazados por cada uno de los *pointers*.

Hay que tener en cuenta que los *pointers* hacen que los valores no tengan que estar siempre en memoria uno a continuación de otro, como si ocurre con los *arrays*. En este caso, se pueden tener los valores ubicados en distintas zonas de la memora y aun así seguir siendo una colección de valores consecutivos.

A continuación, se representa una secuencia de valores que está enlazada por punteros:

En el ejemplo anterior, se puede observar que se tiene una secuencia de valores enlazados por los *pointers*, que son los llamados nodos.

Cuando se utilizan las estructuras enlazadas, es necesario guardar el valor que se requiera, pero, además, se debe guardar en un espacio extra la dirección de memoria del valor.

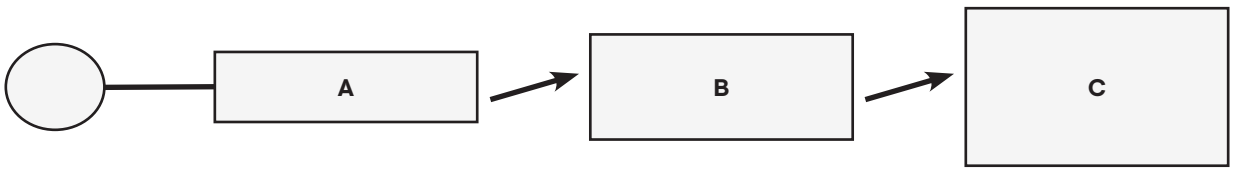
Linkedlist

Una lista es una estructura que representa un número contable de valores que se encuentran ordenados; un mismo valor puede repetirse y considerarse un valor distinto a otro que ya exista.

Las características principales de una lista son:

- Cada nodo tiene que contener los campos que almacenan el valor que se quiere guardar.
- Cada nodo tiene un pointer que apunta a otro nodo, lo que quiere decir que se puede tener un par de punteros que permiten uno, apuntar al nodo anterior y otro, al nodo siguiente, formando una *double linkedlist*, es decir, una lista doblemente enlazada.
- Se necesita que un *pointer* apunte a la cabeza de la estructura para conocer el lugar donde se tiene que comenzar.

A continuación, se muestra la representación gráfica de cómo funciona una lista con sus respectivos punteros:



Las operaciones básicas que soporta una linkedlist son:

- Búsqueda de nodos
- Inserción de nodos
- Eliminación de nodos

Diferencias entra array y linkedlist

Ninguna de las opciones tiene que ser mejor que otra, todo siempre depende del uso que se le quiera dar a cada una y del problema que se pretenda resolver.

A continuación, se enumeran las diferencias más apreciables que se pueden encontrar entre ambas estructuras de datos:

- La eliminación o inserción de datos son bastante más sencillas en una *linkedlist*, ya que no tiene tamaño fijo, por lo que para insertar el dato lo único necesario es apuntar al nodo creado; en un array no se tiene esa flexibilidad.

- Cuando se tiene una gran cantidad de datos, siempre resulta más sencillo cambiar los *pointers* de un valor a otro que mover los valores entre ellos.
- En un array es posible provocar desbordamiento de memoria cuando se necesitar añadir un nuevo valor extra y se excede el tamaño máximo de *array*; este tipo de casos no sucede en una *linkedlist*.
- Se necesita mucho más espacio en una *linkedlist* para poder almacenar todos los *pointers* necesarios.
- Los *arrays* ofrecen una mejor forma de acceder a los valores cuando se pretende acceder de forma aleatoria, ya que aprovechan de mejor forma la caché y determinar de mejor forma la ubicación de los datos.

Contenedores

Este tipo de estructuras tienen como principal característica la forma particular de recuperación ordenada de los datos que soportan. Hay dos tipos principales:

- ***Stack* (pila)**: permite la recuperación ordenada de datos de manera que el ~~primer~~ dato que entra es el primer dato que tiene que salir.

  Las pilas se usan cuando el orden de recuperación de los datos no es prioritario, cuando se pretende solamente apilarlos y luego desapilarlos, por que las operaciones principales que se utilizan en este tipo de estructura son *push* y *pop*, que permiten poner y obtener los datos de la pila.

- ***Queue* (cola)**: este tipo de cola permite la recuperación ordenada de los datos, de modo que el primero dato en entrar es el primer dato en salir.

  En este tipo de estructuras, el orden si tiene importancia, ya que siempre el primero de la cola debe de ser el primero en ser atendido y el resto debe de esperar su turno correspondiente.

  Las operaciones de la estructura son *enqueue*() para encolar y para desencolar *dequeue*().

3.1. Datos estructurados en Python: tuplas, listas y diccionarios

Las listas y tuplas en otros lenguajes se conocen como vectores o *arrays*, aunque tienen ciertas diferencias.

Una lista no es lo mismo que una tupla, aunque ambas son un conjunto ordenado de valores. No obstante, en la tupla puede ser de cualquier tipo de objeto, mientras que las listas tienen una serie de funciones adicionales que otorgan un mayor manejo de los valores que contienen. Por otro lado, las listas son dinámicas, en tanto que las tuplas son estáticas.

Para crear una lista hay dos posibles formas. La primera forma es la más recomendable; sin embargo, conviene conocer la segunda forma para reconocerla si aparece en algún lugar del código o se prefiere hace uso de ella.

A continuación, se muestra un ejemplo de definición de lista:

```
>>> a = []     # Sí.
>>> b = list()  # No.
>>> a == b
True
```

Cuando se utilizan los corchetes lo que se hace es crear una lista vacía, sin ningún tipo de valor. Los valores que debe tener se pueden especificar después de la creación, indicándoselo entre los corchetes también:

```
a = [1, 2, 3, 4]
```

A cada uno de los valores se le llama elemento y no será necesario indicar cuántos elementos es necesario guardar. Tanto las listas como las tuplas son capaces de contener elementos de distinto tipo:

```
a = [5, “Hola mundo!”, (1, 2), True, -2.5]
```

Para crear una tupla, al igual que en las tablas, hay dos formas:

```
>>> a = ()
>>> b = tuple()
>>> a == b
True
```

Dado que las tuplas son estáticas, siempre se deben especificar los elementos que van a formar parte de ella durante la creación:

```
a = [5, “Hola mundo!”, (1, 2), True, -2.5]
```

Para poder acceder a los elementos, tanto en una tupla como en una lista, se puede indicar el índice del elemento, teniendo en cuenta que los índices comienzan en 0 y deben estar indicados dentro de corchetes:

```
>>> a = [“Hola”, “mundo”, “!”]
>>> a[0]
‘Hola’
>>> a[1]
‘mundo’
>>> a[2]
‘!’
```

En el caso de que se intente acceder a un índice que se encuentre fuera de rango, se produce una excepción:

```
>>> a[3]
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
IndexError: list index out of range
```

En el caso expuesto anteriormente, el objeto a tiene una longitud máxima de tres elementos, pero como el índice comienza en cero, realmente el índice 3 corresponde al cuarto elemento, que no se encuentra en el objeto.

Para la asignación de valores, una vez se crea la lista, es posible añadir todos los que se necesite. Para ello se utiliza el método *append*(), que permite agregar el elemento que se pase por parámetro al final de la lista.

También es posible modificar los elementos existentes cuando se combinan los elementos de acceso con los de asignación:

```
>>> a[0] = “Esto es una lista.”
```

Para hacer una inserción de un elemento en un lugar especificado, se utiliza el método insert. Se puede hacer de la siguiente forma:

```
>>>a.insert(1, 2)
```

Para eliminar los elementos de una posición concreta es necesario utilizar la palabra reservada del y la posición del índice del elemento a eliminar:

```
>>> del a[2]
```

En cambio, si lo que se requiere es eliminar todos los elementos de la lista, en vez de indicar el índice se utilizan los dos puntos:

```
>>> del a[:]
```

Hay que tener en cuenta que el carácter : lo que hace es borrar todos los elementos de la lista, pero no la referencia a la lista. Para ello, se ha de usar la palabra reservada del, que borra de forma completa la propia referencia de la lista.

Cuando se tiene que determinar la cantidad de elementos que hay en una lista o una tabla, se puede utilizar el método *len*:

```
>>> a = (“Coche1”, “Coche2”, “Coche3”)
>>>len(a)
3
```

Pero hay que tener en cuenta que, en el caso de que se quiera acceder al último elemento, si se toma como base el valor devuelto de *len*, siempre habrá que restarle un valor, ya que el índice siempre comienza en 0.

Además, se pueden realizar conversiones. Por ejemplo, convertir una lista a tupla o una tupla a lista, en el siguiente código se puede ver la forma de hacerlo:

```
>>> a = tuple(a)
>>> a = list(a)
```

También es posible extraer partes de una lista o tupla: es el denominado *slicing*. Se puede indicar el indicar el inicio del índice y el fin del índice del que se tienen que extraer los datos. Es decir, se puede indicar el rango de los índices de los cuales se pretende extraer los datos de la estructura. Si el comienzo no se especifica, por defecto es 0 y si el fin no es especificado, por defecto es la cantidad de objetos que contenga.

Este es el código ejemplo para obtener un trozo de una lista y de una tupla:

```
objeto[comienzo:fin]
>>> a[2:5]
```

También es posible el uso de índices negativos para comenzar a contar desde el final, lo que puede permitir obtener todos los elementos a excepción del último o quitar elementos del final.

Al *slicing* también se le puede señalar el número de elementos que se tiene que saltar dentro del rango de índices para obtener un valor:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> a[5:16:1]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Cuando se quiere realizar la búsqueda de elementos, la forma más fácil es utilizando la palabra reservada *in*, que, además, se puede usar conjuntamente con la palabra not para poder indicar entonces si los valores indicados no se encuentran en la lista:

```
>>> a = (1, 2, 3, 4)
>>> 3 in a
True
=========================================================

>>> 0.5 not in (1, 2, 3, 4)
True
```

En la interacción, existen varios métodos para recorrer los elementos de una tupla o lista. En lenguaje Python se realiza de la siguiente forma, cuando se utiliza la técnica de uso del índice para recorrer todos los valores:

```
a = (1, 2, 3, 4, 5)
for i in range(len(a)):
print(a[i])
```

Aunque también existe otra forma de hacerlo sin tener que utilizar la técnica de uso del índice que es más limpia y agradable:

```
fornum in a:
print(num)
```

Asignar valores de una lista a un conjunto de objetos se llama unpacking o desempaquetamiento. En el siguiente ejemplo, se muestran las dos formas de hacerlo, el método de la parte superior es equivalente al método de la parte inferior:

```
obj1, obj2, obj3 = lista

obj1 = lista[0]
obj2 = lista[1]
obj3 = lista[3]
```

A continuación, se detallan los métodos más comunes y usados en la administración de las listas:

- *extend*(): permite añadir elementos al final de la lista.
- *pop*(): permite eliminar un elemento de la lista indicando la posición y obteniendo el valor eliminado.
- *index*(): permite obtener la posición del valor indicado.
- *count*(): determina cuántas veces aparece un elemento en la lista.
- *sort*(): permite ordenar de menor a mayor los elementos de la lista.
- *reverse*(): permite invertir el orden de todos los elementos de la lista.

Para realizar la ordenación de menos a mayor, como hemos visto, tenemos el método *sort*(), pero en caso de que se precise una ordenado de mayor a menor, es necesario utilizar la combinación del método *sort*() con el método *reverse*().

Otra función importante es la función ¬*zip*() que permite la creación de una lista de tuplas, creando una iteración clara y sencilla.

Cuando utilizamos un set nos referimos a un conjunto de elementos no ordenados y únicos, por lo que un set no tiene dos o más valores iguales y tampoco es posible acceder a los valores a través de un índice, ya que es una colección desordenada de valores.

3.2. Datos estructurados en R: vectores, listas y dataframes

R es un lenguaje de programación con el cual se obtiene una gran variedad de técnicas de estadísticas y generación de gráficos. El término R se refiere a que es un sistema totalmente planificado y coherente, es decir, no es una acumulación de herramientas específicas que no son flexibles, como si ocurre con otros softwares de análisis de datos.

Como se ha apuntado, el entorno R se utiliza normalmente en la computación estadística y gráfica, ya que tiene una gran variedad de técnicas especializadas en estadística, como pueden ser los modelos lineales y no lineales, así como las pruebas de estadística clásica o el análisis de tiempo, clasificación, agrupamiento y otros tipos de datos.

Funciona en todo tipo de plataformas, como UNIX, FreeBSD, Linux, Windows y MacOS.

Las principales características de R son las siguientes:

- Es muy efectivo en el manejo y almacenamiento de los datos.
- Utiliza un conjunto de operadores para la realización de cálculos con matrices.

- Tiene una gran variedad de herramientas para el análisis de los datos.
- Dispone de utilidades gráficas para la visualización sencilla de los datos.
- Es un lenguaje de programación desarrollado de forma precisa e incluye todas las funcionalidades necesarias para la entrada y salida de datos.
- El formato de la documentación está basado en LaTeX, que permite difundir la documentación de forma física como digital.

El lenguaje R se integra perfectamente con otros lenguajes de programación para las tareas de análisis de datos que sean muy intensivas, es decir, que tengan un alto consumo de recursos como CPU y RAM. También se integra con distintas bases de datos y, además, existe multitud de bibliotecas que permiten la utilización de lenguajes de programación interpretados, como son Perl y Python.

R se utiliza en el ámbito del big data para la manipulación, procesamiento y visualización de forma gráfica de los datos, por lo que nos permite:

- Crear visualizaciones de asombrosa calidad.
- Crear panales de control, o *dashboards*, con los que visualizar los datos y analizarlos.
- Generar informes automáticos.
- Usar herramientas de análisis estadístico para poder sumergirse en el conocimiento de los datos.

Por todo ello, se puede decir que R es mucho más que un simple lenguaje de programación: el usuario realmente no programa, es R el encargado de ensayar, equivocarse y volver a probar tantas veces como sea necesario. Cuando tras todos los pasos termina el ciclo y se obtiene un resultado válido, es cuando genera un resultado final que es un informe detallado.

Normalmente, R utiliza todas las fases de análisis de datos:

1) **Adquisición de datos:** se obtienen los datos de todas las fuentes disponibles, incluso de distintas bases de datos.
2) **Preparación de datos:** permite eliminar los datos duplicados, así como los datos incorrectos o valores que se consideren extremos para mejorar el análisis de los datos.
3) **Análisis de los datos:** crea modelos predictivos, también de agrupamiento y de clasificación, entre otros.
4) **Comunicación de los resultados**: permite la realización de informes para poder mostrar los resultados y las conclusiones obtenidas.
5) **Aplicación de los resultados obtenidos:** se pueden utilizar modelos predictivos que hayan sido desarrollados para una serie de datos históricos; esto permite predecir algunas salidas del modelo.

R es una herramienta básica para los analistas de base de datos.

Vectores

Los vectores son una concatenación de datos, pero para una de estas se considere un vector tiene que cumplir los siguientes puntos:

- Todos los datos deben ser del mismo tipo. En el caso de que sean de distinto tipo, R se encarga de transformarlos a un único tipo de forma automática.
- Cada dato tiene que recibir un índice según el orden en que se realizó su concatenación; esto nos permite acceder a cada dato por su índice.

Para la creación de un vector es necesario utilizar la función *c*():

```
datos ← c(2,1,3,-1,10,0,0,1)
```

Si se requiere generar sucesiones de número, R cuenta con diferentes expresiones que nos ayudan a realizar dicha tarea. A continuación, se muestra una serie de ejemplos muy clarificadores de las diferentes posibilidades existentes para poder crear sucesiones:

```
1:10
1 2 3 4 5 6 7 8 9 10
15:11
15 14 13 12 11
1:10-1
0 1 2 3 4 5 6 7 8 9
1:(10-1)
1 2 3 4 5 6 7 8 9
```

También se pueden utilizar para la misma finalidad las funciones *seq*() y *rep*():

```
>seq(10) #tiene el mismo efecto que 1:10
1 2 3 4 5 6 7 8 9 10
>seq(3,10) #tiene el mismo efecto que 3:10
3 4 5 6 7 8 9 10
>seq(1,10, by=3) #realiza los saltos de 3 en 3
═══════════════════════════════════════════════

>rep(1:4,2) #repite 1:4 dos veces
1 2 3 4 1 2 3 4
```

Para poder acceder a los datos del vector, se puede obtener el índice del dato que queremos acceder. Estas son las diferentes formas en las que se puede acceder a los datos de los vectores:

- Vector completo: datos
- Primer dato: datos[1]
- Todos los datos menos el tercer dato: datos[-3]
- Posiciones 1, 5, y 8: datos[c(1,5,8)]
- Posiciones 2 a 5: datos[2:5]
- Vector lógico: v ← datos>1

Para modificar los datos se puede hacer de la misma forma en la que se pueden acceder pero utilizando el operador de asignación:

```
datos ← c(2,1,3,-1,10,0,0,1)
```

Además, se pueden nombrar los elementos de un vector usando la función names():

```
> v = c(1,2,3,4,5)
>names(v) ← c("uno","dos","tres","cuatro","cinco")
> v
uno dos tres cuatro cinco
1 2 3 4 5
>names(v)
[1] "Lun" "Mar" "Mie" "Jue" "Vie"
```

También los vectores permiten las operaciones aritmético-lógicas. La operación devuelve otro vector con los resultados tras aplicar la operación elemento a elemento:

```
a1 ← c(2,1,3)
a2 ← c(2,1,2)
a1 == v2
 TRUE TRUE FALSE
```

Además, los vectores también cuentan con las funciones imprescindibles para el manejo de los datos. A continuación, se detallan las funciones más comunes:

- *Length*(): devuelve el tamaño completo del vector.
- *Min*(): obtiene el elemento mínimo que se encuentra en el vector.
- *Max*(): obtiene el elemento máximo que se encuentra en el vector.
- *Sum*(): realiza la suma de todos los elementos del vector.
- *Mean*(): permite calcular la media de todos los elementos del vector.
- *Median*(): permite calcular la mediana de todos los elementos del vector.
- *Sort*(): realiza la ordenación de todos los elementos.
- *Unique*(): devuelve los elementos del vector eliminando los resultados repetidos.
- *Which*(): devuelve un valor lógico además de los índices que son verdaderos en la condición. Esta función permite conocer de forma muy sencilla los índices de los elementos que cumplen la propiedad indicada:
  - *Whitch.min*(): devuelve los índices de los elementos mínimos.
  - *Whitch.max*(): devuelve los índices de los elementos máximos.

Listas

Una lista es una colección de elementos que pueden ser de distintos tipos y que, además, generalmente se encuentra identificados por un nombre dado.

Para la creación de listas se usa la función *list*():

```
>Lst← list(hombre = “Paco”, mujer = “Carmen”, casados = TRUE, número.hijos = 3, edad.hijos = c(4, 7, 9))
```

Cuando se quiere obtener los elementos que contiene una lista, se realiza mediante la utilización del operador $:

```
>Lst$hombre
[1] Paco
```

Si se quiere acceder a las sublistas que contiene la lista, es necesario utilizar los corchetes []:

```
>Lst[c(“hombre”, “número.hijos”)]
```

Además, si es necesario, se pueden utilizar vectores de valores lógicos:

```
>Lst[c(TRUE, FALSE, FALSE, TRUE, FALSE)]
```

Para poder acceder por índices, se puede indicar el número de índice al que se quiere acceder de la siguiente forma:

```
>Lst[c(1, 4)]
```

También es posible usar índices negativos para obtener los datos correspondientes en cada caso, sabiendo que los datos se pueden acceder de forma circular:

```
>Lst[c(-2, -3, -5)]
```

Dataframes

Estas son estructuras de datos de dos dimensiones, es decir, de formato rectangular, que contienen datos de diferentes tipos, por lo que son heterogéneas. Es la estructura de datos más usada para el análisis de datos.

Los *dataframes* son como una matriz, pero tienen un comportamiento mucho más flexible. En una matriz, todas las celdas contienen datos del mismo tipo, mientras que en los *dataframes* es posible almacenar datos de distintos tipos, aunque cada una de las columnas sigue manteniendo la condición necesaria de que los datos sean del mismo tipo.

De forma general, los renglones de una *dataframe* representan los casos, observaciones o individuos. En cambio, las columnas son las encargadas de representar los atributos o rasgos.

Para la creación de un *dataframe* se utiliza la función *data.frame*(), en la cual hay que indicar el número de vectores, el cual tiene que ser el mismo que el número de columnas que se quieran utilizar. Además, todos los vectores tienen que tener el mismo largo, por lo que un dataframe está compuesto por vectores.

Se le puede asignar también un nombre a cada uno de los vectores, por lo que se puede convertir en el nombre de la columna. El nombre debe de ser claro, descriptivo y que no presente ambigüedad, para evitar confusiones.

A continuación se muestra un ejemplo de creación de un dataframe en el lenguaje R:

```
mi_df← data.frame(
  “entero” = 1:4,
  “factor” = c(“a”, “b”, “c”, “d”),
  “numero” = c(1.2, 3.4, 4.5, 5.6),
  “cadena” = as.character(c(“a”, “b”, “c”, “d”))
)
```

##	Entero	Factor	Numero	cadena
1	1	a	1.2	a
2	2	b	3.4	b
3	3	c	4.5	c
4	4	d	5.6	d

Para obtener el tipo de variable de un *dataframe* se puede utilizar la función *dim*() y si lo que se quiere es obtener el nombre de las columnas, se pude utilizar la función *name*().

En el caso que intentemos crear un *dataframe* con vectores que no son del mismo tamaño, los datos no se añadirán y el código nos devuelve una excepción.

Cuando es necesario “coercionar” una matriz a un *dataframe* se puede realizar mediante el siguiente código:

```
## Se crea la matriz
matriz ← matrix(1:12, ncol = 4)
##Usamos la función as.data.frame para coercionar una matriz al dataframe
df← as.data.frame(matriz)
```

Como ocurre también en una matriz, cuando se aplica una operación aritmética a un *dataframe* procederá a la vectorización del contenido.

Los resultados que se obtienen vienen determinados por el tipo de cada una de las columnas, por lo que R nos devuelve las advertencias que ocurren como consecuencia del resultado de todas las operaciones realizadas.

3.3. Datos estructurados para el cálculo científico en Python: array de NumPy y dataframe de Panda

Un *array* de NumPy tiene todo lo necesario para ofrecer un buen rendimiento cuando se quieren realizar cálculos con los *arrays*. Esta biblioteca está concebida de forma que su principal virtud es el rendimiento frente a otro tipo de estructuras de datos, como pueden ser listas, tuplas o diccionarios.

Permite realizar la misma operación aritmética con cada uno de los elementos de un *array* por separado y realizar las operaciones con arrays de múltiples dimensiones. Cada uno de los *arrays* puede ser de una dimensión o de múltiples dimensiones, por lo que no es necesario que todos los *arrays* sean de la misma dimensión.

NumPy ayuda a operar con un mismo valor con todos los elementos que forma el *array*, elemento por elemento: se pueden utilizar símbolos matemáticos para realizar las diferentes operaciones.

Cuando se quiere inicializar un *array* NumPy se puede utilizar la función *np.zeros*(). Esta función permite crear una *array* de un determinado tamaño e inicializarlo con valores cercanos. La sintaxis es la siguiente:

```
np.zeros(shape, dtype=float, order='C')
```

- *Shape*: permite indicar las dimensiones que tiene el *array*; si se indica una tupla, se creará una matriz, pero si es un escalar se creará un vector.
- *Dtype*: este parámetro es opcional y ayuda a indicar el tipo de dato; en caso de no indicarlo, el tipo de datos será *float*.
- *Order*: este parámetro es opcional y permite indicar el orden de llenado de las filas:
  - C: indica que primero se llenarán las columnas.
  - F: indica que primero se llenarán las filas.

  El valor por defecto de este campo es C

Otra forma de inicializar un *array* NumPy es con la función *np.ones*(). Esta función permite crear un *array* con matrices con ceros. La forma de uso es similar a la función *np.zeros*():

```
np.ones(shape, dtype=float, order='C')
```

Si se requiere inicializar un *array* con otro tipo de valor por defecto, que no sean ni unos ni ceros, se utiliza el método *np.ones*(), que permite crear un *array* de unos, y luego se multiplican estos unos por el valor deseado:

```
np.ones(3) * 3
```

Las distintas funciones matemáticas que se pueden usar con los arrays NumPy para realizar distintas operaciones son las siguientes:

- *Add*(): permite la suma de dos arrays, elemento por elemento.

- *Subtract*(): permite restar dos *arrays*, elemento por elemento.
- *Multiply*(): permite multiplicar dos *arrays*, elemento por elemento.
- *Dot*(): permite multiplicar un *array* por un valor determinado, elemento por elemento.
- *Np.dot*(): permite la multiplicación de las matrices de dos *arrays*, siempre que el número de columnas de la primera coincida con el número de filas de la segunda.
- *Divide*(): permite la división de dos arrays, elemento por elemento.
- *Mod*(): permite dividir dos matrices y obtener como resultado el resto de la división, elemento por elemento.
- *Divmod*(): permite dividir dos *arrays* y obtener el cociente, además del resto de la división, elemento a elemento.
- *Negative*(): permite cambiar el símbolo de los valores de un *array*.
- *Rint*(): permite redondear todos los valores que se encuentran dentro del *array* al entero más próximo.
- *Sqrt*(): permite calcular la raíz cuadrada, elemento a elemento.
- *Exp*(): permite calcular la exponencial, elemento a elemento.
- *Power*(): permite elevar los elementos a la potencia del número indicado.
- *Sin*(): permite aplicar el seno a un *array*, elemento a elemento.
- *Cos*(): permite aplicar el coste a un *array*, elemento a elemento.
- *Tan*(): permite aplicar la tangente a un *array*, elemento a elemento.
- *Log*(): permite aplicar el logaritmo de todos los elementos del *array*, elemento a elemento.

En el caso de que se quieran realizar operaciones que afecten a los ejes, se pueden utilizar las siguientes funciones:

- *Add.reduce*(): se obtiene un *array* con la suma de los elementos por cada uno de los ejes.
- *Multiply.reduce*(): permite obtener un *array* con la multiplicación de los elementos por cada uno de los ejes.
- *Add.accumulate*(): proporciona un *array* con los acumulados de las sumas de todas las filas.
- *Multiply.accumulate*(): permite obtener un *array* con los acumulados de las multiplicaciones de todas las columnas.
- *Add.reduceat*(): obtiene un *array* con los acumulados de las sumas parciales del *array*.
- *Multiply.reduceat*(): permite obtener un *array* con todos los acumulados de las multiplicaciones parciales del primer eje
- *Add.outer*(): facilita un *array* de dos dimensiones a partir de la suma de dos *arrays* de dos dimensiones, con los datos repetidos de dos vectores. En el primer *array* se obtienen los datos repetidos de a por columnas y en el segundo *array* se obtienen los datos repetidos de b por filas.

- *Multiply.outer*(): permite obtener un *array* de dos dimensiones a partir de la multiplicación de dos arrays de dos dimensiones, con los datos repetidos de dos vectores. En el primer *array* se obtiene los datos repetidos de a por columnas y en el segundo *array* se obtienen los datos repetidos de b por filas.
- *Add.at*(): permite modificar el *array* sumando el valor indicado a los elementos que también se indiquen.
- *Multiply.at*(): modifica el *array* multiplicando el valor indicado por los elementos que también se indiquen.
- *Power.at*(): permite modificar el *array* elevando a la potencia el valor indicado a los elementos que también se indiquen.

En el caso de que se quieran hacer funciones binarias con los arrays, están disponibles las siguientes funciones:

- *Bitwise_and*(): permite calcular el producto lógico a nivel binario.
- *Bitwise_or*(): para calcular la suma lógica a nivel binario.
- *Bitwise_xor*(): permite calcular el OR, pero esta vez en caso excluyente.
- *Invert*(): permite calcular el inverso, es decir, el NOT.
- *Binary_repr*(): permite obtener el resultado de una operación binaria y expresarla en binario.

Cuando se requiere hacer cálculos estadísticos también se dispone de una serie de funciones muy útiles dentro de los arrays. A continuación, se detallan cada una de estas funciones:

- *Max*(): permite obtener el valor máximo dentro de los elementos del *array*.
- *Argmax*(): obtiene los índices que tiene los valores máximos dentro de los elementos del *array*.
- *Min*(): facilita el valor mínimo dentro de los elementos del *array*.
- *Ptb*(): proporciona la diferencia que existe entre el valor máximo y mínimo entre todos los elementos que forman el *array*.
- *Clip*(): permite limitar los valores mínimos y máximos que se pueden guardar en un *array*.
- *Round*(): proporciona el redondeo de todos los valores de los elementos que forman el *array*; se obtiene un número en formato decimal.
- *Trace*(): obtiene la suma de todos los valores de los elementos que forman la diagonal de un *array*.
- *Sum*(): permite obtener la suma de todos los valores que forman un *array*.
- *Cumsum*(): suministra un *array* con todas las sumas acumulas de todos los elementos que forman un *array*.
- *Mean*(): facilita la media aritmética de todos los elementos que contiene un *array*.
- *Var*(): permite obtener la varianza de todos los elementos que contiene un *array*.
- *Std*(): calcula la desviación típica de todos los elementos que contiene un *array*.
- *Prod*(): permite obtener el producto de todos los elementos que contiene un *array*.

- *Cumprod*(): proporciona un *array* con todas las multiplicaciones acumuladas de todos los elementos que contiene un *array*.
- *Median*(): calcula la mediana de todos los elementos que contiene un *array*.

Si se quiere hacer operaciones en los arrays con números flotantes, se podrán utilizar las siguientes funciones:

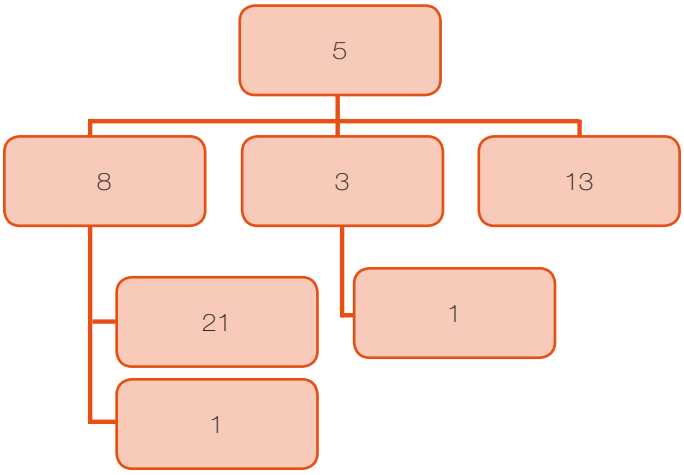
- *Floor*(): se obtiene un *array* con los elementos que sean anteriores.
- *Ceil*(): permite obtener un *array* con los elementos que sean posteriores.
- *Trunc*(): suministra un *array* con los valores de los elementos que forman un *array* truncados.

3.4. Introducción a árboles (trees), colas (queues) y pilas (stacks). Enfoque en Python

Árboles

Los árboles están compuestos de nodos, al igual que las listas encadenadas, pero, a diferencia de estas, el árbol tiene referencia a más nodos hijos o a ceros, y estos nodos, a su vez, tienen referencias a otros nodos hijos y así, sucesivamente.

A continuación, se muestra el esquema que sigue un árbol de nodos:



- **Nodos hoja**: son los nodos que no tienen hijos.
- **Árbol binario**: cada nodo como máximo puede tener dos hijos.
- Un árbol permite la ordenación y la desordenación.

Los árboles se suelen utilizar cuando se pretenden representar jerarquías y optimizar las diferentes búsquedas.

La forma más sencilla de crear un árbol es mediante el uso de nodos interconectados, de forma similar a como se deben implementar las listas encadenadas. La diferencia radica en que las listas encadenadas solo guardan una referencia a otro nodo, mientras que los arboles puede tener varias referencias entre los nodos.

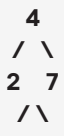
Las operaciones más habituales que se pueden hacer en los árboles son las siguientes:

- Inserción de un nodo
- Eliminación de un nodo
- Recorrer el árbol

Las formas disponibles para recorrer un árbol son las siguientes:

- **Búsqueda por profundidad:** se empieza por la raíz y se recorre la rama izquierda. Una vez la rama está totalmente recorrida, se realiza el mismo proceso con las sucesivas ramas que se desprenden de la misma rama raíz.
- **Búsqueda por anchura**: se empieza por la raíz y se imprimen todos los hijos de la raíz, luego los nietos y así, sucesivamente. Es decir, se va imprimiendo por niveles.

Ejemplo de implementación de un árbol:

consttree = new BinaryTree();
tree.add(4);
tree.add(2);
tree.add(7);
tree.add(1);
tree.add(3);
	

Pilas

Las pilas son tipos abstractos de datos y su estructura está formada por un tipo de lista en la que a los elementos se puede acceder de una forma determinada. Es decir, las pilas determinan la forma en la que se almacenan los datos, que es mediante su apilamiento, por lo que, en consecuencia, se determina la forma en la que se puede acceder a los datos y la forma de poder recuperarlos.

Las pilas se suelen utilizar en las siguientes circunstancias:

- Cuando se quiere evaluar algoritmos en la notación de prefijo o en la notación polaca inversa, en donde los operadores están escritos a continuación de los operandos; este suele ser el uso más particular.
- En el reconocimiento de la sintaxis de lenguajes al margen del contexto.
- Cuando se implementan funciones recursivas.
- En las funciones *Main*().

Como se ha mencionado, las pilas funcionan mediante el apilamiento de los datos, que es cuando se añaden nuevos elementos a la pila, y el desapilamiento de los datos, que es la operación contraria al apilamiento y permite sacar los datos, es decir, el último elemento añadido a la pila.

Para crear una pila en Python es necesario, primero, conocer los diferentes métodos que implementan una pila. A continuación, se detallan cada una de las funciones necesarias para trabajar con pilas:

- *Is_empty*(self): permite verificar si la pila está vacía.

- *Push*(): inserta datos a la pila.
- *Pop*(): permite eliminar el último dato añadido.
- *Print_stack*: permite mostrar los diferentes elementos almacenados en la pila por pantalla.

Un ejemplo básico de una implementación de una pila en Python puede ser el siguiente:

```
pila = Stack() # Creamos una instancia de la pila
# ingresamos algunos elementos a la pila
pila.push('a')
pila.push('b')
pila.push('c')
pila.print_stack() # Mostramos los elementos de la pila
pila.pop() # Utilizamos el metodo pop
pila.print_stack() # Mostramos nuevamente los elementos de la pila
['c', 'b', 'a']
['b', 'a']
```

Colas

Las colas son un tipo de estructuras en las que los datos se van añadiendo por un extremo de la propia cola y se devuelven por el extremo contrario de la misma. Esto permite hacer avanzar los datos desde el inicio hacia el fin. Por tanto, en este tipo de estructura solo se puede acceder al primer y último elemento de la cola: los datos y procesos se van almacenando para su posterior obtención.

Para implantar una estructura de colas en Python hay que tener en cuenta los siguientes conceptos:

- **Encolar**: se refiere a la adición de nuevos elementos al final de la cola.
- **Desencolar**: se refiere a la extracción de los elementos ubicados en la cabecera de la cola.

Uno de los primeros pasos que se tiene que llevar a cabo es la implementación de las listas como colas. Para ello se tienen que usar los métodos *insert*(), para encolar los datos, y *pop*(), para desencolar los datos.

De esta forma se obtiene una lista en la que se podrán ir añadiendo los datos de los diferentes elementos.

A continuación se muestra la forma en la que se crea una cola en Python:

```
>>>from claseCola import Cola
>>> q = Cola()
>>>q.es_vacia()
True
>>>q.encolar(1)
>>>q.encolar(2)
>>>q.encolar(5)
>>>q.es_vacia()
False
```

```
>>>q.desencolar()
1
>>>q.desencolar()
2
>>>q.encolar(8)
>>>q.desencolar()
5
>>>q.desencolar()
8
>>>q.es_vacia()
True
```

En el momento en que se quiere encolar, hay que tener en cuenta diferentes situaciones que se pueden dar, para evitar excepciones o errores:

- **Comprobar si la cola está vacía:** se tiene que comprobar que el último elemento de la cola se encuentre libre. En ese caso, tanto el primer elemento como el último de la cola ahora deberán referenciar al nuevo nodo, ya que este nodo será el primero y el último.
- **Si ya hay nodos en la cola:** se procede a agregar el nuevo dato a continuación del último, además de realizar la actualización de la referencia de último valor.

Cuando se quiere desencolar, es necesario verificar que la cola no se encuentre vacía, ya que, en caso de que no se hallasen más elementos para desencolar, se produciría una excepción que parará la ejecución del programa si no es controlada.

Si la cola no está vacía, se almacena el valor en el primer nodo de la cola y, luego, se mueve la referencia del primer elemento al siguiente.

En el caso de que, tras eliminar el primer elemento del nodo de la cola, la cola quede vacía, hay que actualizar las referencias del primer valor y del último valor, ya que no existe ningún elemento en cola almacenado.

Para comprobar que la cola está vacía hay que verificar que el primer elemento sea *none* y el último elemento también sea *none*.

Resumen

Las estructuras de datos son una forma de organizar los datos, una manera de almacenar los datos que las hace muy eficientes.

El tipo de estructura a utilizar vendrá determinado por el tipo de algoritmo que se quiera utilizar. Se pueden utilizar las estructuras de colas y pilas cuando se quiere obtener los datos almacenados de una forma determinada.

Capítulo 4

Uso de diccionarios y dataframes para el tratamiento de datos

4.1. Aplicaciones de diccionarios de Python

Un diccionario es una colección de datos realizada de forma no ordenada, por lo que para identificar un valor dentro de un diccionario es necesario especificar la clave.

Las claves suelen ser, por norma general, números enteros o cadenas, aunque se puede utilizar cualquier otro tipo de objeto que sea inmutable para actuar como clave. Los valores, en cambio, pueden ser de cualquier tipo; incluso pueden ser otros diccionarios. Un diccionario tiene una relación uno a uno entre las diferentes claves y valores que almacena.

El objeto mapping es el que se encarga de mapear los diferentes valores hashable a objetos de forma arbitraria. Todos los objetos mapeados son objetos mutables.

Los diccionarios se crean poniendo una lista separada por comas de pares, «key:value». Se tiene que poner entre llaves {} para su declaración.

```
p>>> d = {“Python”: 2000, “C”: 2001, “Java”: 2002}
```

Las clases se forman como instancia de la clase primitiva dict:

```
>>>type(d)
<class 'dict'>
>>>isinstance(d, dict)
True
```

Operadores principales

Para poder acceder a los valores de un diccionario es necesario indicar la clave correspondiente entre corchetes:

```
>>> d["Python"]
2000
```

Cuando se quiere asignar un valor al diccionario, es necesario acceder al valor indicando la clave correspondiente entre corchetes y, mediante un operador de asignación, indicar el valor correspondiente:

```
>>> d["Python"] = 2003
>>> d["Python"]
2003
```

~~Si se quiere añadir varios elementos a la vez, se puede hacer mediante la siguiente sintaxis:~~

```
~~>>> d["C++"] = 1985~~
~~>>> d~~
~~{'C++': 1985, 'Java': 2021, 'Python': 2012, 'C': 1960}~~
```

Hay que tener en cuenta que cuando las claves actúan como identificadores, se tiene que controlar que no haya dos iguales, ya que Python no será capaz de saber cuál de los valores asociados es el que tiene que devolver cuando se llame a la clave. No obstante, es posible añadir varias claves con el mismo valor: en este caso, como la clave identifica el valor, no se produce ningún comportamiento anómalo.

Si intentamos acceder a una clave que no existe en el diccionario, se produce una excepción del tipo *KeyError*.

Para poder acceder de forma más segura a los valores del diccionario, se puede usar la función *get*(): en el caso de que la clave no exista, de esta forma no se produce ninguna excepción, sino que se obtiene el valor *none* de retorno:

```
>>>d.get("noExiste") is None
True
```

Cuando se necesita borrar un valor del diccionario, se utiliza la palabra reservada del:

```
>>> del d["Python"]
```

Si se quieren obtener todos los datos que almacena un diccionario, se puede hacer mediante la interacción de todas sus claves, gracias a un bucle:

```
>>>forkey in d:
...    print(key)
```

En el caso de que, en vez de obtener los valores de las claves, se quiera obtener las claves que almacena el diccionario, también se pueden obtener mediante el uso de un bucle:

```
>>>forvalue in d.values():
...    print(value)
```

Si lo que se requiere es obtener tanto las claves como sus correspondientes valores almacenados en el diccionario, se utiliza un bucle que itere todos los elementos que se encuentran guardados el diccionario:

```
>>>forkey, value in d.items():
...    print(key, value)
```

Para saber el número de pares de claves-valor que se guardan en el diccionario, se puede usar la función *len*():

```
>>>len(d)
8
```

Cuando se quiere asegurar que una clave se encuentra en un diccionario, se emplea la palabra reservada in de la siguiente forma:

```
>>> “Java” in d
True
```

Para eliminar todos los pares de clave y valor, se utiliza el método *clear*():

```
>>>d.clear()
>>> d
{}
```

Mediante el método *update*() se puede actualizar un diccionario o unir dos diccionarios; es decir, permite la concatenación entre dos diccionarios:

```
>>> d = {“Python”: 2002, “C”: 2003, “Java”: 2004}
>>> d2 = {“Elixir”: 2005, “Ruby”: 2006}
>>>d.update(d2)
>>> d
{‘Python’: 2002, ‘Ruby’: 2006, ‘C’: 2003, ‘Java’: 2004, ‘Elixir’: 2005}
```

La función *pop*() se usa en diccionarios para eliminar los elementos sin retornar ningún valor. Su funcionamiento es parecido al método *get*():

```
>>> d = {“Python”: 2001, “C”: 2002, “Java”: 2006}
>>>d.pop(“Python”)
2001
>>> d
{‘C’: 1972, ‘Java’: 2006}
```

En el caso de que la clave no exista, el método *pop*() devuelve la clave por defecto que se haya indicado:

```
>>>d.pop(“NoExiste”, “No existe”)
‘No existe’
```

Además existe un método llamado *popitem*() que permite devolver un par de clave-valor de una forma aleatoria, ya que es una colección no ordenada y, luego, lo que hace es modificar su posición de forma aleatoria. En el caso de que esté vacío el diccionario, devuelve el error *KeyError*.

```
>>> d = {“Python”: 2001, “C”: 2002, “Java”: 2006}
>>>d.popitem()
(‘Python’, 2001)
>>>d.popitem()
(‘C’, 2002)
>>>d.popitem()
(‘Java’, 2006)
>>>d.popitem()
Traceback (mostrecentcalllast):
  ...
KeyError: ‘popitem(): dictionary is empty’
```

Diferentes formas de crear diccionarios en Python

Si se hace uso de la clase *dict*, Python permite crear diccionarios de formas alternativas, aunque suelen ser formas poco habituales. No obstante, está bien conocerlas, ya que, si se visualizan en el código, es conveniente saber el comportamiento que tienen y de qué se trata.

En el caso de que a la función *dict*() se le pasen argumentos por nombre, devolverá las claves que debe tener el nuevo diccionario como una cadena de texto:

```
>>> d = dict(Python=2001, C=2002, Java=2006)
>>> d
{‘Python’: 2001, ‘C’: 1972, ‘Java’: 2006}
```

Cuando se utiliza el método *dict*() en vez de la forma tradicional, se suele hacer normalmente por cuestiones estéticas del código.

También es posible mostrar de forma alternativa la representación de pares de clave-valor si se cargan las tuplas de dos ítems dentro de una lista:

```
>>>keys_and_values = [
...    (“Python”, 2001),
...    (“C”, 2002),
...    (“Java”, 2006)
... ]
```

Cualquier objeto iterable, como puede ser una lista, si se estructura de la manera vista en el ejemplo anterior y le introducimos el método *dict*(), devolverá el diccionario correspondiente a los parámetros introducidos.

```
>>> d = dict(keys_and_values)
>>> d
{‘Python’: 2001, ‘C’: 2002, ‘Java’: 2006}
```

En el caso de que se quiera generar un diccionario a partir de un conjunto de claves y se le asigne a todas ellas el mismo valor (que si no se indica ninguno, será el valor *none*), el método encargado es *fromkeys*():

```
>>>dict.fromkeys([“Python”, “C”, “Java”], 0)
{‘Python’: 0, ‘C’: 0, ‘Java’: 0}
```

Implementación de los diccionarios

El código que permite regular la actividad de los diccionarios está escrito en C, como cualquier otro tipo nativo de datos de Python.

Internamente funciona como una tabla de hash. Esto quiere decir que son muy eficientes cuando se requiere almacenar y obtener grandes cantidades de pares clave-valor, por lo que el tiempo para acceder a una clave siempre es el mismo, aunque el número de claves-valor aumente en el diccionario.

En las últimas versiones de Python, los diccionarios funcionan internamente como una colección ordenada de datos. Esta implementación novedosa permite optimizar el rendimiento, por lo que los diccionarios son hasta dos veces más rápidos que los de las primeras versiones de Python.

4.2. Aplicaciones de dataframe del módulo Panda en Python

Panda es un paquete de Python que ofrece estructuras rápidas de datos, flexibles y muy expresivas, que consiguen que el trato con los datos relaciones o etiquetados se haga de una forma fácil y muy intuitiva. Es el elemento por excelencia que pretende ser fundamental para el análisis de datos prácticos en el alto nivel.

Para poder hacer uso de las estructuras que tiene Panda o de cualquier da las otras funciones o métodos que incluye, es necesario importar la librería Panda con el siguiente comando:

```
import pandas as pd
```

Por otro lado, Panda se basa en la funcionalidad de NumPy. Esto quiere decir que numerosas funciones de esta librería son aplicables a las series y a los *dataframes*. Para poder importarla y hacer uso de ella se puede usar la siguiente instrucción:

```
importnumpy as np
```

Las series son las estructuras unidimensionales que contienen un *array* de datos y un *array* de etiquetas, las cuales tienen asociadas los datos; estos son los llamados índices. Los diferentes elementos de la serie se extraen con el nombre de la serie y entre corchetes, el índice de elemento al que tienen que hacer referencia.

Las etiquetas del índice pueden ser diferentes, aunque no necesitan serlo. La única regla que se tiene que cumplir es que sea posible aplicar la función hash sobre cada una de ellas para hacer referencia luego a su valor. La inmutabilidad del índice de etiquetas provoca que si se intenta modificar un único valor del index, devuelve un error.

Los pandas solo pueden contener datos de un mismo tipo. Se puede acceder a los objetos que contienen el índice y los valores mediante los atributos *index* y *values* de la serie. La serie además contiene el atributo *name*, que permite obtener el nombre de la serie y el atributo *axes* que permite el acceso a la lista con los diferentes ejes de la serie.

Si se quiere obtener el tamaño de la serie, es necesario usar el atributo *shape*, que nos devuelve el tamaño completo de la serie.

Los *dataframes* son estructuras tabulares de datos. Están orientados a columnas y tienen etiquetadas las filas y las columnas que permiten su acceso. La columna de un *dataframe* solo puede contener un tipo de datos, aunque cada columna es posible que tenga un tipo de dato diferente. Para acceder a los tipos de las columnas es necesario utilizar el atributo dtypes.

Para poder acceder a las filas y columnas de un *dataframe*, es necesario utilizar los atributos *index* y columna.

El eje 0 corresponde al índice de las filas, que es el eje vertical, y el eje 1 corresponde al índice de las columnas, que son los índices, que corresponden al eje horizontal. Para obtener la lista de los ejes de la estructura se puede utilizar el método *axes*: devolverá dos listas, ya que el *dataframe* tiene una estructura bidimensional.

Los índices y las filas son inmutables, por lo que se pueden asignar nuevos valores, pero si se intenta modificar un único valor, se producirá una excepción.

Tanto el índice de las filas como el índice de las columnas cuenta con el atributo name, que permite imprimir la estructura, y el atributo *values*, que da acceso a cada uno de los valores del *dataframe*. Por su parte, el atributo shape del *dataframe* informa de su dimensionalidad y del número de elementos en cada una de las dimensiones.

Para crear una *dataframe* se tiene que usar su constructor que es *pandas.DataFrame*. El constructor cuenta con los siguientes parámetros de entrada:

- ***Data***: es la estructura de datos; puede ser un array NumPy, otro dataframe e incluso un diccionario.
- ***Index***: es el índice que se aplica a las filas; en el caso de que no se especifique ningún índice, se le asigna uno por defecto, que es un número entero.

- ***Columns***: son las etiquetas a aplicar en las columnas; si no se indica ningún valor, se le aplicará un valor automático formado por número enteros.
- ***Dtype***: permite definir la forma en la que se aplican los datos; solo se permite un tipo, en el caso de que no se especifique, se entiende que el tipo de cada columna proviene de los tipos de datos que contenga.

Los *dataframes* tienen los siguientes atributos disponibles:

- *T*: para transponer el índice y las columnas.
- *At*: para acceder a un único valor para un par de etiquetas de fila/columna.
- *Axes*: devuelve una lista que represente los ejes de *dataframe*.
- *Blocks*: es una propiedad interna, sinónimo de propiedad para *as_blocks*().
- *Columns*: son las etiquetas de la columna del *dataframe*.
- *Dtypes*: obtiene los tipos de las columnas del *dataframe*.
- *Empty*: indica si el *dataframe* está vacío.
- *Ftypes*: permite devolver los tipos de una forma más compleja que *Dtypes*.
- *Iat*: accede a un único valor para un par de fila/columna dada una posición entera.
- *Iloc*: indexación basada completamente en la ubicación.
- *Index*: el índice de la fila.
- *Ix*: devuelve un índice basado en la ubicación de la etiqueta.
- *Lox*: permite acceder a un grupo de filas y columnas mediante el uso de etiquetas o una matriz.
- *Ndim*: devuelve un entero que representa el número de ejes y dimensiones de la matriz.
- *Shape*: obtiene la tupla que representa la dimensionalidad del *dataframe*.
- *Size*: devuelve mediante un número entero el tamaño del *dataframe*.
- *Style*: devuelve todas las propiedades disponibles para representar el *dataframe* en tablas HTML.
- *Values*: obtiene una representación NumPy del *dataframe*.

Existen varios métodos que nos ayudan al manejo de los dataframe. A continuación, se detalla cada uno de ellos.

- ***Función drop_duplicates()***: es la encargada de eliminar todas las filas duplicadas en un *dataframe*.
  - *Subset*: se le indica la etiqueta de la columna o la secuencia de etiquetas que se deben tener en cuenta para detectar los duplicados.
  - *Keep*: permite ignorar los duplicados:
    - *First*: ignora el primer duplicado

    - *Last*: ignora el último resultado
    - *False*: ignora todos los duplicados.
  - *Inplace*: permite modificar el llamador del *dataframe*.
  - *Ignore_index*: permite ignorar los índices originales.

  Valor de retorno:

  - *True*: se han eliminado todas las filas duplicadas.
  - *False*: no se han eliminado las filas duplicadas.

- ***Función replace()***: permite remplazar todos los valores del dataframe con otros valores.
  - *To_replace*: permite indicar los datos que deben de ser remplazados.
  - *Value*: permite indicar los valores por lo que se tiene que remplazar cualquier valor que coincida en el *To_replace*.
  - *Inplace*: modifica el llamador del *dataframe*.
  - *Limit*: permite indicar mediante un número entero el tamaño máximo del hueco a rellenar hacia adelante o hacia atrás.
  - *Regex*: permite activar la función *regex* en el *To_replace*.
  - *Method*: permite indicar el método a utilizar para la sustitución de los valores.

  Valor de retorno: devuelve el *dataframe* con los valores remplazados.

- ***Función shift()***:desplaza el índice por un número determinado.
  - *Periods*: permite indicar mediante un número entero los periodos a desplazar el índice; este puede de ser positivo o negativo.
  - *Freq*: parámetro opcional que permite indicar el desplazamiento de los valores sin tener que realinear los datos.
  - *Axis*: permite el desplazamiento a lo largo de la fila.
  - *Fill_value*: en caso de que haya valores que falten, se escalan con el valor indicado.

  Valor de retorno: devuelve el objeto dataframe con los índices desplazados.

- ***Función melt()***: permite cambiar la ordenación del *dataframe* de un formato ancho a uno largo.
  - *Dataframe*: es necesario indicar el *dataframe* que se quiere cambiar a formato largo.
  - *Id_vars*: se le pasan las columnas para las variables de identificación.
  - *Value_vars*: las columnas que no se especifiquen como variables serán tomadas como variables de valor; estas pueden ser también seleccionadas.
  - *Var_name:* indica el nombre de la columna identificadora.

- *Value_name*: indica el nombre de la columna de los no identificadores.
- *Col_level*: permite transformar el *dataframe* en el caso de que sea de columnas múltiples.

Valor de retorno: devuelve el *dataframe* con una o más columnas identificadoras y solo dos columnas sin identificadores, que son las columnas de variable y valor.

- ***Función transform()***: permite aplicar una función específica a un *dataframe* en concreto. El *dataframe* transformado tiene la misma longitud de eje que el *dataframe* original.
  - *Func*: se le indica la función que se debe aplicar al *dataframe* para convertirlo en otro *dataframe*.
  - *Axis*: permite indicar el eje del objetivo; puede ser las filas o las columnas.
  - *Args*: son argumentos de posición que sirven para pasar a la función.
  - *Kwargs*: son argumento de las palabras claves adicionales que se tiene que pasar a la función.

  Valor de retorno: devuelve el *dataframe* transformado, que debe de tener la misma longitud que el *dataframe* original. En el caso de que el *dataframe* tenga una longitud distinta, el método devuelve un *ValueError*.
- ***Función sum()***: permite calcular la suma de todos los valores del objeto *dataframe*.
  - *Axis*: permite encontrar la suma a lo lago de la fila o de la columna.
  - *Skipna*:
    - Verdadero: permite excluir los valores NaN.
    - Falso: permite incluir los valores NaN.
  - *Level*: en el caso de que el eje sea "multindex", se puede indicar el seleccionado.
  - *Numero_only*: permite incluir solo las columnas que sean de tipo *float, int y boolean*.
  - *Min_count:* el número mínimo de valores que tienen algún valor para devolver la suma; en el caso de que no se cumpla se devuelve NaN.
  - *Kwargs*: son los argumentos de palabras claves adicionales a la función.

Resumen

Un diccionario es un tipo de datos que sirve para asociar pares de objetos. Es una colección de claves y cada una de las claves tiene asociado un valor. No es necesario que las claves estén ordenadas, pero sí es imprescindible que no haya claves repetidas. Para poder acceder a los valores se hace a través de la clave.

El *dataframe* es un tipo de clase especial en el lenguaje de programación R y se suele utilizar cuando se requiere hacer un estudio estadístico de los datos.

Capítulo 5

Librerías para integración, exploración, tratamiento, modelización de datos y accesos a bases de datos

El acceso a las bases de datos en Python se realiza a través del modo estándar, que se encuentra recogido en las especificaciones de DB-API. Esto significa que, independientemente de la base de datos que se tenga que utilizar, los diferentes métodos y procesos que se utilizan para la conexión de lectura y escritura de la base de datos siempre son los mismos, es decir, en Python el conector da igual.

Si queremos trabajar con la base de datos MySQL, se utiliza el módulo MySQLdb. Este tiene que ser instalado de forma manual a través del siguiente comando:

```
sudo apt-get install python-mysqldb
```

Para poder realizar una conexión a la base de datos a través de Python, hay que seguir los siguientes pasos:

- Abrir una conexión y crear un puerto de comunicaciones.
- Ejecutar la consulta correspondiente.
- Obtener los resultados en el caso de que sea una consulta de selección, o realizar una escritura cuando se requiera insertar, modificar o eliminar algunos datos.
- Cerrar el puntero y la conexión a la finalización.

A continuación, se muestra un ejemplo de cómo se debe acceder a una base de datos en Python utilizando la librería MySQLdb:

```
Import MySQLdb
DB_HOST = 'localhost'
DB_USER = 'root'
DB_PASS = 'mysqlroot'
DB_NAME = 'a'
def run_query(query=''):
 datos = [DB_HOST, DB_USER, DB_PASS, DB_NAME]
 conn = MySQLdb.connect(*datos) # Conectar a la base de datos
 cursor = conn.cursor() # Crear un cursor
 cursor.execute(query) # Ejecutar una consulta
 if query.upper().startswith('SELECT'):
 data = cursor.fetchall() # Traer los resultados de un select
 else:
 conn.commit() # Hacer efectiva la escritura de datos
 data = None
 cursor.close() # Cerrar el cursor
 conn.close() # Cerrar la conexión
```

Para la inserción de datos se debe de hacer mediante el método *raw_input()* para indicar los datos a añadir y el método *run_query()* para ejecutar la consulta:

```
dato = raw_input("Dato: ")
query = "INSERT INTO b (b2) VALUES ('%s')" % dato
run_query(query)
```

Para poder seleccionar todos los registros, solo hay que escribir la consulta selecto correspondiente y utilizar el método *run_query()*. El método devolverá el resultado obtenido por la consulta:

```
query = "SELECT b1, b2 FROM b ORDER BY b2 DESC"
result = run_query(query)
print result
```

En el caso de requerir hacer una consulta con solo algunos registros coincidentes —es decir, mediante el uso de condicionales en la consulta—, se hace de forma similar al ejemplo anterior, utilizando el método *run_query()* para la ejecución de la consulta y obtención del resultado que devuelve dicha consulta:

```
criterio = raw_input("Ingrese criterio de búsqueda: ")
query = "SELECT b1, b2 FROM b WHERE b2 = '%s'" % criterio
result = run_query(query)
print result
```

Para la eliminación de registros es necesario preparar la consulta de eliminación indicando un Id o los criterios que se estimen oportunos. Luego, para la ejecución de la consulta, se utiliza el método *run_query()* para realizar las acciones en la base de datos:

```
criterio = raw_input("Ingrese criterio p7 eliminar coincidencias: ")
query = "DELETE FROM b WHERE b2 = '%s'" % criterio
run_query(query)
```

En la actualización de datos será necesario indicar tanto las columnas como los valores nuevos que debe tomar, se tiene que utilizar el método *run_query()* para poder ejecutar la consulta y realizar los cambios en la base de datos:

```
b1 = raw_input("ID: ")
b2 = raw_input("Nuevo valor: ")
query = "UPDATE b SET b2='%s' WHERE b1 = %i" % (b2, int(b1))
run_query(query)
```

5.1. Descripción de herramientas de integración y tratamiento de datos

El volumen de datos que se llega a producir todos los días —además de los tipos de datos, que pueden ser estructurados y no estructurados— es de gran valor y utilidad para las organizaciones y negocios. En caso de tener demasiados datos, provenientes de muchas fuentes, debemos de intentar centralizar los datos para poder gestionarlos de forma eficiente. En el mercado existen numerosas herramientas de *software*.

La **integración de datos** se refiere al proceso de juntar todos los datos provenientes de las diferentes fuentes de información y trasladarlo a un único punto de acceso.

El proceso de integración de los datos se comprende de las siguientes fases:

- Ingesta de los datos
- Limpieza de los datos
- Mapeo de los datos
- Transformación de los datos
- Conversión de los datos

La mejor estrategia de integración de los datos debe permitir ayudar a la empresa u organización en su transformación digital.

Al tener una visión más amplia de los clientes e información acerca de sus gustos o preferencias, se pueden conocer los patrones de comportamiento, además de los movimientos en sus redes sociales. La posibilidad de recoger esa información (hasta los clics que el cliente realiza en cada una de las páginas web que visita, y en qué secciones en cuestión) resulta muy interesante de cara a conocer mejor sus gustos y preferencias.

Para procesar esa información es necesario integrar los datos. Para ello existen diferentes herramientas de *software* que cubren todos los procesos necesarios, como el proceso de **extracción, transformación y carta** de los datos.

En el mercado existen multitud de soluciones, y podemos encontrar diferentes tipos de herramientas:

- Extracción, transformación y carga
- Enterprise Application Integration (EAI)
- Enterprise Information Integration (EII)

Las ventajas que ofrece la integración son infinitas, pero las principales son:

- Mejorar la colaboración entre los diferentes departamentos de una organización o de una empresa.
- Permitir la unificación de los sistemas, ya que se concentran en los datos y que todos puedan ser accedidos desde cualquier lugar.
- Ahorrar tiempo mediante la eliminación de datos duplicados, lo que significa mayor eficiencia en la preparación de los datos y mayor fiabilidad de los datos obtenidos.
- Proporcionar mayor conocimiento de la empresa, ya que permiten tener más detalles sobre los datos que dispone la organización y mejora el valor de los propios datos al ver de forma más clara todo su potencial.

A continuación, se describirán las herramientas más conocidas y utilizadas para la integración de los datos.

Informatica PowerCenter

Este *software* dispone de **conectores para la mayoría de las fuentes de datos** que podamos imaginar. La integración de los datos se realiza **de punto a punto,** utilizando un modo distribuido.

Se encuentra perfectamente integrada con la herramienta PowerCenter y facilita los datos operaciones de forma instantánea y totalmente escalable.

PowerCenter ofrece una base escalable y de un gran rendimiento enfocado en iniciativas de integración de datos locales, además del análisis y almacenamiento de los datos. También permite realizar la migración de datos de forma muy fácil y sencilla. Es capaz de admitir todo el ciclo de vida de la integración de los datos locales, desde el inicio hasta todas las implementaciones necesarias para la empresa.

Sus principales características son:

- **Conectividad universal:** permite integrar todo tipo de datos de fuentes con conectores de forma efectiva y con un importante nivel de rendimiento.
- **Herramientas basadas en roles:** permite habilitar el autoservicio empresarial y entregar los datos que se crean oportunos de la organización.
- **Escalabilidad:** ofrece una gran escalabilidad.
- **Cero tiempo de inactividad:** mediante el pushdown, el procesamiento distribuido entre otras técnicas.
- **Transformación avanzada de los datos**: permite analizar los datos no relacionales de formatos como XML, JSON, PDF y IoT.
- **Reutilización y automatización:** se puede aprovechar todas las transformaciones preconstruidas.
- **Creación rápida de prototipos y perfiles:** los analistas tienen la posibilidad de colaborar con TI para que puedan crear diferentes prototipos y puedan validar los resultados de una forma iterativa y con rapidez.

Microsoft SQL Server Intregation Services (SSIS)

Este *software* proporciona sus propios **servicios de integración de SQL Server** para conectar con SQL Server. Permite conectarse a diferentes bases de datos, además de permitir las migraciones fácilmente.

Las migraciones se realizan a estructura de datos, que permiten realizar las tareas de migración de los datos por lotes y en grandes cantidades de datos. Esto garantiza la consistencia de todos los datos migrados. Además, se eliminan los riesgos que se pueden presentar al realizar las migraciones.

Esta plataforma permite crear soluciones de integración de datos y la transformación de los datos a nivel empresarial.

También ayuda a resolver problemas empresariales complejos copiando y descargando los archivos, almacenando los diferentes datos, realizando las limpiezas necesarias o realizando la minación de datos, a la vez que administra los objetos y los datos de SQL Server.

Este *software* tiene la posibilidad de extraer y transformar los datos de una gran cantidad de orígenes, como los archivos XML, los archivos de planos y orígenes de datos relacionales, además de cargar los datos en uno o varios destinos.

Incluye también un conjunto amplio de tareas y transformaciones totalmente integradas, además de herramientas gráficas para crear nuevos paquetes y la base de datos donde se almacenan, ejecutan y administran los paquetes.

Además, permite programar modelos para crear paquetes mediante programación y codificar las diferentes tereas personalizadas y otro tipo de objetos de paquete.

Alteryx Designer

Permite agilizar el proceso de integración de datos al realizar todo el trabajo a través de los **workflow,** lo que permite a los analistas poder conectarse a casi cualquier fuente de datos y poder combinarlos, además de transformarlos y mapearlos. Además, es capaz de obtener las tareas estadísticas y predictivas de la fuente de datos.

Las principales características de este *software* son:

- **Automatizar cada resultado analítico:** permite automatizar cada paso que produce el análisis, incluida la preparación de los datos, la unión de todos ellos, los informes necesarios, así como el análisis predictivo y la ciencia de datos.

  Permite acceder a cualquier origen de datos de forma simple y sencilla mediante una plataforma de autoservicio con infinidad de bloques de funcionalidades ya prestablecidas, preparadas para la reutilización.

- **Combinación y preparación de datos eficaz:** permite acelerar la preparación de los datos de todas las fuentes y tipos de datos, ya que se pueden obtener resultados en minutos, de forma fiable.

  Es posible combinar datos simplemente arrastrando y soltando las hojas de cálculo, documentos entre otros tipos de datos.

  Permite automatizar los informes y los diferentes análisis predictivos y prescriptivos, y organiza la información de forma visual, ya que se incluyen datos geoespaciales, demográficos y filmográficos.

- **Automatización de las acciones:** permite la automatización de los análisis y los procesos repetitivos. También permite impulsar las acciones más rápidamente, pudiendo publicar los resultados obtenidos en los diferentes paneles interactivos con los que cuenta este software. Se pueden publicar los resultados en diferentes formatos, además de aplicaciones web y Google.

  Permite compartir los resultados entre los diferentes equipos de la organización mediante la utilización de las principales plataformas del mercado.

Pentaho Data Integration

Es el componente de Pentaho precisamente **destinado a las tareas de ETL**. Los principales propósitos de este *software* son:

- La migración de datos entre las diferentes aplicaciones o bases de datos.
- La exportación de datos desde la base de datos.
- La limpieza de datos.

Permite crear e implementar de forma sencilla las canalizaciones de datos a escala, puede integrar los diferentes datos de las diferentes fuentes y organizar flujos de datos en todos los entornos.

Esta solución permite reducir el tiempo y la complejidad para acceder, preparar y combinar las diferentes fuentes de datos para entrar poder entregar datos listos para sus análisis.

Las principales funciones de este *software* son:

- Acceder a la fuente de datos de forma local, en el núcleo o en la nube.
- Permitir operaciones en tiempo real sobre los datos.
- Mayor flexibilidad de clúster a conector.
- Permitir conectarse y mover los datos de cualquier tamaño y en cualquier formato.
- Obtener metadatos, además de entregar los datos para su posterior análisis.
- Realizar un aprendizaje automático.

Denodo Platform

Este *software* permite proporcionar todos los beneficios de la **virtualización e integración** de los datos, y tiene incluida la capacidad de proporcionar acceso a los datos integrados de las diversas fuentes de información en cualquier momento, en tiempo real. Posibilita el acceso de los datos —tanto estructurados como no estructurados— desde sus propias fuentes y de otras empresas.

Las características principales que ofrece Denodo Plataform son las siguientes:

- Una buena **experiencia de usuario,** ya que está dirigida a las necesidades especiales de las diferentes partes, con una interfaz agradable y totalmente accesible directamente desde el navegador.
- Una óptima e inteligente **estrategia de ejecución de consultas,** gracias al optimizador dinámico de consultas, que permite ofrecer un acceso más rápido y eficiente a los datos.

- Capacidad de **aceleración inteligente** para las diferentes consultas y escenarios más complejos (también ofrece diferentes resúmenes).
- Capacidad de **procesamiento en paralelo** directamente en memoria, para acelerar el acceso a los datos.
- **Características automatizadas de administración de ciclo de vida**. Los usuarios tienen la posibilidad de administrar los datos de forma más eficiente con las diferentes características automatizadas que ofrece este software, y aprovechar mejor el tiempo para la toma de decisiones.
- **Acceso directo a los datos:** permite la colaboración y tiene mejorada la recomendación automática basada en MI a través del catálogo dinámico.
- Permite la **gestión automatizada de infraestructuras** para la nube con soporte incluido a PaaS, en este caso para entornos híbridos y en la nube.
- Permite la **autentificación OAuth 2.0** entre otras tecnologías y los estándares en la nube, permite una interoperabilidad con los sistemas en las nubes.
- Tiene la **capacidad para implementar Denobo** a través de los mercados como Amazon Web Services, Microsoft Azure y Google Cloud Plataform, también se incluye Docker.
- Tiene una gran **seguridad y gobernanza,** proporciona acceso seguro y de una forma selectiva a todas las tendencias de datos de una organización a través de un único punto de control.

Attunity Replicate

Attunity Replicate es una herramienta que permite la **replicación como la obtención de datos** y que permite a**utomatizar el proceso de replicación de tipo extremo a extremo.** Está dirigido a las organizaciones que utilizan múltiples plataformas de datos y que trabajan con una gran variedad de formatos.

Esta herramienta permite la arquitectura de datos compleja y en evolución. Asimismo, tiene una herramienta de replicación de datos que mejora de forma notable la eficiencia, además de añadir más velocidad al sistema y reducir los costos asociados.

Las herramientas de replicación que tiene este *software* permiten simplificar de forma notable y acelerar el proceso de migración y consolidación de los datos desde los diferentes orígenes, que pueden ser internos o externos.

Se puede mover de forma rápida los datos de origen al destino sin necesidad de escribir código ETL, además de potenciar el valor de los datos mediante su integración en tiempo real.

Fivetran

Ayuda al negocio posibilitando la **centralización de los datos** de diferentes fuentes de datos, que pueden ser gestionados directamente desde el navegador web. Permite la extracción de los datos de forma automática, que se en *warehouse*.

Esta herramienta tiene conectores precompilados que ofrecen esquemas listos para su uso, lo que posibilita el análisis de datos con una gran capacidad de adaptación a los cambios de los datos de orígenes.

Las características más importantes de este software son las siguientes:

- **Conectores preconstruidos**: permite la centralización de los datos utilizando la gran cantidad de conectores que se encuentran ya preconstruidos en el sistema.
- **Esquemas listos para consultar:** utiliza esquemas y ERD reflexivos, basados en la investigación para todos los orígenes.
- **Permite migraciones automatizadas de esquemas:** permite guardar los recursos con diferentes conectores que se adaptan automáticamente y los cambios de los esquemas y API.
- **Integración de los datos gestionados:** permite reducir la deuda técnica con diferentes conectores escalables, y la administración de origen a destino.
- **Transformaciones basadas en SQL**: permite modelar la lógica de negocio en cualquier destino mediante SQL, que es el estándar actual del sector.
- **Actualizaciones incrementales por lotes:** permite los datos actualizados para todos los orígenes de datos.

InfoSphere DataStage

Es una herramienta desarrollada por IBM, y funciona con ***workflow***. Permite realizar todo el proceso de ETL de forma completa en múltiples sistemas. Asimismo, permite la administración extendida de **metadatos** y la conectividad empresarial con las diferentes herramientas del Big Data y las herramientas de la nube.

Las principales características que ofrece este *software* son las siguientes:

- **Aceleración de la ejecución de la carga de trabajo:** permite realizar cargas hasta un 30 % más rápidamente gracias a que cuenta con equilibrio de carga de trabajo y con un motor en paralelo.
- **Reutilización del diseño:** permite la separación del diseño del trabajo ETL del tiempo de ejecución y se puede implementar en cualquier nube.
- **Modernización de la integración de datos:** permite ampliar las capacidades de la actual base de datos que se esté utilizando.
- **Reducción del coste:** permite reducir los costes al poder utilizar la virtualización y contenedores.
- **Espectro completo de datos y servicios IA:** permite gestionar el ciclo de vida de datos y el análisis en la propia plataforma IBM.
- **Motor en paralelo y equilibrio de carga automatizado:** permite procesar los datos a escala optimizando el rendimiento ETL utilizando el motor paralelo y usando equilibrios de carga.
- **Compatibilidad con metadatos:** permite proteger los datos que son confidenciales con el intercambio de metadatos. Se utiliza el linaje de los datos para saber cómo fluyen los datos a través de toda la información y la integración.
- **Canalizaciones de entrega automatizadas:** permite automatizar las canalizaciones de trabajo de integración continua para el desarrollo de las pruebas y en producción. Asimismo, permite reducir los costes de desarrollo.

- **Amplio conjunto de conectores:** permite utilizar la conectividad precompilada y las etapas para poder mover los datos entre diferentes fuentes de nube y almacenes de datos.
- **Diseño asistido por aprendizaje de automático**: su interfaz es fácil de usar, por lo que ayuda a reducir los costes de desarrollo.
- **Calidad de los datos:** resuelve automáticamente los problemas de calidad cuando los entornos de destino obtienen los datos.
- **Detección automática de fallos:** la propia infraestructura es capaz de detectar los fallos que se producen en el sistema.
- **Plantillas de trabajo reutilizables:** genera trabajos y usa reglas personalizadas para aplicar diferentes patrones, que son reutilizables.

Oracle GoldenGate

Es una herramienta que está destinada a la integración y replicación de datos **en tiempo real en entornos TL.** Oracle tiene soluciones de alta disponibilidad, replicación de datos, integración y proceso ETL, y funciona también con la configuración de extremo a extremo.

Las principales características que ofrece este *software* son:

- **Movimiento de datos de forma eficiente:** mediante la conexión de GoldenGate intenta mejorar el movimiento de los datos.
- **Base de datos optimizada:** permite reducir la latencia del búfer para las diferentes escrituras paralelas en los diferentes destinos.
- **Aceleradores de negocio integrado:** la herramienta Oracle Integration permite limitar el entrenamiento y acelerar la automatización de procesos de extremo a extremo con las integraciones preconstruidas basadas en las diferentes configuraciones, que se incluyen directamente en las aplicaciones determinadas de SaaS de Oracle.
- **Integración de fuentes y objetos:** permite la integración de datos en las diferentes bases de datos y servicios de Oracle, y también para servicios de datos que no son de Oracle.
- **Transferencia de datos fiable:** permite proporcionar la integridad transaccional con detección y resolución de los diferentes conflictos, además se realiza de forma segura con el sistema SSL y el cifrado punto a punto.
- **Sincronización de Oracle y no Oracle:** permite obtener las bases de datos actualizas en cualquier momento.
- **Reduce el tiempo de inactividad:** permite eliminar el tiempo de inactividad que se produce durante el mantenimiento y las actualizaciones rutinarias de las bases de datos, todas las operaciones están protegidas con las capacidades de conmutación por recuperación, lo que elimina de forma clara el riesgo de perder datos.
- **Escalabilidad de la base de datos:** se pueden implementar configuraciones multimaster o activas que permiten lograr mayor escalabilidad en la base de datos, o cuando se realizan las sincronizaciones de datos. Cualquier base de datos puede generar transacciones y luego sincronizarse con las demás bases de datos para mantener la coherencia de los propios datos.

- **Alta disponibilidad:** se asegura de que los datos estén siempre disponibles con la tecnología Oracle Maximum Availability Architecture.
- **Optimización del servidor**: permite la utilización de servidores que se encuentran en espera en el entorno de producción. Estos servidores se mantienen en espera, sobre todo en entornos que necesitan un alto rendimiento. También se realizan almacenamientos de datos y copias de seguridad rutinarias.
- **Soporte para otras bases de datos:** tiene soporte para las distintas bases de datos que existen en el mercado.
- **Integración con el BigData:** se integra perfectamente con las tecnologías Hadoop, Hbase y Katka, entre otros, además de sistemas NoSql.
- ***Streaming* y análisis en tiempo real:** captura los archivos CSV, JSOn y Avro, permite la transmisión de datos transacciones a Kafka y a bases de datos autónomas y a todos los nodos, y analiza de forma gráfica todos los datos que se encuentra en movimiento.
- **Verificación de datos**: identifica los registros que se encuentran desincronizados, y además de permitir la comparación y reparación del conjunto de datos, lo que permite mejorar la calidad de los datos.
- **Experiencia gráfica del usuario:** de forma gráfica, permite a los usuarios y administradores la configuración, supervisión y administración de forma segura de las operaciones desde todos los navegadores.
- **Supervisión de extremo a extremo:** se realiza la verificación de los datos y la visibilidad de las diferentes estadísticas de uso y rendimiento en tiempo real.
- **API operativas:** permite agregar diferentes controles operativos en procesos de replicación, y permite su integración con herramientas de terceros.
- **Escala automática:** se consigue un rendimiento óptimo, independientemente de la carga de trabajo que tenga el sistema, y se utiliza la carga de trabajo dinámica y el procesamiento de transacciones paralelas.
- **Transformación en línea:** se pueden realizar los procesos de replicación de datos con infinidad de funciones que se encuentran ya integradas en el sistema y facilitan la manipulación de los datos para permitir la transformación y la validación a las llamadas a los diferentes procedimientos almacenados.
- **Malla de datos en tiempo real:** permite replicar las bases de datos y los servicios de datos en los sistemas que son Oracle y otros sistemas. Admite las tipologías de movimiento de datos y el patrón de arquitectura moderno.
- **Sincronización de datos después de una carga masiva.**
- **Monitorización automatizada:** ofrece en todo momento el estado de la replicación con las diferentes alertas en tiempo real.
- **Monitorización de replicación:** permite supervisar la finalización y el rendimiento de todos los procesos de replicación mediante diferentes paneles de administración, además de incluir notificaciones en tiempo real.

5.2. Descripción de herramientas de modelización de datos y acceso a base de datos

La importancia del análisis de datos ha aumentado con el paso de los años, lo que ha dado lugar a una importante apertura del mercado mundial. Por ello, las herramientas de análisis de datos han tomado un lugar central, y ahora existe un gran número de ellas.

Para elegir la herramienta adecuada es necesario analizar el rendimiento que ofrece. El análisis de datos no es un proceso único, ya se relaciona con la integración, consolidación y calidad de los datos.

A continuación, se mencionan las herramientas más importantes para la modelación de los datos y su acceso.

Microsoft Power BI

Esta herramienta analítica de Microsoft es una de las más populares, ya que permite obtener visualizaciones interactivas de datos, además de una integración muy sencilla con las demás herramientas que ofrece Microsoft.

Power BI permite la integración con aplicaciones de terceros con diferentes fuentes de datos, entre ellos Hadoop, Spark o SAP, y puede ser utilizada incluso por usuarios que no tengan conocimientos avanzados.

Entre las principales características que ofrece Power BI destacan:

- **Creación de informes personalizados:** permite la conexión a los datos y a los diferentes modelos, además de virtualizarlos con gran facilidad mediante la creación de informe.
- **Conexión a múltiples orígenes de datos**: permite la conexión a todos los orígenes de datos con escala para analizar, además se puede compartir todos los datos con el resto de la organización, a la vez que se encarga de mantener la precisión de los datos, la coherencia y la seguridad.
- **Integridad con Microsoft Office:** es posible trabajar de forma fácil compartiendo la información mediante el uso de las aplicaciones de ofimática de Microsoft Office, lo que permite el acceso a todos los datos a los usuarios que tienen acceso a estas herramientas.
- **Protección de datos de extremo a extremo:** se protegen los datos de extremo a extremo, incluso cuando los datos se comparten fuera de la organización. También en las exportaciones a otros formatos —como puede ser Excel, PDF o PowerPoint— quedan protegidos de extremo a extremo.
- **Conectores de datos extensivos:** cuenta con una gran cantidad de conectores, por lo que se puede conectar a múltiples orígenes de datos, de forma local o en la nube.

Programación en R

Esta herramienta es la que más destaca hoy en día en la industria. Se usa sobre todo para el **modelado de datos y obtener estadísticas.** Permite manipular los datos y presentarlos de una infinidad de maneras distintas.

Se trata de una herramienta muy fácil de usar y destaca, sobre todo, por su rendimiento. También destaca por la capacidad de datos que tiene y por los resultados que puede obtener, además de funciones en múltiples plataformas.

A lo largo de este contenido se han estudiado con detalle sus características y las ventajas que ofrece respecto a otro tipo de tecnologías.

SAS

Es otra herramienta bien posicionada respecto al análisis de datos. Es un lenguaje de programación que permite manipular los datos de forma fácil y sencilla.

Es bastante manejable, además de tener una gran accesibilidad y tiene la capacidad de analizar un dato de forma independiente de la fuente de datos.

Python

Es una herramienta Open Source. Además de ser un lenguaje de programación orientado a objetos, es un lenguaje muy sencillo y fácil de mantener. Tiene diferentes bibliotecas para el aprendizaje automático, y se pueden usar con JSON, Servidores SQL y bases de datos MonoDb, entre otros.

A lo largo de este temario se han especificado con detalle las ventajas y características principales que ofrece este lenguaje de programación.

ER/Stuido

Este software ha sido construido para el diseño de las bases de datos y la arquitectura de datos.

Las características integradas de este *software* permiten a los modeladores poder automatizar las tareas más complejas.

Está centrado en diseñar los libros físicos y las capas de modelo de datos lógicos del modelo, además de poder generar de forma automática códigos para una infinidad de bases de datos.

Las principales funciones que ofrece este *software* son:

- Creación de un modelo de base de datos.
- Representa conceptos con documentación completa de atributos, definiciones y relaciones, entre otros.
- Descubre los documentos activos en todo el entorno de la base de datos.
- Permite visualizar los orígenes de datos y gestionar la transformación de datos entre el área de pruebas y el almacenamiento de datos.
- Determina las bases de un programa de comercio.
- Obtención de metadatos.
- Garantiza la coherencia entre los modelos y las bases de datos y coordina los cambios entre los diferentes equipos de desarrollo y los analistas.
- Realiza la documentación y la mejora de los datos.
- Implementa estándares de nomenclatura y un diccionario de datos, lo que permite mejorar la coherencia.
- Crea un glosario de los términos más importantes.
- Permite comunicar los metadatos y los datos de toda la organización.
- Permite crear una base de datos para programas de gobernanza.

Astera Centerprise

Esta plataforma está especialmente diseñada para respaldar y responder a las necesidades de integración de los datos complejos y que tienen un gran volumen. Con esta herramienta se permite obtener la validación de los datos que se han creado. Además, puede inspeccionar una muestra específica de los datos que se procesan en cada uno de los pasos del proceso de transformación.

Las principales características de este *software* son:

- **Automatización de flujos de trabajo y programación**: tiene un programador de trabajo integrado, lo que permite programar cualquier cosa, desde un trabajo de transformación de datos simple hasta un flujo de trabajo complejo comprendido por varios flujos de trabajo.

  También puede secuenciar los trabajos de integración y transformación, que se pueden ejecutar en serie o en paralelo en varios servidores. Otra característica muy importante es que se incluye la **ejecución de código SQL**, además de ejecuciones fuera del programa y capacidad de cargar y descargar mediante FTP. Asimismo, integra el servicio de correo electrónico.

- **Motor de procesamiento en paralelo:** tiene una arquitectura basada en clúster y un motor ETL de procesamiento paralelo, el software permite que los trabajos de transformación de datos se ejecuten en paralelo

  Todo el flujo de datos (o partes del mismo) se puede procesar en paralelos en varios nodos. Como resultado se obtiene un rendimiento superior, incluso cuando se tiene que procesar datos muy grandes.

- **Vista previa instantánea de datos:** esto permite inspeccionar una muestra de los datos procesados en cada paso del proceso de transformación.

  De esta manera, es posible identificar de forma rápida y eficaz cualquier error de asignación antes de ejecutar cualquier trabajo.

- **Entorno de mapeo sin código:** tiene un panel de control con interfaz visual que cuenta con las opciones de arrastrar y soltar, lo que permite un nivel avanzado de desarrollo, depuración y pruebas en un entorno sin código, totalmente gráfico.

  La plataforma ofrece una gran facilidad de uso —tanto para desarrolladores como para empresas—, con características fáciles de usar, como la vista previa de instantánea de datos, transformaciones integradas y conectividad nativa a varias fuentes de datos.

- **Implementación local y en la nube:** además de su implementación en la nube, tiene compatibilidad con los principales sistemas operativos, por lo que es una solución totalmente independiente de la plataforma.

- **Optimización *pushdown*:** permite insertar un trabajo de transformación de datos en una base de datos relacional, cuando corresponda, para poder hacer un uso óptimo de los recursos de bases de datos y mejorar de forma notable el rendimiento.

  Este sistema ayuda a las empresas a gestionar mejor las necesidades de procesamiento y a ahorrar más tiempo, además de impulsar la productividad de los desarrolladores.

- **Amplia selección de conectores**: el software cuenta con infinidad de conectores que permiten la integración con los orígenes de datos más modernos, además de con los tradicionales.

- **Biblioteca de transformaciones preconstruidas**: simplifica de forma notable el proceso de transformación de los datos jerárquicos complejos, ya que cuenta con un entorno visual de arrastrar y soltar, además de ampliar la selección de transformaciones integradas.

  Las transformaciones integradas tienen la posibilidad de encadenarse para crear un nuevo flujo de datos completo y que se puede automatizar mediante las características integradas de programación y automatización de trabajos.

- **Generación de perfiles y validación de datos**: al generar perfiles de datos se pueden examinar fácilmente los datos de origen, y se puede obtener información detallada sobre la estructura, integridad y calidad de los datos.

  Las reglas de integración y calidad de los datos personalizados también se pueden definir para validar los datos entrantes e identificar los registros que faltan o que no son válidos.

PowerDesigner

Este *software* es una de las herramientas más potentes que existen en el mercado. Es una solución ideal para cuando se manejan modelos de datos muy complejos. También es una herramienta de modelado y gestión de metadatos. Tiene sincronización y tecnología de enlace y análisis de impacto en informes basados en web.

PowerDesigner soporta las diferentes técnicas de modelación:

- **Modelación de datos:** modelación de datos conceptuales, lógicos y físicos.
- **Modelación de aplicación:** tiene UML y disponible de un mapeo relacional de objeto avanzado para realizar una implementación persistente.
- **Modelación del proceso de negocio:** dispone de descripciones de negocios y definición de diagramas. También permite la modelización de lenguajes de ejecución de procesos.
- **Modelos de arquitectura de empresa:** se enfoca en el objetivo profesional para mejorar la comunicación y coordinación de los equipos de las empresas.

Sparx System Enterprise Architect

Este *software* está enfocado en la creación de ideas y gráficos y la verificación de datos. También puede definir los flujos de trabajo y los diferentes modelos de datos. Se trata de un software muy útil para la gestión de datos.

Es compatible con el sistema operativo Windows —en un principio, el software se diseñó para este sistema—, con Linux a través de Wine y con MacOs a través de Mac CrossOver. Entre sus principales características destacan:

- **Estándares abiertos:** además de utilizar estándares abiertos, soporta los marcos de arquitectura empresaria como TOGAF y UPDM y herramientas personalizas e integradas para analizar y visualizar el software que se encuentra en ejecución.

  Permite realizar simulaciones avanzadas, y tiene herramientas de pruebas y repositorios basados en equipos de control de versiones.

- **Seguimiento de tareas:** permite realizar el seguimiento de tareas, además de la priorización y la asignación de tareas a los diferentes miembros del equipo de desarrollo.
- **Pruebas y depuración:** permite probar el comportamiento de los datos y verificar si son correctos. También simula procesos y permite la simulación avanzada mediante diferentes herramientas de testing.
- **Simulación:** es capaz de simular dinámicamente los modelos de comportamiento y estado, confirmar el diseño del proceso y especificar desencadenadores según las necesidades, eventos y restricciones.

Modelador de datos de desarrollador SQL

El modelador de datos de SQL es una herramienta gratuita de Oracle. Se trata de una herramienta gráfica que permite navegar por los datos y crear y actualizar los diferentes modelos de datos.

Tiene distintas capacidades de ingeniería inversa y directa, y permite la instalación tanto en local como la nube.

Las principales características con las que cuenta este *software* son:

- Funciona con casi todos los tipos de modelado de datos —solo le falta un tipo para llegar al total de seis disponibles en el mercado—. Los tipos de modelo que utiliza son el modelado de datos relacional, jerárquico, de red, entidad-relación, orientado a objetos y el modelo relacional.
- Permite realizar ingeniería inversa y el análisis de su impacto.
- Permite anidar los diagramas uniendo diagramas de modelos de datos separados.
- Puede generar informes de forma automática.

Los modelos de datos que se diseñan bien permiten que las organizaciones que lo usan tengan un buen acceso y conocimiento de todos los datos, por lo que es muy importante escoger la herramienta de modelado adecuada.

Los beneficios que se experimentan en la organización al elegir el software de diseño de modelo adecuado son:

- **Rendimiento:** al obtener un modelo de datos bien diseñado este debe de funcionar más rápido por lo que además facilitará la navegación a través de los datos.
- **Coste**: cuando el modelo de datos contiene errores y no se detectan de forma temprana, a la larga pueden resultar muy caros de resolver. Por eso, un software de modelado de datos ayuda a reducir el coste, evitando que el sistema de modelado de datos tenga errores.
- **Eficiencia:** al tener un modelo de datos bien construido, los datos dentro de él fluyen de forma correcta, mientras se hacen todo tipo de operaciones con el modelo.
- **Calidad**: como los datos se ha organizado de la mejor forma posible, se podrá obtener una información de mayor calidad a partir de ellos.

Resumen

Cuando se requiera el acceso a la base de datos, es necesario, mediante la programación, abrir una conexión y un puerto para las comunicaciones. Después, se procederá a ejecutar las consultas deseadas. La función encargada de ejecutar las consultas podrá devolver los datos en caso de que hayan sido solicitados. Tras finalizar todo el proceso, se cierra la conexión.

También es conveniente conocer los diversos programas de *software* que permiten mejorar el tratamiento de los datos y su modelación. Este tipo de herramientas son muy útiles para obtener un nivel de automatización mayor.

Capítulo 6

Concepto de procesos e hilos

En la actualidad, los ordenadores tienen la capacidad de llevar a cabo más de un proceso a la vez, debido a que utilizan varios núcleos. Para entender mejor el funcionamiento de los procesos e hilos, es necesario entender como era posible ejecutar diferentes procesos a la vez en máquinas mononúcleo.

En realidad, no se ejecutan (ni se ejecutaban) varios procesos a la vez: los procesos se turnan y, debido a la velocidad a la que se pueden ejecutar las instrucciones dadas, nos da la impresión de que las tareas se ejecutan de forma simultánea en el sistema. Cuando un proceso distinto tiene que ejecutarse, es necesario realizar un cambio de contexto. Lo que esto significa es que el programa que se ejecuta salva su estado en memoria y se carga el estado del programa que entra para ejecutarse. Podemos crear nuevos procesos mediante la función *os.fork* en Python, que lo que hace es ejecutar la llamada al sistema fork. Sin embargo, también existen otras funciones más avanzadas, como popen2. De esta forma, el programa puede realizar varias tareas de forma paralela mediante el uso de estos métodos.

Cuando se realiza el cambio de contexto, se pierde bastante rendimiento. Esto es así debido a que es un proceso bastante lento y los recursos que son necesarios para mantener el estado son exigentes. En estos casos, es mejor utilizar **hilos de ejecución, *threads* o procesos ligeros,** pues es la forma más eficaz de implementar los procesos paralelos. Los procesos son similares a los *threads* (ya que ambos son código de ejecución), aunque los *threads* se ejecutan siempre de un proceso, y todos los *threads* de un proceso comparten los recursos entre sí.

Los sistemas operativos necesitan más recursos para crear los procesos, por lo que, cuando se crean *threads,* se ahorran más recursos: al compartir los recursos, el cambio de contexto es más rápido y eficiente.

Los *threads* siempre comparten el mismo espacio de memoria global, lo que permite que los *threads* puedan compartir la información. Así, si se ha declarado una variable global en el programa, esta es vista por todos los *threads* del programa.

El GIL

Es el encargado de controlar la ejecución de los *threads* en Python. De esta forma, solo un thread puede ejecutarse a la vez, independientemente del número de procesadores que tenga la máquina donde se está ejecutando el programa.

Como efecto colateral, este comportamiento con los *threads* en Python permite que escribir extensiones en C se pueda llevar a cabo de una forma muy simple, aunque limita bastante el rendimiento. Por ello, en Python, según las necesidades, es más recomendable utilizar más los procesos que los *threads*, ya que los procesos no sufren ese tipo de limitaciones.

Sin embargo, cuando se llevan a cabo un cierto número de instrucciones, la máquina virtual detiene la ejecución del thread y, de entre los que están esperando, elige otro para su ejecución.

Por norma general, el cambio de thread se realiza cada 10 instrucciones. Este valor es modificable mediante el método *sys.setcheckinterval,* en el cual se puede definir el número de instrucciones que se pueden ejecutar antes de cambiar de thread.

El thread también se puede cambiar cuando el hilo principal se pone a dormir con la función time.sleep(), o si empieza una nueva operación de entrada/salida. Cuando se genera una operación de entrada/salida, estas pueden tardar bastante tiempo en terminar, por lo que el thread se duerme a la espera de que el proceso de entrada/salida finalice su ejecución. En el caso de que el proceso de entrada/salida no termine el procesador se quedará esperando a que termine dicho proceso antes de continuar con otro.

Para optimizar el GIL referente al rendimiento, se puede llamar al intérprete con la extensión *-flag* O, para generar un código más optimizado y que use menos instrucciones, tratando de evitar los cambios de contextos.

Es recomendable utilizar procesos en lugar de *threads* cuando el rendimiento sea un tema primordial en la aplicación.

6.1. Aplicaciones de procesos con el módulo multiprocessing de Python

El módulo *multiprocessing* es el que permite dividir el proceso principal en múltiples procesos basados en la interfaz de programación para el *threading*. Esto permite aprovechar mejor los múltiples núcleos que tienen las máquinas actuales y evitar los cuellos de botella que se pueden producir con Python asociados al bloqueo global del intérprete.

La diferencia principal entre el ***threading*** y el ***mutiprocessing*** es la protección con la que cuenta el *__main__*. La diferencia radica en cómo los procesos son iniciados. Para ellos, el proceso hijo es el encargado de importar la secuencia de comandos que contiene la función de destino. Envolver la parte principal de la aplicación con el método *__main__* permite asegurarse de que no se ejecuta de forma recursiva en cada uno de los hijos cuando se importe el módulo. Otra forma de realizarlos es importar primero la función de destino desde otra secuencia de comandos totalmente separada, usando *multiprocessing_import_main.py*.

Determinación del proceso actual

Para determinar el proceso actual, es conveniente evitar el uso de argumentos para identificar o nombrar el proceso, ya que esta forma es bastante complicada y puede inducir a errores.

A cada instancia del proceso se le asigna un nombre con un valor predeterminado, que puede cambiarse una vez se crea el proceso. Los procesos que usan denominaciones son bastante útiles a la hora de realizar el seguimiento de cada uno de ellos. Esto será necesario cuando la aplicación necesite ejecutar múltiples procesos de forma simultánea.

A continuación, se muestra un ejemplo:

```
import multiprocessing
import time

def worker():
        name = multiprocessing.current_process().name
        print(name, 'Starting')
        time.sleep(2)
        print(name, 'Exiting')

def my_service():
        name = multiprocessing.current_process().name
        print(name, 'Starting')
        time.sleep(3)
        print(name, 'Exiting')
```

```
if __name__ == '__main__':
        service = multiprocessing.Process(
            name='my_service',
            target=my_service,
        )
        worker_1 = multiprocessing.Process(
            name='worker 1',
            target=worker,
            )
        worker_2 = multiprocessing.Process(  # default name
             target=worker,
        )
        worker_1.start()
        worker_2.start()
         service.start()
```

Pasar mensajes a procesos

En la creación de múltiples procesos, lo que se hace es dividir el proceso principal en procesos más pequeños con la idea de repartir los diferentes trabajos que tienen que realizar. Al dividir el proceso en varios procesos diferentes es posible paralizar las tareas. Esto permite que se haga un uso más eficaz de los procesos múltiples que requieren que tengan algún tipo de comunicación.

Una de las técnicas que se utilizar para comunicarse entre procesos es con el uso de *queue,* que permite pasar mensajes de ida y vuelta. Los objetos que se quieran pasar entre procesos deben serializarse mediante *pickle,* para luego poder ser almacenados en la *queue* y que luego puedan ser pasados al proceso en cuestión.

A continuación, se muestra un ejemplo de implementación de multiproceso con transferencia de datos entre ellos:

```
if __name__ == '__main__':
    queue = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()
    queue.put(MyFancyClass('Fancy Dan'))
```

Señalizar entre procesos

Cuando múltiples procesos se están ejecutando, a veces es necesario comunicar el estado de cada proceso entre ellos. Para eso se utiliza la clase *event*, que permite comunicar la información de estado entre procesos de manera sencilla.

Los eventos tienen dos posibles estados: armado y desarmado. Los procesos que usen el objeto pueden esperar a que el evento cambie de estado (en este caso, de desarmado a armado) mediante un valor opcional de tiempo de espera.

```
def wait_for_event(e):
        """Wait for the event to be set before doing anything"""
        print('wait_for_event: starting')
         e.wait()
        print('wait_for_event: e.is_set()→', e.is_set())
def wait_for_event_timeout(e, t):
        """Wait t seconds and then timeout"""
        print('wait_for_event_timeout: starting')
         e.wait(t)
        print('wait_for_event_timeout: e.is_set()→', e.is_set())
```

Controlar el acceso a los recursos

Si un recurso único tiene que ser compartido entre más procesos, se tiene que bloquear en algún momento para que un único proceso pueda acceder a él. Para ello se utiliza lock, para evitar conflictos.

```
def worker_no_with(lock, stream):
        lock.acquire()
        try:
            stream.write('Lock acquired directly\n')
```

Sincronización de operaciones

Cuando se busca que algunos de los flujos de trabajo se ejecuten en paralelo y otros de forma secuencial, se utiliza el objeto *condition*, incluso aunque estén en procesos separados.

```
s2_clients = [
            multiprocessing.Process(
                name='stage_2[{}]'.format(i),
                target=stage_2,
                args=(condition,),
            )
             for i in range(1, 3)
]
```

Controlar el acceso concurrente a los recursos

En ocasiones, es necesario permitir que más de un proceso puedan acceder a un mismo recurso a la vez y, al mismo tiempo, limitar el número máximo de conexiones que puede soportar de forma simultánea, se utiliza el objeto *Semaphore,* que permite gestionar las conexiones:

```
if __name__ == '__main__':
            pool = ActivePool()
            s = multiprocessing.Semaphore(3)
            jobs = [
                multiprocessing.Process(
                    target=worker,
                    name=str(i),
                    args=(s, pool),
                )
                for i in range(10)
```

Gestionar el espacio compartido y espacios de nombre

Cuando se pretende gestionar el espacio compartido que usan varios procesos a la vez, se utiliza el objeto Manager, que es el responsable de coordinar el estado de la información compartida entre los usuarios que acceden al recurso.

Manager es capaz de gestionar un espacio de nombre compartido para todos los procesos.

Agrupación de procesos

Cuando se necesita agrupar un número de procesos para que trabajen de forma conjunta sobre un problema, se utiliza la función pool. Esta función permite administrar un número fijo de procesos para los casos simples, donde el proceso que se tiene que realizar permite que se puede dividir entre procesos de forma independiente.

6.2. Aplicaciones de hilos con el módulo threading de Python

El uso de hilos permite a los programas ejecutar múltiples operaciones de forma simultánea, usando el mismo espacio de proceso.

Para usar un hilo es necesario declarar un *thread,* es decir, crear una instancia con la función de destino y empezar a ejecutar el hilo utilizando la función *start()*.

A continuación, se muestra una implementación de un *thread:*

```
import threading
def worker():
        """thread worker function"""
        print('Worker')
threads = []
for i in range(5):
        t = threading.Thread(target=worker)
        threads.append(t)
        t.start()
```

```
$ python3 threading_simple.py

Worker
Worker
Worker
Worker
Worker
```

Cuando se trabaja con hilos, lo habitual es que se requiera pasar parámetros o argumentos para indicarle el trabajo que tienen que realizar, cualquier objeto se le puede pasar a un hilo. A continuación, se muestra un ejemplo:

```
threads = []
for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
```

Determinar el hilo actual

A igual que en los procesos, no es del todo recomendable utilizar argumentos para identificar los hilos, ya que es más complejo y puede inducir a error.

Cada uno de los hilos tiene asignado su nombre con el valor predeterminado que se puede indicar cuando se crea el hilo. Nombrar los hilos es bastante útil, ya que cuando hay muchos procesos e hilos corriendo a la vez, se pueden identificar de una forma más clara e inequívoca.

A continuación, se muestra un ejemplo en la que se le indica el nombre que se le asigna a cada uno de los hilos:

```
t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker)  # use default name
```

Cuando se requiere hacer una depuración, el módulo *logging* también admite añadir el nombre del hilo en cada mensaje que imprime. Esto permite que, al incluir los nombres a los hilos, se pueda depurar más fácilmente el código y el comportamiento de cada uno de los hilos.

Hilos de Daemon vs. No-Daemon

Los hilos se pueden ejecutar como demonios o no demonios. En caso de ejecutarlos como demonios, estos se ejecutan **sin bloquear el hilo principal.** En los programas donde no hay una forma clara de interrumpir el hilo, se puede utilizar e hilo como demonio siempre que no se pierdan los datos.

Para marcar un hilo como demonio se utiliza *Daemon*=true en la construcción del hilo. También, de forma alternativa, se puede llamar al método *set_daemon* con el parámetro True.

De forma predeterminada, los hilos se ejecutan como no demonios, por lo que, en caso de necesitar que sean demonios, es necesario indicarlo.

```
d = threading.Thread(name='daemon', target=daemon, daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)
```

Se puede utilizar el método *join()* en el caso de que se requiera esperar que un subproceso demonio termine su trabajo.

```
d.start()
t.start()
d.join()
t.join()
```

Por defecto, el método *join()* realiza un bloqueo de forma infinita, aunque es posible pasar mediante argumento el número de segundos que se quiere esperar por el hilo hasta pasar a estar inactivo.

En el caso de que el hilo no se complete en el tiempo indicado, el *join()* se vuelve a ejecutar de nuevo, y así sucesivamente hasta que el hilo termine su trabajo.

Enumerar los hilos

Cuando se necesita saber la cantidad de hilos activos, se utiliza el método *enumerate()*. De esta manera, no es necesario guardar un identificador para cada uno de los hilos demonio para intentar asegurar que todos se han completado antes de salir del proceso principal.

Señalización entre hilos

La idea de usar múltiples hilos se basa en ejecutar de forma separa las operaciones. Sin embargo, cuando se necesita sincronizar las operaciones de varios hilos, se utilizan los eventos, que son la forma más sencilla de comunicarse entre los diferentes hilos.

El *event()* es el que se encarga de gestionar la bandera que permite la comunicación entre hilos, además controla también los métodos *set() y clear()*.

Además, se puede usar el método *wait()* para pausar un hilo hasta que se establezca una bandera, lo que permite bloquear el hilo hasta nuevo aviso.

Controlar el acceso a los recursos

Para prevenir la corrupción de datos o pérdidas de información, es importante controlar el acceso de los hilos a lo recursos compartidos.

Python cuenta con estructuras de datos incorporadas, como listas o diccionarios, que son seguras cuando se trabaja con subprocesos. Sin embargo, otros tipos de datos (los más simples, como los números enteros o flotantes) no cuentan con esta protección.

Cuando se necesita proteger un objeto contra el acceso simultáneo, se utiliza el objeto ~~look.~~

Resumen

Las máquinas pueden simular que se ejecutan varios procesos a la vez mediante el uso de procesos e hilos. El sistema encargado de gestionar la ejecución de estos procesos e hilos es el sistema GIL.

De esta manera, mediante el multiprocesamiento se puede dividir un proceso muy pesado en varios procesos, más pequeños, que sean más fáciles de ejecutar. Estos procesos más pequeños podrán ser procesados en diferentes procesos o hilos, según las técnicas que se quieran utilizar. De esta manera, entre todos, obtendrán un resultado final que será leído por el proceso principal.

Capítulo 7

Introducción al procesamiento distribuido

7.1. Introducción al procesamiento distribuido en el tratamiento de datos

Solucionar los problemas que se pueden presentar mediante el procesamiento distribuido ya se hace desde hace algún tiempo: se basa en el método de dividir para hacer más simples los problemas y poder solucionarlos de una forma más fácil.

Un sistema distribuido es un conjunto de máquinas que están conectadas en red y que producen la sensación de que se está trabajando directamente en una sola máquina. Aunque esta forma de trabajar agrega mayor complejidad al *software* y, como contrapartida, disminuye los niveles de seguridad, tiene ventajas si lo vemos desde la perspectiva precio-desempeño.

Estos sistemas pueden aumentar de forma gradual su tamaño para también aumentar de forma gradual la carga. Lo que se persigue entonces es, mediante paralelismo, poder reducir la carga de ejecución de un programa, o lo que es lo mismo, reducir el número de ciclos que se tienen que realizar para completas todas las tareas.

Las principales ventajas que tiene el procesamiento distribuido son las siguientes:

- **Mayor economía:** en los sistemas centralizados la relación de precio y rendimiento es mucho peor que en los sistemas distribuidos, ya que en estos se consigue mejor ratio entre ambas variables. Se nota aún más la diferencia entre ambos tipos cuando se necesitan altas prestaciones.
- **Velocidad**: a un sistema centralizado llegará un momento en que le sea imposible manejar toda la información y datos, cuando crezca demasiado, ya que se tendría que usar una máquina muy potente.

  Por ello, se utiliza el sistema de "divide y vencerás": es decir, en los sistemas distribuidos la carga de trabajo se reparte entre todas las máquinas que componen el sistema y cada máquina añade más potencia al sistema distribuido.
- **Distribución de máquinas**: las máquinas se pueden encontrar en cualquier lugar, esparcidas por todo tipo de localizaciones, no es necesario que todo el sistema se encuentre en un único lugar.
- **Alta disponibilidad**: cuando una máquina falla en un sistema distribuido, no hace que caiga el resto del sistema, ya que, al contar con más máquinas, otra cogerá el relevo y hará mientras las tareas de la máquina que ha dejado de funcionar.

  De este modo, el sistema es capaz de recuperarse de caídas de los nodos, aunque quizás pueda disminuir algo la velocidad.
- **Escalabilidad**: se puede empezar a utilizar un sistema distribuido con pocas máquinas al principio y, luego, conforme se va encontrando que la carga empieza a ser elevada para el sistema, se pueden añadir nuevas máquinas.

  Por tanto, para realizar una actualización no es necesario dejar de utilizar ninguna máquina anterior ni tampoco se requiere una gran inversión inicial.
- **Comunicación**: en un sistema distribuido, todo el sistema se encuentra comunicado. Las comunicaciones entre máquinas tienen que ser rápidas, por lo que se crean nuevas funcionalidades avanzadas de comunicación que permiten mejorar la conexión entre las distintas máquinas.

  Estas nuevas funcionalidades de comunicación las pueden utilizar los programas y también pueden ser usadas por los usuarios para mejorar todas las comunicaciones entre las máquinas.
- **Sistema de ficheros con raíz única:** tener un sistema de fichero con raíz única hace que el sistema de administración sea más sencillo, ya que no es necesario administrar varios discos independientes y el sistema se encarga de las diferentes tareas relacionadas con el almacenamiento.
- **Capacidad de comunicación de procesos y de intercambio de datos universal:** es posible enviar cualquier tipo de señal a cualquier máquina que se encuentre dentro del clúster. Se pueden realizar trabajos de forma conjunta entre todas las máquinas o intercambiar información entre ellas.

Otra ventaja que se da en los sistemas distribuidos de procesamiento es que cumplen de forma inequívoca con todos los criterios de transparencia. Para ello se tienen que implementar los siguientes mecanismos:

- **Transparencia de acceso**: es necesario mantener el antiguo sistema para el nuevo clúster, es decir, no se deben de romper las diferentes API con la que se trabaja para poder introducir nuevas funcionalidades.

- **Transparencia de localización:** es el nivel más bajo en el que se sabe dónde se encuentra los recursos. Para ello, tiene que haber implantado un *software* que nos provea de esa información.

  Cada vez se tiene más tendencia a que la información de la localización de los recursos esté más esparcida en la red, pero hasta hace no mucho el encargado de guardar dicha información era un sistema centralizado.

- **Transparencia de concurrencia:** el primer problema a resolver es que todas las máquinas deben tener sus relojes sincronizados. No obstante, es muy difícil conseguir que el *hardware* que se encarga de indicar el tiempo lo haga exactamente igual en todas las máquinas que forman el sistema. Esto hace que algunas máquinas puedan ver algunos hechos como pasados o futuros, según cómo tengan sincronizado el reloj del sistema.

- **Transparencia de replicación:** el sistema tiene que saber que las réplicas existen y se encuentran en el sistema, además de mantenerlas de forma coherente y sincronizadas en todo momento.

  En caso de que ocurra cualquier error, es necesario que se activen todas las acciones necesarias y requeridas para poder solucionarlo de forma transparente.

- **Transparencia de fallos:** esto permite que, aunque se puedan producir fallos en el sistema, este pueda seguir funcionando de forma correcta. Los usuarios del sistema distribuido no tienen por qué saber que se han producido fallos: el sistema debe de ser capaz de gestionarlos o deben ser minimizados de forma transparente.

- **Transparencia de migración**: se activa cuando se tienen que solucionar los inconvenientes que provoca una migración de datos, teniendo en cuenta las diferentes políticas de migración y las ubicaciones.

  Las aplicaciones que usan el sistema distribuido no tienen por qué enterase de que el sistema ha sido migrado: debe de ser un proceso transparente y que pase desapercibido para el resto de usuarios del sistema.

- **Transparencia de los usuarios:** esto implica contar con una capa de abstracción de software que tenga una apariencia parecida a las anteriores.

- **Transparencia para programas**: los programas no tienen por qué usar nuevas llamadas específicas para los nuevos clústeres. Todo debe de funcionar de forma transparente y sin cambios que puedan apreciar los programas.

Si nos enfocamos ahora en las desventajas que representa el sistema distribuido de datos, hay que tener en cuenta que la principal es la complejidad que implica su creación, además de los problemas típicos que se tendría en caso de que no estuviera escalado: al estar escalado, los problemas se amplifican.

Las desventajas que se han de tener en cuenta en un sistema distribuido de datos son las siguientes:

- **Alta disponibilidad:** es posible tener una alta disponibilidad al contar con varias máquinas independientes, lo que hace que haya muchas menos posibilidades de que el sistema caiga al completo. Aun así, esto por sí solo no supone que realmente se tenga una alta disponibilidad, ya que será necesario implantar previamente todos los mecanismos imprescindibles para que cuando un nodo de deje de responder de forma abrupta, se puedan seguir dando los diferentes servicios.

- **Escalabilidad**: como se pueden ir añadiendo nodos al sistema, esto implica la necesidad de mejorar las comunicaciones entre nodos. Por ello, el sistema se debe diseñar desde un principio para que sea lo más escalable posible y no pierda eficiencia a lo largo del tiempo.
- **Comunicación**: los sistemas normales no tienen tantas necesidades de comunicación como un clúster, por lo cual hay que volver a hacer hincapié en la mejora de las comunicaciones en todo el sistema distribuido.
- **Sistema de ficheros con raíz única**: es necesario tener que independizar los sistemas de ficheros de las distintas máquinas que forman el sistema y crear un sistema general en el que se encuentren todos los datos.
- **Capacidad de comunicación de procesos y de intercambio de datos universal:** para conseguir esto es necesario que cada uno de los procesos del clúster pueda ser identificado. A cada proceso se le puede asignar un ID que permita su fácil identificación por el sistema; de esta forma, se podrá hacer referencia al proceso y comunicarnos con él.

Clústeres

Un clúster es un conjunto de máquina unidas por una red de comunicación y trabajando todas por un objetivo común. La principal característica que tiene un clúster es que existe un medio de comunicación que permite a los procesos migrar para procesarse en los diferentes nodos.

Un único nodo no se puede considerar un clúster, ya que, por su condición de aislamiento, no puede compartir información. Por ello, para que un sistema tenga consideración de clúster es necesario, al menos, que haya dos nodos.

Las arquitecturas con varios procesadores en placa tampoco son consideradas clústeres, ya que el medio de comunicación es interno, no una red.

Las principales características de un clúster son las siguientes:

- Debe tener más de dos nodos que han de estar conectados por medio de una red para poder funcionar.
- Los nodos de un clúster están conectados por una canal de comunicación que les permite comunicarse entre sí.
- Los clústeres requieren de un *software* especializado que permite hacerlos funcionar de forma correcta:
  - ***Software* de aplicación**: cuando es generado por diferentes bibliotecas, estas permiten la abstracción para crear programas de la manera más abstracta posible. Los elementos que utiliza este tipo de software son de tipo rutinas, tareas o procesos que se comunican entre sí cuando se ejecutan en varios nodos del clúster.
  - *Software* de sistema: este tipo de software suele estar implementando como parte funcional del sistema operativo de cada nodo o de la totalidad de todos ellos. Es mucho más complejo que el *software* de aplicación y resuelve problemas de carácter mucho más general.
- Todos los elementos del clúster tienen que trabajar para cumplir unos propósitos de forma conjunta.

Para la catalogación de los clústeres nos fijamos en sus factores de diseño. Teniendo en cuenta esto, los podemos diferenciar gracias a diferentes tipos de factores:

- **Acoplamiento**: que se puedan añadir más nodos al sistema.
- **Control**: indica el modelo de gestión que propone el clúster, es decir, la manera en la que se tiene que configurar y cómo es dependiente de los demás nodos del sistema y de cómo el resto del sistema se configure. Según el tipo de control, pueden ser:
  - **Centralizado**: existe un nodo que es el maestro, desde el cual se realizan todas las configuraciones de comportamiento de todo el sistema. Permite gestionar de una forma más fácil el conjunto de clústeres.
  - **Descentralizado**: en este caso, cada nodo debe administrarse a sí mismo, aunque también puede haber aplicaciones de alto nivel que ayuden a la gestión. Presenta mejor tolerancia a fallos y es el más común en los sistemas distribuidos, aunque este tipo de administración implica que la configuración requiere de más tiempo.
  - **Homogeneidad**: depende de los equipos y recursos de los que esté formado el sistema. Puede haber varios tipos:
    - **Homogéneos**: tienen la misma arquitectura y recursos parecidos, por lo cual no hay muchas diferencias entre nodos.
    - **Heterogéneos**: están formados por nodos que no son totalmente iguales. Puede haber diferencias en los tiempos de acceso, en la arquitectura, en el rendimiento o en los sistemas operativos, entre otros.
  - **Seguridad del sistema.**

Clústeres de alto rendimiento

Han sido creados para compartir el tiempo de proceso. Se suelen utilizar en aplicaciones que requieren de una gran cantidad de procesamiento, como puede ser en ambientes científicos o grandes empresas, donde se necesitan para realizar compilaciones y renderizaciones muy pesadas que en cualquier otro sistema no se podrían hacer.

Cualquier operación que requiera gran cantidad de procesamiento debe ser ejecutada en un clúster de alto rendimiento.

Este tipo de clústeres permite mejorar el rendimiento general del sistema y, sobre todo, el de la solución del problema en concreto.

El procesamiento necesario en los sistemas distribuidos se resuelve entre los diferentes nodos del sistema, pero si alguno está especializado en el campo de trabajo, se le otorga la posibilidad de realizar la tarea en la mayor parte o en su totalidad.

No obstante, los clústeres que trabajan a nivel de software no suelen llevar ningún sistema de balanceo de carga, sino que suelen basar toda su política de funcionamiento en la localización de los procesos entre los diferentes nodos del sistema. Aunque resuelve el problema, es necesario contar con un software que esté programado para solucionar esos problemas en concreto de forma eficiente en todo el sistema de nodos.

En cambio, los clústeres que trabajan a nivel de sistema basan todo su funcionamiento en la comunicación constante entre los demás nodos del sistema, también a nivel de sistema operativo, por lo que suelen ser

nodos con la misma arquitectura. En este caso, se trata de compartir todos los recursos del sistema para resolver la tarea, además de tener un balanceo de carga de forma dinámica.

Clústeres de alta disponibilidad

Esto tipo de clústeres son de tipo ortogonal en lo que se refiere a la funcionalidad que tienen respecto a un clúster de alto rendimiento. Los clústeres de alta disponibilidad tienen como prioridad máxima que los servicios se encuentren en constante funcionamiento: se han diseñado para proporcionar una alta disponibilidad sobre los servicios que ofrece el propio clúster.

Este tipo de clústeres permite abaratar el coste de los sistemas redundantes y su base filosófica está en que tienen que ser confiables, tienen que estar disponibles y tienen que proporcionar siempre servicio. Por ello, se utilizan cuando es necesario dar un servicio de cualquier tipo a distintos usuarios, de una forma ininterrumpida.

La forma de mantener funcionando el sistema de forma constante es eliminar de este aquellos puntos que sean críticos y que puedan producir algún tipo de fallo, el cual, a su vez, haga que el sistema quede fuera de servicio.

Por ello, se recurre a la redundancia, para que, en caso de que algún sistema falle, otro entre en su lugar.

Clústeres de alta confiabilidad

Estos clústeres están destinados a garantizar que siempre se van a comportar de la misma forma. Son los clústeres más complicados de implementar, ya que, aparte de dar servicios de alta disponibilidad, en la mayoría de los casos el entorno tiene que ser muy confiable en todo momento. Esto quiere decir que, además, hay que tener especial cuidado con el *software* para que este responda como debe y no caiga antes posibles fallos.

Se suelen utilizar en entornos empresariales y mediante un *hardware* especializado, ya que es imposible obtener clústeres de alta confiabilidad solo por software que puedan ser lo suficientemente eficientes.

7.2. Aplicación de procesamiento distribuido con el módulo Celery de Python

Una cola de tareas se utiliza como un mecanismo para distribuir el trabajo entre los diferentes subprocesos y máquinas. La tarea es la unidad que puede entrar por la cola de entrada y los procesos de trabajo son los encargados de supervisar de manera constante todas las colas de tareas para que se pueda realizar un nuevo trabajo.

En el caso de Celery, esto se comunica mediante mensajes y, generalmente, utiliza un intermediario entre clientes y procesos. Para que una tarea se pueda iniciar es necesario que el cliente añada un mensaje a la cola; luego, el intermediario es el que se encarga de añadir el mensaje al proceso. Y un sistema Celery puede tener varios procesos, por lo que tiene alta disponibilidad y escalabilidad horizontal.

Por tanto, para que Celery funcione es necesario que alguien transporte los mensajes que necesita enviar y recibir: los principales transportes pueden ser RabbitMQ o Redis, entre otros, incluso SQLite, pero en desarrollo local.

Celery está escrito en Python, aunque el protocolo que utiliza es posible implementarlo en cualquier lenguaje. La interoperabilidad del lenguaje también se puede lograr exponiendo un punto de conexión HTTP y teniendo una tarea que lo solicite (webhooks).

Se puede ejecutar en una única máquina o en varias, según las necesidades, e incluso en más de un centro de datos.

Lo primero para empezar a usar Celery es la elección de un bróker. Esta será la solución que permitirá a Celery enviar y recibir los mensajes y podemos elegir, preferentemente, entre RabbitMQ o Redis.

RabbitMQ es fácil de instalar, además de completo. También es estable y se recomienda para los entornos de producción. Para instalar RabbitMQ, se puede ejecutar la siguiente instrucción:

```
$ sudo apt-get install rabbitmq-server
```

Redis también es una opción válida, aunque es más susceptible de pérdida de datos en caso de que pare de forma abrupta o se produzcan fallos de energía. Se puede instalar Redis ejecutando la siguiente instrucción:

```
$ docker run -d -p 6379:6379 redis
```

Existen otros tipos de conectores, pero estos son los más conocidos y los más usados en la actualidad.

Ahora hay que instalar Celery. Este se encuentra entre los paquetes de Python, por lo que se puede instalar directamente ejecutando el siguiente comando:

```
$ pip install celery
```

Para empezar a usar Celery se necesita crear una instancia. Esta instancia es ApplicationCelery y es la que nos permite utilizarlo como punto de entrada para todo lo que se quiera hacer, como puede ser crear nuevas tareas y administrar procesos, además de que los otros módulos disponibles en el sistema lo puedan importar.

```
$From celery import Celery
app = Celery('tasks', broker='pyamqp://guest@localhost//')
@app.task
defadd(x, y):
return x + y
```

El primer argumento es el nombre del módulo actual. Este solo es necesario en caso de que se quiera definir, ya que el nombre se genera automáticamente cuando se declara Celery. No obstante, para poder identificarlo de forma más fácil es recomendable indicarlo.

El segundo argumento especifica la URL del agente de mensajes que se quiere utilizar. En este caso, habría que indicar el agente de RabbitMQ, en caso de querer usarlo.

Ya se puede ejecutar el proceso deseado en Celery, indicando el nombre de dicho proceso que se quiere ejecutar:

```
$ celery -A tasks worker --loglevel=INFO
```

Normalmente, en producción este tipo de procesos se deberían ejecutar en segundo plano, como demonio, para que trabaje sin que interrumpan los procesos principales. Para ello se puede utilizar la opción Daemon=-true, para indicar que es un proceso demonio.

Cuando se quiera llamar a una tarea, se utiliza el método *delay*(). Este método nos permite acceder al método *apply_sync*() que nos asegura un mejor control de las tareas que se ejecutan en segundo plano.

```
>>>from tasks import add
>>>add.delay(4, 4)
```

Para almacenar los resultados de los procesos, Celery necesita almacenar o enviar los estados a algún sitio para que luego puedan ser vistos. Por ejemplo, se podría usar MongoDB para la interpretación de los resultados.

Para ello, en el apartado de backend le podemos pasar el manejador de resultados elegidos. En este caso se ha de elegir Redis:

```
app = Celery('tasks', backend='redis://localhost', broker='pyamqp://')
```

El método *ready*() indica si la tarea a terminado o no.

Se detallan cada una de los serializaciones que componen el Celery:

- *celery[auth]*: permite utilizar el serializador de seguridad *auth*.
- *celery[msgpack]*: permite el uso del serializador *msgpack*.
- *celery[yaml]*: permite el uso del serializador *yaml*.

En caso de necesitar concurrencia, se pueden utilizar las siguientes opciones:

- *celery[eventlet]*: para el uso del grupo de eventlets.
- *celery[gevent]:* para utilizar el grupo de eventos.

En el ámbito de transportes y backends en Celery, contamos con los siguientes elementos:

- *celery[librabbitmq]*: uso de la biblioteca *librabbitmq* C.
- *celery[redis]*: permite usar Redis como transporte de mensajes o como *backend* de resultados.
- *celery[sqs]*: utilizar Amazon SQS como transporte de mensajes (experimental).

  celery[tblib]: permite utilizar la función *task_remote_tracebacks*.
- *celery[memcache]*: permite el uso de Memcached como resultado *backend* (usando *pylibmc*).
- *celery[pymemcache]:* permite el uso de Memcached como *backend* de resultados (implementación de Python puro).
- *celery[cassandra]:* utilizar Apache Cassandra como *backend* resultante con el controlador DataStax.
- *celery[couchbase*]: usar Couchbase como *backend* de resultados.

- *celery[arangodb]*: uso de ArangoDB como resultado *backend*.
- *celery[elasticsearch]:* permite utilizar Elasticsearch como *backend* de resultados.
- *celery[riak*]: uso de Riak como resultado *backend*.
- *celery[dynamodb*]: AWS DynamoDB como *backend* de resultados.
- *celery[zookeeper]*: permite utilizar Zookeeper como transporte de mensajes.
- *celery[sqlalchemy]:* usar SQLAlchemy como *backend* de resultados (compatible).
- *celery*[*pyro* uso del transporte de mensajes Pyro4 (experimental).
- *celery[slmq]*: utilizar el transporte de cola de mensajes SoftLayer (experimental).
- *celery[consul]*: permite utilizar el almacén de Consul.io clave/valor como transporte de mensajes o *backend* de resultados (experimental).
- *celery[django]:* especifica la versión más baja posible para la compatibilidad con Django.

De forma práctica, Celery es capar de ejecutar varias tareas a la vez. En el siguiente ejemplo se puede comprobar cómo se puede programar cinco procesos de compra mediante el uso de *stagning_time*.

```
@product_router.get("/buy/{name}")
asyncdefbuy(name: str):
for i in range(0, 5):
move_to_next_stage.apply_async((name, stages[i]), countdown=i*STAGING_TIME)
return True
@celery.task
defmove_to_next_stage(name, stage):
redis_store.set(name, stage)
returnstage
@product_router.get("/status/{name}")
asyncdef status(name: str):
returnredis_store.get(name)
```

Las principales características que Celery son:

- **Simple**: es muy fácil de usar y mantener, ya que no necesita archivos de configuración. Además, tiene una comunidad muy activa, con lo que, en caso de necesitar soporte o algún tipo de apoyo, se puede consultar a la propia comunidad de desarrolladores.

Es una de las herramientas más sencillas que se pueden utilizar en el procesamiento distribuido.

- **Alta disponibilidad:** los diferentes procesos y clientes, en caso de error o pérdida de conexión, vuelven de nuevo a intentar la ejecución. Además, es posible configurar replicaciones.
- **Rapidez**: un único proceso de Celery es capaz de procesar millones de tareas por minuto con muy poca latencia. Todo eso, por supuesto, con los ajustes bien optimizados, según donde se use.

- **Flexible**: casi todas las partes de Celery se pueden ampliar. Se pueden añadir implementaciones, serializaciones, esquemas de compresión, registros, programadores, consumidores, productores o *brokers*, entre otros.

Las principales funciones que provee Celery son las siguientes:

- **Monitoreo**: los procesos emiten un flujo de eventos de supervisión. Las herramientas externas se encargan de indicar lo que está haciendo el clúster en tiempo real.
- **Flujos de trabajo:** los flujos de trabajos se pueden formar utilizando un conjunto de componentes primitivos, que son el lienzo, y, luego, incluir la agrupación, el encadenamiento o la fragmentación, entre otros.
- **Límites de tiempo:** se pueden controlar las tareas que se pueden ejecutar por un tiempo determinado o cuánto tiempo se puede permitir que dure una tarea en ejecución. Se pueden establecer para un proceso específicamente o para cada tipo de tarea.
- **Programación**: se pueden usar tareas periódicas para trabajos que se tengan que hacer de forma repetida, a lo largo del tiempo; es decir, se quedan programados para ejecutar en una fecha y hora determinada. Se puede utilizar las expresiones crontab para programas dichas tareas.
- **Protección contra fugas**: permite controlar las descripciones de memoria o archivos que se encuentra fuera de control de Celery. Para ello se utiliza la opción:

  --max-tasks-per-child
- **Componentes del usuario:** cada componente puede personalizar su usuario y definir los componentes adicionales.

Celery también permite la integración con varios *frameworks*, entre los cuales destacamos:

Pyramid	*pyramid_celery*
Pylons	*celery-pylons*
Flask	No necesita ninguno
web2py	*web2py-celery*
Tornado	*tornado-celery*
Tryton	*celery_tryton*

Hay que tener en cuenta que Celery es una librería que ya está bastante madura. Es exclusiva de Python y, gracias a ella, se pueden crear aplicaciones muy eficientes aprovechando toda su tecnología.

También se pueden ejecutar las tareas de forma simultánea y es capaz de obtener un rendimiento muy superior al que se consigue con otras formas de ejecución. Por otro lado, reduce la carga de rendimiento y, ejecutando parte de la funcionalidad como tareas retenidas, se puede ejecutar en el mismo servidor o en otros servidores.

Normalmente, los desarrolladores suelen usar a Celery para el envío de correos electrónicos, pero Celery también es capaz de ejecutar tareas de una gran duración.

Por ejemplo, cuando se necesita realizar tareas que tienen que pasar por los registros de las bases de datos, estas tienen que realizar varias operaciones y varias escrituras en disco. Esto puede suponer un problema, ya que el proceso puede durar bastante tiempo desde que es iniciado hasta que se termina de ejecutar. La idea es realizar todas estas operaciones y que se puedan subdividir en tareas más pequeñas. Aquí es cuando entra en juego Celery, que es capaz de acelerar todos esos procesos, al obtener las subtareas en las que se pueden dividir y ejecutarlas.

Como funciona todo de forma asíncrona, el proceso principal puede seguir corriendo sin tener que esperar a realizar todas las tareas. Es más, incluso en el caso de que estemos en un sistema distribuido, las tareas se podrían dividir entre los diferentes clústeres.

Resumen

En los sistemas distribuidos todos sus componentes se encuentran conectados en red. Esto permite la comunicación entre todos los nodos del sistema y su coordinación.

En el lenguaje de programación Python, se encuentra el módulo Celery que se utiliza para distribuir las tareas entre todos los componentes del sistema distribuido. Así se pueden ejecutar las tareas a altas velocidades, al usar toda la potencia de los nodos que componen el sistema.

Glosario

Algoritmo

Secuencia de instrucciones que son finitas y llevan a cabo una serie de procesos para poder dar repuesta a determinados problemas.

API

Conjunto de definiciones y protocolos que se utilizan para integrar un software.

Clave

Entrada a un índice que identifica un registro.

Codificación

Envío y recepción de datos tras realizarles un procesamiento para combinarlos con más códigos y operaciones.

Código

Texto desarrollado en un lenguaje de programación y que tiene que ser compilado o interpretado para poder ejecutarse.

Colección

Representa a un grupo de objetos, es el lugar donde se almacena.

Comando

Orden o instrucción que un usuario proporciona al sistema informático para que efectúe una acción determinada.

Concurrente

Proceso de coincidencia de las acciones con otros elementos del mismo sistema y el mismo momento.

Contexto

Mínimo conjunto de datos que se puede utilizar por una tarea, que debe almacenarse para poder interrumpir el proceso en algún momento adecuado.

Declaración

Forma en la que se debe llamar a los objetos, funciones y otros tipos de elementos en el código.

Desempeño

Se refiere a la productividad y calidad de los trabajos realizados.

Depuración

Proceso de identificar y corregir los errores que se producen en un programa.

Entidad

Objeto (puede ser abstracto) que recoge información que puede ser representada en un sistema de base de datos.

Excepción

Es la indicación que se produce cuando ocurre un error en el programa.

Hash

Es un algoritmo matemático que permite transformar cualquier bloque a una nueva serie de caracteres con la característica de que tendrá una longitud fija.

Identificador

Nombre que se define y que permite denotar los elementos en un código.

Índice

es la estructura de datos que permite recuperar las filas de una tabla; además es posible que indique el orden de los datos.

Librería

Conjunto de implementaciones funcionales que han sido codificadas en un lenguaje de programación.

Mapeo

Técnica de convertir los datos entre el sistema de tipos, utilizando un lenguaje de programación orientado a objetos.

Microprocesador

Circuito electrónico que se encarga de ejecutar los procesos. Es la parte principal de un ordenador.

Migración

El proceso de la transferencia de datos de un sistema a otro con la intención de cambiar el sistema de almacenamiento.

Módulo

Parte del código de un programa que tiene determinadas funciones.

Mononúcleo

Referente a que el procesador solo cuenta con un núcleo para realizar las operaciones de procesamiento.

Nodo

Es cada uno de los elementos de una lista enlazada.

Paralelismo

Forma de computación en la que se permite realizar varios cálculos de forma simultánea.

Parámetro

Es la variable que se utiliza para recibir valores de entrada en una función.

Portabilidad

Característica de un software que permite que se ejecute en diferentes plataformas.

Procesador

Es la unidad central de procesamiento, encargada de interpretar las instrucciones y procesar los datos de los programas.

Prototipo

Implementaciones que se realizan con técnicas de programación para reproducir el funcionamiento deseado y realizar sobre él las pruebas.

Puntero

Es un objeto en el cual su valor se refiere al valor almacenado en otra parte de la memoria obteniendo su dirección.

Raíz

Es el nivel superior de los directorios de un sistema de archivo.

Relé

Dispositivo electromecánico que funciona como un interruptor y es controlado por un circuito eléctrico que se puede accionar por medio de una bobina y un electroimán.

Replicación

Es el proceso de copiar y mantener actualizados los datos en los diferentes nodos del sistema.

Rol

Colección de permisos que tiene asignado cualquier usuario de un sistema.

Servicio

Se utiliza para realizar acciones en segundo plano, preferiblemente, y que no dependen del proceso principal.

Bibliografía

Cao Abad, R. (2002). *Introducción a la Simulación y a la Teoría de Colas*. Netbiblo.

Cerrada Somolinos, J. A. y Collado Machuca, M. E. (2010). *Fundamentos de programación*. Editorial Universitaria Ramón Areces.

Cuevas Álvarez, A. (2016). *Python 3*. Curso práctico. Ra-Ma.

Franch Gutiérrez, X. (2002). *Estructuras de datos: especificación, diseño e implementación*. UPC.

Liberty Vittert, M. (2021). *50 principios de la ciencia de datos*. Blume.

Moreno Pérez, J. C. (2015). *Programación orientada a objetos*. Ra-Ma.

Rodríguez Artalejo, M., González Calero, P. A. y Gómez Martín, M. A. (2011). *Estructuras de datos. Un enfoque moderno*. Editorial Complutense.

Sullivan, (2019). *Introduction to Spark SQL and DataFrames*. LinkedIn Learning.

Autor
José Miguel Acosta Martín

Universidad Internacional de Valencia