Programación de Tareas de Sistema

Python para Administradores de Sistemas

Sergio de Mingo Gil

Última revisión: 26 de septiembre de 2024

Índice general

Sobre este libro

Este libro se ha focalizado en la tarea de enseñar los conceptos básicos de la programación de pequeños programas para ser usados en el ámbito de la administración de sistemas. Sus contenidos y las exposiciones están pensadas para lectores que carezcan de formación técnica orientada a la programación o a los computadores en general. Aún así, se supone que el lector debe tener ciertas destrezas básicas en el uso de ordenadores: editar y guardar contenido en ficheros de texto, uso básico de aplicaciones, crear directorios y otras de similar dificultad todo ello en sistemas Linux.

Como lenguaje de transmisión para aprender estas habilidades se usa el lenguaje Python. Creado por Guido van Rossum durante la década de los 90 del siglo pasado y ampliamente extendido en diferentes industrias del sector tecnológico. Además de mostrar estos fundamentos básicos sobre programación, el libro se centra en su parte final en como adaptarlos al entorno laboral en el que los estudiantes del ciclo de Administración de Sistemas y Redes acabarán integrados. Al final de cada capítulo se pueden encontrar actividades propuestas para fortalecer el proceso de aprendizaje y facilitar al alumno la puesta en práctica de los contenidos desarrollados. Si se desea obtener una copia de las soluciones a estos ejercicios póngase en contacto con el autor en la dirección de correo electrónico: `sergio.demingogil@educa.madrid.org`. De igual manera puede usarse esa dirección de contacto para solicitar una copia del contenido fuente usado para construir este documento. Para terminar hay que señalar que todo el material del libro está en permanente revisión. Esto quiere decir que puede contener erratas o errores tipográficos que serán subsanados lo antes posible.

1

Introducción a Python

El computador es la primera y por ahora única máquina que el hombre ha construido sin que sea fabricada con un propósito específico. Un ordenador o computador puede utilizarse para cualquier propósito siempre y cuando seamos capaces de dotarle de un conjunto de instrucciones adecuado. Hemos de entenderla como un lienzo en blanco que podemos utilizar para cualquier objetivo. La única dificultad es que hemos de darle las instrucciones que debe llevar a cabo para completarlo de una forma precisa y sin ambigüedades. Aprender a escribir estas instrucciones forma parte de la disciplina de la programación de computadores. Una disciplina que mezcla a partes iguales creatividad, imaginación y productividad.

1.1. Lenguajes y algoritmos

La definición formal de **algoritmo** es conjunto de instrucciones o reglas definidas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades. Dado un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución. La relación que existe en informática con el concepto de algoritmo es la necesidad que tenemos de que sea el computador el que ejecute esas instrucciones por nosotros para llegar a ese estado final. Eso nos implica que las instrucciones con las que escribamos el algoritmo deben ser interpretadas por el computador. Las tenemos que escribir pensando en que debe saber como ejecutarlas.

La forma de escribir estas instrucciones ha cambiado mucho durante la historia de la informática. El catálogo de instrucciones con que podemos codificar un algoritmo y el conjunto de reglas para combinar estas, están determinadas por el **lenguaje de programación** que utilicemos. Estos lenguajes han cambiado mucho desde el origen de la informática. Los primeros lenguajes tenían todos una estructura sintática muy básica. Dicho de otra manera, teníamos que escribir programas adaptándonos a la limitada capacidad de interpretación de los computadores. Se usaban repertorios de instrucciones muy pequeños que se debían combinar en extensos programas para conseguir trazar algoritmos sencillos.

Esto cambió ha finales de la década de 1950 con la aparición de Fortran y algo más tarde COBOL. Los considerados dos primeros **lenguajes de alto nivel**. Estos lenguajes se caracterizaban por permitir usar instrucciones de alto nivel de abstracción, más sencillas de usar para los humanos que luego debían de ser traducidas a secuencias de instrucciones más sencillas de interpretar por el computador. Este proceso de adaptación se denominaba compilación. A finales de la década siguiente, entorno a 1970 aparece el lenguaje C que tendría una grandísima repercusión en la historia de la informática moderna.

1.2. El lenguaje Python

El lenguaje Python es un lenguaje de los considerados de alto nivel pero que posee una característica diferente a los mencionados en el apartado anterior. No está pensando para ser compilado. Es un lenguaje interpretado. Esto quiere decir que el código que escribamos con Python no será compilado y adaptado para que el computador lo ejecute directamente. Será leído por otro programa llamado intérprete. Este intérprete es un programa que está siendo ejecutado por el computador en un momento dado y que está leyendo mientras nuestro código fuente. El intérprete entonces hará que el computador realice las operaciones que se indican en nuestro código como si fuera el mismo el que las hiciera. En la actualidad existen muchos otros lenguajes interpretados con gran éxito en la industria del software: PHP, Javascript, Ruby, Lua, etc. Debido a su naturaleza de lenguaje interpretado, es un lenguaje ideal para programar aplicaciones y programas para que funcionen sobre diferentes arquitecturas hardware y sistemas operativos. Nuestro código será totalmente portable entre plataformas y sistemas siempre y cuando dispongamos de un intérprete de Python instalado en la máquina en la que queramos hacer funcionar nuestro programa.

Python fue creado a finales de los años ochenta del siglo pasado por Guido van Rossum mientras trabajaba en un centro de investigación en los Países Bajos[1]. El 20 de febrero de 1991, van Rossum publicó el código por primera vez una lista de correo de programadores. En enero de 1994 alcanzo la versión 1.0. En la actualidad es uno de los lenguajes de programación más extendidos en el mundo. Es usado en numerosos ámbitos y para todo tipo de funciones. Implantado en el desarrollo de aplicaciones de escritorio, servicios en red, software coorporativo, inteligencia artificial, domótica, microcontroladores, etc.

En nuestro caso particular lo vamos a estudiar como un lenguaje ideal para complementar las habilidades que aprenderéis como administradores de sistemas. Es un lenguaje sencillo y ágil que os permitirá realizar pequeños programas o *scripts* para vuestro día a día en vuestras labores de administración.

1.2.1. Entorno de desarrollo

Python es un intérprete que puede ser fácilmente instalado sobre cualquier sistema operativo comercial. En la actualidad, la mayoría de distribuciones de Linux vienen con el intérprete instalado y listo para ser usado. Por ahora usaremos Python sobre Windows pero poco a poco, a medida que avancemos en el curso empezaremos a usar Python sobre Linux.

Una vez tengamos instalado Python en nuestro sistema Windows la manera para probar

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

que la instalación es correcta será abriendo una consola de comandos o `cmd` y ejecutando el comando `python -version`. Si obtenemos una salida simlar a la que se muestra a continuación, el intérprete ha sido correctamente instalado en nuestro sistema:

```
C:\Windows\system32>python --version
Python 3.10.8
C:\Windows\system32>
```

Para ejecutar nuestro primer *script* o programa basta con que abramos el editor de texto plano de nuestro sistema, por ejemplo «Bloc de Notas» y escribimos nuestro primer programa:

```
print ("hola mundo!!")
```

Lo guardamos con el nombre `prueba.py` en cualquier directorio de nuestra carpeta personal. Ahora abrimos de nuevo la consola de comandos y nos dirigimos a este directorio del sistema. Por ejemplo yo lo he guardado en el directorio de mis «Documentos». Así que abro la consola de comandos y me dirijo a ese directorio y ejecuto el intérprete pasándole el nombre del archivo:

```
C:\Windows\system32>cd c:\users\sergio.de.mingo\documents
C:\users\sergio.de.mingo\documents>python prueba.py
Hola mundo!!
```

Como podéis ver, usando un simple editor de textos y la consola de comandos podemos escribir código Python sin más. De todas formas, el entorno de desarrollo que usaremos sobre Windows por ahora será Geany. Este sencillísimo entorno es ideal para comenzar a escribir código sin complicaciones. No es el entorno ideal para un uso profesional pero poco a poco, según avancéis en el lenguaje podéis ir cambiando a otro que os guste más.

Una vez escrito el primer programa sobre Geany bastará con que pulsemos F5 para que el intérprete ejecute nuestro programa directamente.

1.3. Datos

El pensamiento algorítmico se basa en los datos y en las instrucciones para manipularlos. Los datos son, por lo tanto, la materia prima fundamental de todo algoritmo. Un dato puede ser una palabra, un número, una frase, etc. Un programa informático, en definitiva es un conjunto de datos y de instrucciones que los manipulan. Los datos pueden tener un origen variable y distinto todos ellos: un dato puede ser introducido por teclado, leído desde un fichero del disco, tomado desde memoria principal, sondeado desde un sensor, etc. De igual manera, tras su manipulación, el dato puede tener un destino variable: impreso en pantalla, o en un papel, enviado a un fichero de disco, etc. Lo más importante de cara la programación es tener siempre clara dos cosas:

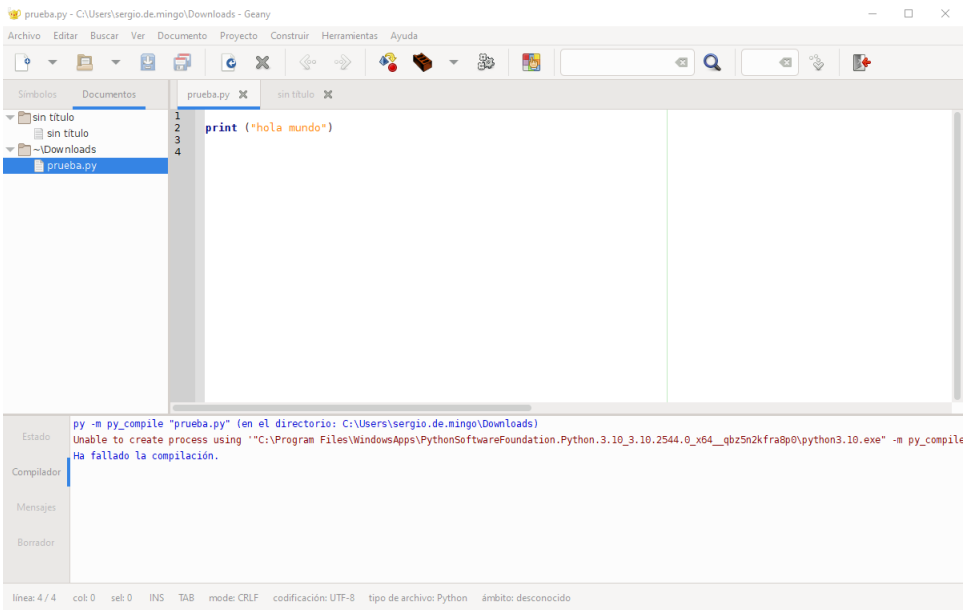

Figura 1.1: Pantalla principal de Geany

- Los datos que maneje nuestro programa se guardan en la memoria principal de nuestro computador codificado, como toda la información de la memoria en formato binario.
- Existen diferentes tipos de datos y cada uno de ellos tiene un tamaño concreto.

1.3.1. Variables y tipos

Como ya hemos mencionado estos datos a la hora de ser usados por las instrucciones del programa residen todos en memoria principal. Para encontrarlos dentro de la memoria se les asigna una etiqueta llamada **variable**. Una variable por tanto no es más que un dato que reside en memoria y a la que hemos puesto un nombre para poder utilizarla en cualquier punto de nuestro programa.

Para crear una variable en nuestro programa elegimos un nombre para ella y la damos un valor inicial. El nombre puede ser cualquier cosa pero debería ser preferiblemente descriptivo, no muy largo y sin espacios. Por ejemplo:

```
mivariable=96
```

Las variables no ocupan todas lo mismo en memoria. Las variables pueden tener diferentes **tipos** y estos suelen indicar el tamaño que cada una ocupará en la memoria principal. Ahora vamos a introducir algunos tipos básicos con los que empezaremos a trabajar pero más adelante aprenderemos a usar otros tipos algo más complejos. Los tipos básicos de Python serían:

Enteros (*Integers*) Representan números enteros (positivos y negativos).

Reales (*Floats*) Representan números reales o decimales.

Booleanos (*Booleans*) Representan los dos valores lógicos de verdad para una proposición:

verdadero (*True* o falso (*False*).

Cadenas (*Strings*) Representan cadenas de caracteres para representar palabras o frases.

Listas (*Lists o Arrays*) Representan colecciones de elementos de otros tipos. Una colección es un agrupamiento de valores de igual o de distinto tipo.

Empezaremos a trabajar con todos ellos aunque las cadenas de caracteres y las listas las estudiaremos algo más adelante en profundidad. A continuación se muestran ejemplos como declarar variables de los tipos anteriores:

```
un_entero=-45
un_real=34.6876
un_valor_de_verdad=True
una_cadena="Esto es una cadena de texto"
una_lista=[3,5,-4]
```

Un valor establecido en el propio código de programación de forma directa y estática, como los valores que hemos usado en esas variables, se llaman **valores literales**. Normalmente los datos almacenados en variables suelen cambiar y ser actualizados durante la ejecución de nuestro programa. Si sabemos a priori que un dato de nuestro código nunca va cambiar, ni debe hacerlo, podemos indicarlo poniendo su nombre todo en mayúsculas. A esto se le suele llamar **constante**. Un ejemplo puede ser:

```
NUM_MAX=80
```

1.3.2. Imprimir y tomar datos

Para mostrar el valor de una variable por pantalla podemos utilizar la instrucciones `print`. Esta instrucción volcará a pantalla el valor de la variable seguida por un salto de línea. Por ejemplo:

```
print ("Esto imprime este valor literal")
mivariable=8860
print (mivariable)
```

A través del teclado también podemos darle a nuestro programa el valor de una variable y así introducir datos de entrada para que sean manipulados en el programa. Esto lo haremos fácilmente usando la instrucción `input`. A esta instrucción le podemos pasar de forma opcional una cadena de texto que informe al usuario con el dato que esperamos. Cuando el usuario introduzca el dato y pulse la tecla *intro*, el programa continuará su ejecución y la variable asignada a `input` tomará el valor que le hayamos introducido por teclado:

```
tunombre=input("Introduce tu nombre: ")
print ("Tu nombre es:")
print (tunombre)
```

El valor devuelto por la instrucción `input` siempre será una cadena de caracteres. Aunque le pidamos un número y el usuario lo introduzca tal cual, `input` siempre lo retornará como una cadena de texto. Esto hemos de tenerlo en cuenta pues puede producir situaciones extrañas si no entendemos bien los tipos de datos explicados anteriormente. Ejecuta el código del recuadro siguiente y trata de entender lo que está ocurriendo. En un principio podemos asumir que se debería imprimir la suma aritmética de ambos números pero no es así. Lo que se está imprimiendo es la suma de las dos cadenas de texto, osea su fusión. Si queremos que los datos de los usuarios sean tratados como valores numéricos debemos de realizar una conversión, tal y como explicaremos en la sección siguiente.

```
num1=input("Número 1: ")
num2=input("Número 2: ")
print (num1+num2)
```

1.4. Operadores y conversiones

Ya hemos hablado sobre como introducir datos a nuestro programa, como almacenarlos mientras los usamos y como mostrarlos ya manipulados. Ahora vamos a ver que tipo de manipulaciones podemos hacer con esos datos. Las operaciones que podemos hacer con un dato vienen determinadas por el tipo de este. Cuando queramos operar con varios datos **es necesario que estos sean todos del mismo tipo**. En caso de que sean de tipos diferentes hemos de realizar una conversión de alguno de ellos para igualarlo al resto. Al final de este apartado veremos como hacer esto.

Para operar de forma aritmética con datos numéricos tenemos los operadores aritméticos clásicos:

Operador	Definición	Ejemplo
+	Realiza la suma entre los operandos	12 + 3 = 15
-	Realiza la resta entre los operandos	12 - 3 = 9
*	Realiza la multiplicación entre los operandos	12 * 3 = 36
/	Realiza la división entre los operandos	12 / 3 = 4
%	Realiza el resto entre los operandos	16 % 3 = 1
**	Realiza la potencia de los operandos	12 ** 3 = 1728
//	Realiza la división con resultado de número entero	18 // 5 = 3

Los operadores para los números reales serían los mismos. En caso de que se opere un número entero con uno real, el intérprete realizará una conversión automática o *casting* implícito y el resultado siempre será real. También podemos relacionar datos. La relación siempre nos devuelve un valor de verdad: `True` o `False`.

Operador	Definición	Ejemplo
>	Devuelve `True` si el operador de la izquierda es mayor que el operador de la derecha	12 > 3 devuelve `True`
<	Devuelve `True` si el operador de la derecha es mayor que el operador de la izquierda	12 < 3 devuelve `False`
==	Devuelve `True` si ambos operandos son iguales	12 == 3 devuelve `False`
>=	Devuelve `True` si el operador de la izquierda es mayor o igual que el operador de la derecha	12 >= 3 devuelve `True`
<=	Devuelve `True` si el operador de la derecha es mayor o igual que el operador de la izquierda	12 <= 3 devuelve `False`
!=	Devuelve `True` si ambos operandos no son iguales	12 != 3 devuelve `True`

Por último hablaremos de los operadores lógicos. Estos operadores los usaremos sobre todo en expresiones condicionales un poco más adelante para poder tomar decisiones en base a valores de verdad de múltiples expresiones:

and	Devuelve True si ambos operandos son True	a and b
or	Devuelve True si alguno de los operandos es True	a or b
not	Devuelve True si alguno de los operandos False	not a

Los operadores y las transformaciones con cadenas de caracteres las veremos más adelante en profundidad pero, por ahora, un operador que usaremos mucho con cadenas será el de `+`. Este operador, al igual que con datos numéricos, nos permite realizar adición de cadenas:

```
tunombre = input("Introduce tu nombre: ")
print ("Tu nombre es " + tunombre)
```

Hemos comentado con anterioridad que Python necesita que los tipos a operar sean siempre del mismo tipo. Para realizar conversiones de tipos usaremos las instrucciones: `int()` para convertir en número entero, `float()` para convertir en número real y `str()` para convertir algo a cadena. Por ejemplo, vamos a unir un número entero a una cadena para poder imprimirlas juntas en la pantalla:

```
edad=45
print ("La edad del usuario es " + str(edad) + " años")
```

Otro ejemplo de conversión puede usarse al pedir un valor numérico por teclado. Como `input()` siempre retorna una cadena de caracteres realizaremos la conversión de forma explícita:

```
semanas=input("Introduce las semanas que has trabajado: ")
dias=int(semanas) * 7
print ("Has trabajado " +str(dias)+ " días seguidos")
```

1.5. Comentarios y documentación

Los programas que realizaremos al principio de este módulo no serán de gran tamaño. Tan solo de unas pequeñas líneas pero poco a poco estos programas pueden ir creciendo en tamaño y complejidad. En este caso puede sernos de utilidad la integración de comentarios descriptivos dentro del propio código. Estos comentarios serán ignorados por el intérprete en el momento de la ejecución del programa. Para que esto ocurra debemos iniciar la línea con el carácter `#`. Cualquier línea que comience con este carácter será ignorada. De esta forma podemos incluir comentarios junto con el código que escribamos. Por ejemplo:

```
# Declaramos dos variables:
num1=9545
clave="Clave de usuario"
```

Si queremos introducir comentarios largos separados en varias líneas puede que usar `#` al comienzo de cada una de ellas sea molesto. En Python hay una segunda forma de comentar párrafos multilínea. Usando tres comillas dobles o tres comillas simples al comienzo y al final de cada comentario largo. Por ejemplo:

```
'''
Este es un comentario largo en Python usando
varias líneas y sin necesidad de meter un prefijo
al comienzo de cada una de estas.

Puede ser más cómodo para comentar varias líneas.
'''
```

Siempre que consideremos que un fragmento de nuestro programa no queda suficientemente claro podemos incluir comentarios de forma similar. Es importante señalar que en el futuro puede que no solo tú leas tus programas. Es muy posible que parte del equipo en el que trabajas compartáis programas que pueden estar hechos incluso por compañeros que ya no están. Dejar tus programas bien documentados es una práctica sana y muy productiva que ayudará a todo aquel que llegue más tarde a tu código a entenderlo y mejorarlo si es necesario. Ahora que ya sabemos como introducir comentarios en nuestro código sería importante seguir un convenio de documentación siempre de nuestros programas. Para reconocer siempre un programa al abrirlo y recordar su funcionalidad básica es muy recomendable empezar siempre con una cabecera similar a la que se muestra a continuación:

```
# Breve descripción del programa y de su fucionalidad. Puede ser una sola línea
# o un párrafo.
# Autor: Nombre del autor (direccion@decorreo.com)
# Fecha: 25/08/2024
# Temas pendientes:
# - Errores encontrados o asuntos que todavía están por resolver
# - ...
```

Actividades

1. Crea un programa en Python donde declares una variable de tipo string y otra de tipo entero e imprime las dos en pantalla.

2. Escribe un programa que declare una variable llamada `lado` donde se indique el lado de un cuadrado y te calcule su área. Asigna un valor entero a esta variable y prueba tu programa varias veces con diferentes valores.

3. Haz un programa similar al anterior pero calculando ahora el área de una circunferencia y usando una variable llamada `radio`

4. Crea dos variables llamadas `x` e `y`, asígnales dos valores enteros a tu elección e imprime el resultado resultante al resolver la función matemática:

$$\frac{(3x^3 + 2x^2)}{(3y^3 - 2y^2)}$$

5. Programa un pequeño script en Python que posea una variable entera llamada `dias` y calcule los meses y semanas de esa cantidad de días. Puedes suponer que todos los meses tienen 31 días.

6. Escribe un programa que pida tres palabras e imprima un mensaje en una sola línea donde se enumeren las tres palabras separadas por comas.

7. Escribe un programa similar al del ejercicio 5 pero ahora introduciendo el valor de los días por teclado y ofrenciendo la salida en una sola línea. Por ejemplo: `Son 2 meses y 3 semanas`.

8. Escribe un programa para convertir un valor introducido por teclado equivalente a un peso en kilos a un peso en gramos.

2

Estructuras de control

Ahora que ya conocemos las instrucciones y estructura básica de un programa en Python vamos a comenzar a estudiar las diferentes formas que tenemos de controlar y modificar la ejecución de estas instrucciones. El computador no suele leer los programas como nosotros hacemos con una novela: de principio a fin. La lectura y ejecución de un programa suele implicar saltos, bifurcaciones y repeticiones de fragmentos a elección nuestra. En este capítulo estudiaremos las diferentes formas que tenemos de modificar el flujo principal de instrucciones para realizar estas variaciones.

2.1. Instrucciones de selección

En este tema vamos a estudiar un mecanismo básico en la programación sobre cualquier lenguaje: la estructura de selección. Esta estructura es común a la mayoría de lenguajes de programación y nos permite hacer una bifurcación en el código en base a una condición que se evaluará como cierta o falsa.

La instrucción básica para hacer esta bifurcación es `if`. Esta instrucción va acompañada de una condición o expresión lógica cuyo valor será evaluado. Opcionalmente podemos acompañar a esta instrucción de otra llamada `else`. Si la expresión lógica se evalúa a cierto se ejecutará las instrucciones bajo el `if`. En caso contrario se ejecutarán las instrucciones bajo el `else`. Veamos un ejemplo en el que declaramos dos variables enteras y queremos mostrar un mensaje en el que se imprima cual de las dos tiene el mayor valor.

```
a=9
b=-1

if (a<b):
   print ("El número mayor es :" + str(a))
else:
   print ("El número mayor es :" + str(b))
print ("FIN)
```

Es muy importante notar la tabulación de las dos ramas de código. Tanto bajo la instrucción `if` como bajo la instrucción `else` podemos incluir tantas instrucciones como queramos y todas ellas pertenecerán a ese mismo bloque **siempre y cuando estén al mismo nivel de tabulación**. En Python respetar el nivel de tabulación es crucial para que el intérprete sepa si dos instrucciones son del mismo bloque o de bloques diferentes. Los bloques de instrucciones siempre comienzan con el carácter `:` y se forman por todas las instrucciones que tengan el mismo nivel de tabulación.

En el ejemplo anterior la última instrucción que imprime `FIN` se ejecutará siempre, sea cual sea la rama elegida en la instrucción condicional ¿por qué? Porque esta está fuera del bloque del `if` y fuera del bloque del `else`. Probamos a reescribir el ejemplo anterior e iguala la tabulación de las dos últimas instrucciones para que ambas estén dentro del bloque del `else`. Ahora solo verás el mensaje `FIN` si resulta que `b` es menor que `a`.

```
a=9
b=-1

if (a<b):
   print ("El número mayor es :" + str(a))
else:
   print ("El número mayor es :" + str(b))
   print ("FIN)
```

Dentro de un bloque de instrucciones de Python podemos encontrar cualquier otra instrucción. Dentro de un bloque asociado a una instrucción condicional puede haber otra instrucción condicional perfectamente. A esto se le suele denominar *ifs anidados*. Por ejemplo:

```
a=9
b=-1

if (a<b):
   print ("El número mayor es :" + str(a))
   if ((b<0) or (a<0)):
      print ("Y uno de los dos números es negativo")
else:
   print ("El número mayor es :" + str(b))
print ("FIN)
```

En este caso vemos como dentro de la rama de la instrucción `if` tenemos otra instrucción

condicional que se evalúa a cierto si alguno de los números es negativo. Este ejemplo también nos permite estudiar la expresión condicional de este segundo `if`. Esta expresión se ha construido de forma algo más compleja usando un operador lógico `or`. Este operador compara dos valores de verdad situados a su izquierda y derecha y retorna un valor resultante:

```
(b<0)   or   (a<0)

true         true     => true
true         false    => true
false        true     => true
false        false    => false
```

Como vemos en el recuadro anterior, con que una de las expresiones que comparan cada variable con `0` sea cierta, la expresión final será cierta. Por último, en este mismo `if` que estamos estudiando se pone de manifiesto otra característica de estas instrucciones y es la opcionalidad de la instrucción `else`. En este caso, no tenemos ninguna rama asignada a esta instrucción por lo que si la expresión condicional se evalúa falso (por ejemplo porque ambos números son positivos) no se mostrará ningún mensaje indicando este aspecto.

Otra estructura que se deriva de la instrucción `if` es la que permita evaluar diferentes condiciones en lugar de una sola. En Python esta estructura suele denominarse instrucción `elif` y tiene una forma similar a la del siguiente ejemplo:

```
producto=input("Introduce un alimento: ")

if (producto == "naranja"):
   print ("Es una fruta")
elif (producto == "tomate"):
   print ("Es una hortaliza")
elif (producto == "pan"):
   print ("Es un derivado del cereal")
elif (producto == "queso"):
   print ("Es un lácteo")
else:
   print ("No se lo que es")
```

Como puedes ver, esta variación permite múltiples bloques en base a diferentes expresiones que se evalúan de forma independiente. Una vez se evalúe una a cierto, se ejecuta el bloque asociado a ella y se continúa fuera de la instrucción. Nunca se dará que se ejecuten varios bloques asociados a diferentes condiciones. Además, de forma opcional podemos añadir un último bloque asociado a un `else` que se ejecutará siempre que todas las condiciones anteriores sean falsas.

2.2. Instrucciones iterativas o repetitivas

Otro tipo de instrucciones fundamentales en el mundo de la programación son las instrucciones repetitivas o iterativas. Estas instrucciones se basan en la repetición de un bloque de instrucciones mientras se evalúa una condición. La primera instrucción de este tipo es `while`.

Esta instrucción se construye en base a una condición que repite el bloque de instrucciones mientras la condición sea cierta. Veamos un ejemplo:

```
x=5
while (x<100):
    x=x*2
    print(x)
print ("FIN")
```

Este código imprimirá 10, 20, 40, 80, 160 y por último la palabra `FIN`. El bloque de código asociado a la instrucción `while` se repite mientras la condición `x<100` sea cierta. En el momento en que la variable `x` vale 160 esta condición se evalúa a falsa y el bucle termina continuando el programa por la siguiente instrucción tras el.

Otra instrucción repetitiva fundamental es `for`. Esta instrucción utiliza un tipo de datos del que hablaremos algo más adelante: las listas. De las listas solo hemos dicho hasta ahora que son colecciones de elementos o valores y con eso nos basta por ahora para entender su uso en la instrucción `for`. Esta instrucción utiliza una variable «pivote» cuyo valor se va actualizando entre los que tenga la lista. Vamos a ver un ejemplo sencillo en el que creamos una lista con varias palabras predefinidas:

```
palabras = ["perro", "gato", "pájaro", "pez"]

for animal in palabras:
    print ("Yo en casa tengo un "+animal)
```

La instrucción `for` toma la variable `animal` y en cada repetición del bloque de instrucciones esta variable va tomando un valor diferente de la lista. Por lo tanto, una instrucción `for` repetirá su bloque de instrucciones tantas veces como elementos haya en la lista y además, la variable de control pivotará su valor entre todos estos valores.

Un uso muy habitual de la instrucción `for` es junto con la instrucción `range()`. Esta instrucción devuelve una lista de valores enteros y que puede usarse para hacer pivotar la variable de un `for` tantas veces como queramos. Por ejemplo, si queremos repetir un bloque de instrucciones 100 veces no es necesario que creemos de forma literal una lista de 100 números. Podemos en su lugar usar `range` tal y como se ve en el ejemplo:

```
for contador in range(100):
    print ("El valor de contador es "+str(contador))
```

Cuando el intérprete llega a la instrucción `range(100)` la sustituye por una lista de 100 elementos, desde el 0 hasta el 99. De ahí que la ejecución del código anterior nos devuelva algo parecido a:

```
El valor de contador es 0
El valor de contador es 1
El valor de contador es 2
El valor de contador es 3
...
El valor de contador es 98
El valor de contador es 99
```

Antes de terminar hablaremos de la instrucción `break`. Esta instrucción nos permite romper un bloque de instrucciones asociadas a una instrucción repetitiva de tipo `while` o `for` y continuar la ejecución por la siguiente instrucción tras el bloque. Normalmente la condición de salida del bucle suele ser la propia que acompaña a la instrucción `while` o `for` pero, a veces, también podemos encontrar cómodo incluir una segunda condición de salida forzada usando una instrucción `break` dentro del propio bucle.

Este tipo de rupturas suelen ser útiles cuando nos interesa repetir un bucle indefinidamente y dentro de el evaluar las condiciones de salida. Veamos un ejemplo de este tipo de bucles:

```
while True:
    numero = int(input("Introduce un número: "))
    print (numero)
    if (numero % 2) == 0:
       break
print ("Fin del programa")
```

Este programa repetirá el bloque del bucle indefinidamente mientras no introduzcamos un número par. El bucle es un tipo de los llamados infinitos porque su condición siempre se cumplirá. Ya que siempre será cierta claramente. Es por ello, que la única forma de que el programa avance en algún momento es evaluando alguna otra condición dentro del bucle y creando una condición de salida interior con la ayuda de `break`.

2.3. Control de excepciones y errores

A veces un programa puede entrar por un camino o rama por la que no queremos que continúe su ejecución. Esto puede deberse a que hemos creado una condición de salida a priori y, por lo que sea, esta se ha cumplido dentro de la ejecución de nuestro programa. En el futuro veremos otras maneras pero por ahora, si queremos que un programa termine su ejecución podemos usar la instrucción `exit()`.

Imaginad que hemos pedido al usuario que introduzca una frase y este, al serle pedida por teclado se limita a pulsar *intro* introduciendo una frase o cadena vacía. En caso de que esto ocurra imprimiremos un mensaje de error para que el lo sepa y abortaremos el programa:

```
frase = input("Introduce una frase: ")

if frase == "":
   print ("Error: La frase no puede estar vacía")
   exit()

print ("El programa puede continuar ...")
```

A veces pueden ocurrir situaciones inesperadas que no hemos controlado al detalle. Estas situaciones se denominan **excepciones**. Una excepción, si no es capturada, suele producir que el intérprete aborte la ejecución del programa. Por ejemplo una división por cero es una excepción muy clásica. Pero veamos otro ejemplo que se suele dar con más frecuencia. Tenemos un programa que pide al usuario que introduzca un número por teclado. Usamos la instrucción `input()` para leerlo del teclado y lo convertimos a entero tal y como hemos visto anteriormente:

```
numero = int(input("Introduce un número: "))
```

La instrucción `input()` siempre retorna una cadena de caracteres que la instrucción `int()` convierte a número entero. ¿Qué ocurre si el usuario introduce una cadena que no puede ser convertida a número entero? Por ejemplo escribimos la cadena `bla` y pulsamos la tecla intro. El intérprete aborta nuestro programa y nos muestra en consola el siguiente error:

```
Traceback (most recent call last):
  File "<string>", line 17, in __PYTHON_EL_eval
  File "/tmp/prueba.py", line 3, in <module>
    numero = int(input("Introduce un número: "))
ValueError: invalid literal for int() with base 10: 'bla'
```

Se ha producido una excepción y no se ha capturado. Capturar una excepción indica al intérprete como debe actuar ante ella. La información que nos muestra además el intérprete nos ayuda a localizar el origen de la excepción. Se nos indica el archivo y la línea donde se ha producido, en este caso en la línea 3 de `/tmp/prueba.py`. Por último se nos dice que tipo de excepción se ha producido. En este caso es una excepción del tipo `ValueError` y la descripción de esta, donde se indica que se ha introducido un valor literal inválido para `int()`.

```
cantidad=6000
suma=0
try:
    suma=int(input("Introduce la cantidad a sumar: "))
except:
    print ("Cantidad mal introducida. Se sumará 0")

cantidad = cantidad + suma
print (cantidad)
```

Para capturar una excepción usaremos la instrucción `try-except`. Esta instrucción se forma básicamente uniendo dos bloques. En el primero es donde sospechamos que puede producirse la excepción y el segundo indicamos lo que debe ejecutar el intérprete en caso de que se produzca. Vamos a imaginar un ejemplo en el que nuestro programa tiene declarada previamente una variable numérica llamada `cantidad`. Queremos pedir al usuario que introduzca un valor por teclado que sumaremos a esta `cantidad`. En caso de que el usuario nos introduzca un valor no numérico, capturaremos la excepción y sumaremos un 0 a la cantidad para que el programa continúe. En el primer bloque, el del `try` agrupamos todas las instrucciones donde sospechamos que pueda haber una excepción. Si no la hay, el bloque terminará y se continuará el programa ignorando el bloque del `except`. En el caso de que la haya, el intérprete salta directamente al bloque de `except` donde informamos al intérprete de que hacer. Terminado este, se continúa el programa de forma normal.

Actividades

1. Escribe un programa que pida dos números por teclado y te indique si el segundo es divisor del primero.

2. Escribe un programa que pida dos palabras por teclado y te indique si son iguales o no.

3. Escribe un programa que te diga si un número es primo o no (un número primo solo tiene como divisores a 1 y a el mismo).

4. Crea un programa que muestre los 10 primeros múltiplos de un número introducido por teclado.

5. Queremos calcular el salario final de un empleado usando un pequeño script. Este script debe solicitar primeramente el salario base. Luego añadirá un bonificador del 10 % por productividad del año, le restará 15€ debido al seguro dental y por último calculará el complemento de responsabilidad que se sumará a lo calculado. Este complemento de ser del 5 % del base para un empleado «junior», del 10 % para uno de tipo «senior», un 15 % para uno de tipo «master» y un 25 % para uno de tipo «exclusive».

6. Haz un programa que pida un número entero por teclado. Comprueba que el número entero no tiene más de tres cifras. En caso contrario aborta el programa mostrando un error. En caso correcto, ve pidiendo números al usuario para ver si es capaz de adivinarlo. Cada vez que el usuario introduzca un número el sistema le dirá si el número que busca es mayor o menor. El programa terminará cuando el usuario encuentre el número buscado. Supón siempre que el usuario te introducirá números bien formados.

7. Modifica el programa del ejercicio 6 teniendo en cuenta que el usuario pude equivocarse a la hora de introducir los números y controla que el programa no se interrumpa en ese caso.

3

Funciones y módulos

El objetivo de este capítulo es entender otro mecanismo fundamental de la programación: la modularización. Este concepto permite al programador ir construyendo su programa en base a piezas o bloques que luego podrá reutilizar en diferentes partes o incluso recombinar para formar otros bloques nuevos. De esta forma, programar comienza a parecerse a un clásico juego de construcción donde nosotros mismos crearemos los bloques que luego usaremos para levantar el resto de la figura.

3.1. Funciones

Cuando el número de instrucciones aumenta en nuestro programa podemos tener la sensación que este se vuelve más complicado de leer. Además, a medida que lo vamos escribiendo puedo que haya momentos en que necesitemos volver a escribir patrones de instrucciones que ya habíamos escrito anteriormente. Para este tipo de situaciones se han creado en el mundo de la programación las subrutinas. Estas no son más que bloques de instrucciones a los que asignamos un nombre. A partir de ese momento, podemos referirnos a ellas a través de este nombre. En Python estas subrutinas se denominan **funciones**.

Una función en Python es algo muy parecido en concepto a una función matemática. Tiene un nombre por el que nos referiremos a ella, unos valores de entrada y un valor de salida. Bajo su nombre, escribiremos un bloque de instrucciones que serán el cuerpo de la función. Estas instrucciones normalmente manipulan los datos de entrada y construyen el dato de salida. Veamos un ejemplo sencillo de función. Necesitamos sumar el área de varios polígonos y vamos a crear una función inicial para calcular el área de un círculo.

```
def area_circulo(radio):
    area = 3.14 * radio^2
    return area
```

Lo primero que vemos es que definimos la función usando la instrucción `def` seguida del

nombre que queremos que esta tenga. Luego tras el nombre se colocan los datos de entrada de la función entre paréntesis y, en el caso de que haya varios, separados por comas. Esta función toma un dato de entrada al que llamaremos `radio` y con el, calcularemos el área de cualquier círculo (recuerda que el área del círculo es $\pi * r^2$). Bien, ahora calculamos el área en la variable `area`. Pero esta variable, al estar definida dentro de la función solo será visible dentro de ella. ¿Como podemos hacerla visible fuera? Muy sencillo, haremos que la función retorne el valor cuando sea invocada (ahora entenderemos que quiere decir esto) usando la instrucción `return`. Es muy importante destacar que el bloque de instrucciones de una función se ha definido teniendo en cuenta las reglas de justificación que se explicaron anteriormente. **Si no respetamos la justificación, Python no entenderá que una instrucción pertenece a la función o no**. Bien ahora ya tenemos definida la función así que, vamos a usarla para calcular las áreas de tres círculos de radios 3,10 y 50 respectivamente. Veamos el código completo del programa:

```
# Definimos la función:

def area_circulo(radio):
    area = 3.14 * radio^2
    return area


# Ahora la invocamos con distintos radios:

area_3 = area_circulo(3)
area_10 = area_circulo(10)
area_50 = area_circulo(50)

print (area_3, area_10, area_50)
```

Vemos que el programa termina invocando tres veces a la misma función. Cada vez que se la ha invocado la función ha recibido un parámetro de entrada diferente y por lo tanto ha retornado un valor resultado diferente.

Los parámetros de entrada de una función pueden ser varios y además de diferentes tipos (enteros, reales, cadenas, etc). Vamos a reescribir ahora nuestra función anterior para que calcule varios tipos de áreas de diferentes polígonos: cuadrados y círculos. La función ahora se llamará `area` y recibirá dos parámetros de entrada: una cadena con el nombre del tipo de polígono y el valor del lado o radio de este. Veamos un ejemplo completo donde la definimos y la usamos:

```
def area(tipo,radio_lado):
    if (tipo == "cuadrado"):
        return radio_lado**2

    if (tipo == "circulo"):
        return 3.14 * (radio_lado**2)


area_cuad=area("cuadrado",10)
area_cir=area("circulo",30)
area_mal=area("no existe", 12)

print (str(area_cuad),str(area_cir),str(area_mal))
```

¿Qué ocurre en la tercera invocación? Aquí hemos invocado una función y esta ha salido sin retornar ningún valor efectivo. En este caso se dice que la variable `area_mal` ha quedado sin inicializar y de ahí que cuando intentemos imprimirla nos ofrece un valor llamado `None`. Este tipo de datos de Python nos informa de un valor que todavía no ha sido definido o inicializado.

3.1.1. Parámetros variables y valores por defecto

Podemos definir funciones con un número mayor de parámetros que los que luego encontremos en la invocación de esta. Vamos a imaginar que queremos ahora crear una función para calcular el área de cuadrados o rectángulos. El área de ambos se calcula de igual manera, multiplicando dos lados, pero en el caso del cuadrado estos son iguales, por lo que será innecesario introducir dos lados de igual tamaño. Queremos una función a la que haya veces que le introduzcamos un solo parámetro (el caso del cuadrado) y retorne este lado multiplicado por si mismo y otras veces la introduzcamos dos parámetros correspondientes a la base y a la altura del rectángulo. Veamos como haríamos esta función:

```
def areacuad(base,altura=None):
    if altura==None:
        return base*base
    else:
        return base*altura

print (areacuad(5))
print (areacuad(5,3))
```

Lo primero que vemos es que en la definición de la función encontramos dos parámetros pero el parámetro `altura` tiene asociado un valor por defecto. Este es el valor que tendrá este parámetro si en la invocación de la función no definimos otro. Vamos ahora al cuerpo de la función donde lo que hacemos es comprobar el valor de `altura`. En el caso de que este valor sea `None` indicará que el invocador de la función solo la llamó con un parámetro (es el caso de la primera invocación del ejemplo) y en ese caso el área se calculará multiplicando el lado `base` por si mismo. En caso de que `altura` tenga un valor distinto a `None` querrá decir que el invocador usó los dos parámetros de entrada en la invocación y en ese caso se

entrará en la rama del `else` para retornar el valor del área asociada a un rectángulo.

3.1.2. Comentarios en funciones

Utilizar comentarios en funciones puede ser útil para describir de forma breve sus detalles: el tipo de los parámetros de entrada, el tipo de dato que devuelve, etc. Esto lo podemos hacer usando lo conocido por los comentarios de temas anteriores.

Además Python incorpora un mecanismo interno para organizar estos comentarios y asociarlos fácilmente a las funciones. Este mecanismo se basa en los llamados `docstring`. Para introducir un `docstring` en una función basta con que escribamos un comentario multilínea bajo su declaración. Este comentario además debe tener la siguiente estructura:

- La primera línea del comentario debe ser una sola línea con la descripción. Comienza en mayúscula y termina siempre en punto.
- Si existen más líneas, la segunda deberá ser un espacio en blanco, de tal forma que sea fácil distinguir entre la descripción y el resto del comentario.
- Las siguientes líneas deben ser uno o más párrafos que describan las convenciones de llamada a la función.

```
def areacuad(base,altura=None):
    '''
    Esta función retorna el área de un cuadrado o un rectángulo.

    Parámetros:
        base (int): Es la base del cuadrado o el rectángulo
        altura (int): Es la altura del rectángulo. Para cuadrados se omite.
    '''

    if altura==None:
        return base*base
    else:
        return base*altura
```

En el recuadro anterior se ve con detalle como hemos usado un `docstring` para documentar la función `areacuad`. En dicho comentario hemos descrito la función en general y los parámetros que recibe. A la hora de procesar el fichero, el intérprete detectará el `docstring` y lo insertará en su espacio de memoria asignado por el sistema operativo. Esto puede ser útil en caso de que estemos usando la consola interactiva de Python. En ese caso, invocando a la función `help()` y pasándole el nombre de nuestra función, nos mostrará en pantalla la descripción de esa función que hemos escrito en el `docstring`.

```
>>>
>>> help(areacuad)
Help on function areacuad in module __main__:

areacuad(base, altura=None)
    Esta función retorna el área de un cuadrado o un rectángulo.

    Parámetros:
        base (int): Es la base del cuadrado o el rectángulo
        altura (int): Es la altura del rectángulo. Para cuadrados se omite.

>>>
```

3.2. Funciones como parámetros

Una de las propiedades que convierte a Python en un lenguaje de altísimo potencial es que sus funciones son *elementos de primer orden*. Esta característica implica ciertos conceptos sobre teoría de los lenguajes que se escapan a este libro pero, resumiendo mucho podríamos concluir diciendo que, en Python, las funciones pueden ser pasadas como parámetros a otras funciones o incluso devueltas como valores usando `return`. En niveles básicos de Python esta capacidad puede ser vista como poco útil e incluso ignorada pero aún así, es recomendable no pasar de largo y conocerla aunque sea de forma superficial.

Vamos con el primer uso de esta propiedad. ¿Qué significa que una función puede ser pasada como parámetro a otra? Pues justo eso. Imaginemos que queremos definir una función llamada `operar()` que realice operaciones de cualquier tipo sobre listas de datos. Esta definición es tremendamente genérica ¿Cómo podríamos implementarla? Lo que haremos será una función que reciba por parámetro la lista de datos sobre la que queremos que trabaje y la operación que queramos que aplique a cada dato. Veamos un ejemplo de esto:

```
def operar(func, datos):
    for n in datos:
        print(func(n))
```

Hemos definido la función `operar` que lo único que hace es iterar entre los elementos de la lista `datos` y ejecutar sobre cada uno de ellos la función `func`. Esta función irá recorriendo la lista y asignando a la variable `n` cada uno de sus valores. Vemos que en cada vuelta del bucle se ejecuta la función `func` pasándole el valor almacenado en `n` e imprimiéndolo. Pero ¿donde está definida la función `func`? Esta función no está definida todavía y realmente no tiene porqué ser una sola función. Veamos ahora un ejemplo completo. En el siguiente recuadro hemos definido dos funciones que realizan el cuadrado y la raíz cuadrada sobre un número que le pasamos por parámetro. Ambas funcionen retornan el resultado. También hemos incluido la función definida anteriormente llamada `operar`. Para entender el ejemplo completo hemos de ir a las últimas líneas del recuadro. Hemos usado la misma función `operar` para calcular tanto la secuencia de cuadrados como la secuencia de raíces cuadradas. Esto ha sido gracias a que hemos invocado dos veces esta función, pasándole cada vez la función que queríamos que ejecutara junto con los datos sobre los que queríamos operar.

```
def cuadrado(x):
    return x ** 2

def raiz_cuadrada(x):
    return x ** (1/2)

def operar(func, datos):
    for n in datos:
        print(func(n))


operar(cuadrado,[4,5,6,7])
operar(raiz_cuadrada,[4,25,36,49])
```

3.3. Módulos

A medida que tu programa crezca, quizás quieras separarlo en varios archivos para que el mantenimiento sea más sencillo. Quizás también quieras usar una función útil que has escrito en distintos programas sin copiar su definición en cada programa. Es posible que ni siquiera hayas sido tu el que ha escrito esa función. Esta haya sido escrita por otro y tu quieras utilizarla en tu código directamente. Para todo esto, Python incorpora el concepto de módulo. **Un módulo es un fichero conteniendo definiciones y declaraciones de Python**. En esta sección aprenderemos tanto a crear nuestros propios módulos como a utilizar módulos de otros.

3.3.1. Creación de módulos propios

La creación de nuestro propios módulos es útil para organizar nuestro código. Si nuestro programa comienza a crecer y queremos separar parte de las funciones que tienen relación en diferentes archivos o reutilizarlas el otros programas, puede ser útil crear nuestros propios módulos. Para crear un módulo simplemente dejaremos las funciones que allí queramos meter en un fichero separado. El nombre de archivo será el nombre del módulo con el sufijo `.py` agregado. Vamos a crear un módulo de ejemplo y a utilizarlo en otro programa Python. Creamos un fichero llamado `areas.py` en el que escribimos varias funciones que calculan el área de diferentes polígonos.

```
def areacuad(base,altura):
    ...

def areatrian(base,altura):
    ...

def areacir(radio):
    ...
```

Ahora tenemos en otro fichero, código en el que queremos utilizar funciones de nuestro

módulo para operar con áreas de diferentes polígonos. Para utilizar nuestro anterior módulo en este nuevo programa hemos de indicarlo al comienzo usando la cláusula `import` seguido del nombre del módulo. Luego podemos usar las funciones del módulo añadiendo el nombre del módulo como prefijo de estas. Veamos un ejemplo de programa en el que se usan las funciones del módulo:

```
import areas

# Ahora puedo usar las funciones del módulo
# en este programa directamente

cuadrado1 = areas.areacuad(7)
rect1 = areas.areacuad(9,4)
rect2 = areas.areacuad(10,40)
```

Si queremos evitar poner en todas las invocaciones el nombre del módulo podemos usar la cláusula `from` antes del `import` para indicar que nombres de funciones queremos importar del módulo. En esta cláusula, además de indicar nombres sueltos de funciones en el caso de solo querer importar estas también podemos indicar que queremos importar todas con un `*`. Así quedaría el ejemplo anterior usando esta clausula.

```
from areas import *

cuadrado1 = areacuad(7)
rect1 = areacuad(9,4)
```

Tanto el módulo `areas.py` como el fichero que lo utiliza deberían estar situados en el mismo directorio. Realmente esto no es necesario si se configura el intérprete pero esto queda fuera del contenido de este libro.

3.3.2. Uso de módulos de terceros

En Python existen muchísimos módulos ya construidos y listos para ser importados en tu código junto con la instalación básica del intérprete. En la página `https://docs.python.org/3/py-modindex.html` puedes encontrar todo el índice de estos.

En caso de querer importar un módulo que no esté con la distribución básica hemos de instalarlo en nuestro sistema. Instalar un módulo de un tercero es algo complicado pero por fortuna tenemos la herramienta `pip`. Para instalar el nombre de un nuevo módulo usaremos esta herramienta de la siguiente manera:

```
python3 -m pip install "NombreDelModulo"
```

Actividades

1. Crea una función que reciba una frase y cuente el número de palabras que tiene.

2. Crea una función que reciba una lista de números positivos y retorne el mayor.

3. Crea una función que reciba un número y calcule su factorial.

4. Crea dos funciones que trabajen sobre listas de números. Una de ellas debe calcular el sumatorio de toda la lista, la otra su media aritmética[1].

5. Haz una función que calcule la raíz cuadrada de un número[2]. Haz después otra función que reciba cuatro números correspondientes a las coordenadas de dos puntos en el plano: $x0$, $y0$, $x1$, $y1$ y calcule la distancia entre estos dos puntos. La función matemática para calcular la distancia entre dos puntos del plano es:

$$distancia = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

6. Utiliza la función `sqrt()` del módulo `math` que viene incluido en la distribución de Python. Importa el módulo y reescribe el código anterior usando dicha función.

7. Utiliza las funciones programadas en los ejercicios 2, 3 y 4 y construye un módulo llamado `numeros.py`. Ahora crea un pequeño programa en un fichero aparte que importe ese módulo y use sus funciones a modo de prueba.

[1] Para saber el número de elementos de una lista puedes usar la función len()
[2] Para calcular la raíz cuadrada de un número basta con elevarlo a 1/2

4

Listas, cadenas y otros tipos

En este tema vamos a profundizar en algunos tipos de datos que ya hemos estudiado superficialmente pero en los que todavía no nos hemos parado lo suficiente como las listas y las cadenas. Además conoceremos nuevos tipos de datos fundamentales en Python como las tuplas y los diccionarios. Las tuplas son básicamente similares a las listas pero carecen de su propiedad de «mutabilidad» lo que las hace especialmente útiles en muchos contextos. Los diccionarios son estructuras que permite el agrupamiento de datos pero que ofrecen una búsqueda de estos muy rápida.

4.1. Listas

Durante los temas pasados hemos utilizado las listas pero sin profundizar en su potencial. Una lista, como ya sabemos, es una colección de valores que pueden ser del mismo tipo o de tipos diferentes. Para declarar una lista usaremos los corchetes y dentro de ellos indicaremos los valores que queremos que contenga separados por comas. Para acceder a los elementos de una lista usaremos también el nombre de esta seguido de corchetes y entre ellos indicaremos el índice del valor de la lista al que queremos acceder. **Es muy importante recordad que los valores de la lista siempre empiezan en 0**. Esto quiere decir que el primer valor será el índice 0, el segundo el índice 1, el tercero el 2, y así sucesivamente. Veamos un ejemplo de declaración y acceso a una lista:

```
milista = ["ana", "sergio", "luis", "marta", "maría", "juan"]

print(milista[2]) # imrpime "luis"

milista[2]="antonio"
print(milista[2]) # ahora se imprime "antonio"
```

En este caso hemos declarado una lista y tras ello hemos accedido a su tercer elemento usando `milista[2]` para pasárselo a `print`. Luego hemos vuelto a acceder a esta posición

para actualizar su contenido y hemos vuelto a imprimirla por pantalla.

Para conocer el número de elementos de una lista podemos usar la función `len()`. También puede ser muy útil acceder a una lista desde su cola o final. Para ello podemos usar el índice precedido del signo negativo. Por ejemplo si accedemos a `milista[-1]` estaremos accediendo a la cadena de texto `juan`. Con `milista[-2]` a `maria` y así sucesivamente. Una última posibilidad para trabajar con listas es crear «rebanadas» o sublistas. Para crear una sublista hemos de usar los corchetes e indicar el rango de elementos que queremos abarcar usando el separador ':'. Por ejemplo, vamos a tomar una sublista que no contenga ni el primer ni el último elemento de la lista anterior:

```
sublista1 = milista[1:5]
print(sublista1)

sublista2 = milista[1:len(milista)-1]
print(sublista2)
```

Ambas sublistas son similares pero de la primera forma tenemos que saber *a priori* el número de elementos de la lista mientras que de la segunda forma funcionará siempre, sea cual sea el número de elementos que tenga.

4.1.1. Map y filter

En Python existe una forma de unir las listas con la teoría sobre funciones y paso de parámetros de funciones que vimos en temas anteriores. Tenemos varias funciones predefinidas que trabajan sobre listas y que, además, reciben una función que harán funcionar sobre cada elemento de la lista. Empecemos con la más sencilla, la función `map`. Esta función sirve para ejecutar sobre cada elemento de una lista una función que ya tengamos programada. Imaginemos una lista similar a la que teníamos en la sección anterior y la queremos recorrer e imprimir sus nombres entre caracteres #. Usaremos la función `encierra` para imprimir una palabra cualquiera entre estos caracteres y luego usaremos `map` para ejecutar esta función sobre todos los elementos de la lista:

```
def encierra(p):
    return "#"+p+"#"

nuevalista = list(map(encierra, milista))
print(nuevalista)
```

Con la función `encierra` no hay ninguna duda pero ¿qué esconde la penúltima línea del recuadro? Empecemos por `map`. Esta función recibe dos parámetros que son la lista sobre que la que va a trabajar y la función que hará ejecutar sobre cada elemento de la lista. La función `map` irá recorriendo la lista y aplicando `encierra` a cada elemento y guardando el contenido resultante en cada posición. Por último, como `map` no retorna una lista sino una estructura llamada *iterador* la convertimos en lista usando la función `list()`. De esta forma obtenemos una nueva lista. Es posible que estés pensando en que podríamos haber hecho esto mismo sin usar `map` y estás en lo cierto. Un bucle recorriendo la lista y ejecutando

`encierra` en cada elemento haría básicamente lo mismo. Usar o no este tipo de funciones es decisión tuya pero, normalmente, suele ayudar a crear código más limpio y fácil de leer.

Otra función de utilidad que trabaja de forma similar con listas es `filter`. Esta función nos ayuda a eliminar o filtrar rápidamente los elementos de una lista que cumplan un patrón. El patrón se lo daremos en forma de función pasada por parámetro. Esta función, a su vez recibe un elemento de la lista y retorna `True` si queremos que se quede en la lista resultante o `False` si queremos que sea filtrado o eliminado. Veamos un ejemplo sencillo. En el recuadro siguiente tenemos una lista de números y usaremos `filter` para eliminar los negativos. Para ello le pasamos una función que le indique a `filter` si un número es negativo o no. Igual que ocurría antes, `filter` nos retorna un *iterador* que convertiremos a lista usando la función `list()`.

```
numeros=[3,-1,10,-23,-11,45,61,12]

def quitaNegativos(x):
    if x<0:
        return False
    else:
        return True

naturales=list(filter(quitaNegativos, numeros))
print(numeros)
print(naturales)
```

4.2. Cadenas o strings

Otro tipo con el que llevamos todo el curso trabajando son las cadenas de caracteres o *strings*. Es un tipo de datos fundamental y muy utilizado en cualquier lenguaje de programación y ahora ha llegado el momento de profundizar algo más en el para sacarle todo su potencial. En principio las cadenas son muy similares a las listas todo lo explicado anteriormente con los operadores `[` y `]` puede ser utilizado sin problemas: básicamente indexación y construcción de subcadenas. Pero también tenemos que tener en cuenta que las cadenas en Python pertenecen a una familia de tipos llamada objetos. Un objeto es un tipo de datos del que hablaremos más adelante. Por ahora nos vale con saber que es un tipo de datos que lleva su propia funcionalidad (las cosas que puedes hacer con ella) en forma de métodos asociados. Estos métodos se invocarán usando el nombre de la variable de la cadena junto con un ”.” y el nombre del método que queremos invocar. Por ejemplo, vamos a usar dos métodos sencillos que nos convierten una cadena a minúsculas (`lower()`) y otro a mayúsculas (`upper()`). Veamos como se invocan ambos:

```
cadena="Esta es una cadena de ejemplo"
print(cadena.upper())
print(cadena.lower())
```

Los objetos de tipo cadena disponen de muchos métodos que podemos usar para multitud de operaciones. A continuación listamos alguno métodos interesantes que son frecuentemente

utilizados:

- `replace(c1,c2)`: Retorna una cadena como la original con todas las apariciones de la primera subcadena cambiadas por la segunda subcadena.
- `find(c)`: Nos retorna el índice de la cadena original donde empieza la primera aparición de la subcadena `c`.
- `split(c)`: Recibe una subcadena como parámetro y fragmenta la cadena original en trozos usando esta subcadena como delimitador. Retorna una lista con todos los trozos.
- `join(lista)`: Es un método inverso al anterior. Recibe una lista de cadenas retorna una cadena con todos los trozos unidos, tomando la cadena original sobre la que se invoca como separador.
- `strip(c)`: Retorna una cadena a la que se la han limpiado los caracteres en blanco de su inicio y de su final.

Es importante remarcar que ninguno de estos métodos modifica la cadena original sobre la que se invocan. Siempre retornan una cadena nueva. Antes de terminar puede ser útil conocer la posibilidad de declarar un literal de tipo cadena en varias líneas o en forma de párrafo usando la triple comilla (algo similar a lo explicado anteriormente para comentarios multilínea). Veamos un ejemplo sencillo:

```
parrafo = '''Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi
nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis
odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum.
Aliquam posuere. Nam a sapien.
'''

print (parrafo)
```

4.3. Tuplas

Las tuplas son otro tipo muy utilizado en Python. A primera vista son similares a las listas pero tienen una característica que las hace muy particulares. Son «inmutables». Esto quiere decir que una vez creada una tupla esta no puede ser modificada. Si queremos hacer algún cambio hemos de crear otra nueva en base a los valores de la primera. Esta característica no es nueva para nosotros y ya la hemos visto con las cadenas de texto. Es importante recordar que, al igual que con las listas, los valores contenidos en una tupla no tienen porque ser todos del mismo tipo.

Para declarar una nueva tupla lo haremos de forma similar a una lista pero usando los paréntesis en lugar de los corchetes. Para acceder a los miembros de una tupla, si usaremos los corchetes igual que hacíamos con las listas.

```
unalumno = ("Sergio", 35, 8.7)
otroalumno = ("María", 28, 9.0)

print ("La nota del alumno "+unalumno[0]+" es "+str(unalumno[2]))
print ("La nota del alumno "+otroalumno[0]+" es "+str(otroalumno[2]))
```

En el ejemplo anterior hemos definido dos tuplas para almacenar la información de dos alumnos. Las hemos creado siguiendo el criterio de usar la primera posición para el nombre, la segunda para la edad y la tercera para la nota media. Luego hemos accedido a algunos de sus campos para mostrarlos por pantalla.

Podemos recorrer una tupla usando un bucle `for` o `while` de forma similar a como recorremos una lista. Y en el caso de querer convertir una tupla en una lista (para poder modificarla, por ejemplo) lo podemos hacer usando la función `list()`. Una de las utilidades más usadas de las tuplas es cuando queremos que una función retorne varios valores. En ese caso es común utilizar una tupla para devolver todos esos valores como un sola variable a través de `return`. Imaginemos una función que queremos que retorne un punto cartesiano aleatorio compuesto por dos valores: x e y.

```
import random

def punto_aleatorio():
    x=random.randint(0,100)
    y=random.randint(0,100)
    return (x,y)

punto1 = punto_aleatorio()
punto2 = punto_aleatorio()

print (punto1)
print (punto2)
```

4.4. Diccionarios

Los diccionarios son el último tipo para agrupar valores que vamos a estudiar. Un diccionario es, al igual que una lista o una tupla, una colección de otros valores de igual o de distinto tipo. ¿Qué le hace diferente entonces a estos? Que los valores que agrupa no serán indexados usando índices numéricos como en las listas o las tuplas. En los diccionarios, cada vez que insertemos un valor lo haremos aparejado a una **clave o llave**. Esta clave será necesaria luego para recuperar el valor del diccionario. Veamos un ejemplo sencillo en que definimos un diccionario de forma literal para almacenar los datos de un alumno.

```
unalumno={
    "nombre":"Sergio",
    "edad":35,
    "nota":8.7
}

print ("El alumno "+unalumno["nombre"]+" tiene una nota de "+str(unalumno["nota"]))
```

Lo primero de todo es ver que el diccionario se ha definido usando los caracteres `{` y `}` llamados comúnmente «llaves » en lugar de corchetes o paréntesis. Dentro de estas llaves hemos incluido, en lugar de valores como hacíamos antes con tuplas y listas, parejas formadas por valores y claves. En concreto hemos introducido tres parejas con la forma valor/clave. Estas parejas se separan por comas simplemente. La separación en líneas diferentes no es necesaria y se ha hecho por simple claridad. Tras la definición del diccionario hemos accedido a sus valores usando la misma sintaxis que con listas y tuplas pero ahora, en lugar de usar índices numéricos como antes, usamos como índices las claves asignadas a cada valor. En caso de que queramos comprobar si una clave existe en un diccionario usaremos la instrucción `in`.

```
if "apellido" in unalumno:
   print (unalumno["apellido"]
else:
   print ("El alumno no tiene definido un apellido")
```

Para iterar sobre un diccionario podemos hacer de forma similar a listas y tuplas. Podemos iterar entre la lista de claves u obtener los valores asociados a cada clave indexando estas.

```
# Se imprimen solo las claves
for c in unalumno:
    print(c)

# Se imprimen solo los valores asociados a las claves
for c in unalumno:
    print(unalumno[c])

# Se imprimen claves y valores separados por un caracter ':'
for c in unalumno:
    print(c+": "+str(unalumno[c]))
```

Si queremos obtener listas separadas de claves y valores podemos usar las funciones `key()` y `values()` respectivamente sobre el dato de tipo diccionario.

4.5. Otras consideraciones

En este tema hemos dejado de comentar otro tipo de datos de tipo colección que también puede resultar interesante para ciertos ámbitos, como son los conjuntos. Además, un tema

interesante sobre los tipos colección es su capacidad de anidamiento en cualquier orden. Esto quiere decir que es perfectamente posible construir un tipo colección compuesto por otro de estos tipos. De esta forma podríamos tener una lista formada por listas, por ejemplo. Estructura que suele ser la usual para representar matrices. Cualquier otra posibilidad de este tipo podría darse: Un diccionario formado por listas, una tupla de diccionarios que su vez contengan una lista, etc. Como ejemplo que ilustre esto vamos a completar la estructura anterior para representar a un alumno manejando, en lugar de una sola nota, tres notas en forma de lista. Además, se añade un código sencillo para calcular la nota media de este alumno para mostrar como accedemos a esta estructura combinada por diccionario y listas.

```
unalumno={
    "nombre":"Sergio",
    "edad":35,
    "notas": [5.5, 6.0, 7.8]
}

# Calculamos la nota media

nota_media = 0
for nota in unalumno.notas:
    nota_media = nota_media + nota

nota_media=nota_media / len(unalumno.notas)
```

Actividades

1. Crea un script en el que declares una lista de palabras y luego elimina usando filter, todas aquellas que tengan más de 5 letras.

2. Declara una variable de tipo cadena en varías líneas (estilo párrafo) y programa dos funciones llamadas `palabra_mas_larga()` que reciba el string y retorne la palabra más larga encontrada y otra llamada `palabras_capitalizadas()` que reciba el párrafo y retorne otro string con las mismas palabras pero capitalizadas[1].

3. Programa una función que convierta un string con un número en binario en un valor decimal

4. Siguiendo el mismo razonamiento que en el ejercicio anterior, programa ahora una función parecida para convertir un número hexadecimal en decimal.

5. Dado un párrafo definido en una variable de forma similar a los anteriores realiza una análisis de frecuencias de caracteres. Este análisis se basa en contar las apariciones de cada carácter. Para ello, debes mantener un diccionario donde cada letra sea una clave diferente y el valor aparejado a ella sea el número de apariciones que has contado.

6. Crea un pequeño programa que le vaya pidiendo nombres al usuario hasta que este introduzca la palabra `fin`. En ese momento debes mostrar por pantalla las veces que ha introducido cada nombre diferente.

[1] Capitalizar significa escribir la misma palabra toda en minúscula y la primera letra en mayúscula.

5

Objetos y clases

Hasta ahora hemos hablado de los tipos con los que Python cuenta para almacenar datos. Hemos hablado de tipos simples como los numéricos o los booleanos y tipos algo más compelejos como las listas, tuplas o diccionarios. Ahora vamos a ir un poco más allá y veremos la forma de crear nosotros nuestros propios tipos de datos adaptados a nuestras necesidades. Para ello usaremos una metodología muy utilizada hoy en el mundo de la programación como son los objetos y las clases.

5.1. Concepto de clase y objeto

Una **clase** es la forma que usaremos en Python de definir nuestros propios tipos de datos. Una clase define un nuevo tipo de dato que suele estar compuesto por tipos más simples y además, lleva aparejada una funcionalidad asociada a el. ¿Esto que quiere decir? Que no solo definimos los tipos que componen el nuevo tipo sino que además definimos las cosas que vamos a poder hacer con el, las funciones asociadas al nuevo tipo. A estas funciones se las llama **métodos**.

Veamos un ejemplo que ilustre esto. Queremos manejar en nuestro programa grupos de alumnos. Un alumno en nuestro programa está compuesto por un nombre, un apellido, una edad y tres números enteros que definen la nota de los tres trimestres. Si solo se tratara de un alumno podemos definir a ese alumno como una tupla con esos subtipos. Pero como usaremos varios alumnos y queremos que todos tengan el mismo aspecto es mejor que definamos un tipo concreto nuevo llamado `Alumno`. Para ello definiremos una nueva clase llamada de esa forma y crearemos variables de ese tipo:

```
class Alumno:
    # definición de alumno
    # ....


alumno1 = Alumno()
alumno2 = Alumno()
```

¿Qué está ocurriendo en estas dos líneas? Hemos definido dos nuevas variables en base a lo devuelto por la función `Alumno()`. Esta función se ha creado de forma automática al definir la clase `Alumno`. Esta función es la que definirá como está construida una variable cualquiera de tipo `Alumno`. A continuación vemos como debemos completar la clase para que esta función esté bien definida:

```
class Alumno:
    def __init__(self):
        this.nombre=""
        this.edad=0
        this.notas=[0,0,0]

alumno1 = Alumno()
alumno2 = Alumno()
```

Antes de pasar a explicar en detalle el funcionamiento de la función `__init__()` terminaremos este apartado diciendo que tanto `alumno1` y `alumno2` son objetos. Un **objeto** no es mas que una variable definida en nuestro código sobre un tipo de datos definido como una clase.

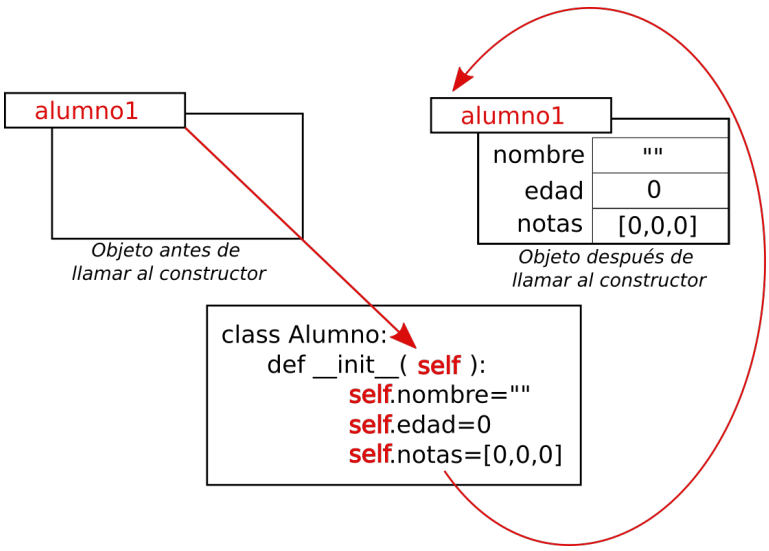

Figura 5.1: Proceso de construcción del objeto `alumno1`

5.1.1. Constructores de clase

Nos habíamos quedado en explicar el objetivo de la función `__init__()`. A esta función se la denomina función constructora. En ella introduciremos el código que queremos que se ejecute siempre cuando un nuevo objeto de esa clase es creado. Vemos que recibe un parámetro llamado `self`. Este parámetro equivale a la nueva variable que estamos creando. La principal tarea de esta función es ir rellenando la variable de los atributos que queremos que compongan al nuevo objeto. En el caso de nuestro anterior ejemplo estos atributos serán:

- `nombre`: Un string para almacenar el nombre de un alumno
- `edad`: Un número para almacenar su edad
- `notas`: Una lista para guardar las notas del alumno

Estos atributos se rellenan gracias a la variable `self`. Esta variable realmente es un parámetro que se le pasa al constructor de forma automática por parte del intérprete. Contiene la dirección de memoria de la nueva variable que estamos creando fuera del constructor. En este caso `alumno1`. Para definirnos un nuevo atributo dentro del nuevo objeto usaremos el caracter '.' seguido del nombre del atributo y de su nuevo valor. En la figura 5.1 vemos en detalle el proceso de construcción. Tras declarar vacía la variable `alumno1` se llama al constructor de `Alumno`. A este se le pasa, a través de `self` la dirección de memoria definida en `alumno1`. Sobre este parámetro se van creando nuevos atributos con sus respectivos valores.

Es muy normal querer introducir parámetros en el constructor y así personalizar el método de construcción para cada objeto. En el caso que nos ocupa vamos a pasar a nuestro constructor los valores del nombre del alumno y de su edad. Dejando las notas siempre en blanco. Para ello definimos el constructor con los dos parámetros que hemos decidido añadir y **dejando siempre como primer parámetro a self**. A la hora de invocar el constructor, lo hacemos asignando valor a estos parámetros pero no a `self`. Recordemos que este parámetro se introduce de forma automática por parte del intérprete.

```
class Alumno:
    def __init__(self,nombre,edad):
        self.nombre=nombre
        self.edad=edad
        self.notas=[0,0,0]

alumno1 = Alumno("Sergio",40)
alumno2 = Alumno("Marta",23)
alumno3 = Alumno("Luis",33)

print (alumno2.nombre)
```

5.2. Métodos y atributos

Ahora que ya tenemos claro como se construye un objeto veamos como acceder a sus atributos. Para acceder a un atributo usaremos el nombre del objeto seguido de punto y del nombre del atributo. Podemos acceder a un atributo para consultar su contenido o bien para modificarlo.

```
alumno1=Alumno("Sergio",40)
print(alumno1.nombre)

alumno1.nombre="Luis"
print(alumno1.nombre)
```

Además de los atributos, un aspecto fundamental de la programación usando objetos es que cada variable puede llevar asociada su funcionalidad en forma de métodos. Un **método** es una función asociada a cada objeto de una misma clase (un mismo tipo) que normalmente accederá o modificará el valor de uno o varios de sus atributos. Los métodos se definen, igual que los atributos, a la hora de definir la clase. Para invocar un método sobre un objeto se usa la misma sintaxis que con los atributos: nombre del objeto, caracter punto y nombre del método con los posibles parámetros que pueda tener. Veamos un ejemplo de tres métodos de ejemplo para la clase `Alumno`:

```
class Alumno:
    def __init__(self,nombre,edad):
        self.nombre=nombre
        self.edad=edad
        self.notas=[0,0,0]

    def esMayorEdad(self):
        return self.edad>=18

    def notaMedia(self):
        return (self.notas[0]+self.notas[1]+self.notas[2])/3

    def nombreFormal(self,titulo):
        return titulo+ +self.nombre
```

En el ejemplo anterior hemos definido tres métodos que pueden ser invocados sobre cada objeto de la clase `Alumno`. El primero retorna un valor booleano en base a si el alumno es o no mayor de edad, el segundo nos da la media aritmética de sus calificaciones y el tercero nos retorna el nombre junto con un título en forma de prefijo. Veamos un ejemplo de uso de estos métodos:

```
alumno1=Alumno("Sergio",40)
alumno2=Alumno("Marta",38)

alumno1.notas=[7,8,8]

print(alumno1.esMayorEdad())              # Imprime True
print(alumno1.notaMedia())                # Imprime 7.6666667
print(alumno2.nombreFormal("Doña"))       # Imprime Doña Marta
```

Es **muy importante entender el uso de self** en la definición de los métodos. Siempre aparece como primer parámetro en la definición de un método y guardará la referencia al objeto o variable sobre el que ha sido invocado el método. Sin embargo, en la invocación del método no se le asigna un valor explícito. Su asignación se hace de forma oculta por parte

del intérprete. Véase el caso del último método que se ha definido con dos parámetros: `self` y `titulo` y sin embargo en su invocación solo recibe el valor para uno. El parámetro `self` es asignado de forma automática.

5.2.1. Visibilidad

Anteriormente hemos visto como hemos accedido a un atributo de un objeto ya creado para modificarlo. Esto en la práctica no es recomendable. Si queremos que un atributo pueda ser modificado, deberíamos incluir un método que lo actualice. Dicho de otra manera, bf la única forma de actualizar el estado de un objeto (sus atributos) debería ser a través de sus métodos. Veremos esto en detalle en la siguiente sección.

Podemos evitar que un atributo sea actualizado «desde el exterior de la clase» haciendo que tenga una visibilidad privada. Esta característica de privado se la podemos dar al crear el atributo añadiéndole un prefijo formado por dos guiones bajos. Una vez hecho esto el atributo no podrá ser invocado desde cualquier otro archivo que no sea en el que hemos definido la clase.

5.3. Encapsulado del estado de un objeto

Como ya hemos avanzado, la forma más limpia de trabajar con objetos es no permitir que su estado sea actualizado desde el exterior de estos. Lo normal será dotar al objeto de métodos para la consulta de cada atributo y para su actualización. Los métodos que se encarguen de consultar un atributo suelen denominarse *getters* y se indican con un prefijo `get_` en el nombre del método. Retornarán el valor de ese atributo. Los métodos encargados de actualizar un atributo se llamarán *setters* y suelen indicarse con un prefijo `set_` seguido del nombre del atributo que actualizan. Vemos un ejemplo con algunos *getters* y *setters* de la clase `Alumno`.

```
class Alumno:
    def __init__(self,nombre,edad):
        self.nombre=nombre
        self.edad=edad
        self.notas=[0,0,0]

    def get_nombre(self):
        return self.nombre

    def set_nombre(self,n):
        self.nombre=n

    def get_edad(self):
        return self.edad

    def set_edad(self,e):
        self.edad=e

    ....
```

Si consideramos que un atributo de un objeto puede no ser actualizable por su propia naturaleza, no es necesario que tenga un método *setter*. Un ejemplo puede ser el DNI de un usuario. Este no puede ser modificado tras crear el objeto porque el DNI no cambia nunca.

5.4. Herencia de objetos

Para terminar con el manejo de objetos vamos a hablar de como crearlos a partir de otros. Imaginemos que queremos crear un programa que maneje alumnos y profesores. Ambos objetos tendrán un montón de atributos comunes: nombre, apellidos, edad, dni, email, etc. Aunque también tendrán atributos propios. Los alumnos tendrán notas, aulas, etc. mientras que los profesores despacho, listas de guardias, etc. ¿Es necesario que creemos dos clases tan parecidas con muchos atributos, *getters* y *setter* iguales? Gracias a la herencia, no.

La herencia permite crear una clase en base a otra sin tener que reescribirla por completo. Veamos esto con un ejemplo en base a lo planteado en el párrafo anterior. La parte común a profesores y alumnos la pondremos en una clase llamada `Persona`. Luego crearemos las clases `Profesor` y `Alumno` que hereden de `Persona` y será como si en ambas hubiéramos escrito el mismo código. La primera parte de este planteamiento se muestra en este código. Se deja como ejercicio al lector añadir la clase `Profesor` y terminar los métodos de las tres clases.

```
class Persona():
    def __init__(self, nombre,edad):
        self.__nombre = nombre
        self.__edad = edad

    def get_nombre(self):
        return self.__nombre

    def get_edad(self):
        return self.__edad


class Alumno(Persona):
    def __init__(self,nombre,edad):
        super().__init__(nombre,edad)
        self.notas=[0,0,0]

    def notaMedia(self):
        return (self.notas[0]+self.notas[1]+self.notas[2])/3


## Código principal

p1 = Persona("Sergio",40)
a1 = Alumno("Marta",38)

print (p1.get_nombre())
print (a1.get_nombre())
```

Actividades

1. Crea una clase llamada `Cuenta` para guardar los movimientos bancarios de una cuenta. Una cuenta debe esta compuesta por un propietario (que será un identificador de texto) y un saldo numérico (un número entero). Todas las cuentas deben comenzar con un saldo inicial de 0. Además debe tener, al menos dos métodos llamados `ingreso()` que reciba una cantidad a ingresar en la cuenta y `gasto()` que reciba una cantidad a retirar de la cuenta. Si una cuenta está en valores negativos de saldo no podrá efectuar ningún otro gasto. El atributo saldo debe ser privado no podrá ser accedido ni modificado directamente.

2. Crea un pequeño programa que utilice la clase anterior y la pruebe. En este ejemplo crea varias cuentas para diferentes usuarios y genera en ella movimientos de prueba.

3. Haz una clase ahora llamada `Banco` que mantenga en su interior una lista de cuentas creadas. Esta clase debe tener al menos dos métodos: uno que permita añadir una nueva cuenta al banco y otra que busque retorne una cuenta en base a un identificador.

4. Crea un pequeño programa que utilice la clase anterior y la pruebe. En este ejemplo crea varias cuentas, insértalas en el banco y luego busca alguna de ellas. Prueba también a buscar alguna que no exista.

5. Haz un pequeño programa que nos muestre un menú con cuatro opciones: añadir cuenta, retirar dinero, ingresar dinero y salir. Tras seleccionar la primera opción se nos pedirá un identificador para esa nueva cuenta y esta será creada y añadida a una lista de cuentas creadas. Seleccionado la segunda o la tercera nos pedirá tanto el identificador de la cuenta como la cantidad a ingresar o retirar. La opción de salir debe abortar el programa.

6

Entrada/Salida

Cuando hablamos de entrada/salida en programación nos referimos a los mecanismos que tenemos disponibles para introducir bytes en nuestros programas y emitir bytes desde estos. Hasta el momento hemos visto dos métodos básicos para ambos flujos: el uso de la instrucción `print` para sacar datos a la pantalla y la instrucción `input` para tomar datos desde teclado. En este tema vamos a profundizar en este aspecto y aprenderemos a tomar y guardar datos en ficheros de disco. Además, también aprenderemos a emitir datos a la pantalla siguiendo formatos que enriquezcan la visualización (tabulados, columnas, etc.). Por último aprenderemos a lanzar nuestros programas suministrándoles los datos durante el inicio en forma de modificadores. De forma similar a como funcionan los comandos propios del sistema operativo.

6.1. Lectura y escritura de ficheros

Con Python podemos acceder a todo tipo de ficheros en el sistema. Cuando hablamos de todo tipo nos referimos a los dos tipos básicos de ficheros que podemos encontrar:

- Los llamados **ficheros de texto** que son aquellos codificados con algún sistema de codificación alfanumérico. Normalmente UTF-8.
- Todos los demás que se denominan normalmente **ficheros binarios**.

Esta terminología es algo confusa porque implica que los primeros que hemos mencionado no lo son, cuando no es así. Cuando hablamos de ficheros binarios nos referimos a aquellos que usan una codificación diferente a un sistema estandarizado alfanumérico. En este libro vamos a centrarnos en los ficheros de texto para trabajar. En Linux son los ficheros más abundantes en relación a la administración y configuración del sistema y también nos servirán para guardar lotes de datos de nuestras aplicaciones. Por lo tanto este tipo cubrirá todas nuestras necesidades por ahora.

Para trabajar con ficheros necesitamos siempre seguir los siguientes pasos:

1. **Apertura del fichero**: En esta fase es cuando indicamos al sistema operativo con que

ficheros queremos trabajar y que operación queremos realizar sobre el (añadir datos, sobreescribir los que hay, etc.)

2. **Lecturas/escrituras**: En esta fase realizaremos las operaciones de lectura de bytes o escritura de bytes que correspondan.
3. **Cierre del fichero**: Es una operación que no puede obviarse. Normalmente los sistemas operativos realizan la sincronización entre la imagen de memoria de los datos y el disco físico en el momento del cierre. No hacer un cierre de un fichero puede hace que se pierdan datos del fichero con el que estamos trabajando.

Para abrir un fichero usaremos la función `open()`. Esta función recibe normalmente dos parámetros: la ruta del fichero con el que queremos trabajar y el tipo de operación que realizaremos sobre el. A continuación indicamos el tipo de operaciones más usuales:

- Solo lectura (`r`)
- Lectura y escritura (`r+`)
- Agregación (`a`)

La diferencia entre la escritura y la agregación es que la segunda mantiene el contenido del fichero intacto y permite escribir datos al final de este. La segunda opción también crea el fichero si este no existe. La función `open()` retorna un objeto de tipo `File` sobre el que podemos invocar los siguientes métodos:

- `write()`: Inserta en el fichero una cadena de texto.
- `writelines()`: Inserta en el fichero varias cadenas de texto.
- `read()`: Retorna todo el contenido del fichero en forma de una sola cadena de texto.
- `readlines()`: Retorna una lista de cadenas de texto. Cada elemento de esta lista es una línea diferente del fichero.

Una vez hemos trabajado con el fichero realizando tantas operaciones de lectura y escritura como queramos es fundamental cerrar el fichero usando la función `close()`. Veamos un ejemplo sencillo en el que se irán pidiendo al usuario nombres de personas que se guardarán en un fichero.

```
fichero = open("nombres.txt","w")

while(True):
    nombre=input("Introduce un nombre (en blanco para salir): ")
    if (len(nombre) == 0):
        break
    fichero.write(nombre+"\n")

fichero.close()
```

Una de las grandes ventajas de Python sobre otros lenguajes se puede apreciar en el código anterior. Hay poco lenguajes donde trabajar con ficheros de la forma en la que lo hemos hecho ocupe tan poco espacio y quede tan condensado y fácil de leer.

6.1.1. Ficheros XML

El formato XML estructurar internamente ficheros con información y hacerla fácilmente procesable por nuestros programas. Imaginemos que queremos guardar una pequeña lista o base de datos con los libros que hemos leído. Una forma sencilla sería guardarlos de cualquier forma en un fichero de texto (por ejemplo, un libro por línea, usando un carácter fijo para separar los campos). Leerlos así del fichero sería posible pero implicaría ir procesando línea a línea y separando los campos manualmente. Una solución más sencilla sería usar un fichero en formato XML tal y como se muestra a continuación:

```
<bd>
  <info>
    <propietario>Sergio</propietario>
    <actualizado>Febrero 2024</actualizado>
  </info>
  <libros>
    <libro>
      <titulo>El señor de los anillos</titulo>
      <autor>J.R.R. Tolkien</autor>
      <anio_publicacion>1954</anio_publicacion>
    </libro>
    <libro>
      ...
    </libro>
    <libro>
      ...
    </libro>
  </libros>
</bd>
```

Leer este fichero con un programa sería muy sencillo usando la librería `xml.etree.ElementTree` que podemos importar de forma sencilla. Una vez importada, lo primero de todo es *parsear* el documento y cargarlo en la memoria del proceso. Tras esto, llamamos a `getroot()` para obtener acceso al elemento raíz del documento. En este caso la etiqueta `bd`.

Dado un elemento al que podemos acceder como un objeto cualquiera podemos usar el método `find()` para encontrar una marca dentro de este. Por ejemplo, para acceder al elemento `info` usaríamos:

```
root.find("info")
```

Si queremos acceder a un elemento dentro de otro podemos encadenar estas llamadas perfectamente. Por ejemplo, para acceder al propietario haríamos algo como:

```
root.find("info").find("propietario")
```

Dado un elemento, si invocamos la propiedad `text` nos retornará el texto dentro de las etiquetas. Tomando como base el anterior, para acceder al texto de la etiqueta propietario usaríamos:

```
root.find("info").find("propietario").text
```

Veamos ahora un ejemplo completo con el fichero XML anterior. En este ejemplo, además de acceder a etiquetas de forma aislada vemos como navegar a través de los libros usando el método `findall()` que nos retorna una lista con todos los objetos del XML del tipo que le pasemos por parámetro al método. Luego, tras obtener esta lista navegamos por ella usando un bucle clásico y vamos imprimiendo el título y el autor de cada uno de ellos.

```
import xml.etree.ElementTree as ET

archivo_xml = "ejemplo.xml"
tree = ET.parse(archivo_xml)
root = tree.getroot()

prop = root.find("info").find("propietario").text

print("Propietario:", prop)
print("**********************************")

for li in root.find("libros").findall("libro"):
    titulo = li.find("titulo").text
    autor = li.find("autor").text

    print("Título:", titulo)
    print("Autor:", autor)
    print("--------------------")
```

Además de leer un fichero XML también podemos actualizarlo con nueva información. En este caso podríamos estar interesados en añadir un nuevo libro. El código siguiente muestra como hacer esto:

```
# Creamos el nuevo elemento y lo rellenamos:
nuevo_libro = ET.Element('libro')
nuevo_titulo = ET.SubElement(nuevo_libro, 'titulo')
nuevo_autor = ET.SubElement(nuevo_libro, 'autor')
nuevo_anio = ET.SubElement(nuevo_libro, 'anio_publicacion')

nuevo_titulo.text = "Nuevo Libro"
nuevo_autor.text = "Autor Desconocido"
nuevo_anio.text = "2022"

# Añadimos el nuevo elemento a su elemento padre:
root.find("libros").append(nuevo_libro)

# Volcamos la estructura en XML en el fichero:
ET.indent(tree, space="\t", level=0)
tree.write(archivo_xml, encoding='utf-8')
```

En este ejemplo vemos como hemos creado el nuevo elemento usando los métodos `Element()` y `SubElement()` y tras esto les hemos dado los valores que hemos querido. El paso final sería unir este nuevo elemento a la estructura eligiendo el elemento padre (en

este caso es `libros`) y llamando sobre el al método `append()`. Por último, volcamos la estructura al fichero.

6.1.2. Otros tipos de ficheros

En el ámbito de la administración de sistemas podemos encontrarnos con otros tipos de ficheros que necesitemos leer y procesar con nuestros programas Python. Uno muy común usado en muchos ámbitos es el formato CSV. Un fichero CSV no es más que un fichero de campos separados por un delimitador. Cada línea del fichero es un registro, similar a una fila de una hoja de cálculo y cada celda de la línea se separa con un separador (coma o punto y coma normalmente). A continuación se muestra el aspecto de un fichero CSV cualquiera:

```
Luis;Martín;33.123;12;izquierda
María;García;32.123;18;izquierda
Jose Miguel;Antunez;43.32;derecha
Antonio;de Miguel;29.234;izquierda
```

Los campos están separados por un delimitador común y normalmente todas las líneas suelen tener un número fijo de campos. Este formato es muy usado debido a que es una forma sencilla de exportar una hoja de cálculo a texto plano y a la inversa. Para leer o procesar un fichero de este tipo podemos usar un patrón similar al que se muestra en el recuadro:

```
fichero = open("fichero.csv")
lineas = fichero.readlines()
for alumno in lineas:
    campos = alumno.strip().split(";")
    ## Ahora accederemos a cada campo de esa linea usando
    ## campo[0], campo[1], campo[2], ...
```

Como se ve en el recuadro se ha usado la función `readlines()` que nos retorna una lista de líneas. Luego se itera en esta lista limpiando cada línea de espacios en blanco y saltos del línea adicionales usando la llamada a `strip()`. Por último, la línea se trocea en campos con `split()` para poder acceder a cada campo de la línea de forma aislada.

Otro tipo de ficheros que podemos encontrar son los llamados ficheros de *properties* o propiedades. Estos suelen ser colecciones de pares de tipo clave-valor donde estas van separadas por un caracter `=`. Un ejemplo de este tipo de ficheros podría ser el siguiente:

```
url=localhost
user=mkyong
password=12345
lastaccess=20240215
target=work
```

El procesado de este tipo de ficheros sería similar a lo explicado antes. Iterando línea por línea pero ahora *spliteando* cada línea por el caracter `=` y así obteniendo la clave y el

valor de esta por separado.

Podemos encontrar variaciones de cualquiera de estos ficheros y siempre, su procesado pasará por ir línea línea usando `readlines()` y realizando un troceado de estas líneas con el método `split()`.

6.2. Escritura formateada por pantalla

Hasta ahora hemos visto la función `print()` como única forma de mostrar datos por pantalla. Para mostrar datos aislados esta función nos basta pero, si queremos combinar datos numéricos con cadenas de caracteres puede que se nos quede muy limitada. Para combinar texto con numéros, espacios, tabuladores, etc. usaremos el método `format()` del objeto String.

El método `format()` se invoca siempre sobre un string. Este string tendrá una serie de marcadores señalados por parejas de {}. Estos marcadores se irán sustituyendo por los parámetros de invocación de `format()`. Veamos un ejemplo sencillo:

```
mensaje="Esta será mi prueba número {} para hacer {}".format(1,"salida formateada")
print(mensaje)
```

En este ejemplo tomamos como base una cadena de texto e invocamos sobre ella al método `format()` con dos valores. Estos valores serán sustituidos por los marcadores en orden y de esa forma, la función `print()` nos devolvería algo como:

```
Esta será mi prueba número 1 para hacer salida formateada
```

Podemos incluir nombres en los marcadores y asociarlos a parámetros del método `format()` para implicar su lectura:

```
mensaje="Titulo: {titulo} Autor:{autor}".format(titulo="It", autor="Stephen King")
```

Otro uso del que podemos sacar mucha utilidad a esta función es la posibilidad de alinear valores. Veamos algunos ejemplos para la alineación de textos. Para alinear valores a la derecha usaremos el carácter `:>`. Por ejemplo, en el código que se ve a continuación alineamos todas las barras entre diez espacios y alineamos los strings a la derecha.

```
print( "|{:>10}|".format("Sergio"))
print( "|{:>10}|".format("Pepe"))
print( "|{:>10}|".format("Francisco"))
print( "|{:>10}|".format("Marta"))
```

Para alinear a la izquierda cambiaríamos el caracter de alineación por `<` y para alinear al centro usaríamos el caracter `^`.

```
print( "|{:^10}|".format("Sergio"))
print( "|{:^10}|".format("Pepe"))
print( "|{:^10}|".format("Francisco"))
print( "|{:^10}|".format("Marta"))
```

También podemos especificar un carácter de relleno indicando dicho carácter después del signo de alineación.

```
print("{:0>8}".format("11"))
print("{:0>8}".format("1101"))
print("{:0>8}".format("1001"))

print ("{:~^10}".format("Sergio"))
print ("{:~^10}".format("Ana"))
```

Para terminar veremos que si estamos formateando números decimales, puedes controlar la alineación de los dígitos y la posición del punto decimal.

```
numero = 123.456
resultado = "{:10.2f}".format(numero)
print(resultado)
```

Actividades

1. Para este ejercicio poder leer un fichero CSV. El fichero CSV debe contener una fila por alumno y en cada fila encontraremos la siguiente información: nombre, apellido, edad, notas de las 3 evaluaciones y número de faltas de asistencia. A continuación se muestra un ejemplo que deberías completar con más alumnos:

```
Marta;López;19;8;9;7;23
Luis;Rodríguez;20;7;7;6;0
Antonio;de Miguel;19;9;9;7;0
María;García;21;7;6;5;12
```

   Tomando como entrada el fichero anterior, lee el fichero de forma completa y ve mostrando por pantalla el nombre, el apellido y la nota medía de los alumnos del fichero.

2. Tomando ahora el fichero CSV del ejercicio anterior como base, haz un programa que lo convierta a un fichero XML.

3. Haz un programa que realice la conversión inversa del ejercicio anterior. Tomando como base un fichero XML (puede ser el que generaste en el ejercicio anterior u otro a tu elección con un esquema que tu hayas diseñado) conviértelo en un fichero CSV.

4. Crea un programa que se utilice a través de la línea de comandos para gestionar los contactos de tu correo electrónico. El programa se llama `contactos` y debe tener dos modificadores. El primero será `-a` para añadir un contacto. Tras el escribiremos el identificador del contacto y su correo electrónico. El segundo será `-s` para buscar un contacto. Tras este modificador escribiremos el identificador del contacto que queremos buscar y el comando nos retornará la información de este. Guarda los contactos en un fichero de la forma que prefieras. A continuación se muestra un ejemplo de uso:

```
$ contactos -a sergio sergio@correo.es
$ contactos -a pepe pepe@correo.es
$ contactos -s pepe
id: pepe   correo: pepe@correo.es
```

5. Toma ahora el fichero CSV del primer ejercicio y genera una tabla formateada tal y como se muestra en el cuadro inferior. La tabla debe mostrar el nombre, los apellidos, la nota media y la nota textual del alumno en base a su nota media. La tabla que muestres debe tener un aspecto como el que se muestra a continuación:

```
|NOMBRE   |APELLIDOS | MEDIA |   NOTA   |
|---------|----------|-------|----------|
|Marta    |López     |  8.00 | NOTABLE |
|Luis     |Rodríguez |  6.67 |   BIEN   |
|Antonio  |de Miguel |  8.33 | NOTABLE |
|María    |García    |  6.00 |   BIEN   |
```

7

Administración básica del sistema

Aunque Python es un perfecto lenguaje de propósito general con el que podemos desenvolvernos en múltiples ámbitos, la administración de sistemas UNIX es uno en los que mejor encaja. El hecho de ser software libre y de estar tan vinculado a la comunidad GNU ha hecho que desde sus inicios, un intérprete de Python venga instalado de serie en todos los sistemas Linux del mercado. En esta unidad vamos a utilizar todo lo aprendido con Python y lo estudiado sobre administración de sistemas para fusionarlo y así empezar a crear tareas personalizadas que nos permitan optimizar el flujo cotidiano de administración de nuestros sistemas.

7.1. Variables de entorno y librerías

Dentro de las miles de librerías que podemos encontrar en el ecosistema de Python hemos de destacar dos que vienen instaladas junto con el intérprete que nos permitirán trabajar directamente con el sistema operativo. Estas son `os` y `sys`.

El módulo `os` proporciona una interfaz para interactuar con el sistema operativo subyacente en el que se está ejecutando Python. Esto quiere decir que podremos realizar tareas comunes tanto en Linux como en otros sistemas en los que Python funciona. Permite realizar una amplia variedad de tareas relacionadas con el sistema de archivos, la manipulación de rutas de archivos, la gestión de procesos y otras operaciones del sistema. De menor importancia a este nivel es el módulo `sys`. Proporciona acceso a variables y funciones específicas del intérprete de Python (versión, rutas de búsqueda, etc.) y del entorno del sistema y de ejecución de nuestro programa.

Una primera aproximación a estas librerías sería la gestión de las variables de entorno de nuestra sesión de trabajo gracias al diccionario llamado `os.environ`. Veamos un ejemplo de como consultamos las variables de entorno ya definidas, consultamos el valor de una de ellas y creamos una nueva para la sesión actual:

```
import os

# Muestra todas las variables de entorno definidas
for variable, valor in os.environ.items():
    print (variable+" = "+valor)

# Muestra el valor asociado a la variable HOME
print (os.environ.get("HOME"))

# Crea una variable llamada NUEVA con ese valor
os.environ["NUEVA"]="Nuevo valor de la variable"
print (os.environ.get("NUEVA"))
```

7.2. Gestión del sistema de ficheros

Una de las principales tareas que puede que nos interese automatizar es la gestión de diferentes procesos sobre el sistema de ficheros. Hasta ahora, relacionado con ficheros solo hemos estudiado su creación, su escritura y su lectura pero podemos ir mucho más allá: creación y lectura de directorios, asignación de permisos, enlaces, etc.. Veamos algunas funciones interesantes relacionadas con estos aspectos:

- `os.rename(origen,destino)`: Renombra o mueve un fichero o directorio de una ruta a otra.
- `os.remove(ruta)`: Borra un fichero.
- `os.rmdir(ruta)`: Borra un directorio vacío.
- `os.removedirs(ruta)`: Borra un directorio con todo su contenido.
- `os.listdir(ruta)`: Lista el contenido de un directorio.
- `os.mkdir(ruta)`: Crea un directorio en esa ruta.
- `os.walk(ruta)`: Lista el contenido de un directorio y de todos los subdirectorios contenidos en el.
- `os.chmod(ruta, permisos`: Asigna la combinación de permisos a ese fichero. La combinación de permisos es un número en octal. Por ejemplo `0o664`.

A estas funciones podemos incluir las del módulo `os.path` que pueden ser útiles para trabajar con rutas de ficheros:

- `os.path.join(elem1,elem2)`: Fusiona varios elementos para generar una ruta absoluta
- `os.path.split(ruta)`: Retorna una tupla con el la ruta del directorio y el nombre del último elemento de esta.
- `os.path.basename(ruta)`: Retorna el último componente de la ruta.
- `os.path.dirname(ruta)`: Retorna el directorio que contiene al último elemento de la ruta.
- `os.path.getsize(ruta)`: Retorna el tamaño ocupado por el fichero o el directorio de la ruta.

- `os.path.exists(ruta)`: Retorna verdadero o falso en base a si la ruta existe o no.
- `os.path.isdir(ruta)`: Retorna verdadero o falso en base a si la ruta es de un directorio o no.

Una de las funciones más extrañas de las mencionadas es `os.walk()`. Esta función es muy útil cuando queremos recorrer o «caminar» sobre un árbol de ficheros y directorios completo. Una vez llamada sobre una ruta nos retorna una tupla de tres elementos: el directorio actual, los subdirectorios y los ficheros encontrados que se irán actualizando en cada iteración. En cada vuelta del bucle podemos recorrer estos elementos de forma separada o conjunta como veremos un poco más adelante. Por ejemplo, si queremos recorrer todas las rutas de ficheros bajo una ruta de un directorio haríamos algo como lo que se muestra en el recuadro siguiente:

```
import os

mydir = "/home/sdemingo/varios"
for actual,subdirs,ficheros in os.walk(mydir):
   for fichero in ficheros:
      print(os.path.join(actual, fichero))
```

Si por contra quisiéramos navegar por entre las rutas de los subdirectorios contenidos bajo el directorio principal podríamos usar algo parecido pero usando el elemento `subdirs`:

```
import os

mydir = "/home/sdemingo/varios"
for actual,subdirs,ficheros in os.walk(mydir):
   for d in subdirs:
      print(os.path.join(actual, d))
```

En el ejemplo se ve también la utilidad de la función `os.path.join()` que nos permite unir rutas relativas y formar una ruta completa. Por supuesto, ambos ejemplos podrían unirse perfectamente en un solo recorrido sumando en cada iteración los elementos de `subdirs` y de `ficheros`:

```
import os

mydir = "/home/sdemingo/varios"
for actual,subdirs,ficheros in os.walk(mydir):
    hijos = subdirs + ficheros
    for hijo in hijos:
        ruta = os.path.join(actual, hijo)
        if os.path.isdir(ruta):
            print("D "+ruta)
        else:
            print("F "+ruta)
```

Mención aparte se merece la función `os.stat(ruta)`. Esta función nos retorna los metadatos asociados a un fichero y un directorio: permisos, fecha de creación, fecha de modifi-

cación, etc. Esta información se nos retorna en un objeto de tipo `stat_result`. Los campos más interesantes de esta estructura son `st_mtime` y `st_atime` que nos retorna la última modificación y el último acceso del fichero o directorio en segundos[1]. Otro campo importante es `st_mode` que almacena en octal tanto los permisos como el tipo de inodo (fichero, directorio, enlace, ...). De este número en octal los permisos son almacenados en las últimas tres cifras. A continuación se muestra un ejemplo donde examinamos este objeto sobre un fichero cualquiera:

```
import os

info = os.stat("/home/sdemingo/.bashrc")

permisos=oct(info.st_mode)[-3:]
ult_mod=info.st_mtime
ult_acc=info.st_atime

print(permisos)
print(ult_mod)
print(ult_acc)
```

7.3. Gestión de procesos

Una de las características relacionadas con el sistema operativo más útiles dentro de Python es la posibilidad de lanzar otros programas como procesos de sistema. Para realizar estas operaciones podemos ayudarnos del módulo `subprocess`. Veamos un ejemplo sencillo en el que desde nuestro código Python lanzamos un comando `ls -l` para mostrar el listado de elementos del directorio actual:

```
import subprocess

subprocess.run(["ls","-l"])
```

Este código invoca una llamada al sistema para que este cree un proceso con el comando `ls` al que le pasamos el modificador `-l`. Como se ve, la función `run()` recibe tanto el comando como sus modificadores en forma de lista de strings. Si queremos capturar la salida del comando para luego poder analizarla, trabajar con ella o enviarla a otro lugar podemos usar los parámetros `capture_output` y `text` tal y como se ve en el ejemplo:

```
import subprocess

resultado = subprocess.run(["ls","-l"],capture_output=True, text=True)
```

De esta manera, dentro del objeto `resultado` tendremos dos campos llamados `resultado.stdout`

[1]Recordar que los segundos siempre se miden desde el 1 de enero de 1970. Para convertir este número en una fecha normal podemos usar la función `fromtimestamp(segundos)` del módulo `datetime` que previamente debemos importar

con la salida estándar que ha producido el comando y `resultado.stderr` con la salida de error. Si el programa que queremos ejecutar requiere una entrada desde teclado, esta podemos suministrarla a través del parámetro `input`. Un último argumento interesante de este método es también `timeout` que recibe un número de segundos que entendemos como máximo para que se ejecute y termine ese programa. En caso de demorarse más el programa será interrumpido y el método `run()` generará una excepción.

```
res = subprocess.run(['comando'], input=b'dato_de_entrada', capture_output=True,
    text=True)
```

7.4. Creación de comandos de sistema

Cuando escribamos programas Python orientados a la administración de sistemas es muy común que queramos utilizarlos desde la propia consola de comandos de la misma manera que usamos el resto de comandos. Esta forma es escribiendo el nombre del programa seguido de los argumentos o modificadores que queremos utilizar para llamarlo. Existen varias maneras de introducir modificadores y argumentos a nuestro programa pero vamos a estudiar la más sencilla. Se llaman modificadores y argumentos las cadenas de texto que siguen al comando en una línea de comandos de la consola de sistema. Estos modificadores suelen ser o bien letras precedidas de guiones o bien cadenas largas (siempre sin espacio). Por ejemplo:

```
$ programa -o -r fichero1 fichero2
```

La manera conseguir que nuestro programa pueda ser invocado en forma de comando de sistema y recupere los datos introducidos en su invocación sería la siguiente:

```
#!/usr/bin/python3

import sys

mod1=sys.argv[1]
mod2=sys.argv[2]
f1 = sys.argv[3]
f2 = sys.argv[4]

print ("Se han introducido "+str(len(sys.argv))+" argumentos")
```

A través del módulo `sys` podemos utilizar la lista `argv`. La longitud de esta lista nos retorna el número de argumentos usado y el primer elemento de la lista, `sys.argv[0]`, siempre será el nombre del comando. Para que el programa sea lanzable desde la consola hemos de incluir el llamado *shebang* con la ruta del intérprete. Obviamente, si trabajamos en un sistema de tipo UNIX el fichero donde está escrito el programa deberá tener también permisos de ejecución.

Actividades

1. Escribe un pequeño programa que busque los usuarios con shell del sistema (aquellos que tengan asignada una shell válida) y cree para cada uno de ellos un directorio privado al que solo tengan acceso ellos mismos y un usuario llamado `profe1`. Este directorio lo usarán para la entrega de una serie de prácticas. Los directorios privados los debes crear bajo `/tmp/entregas`.

2. Debes crear un programa que revise el contenido del directorio personal de un alumno y le genere un aviso. El nombre del alumno lo puedes pasar a tu programa a través de la línea de comandos o pidiéndolo de forma interactiva. Este programa debe recorrer el directorio personal buscando ficheros de más de 50 megabytes de tamaño. En caso de encontrar alguno has de dejarle un fichero de aviso en el raíz de su directorio personal indicando que debe borrar esas rutas.

3. Crea un programa que se utilice a través de la línea de comandos para gestionar los contactos de tu correo electrónico. El programa se llama `contactos` y debe tener dos modificadores. El primero será `-a` para añadir un contacto. Tras el escribiremos el identificador del contacto y su correo electrónico. El segundo será `-s` para buscar un contacto. Tras este modificador escribiremos el identificador del contacto que queremos buscar y el comando nos retornará la información de este. Guarda los contactos en un fichero de la forma que prefieras. A continuación se muestra un ejemplo de uso:

```
$ contactos -a sergio sergio@correo.es
$ contactos -a pepe pepe@correo.es
$ contactos -s pepe
id: pepe   correo: pepe@correo.es
```

8

Tareas en red

El control y la gestión de la actividad del sistema sobre una red de comunicaciones es una de las principales responsabilidades de todo administrador de sistemas. Además de sus propios conocimientos, el administrador puede ayudarse de Python de su extenso catálogo de librerías para construir herramientas personalizadas que le ayuden en estas tareas. En este capítulo veremos las principales librerías y módulos que podemos usar para estas labores.

8.1. Modelo cliente/servidor

El modelo cliente/servidor es el principal modelo de desarrollo de procesos que se quieren comunicar sobre una red TCP/IP como puede ser Internet. Este modelo se basa en un proceso que hará las veces de cliente y que iniciará la comunicación contra otro proceso que hará de servidor. El cliente debe conocer siempre el denominado *endpoint* del servidor: la dirección IP de la máquina donde este está ejecutando y el puerto en el que está escuchando. Una vez conocido este *endpoint* el cliente puede iniciar una conexión contra ese servidor y si el servidor la acepta, iniciarse una comunicación en ambos sentidos. Para comunicar procesos en Python tenemos una librería `socket`. Un socket es una conexión para la comunicación entre dos máquinas en una red. En el contexto de Python, un socket es una abstracción de una interfaz de comunicación de red que permite a los programas enviar y recibir datos a través de la red.

La biblioteca socket permite crear diferentes tipos de sockets, incluyendo sockets TCP y UDP. Para crear un socket, se utiliza la función `socket.socket()`, especificando el tipo de socket deseado (TCP o UDP) y la familia de direcciones (IPv4 o IPv6). Una vez creado el socket o conexión usaremos la función `connect()` a la que indicaremos la dirección IP o nombre de dominio de la máquina a la que queremos conectarnos y el puerto de escucha del socket.SOCK_STREAM servidor con el que queremos comunicarnos. Veamos un ejemplo completo de esto donde se crea un socket TCP que conectaremos a nuestra misma máquina (dirección de *loopback*) y al puerto 5000.

```
import socket

# Usaremos socket.SOCK_STREAM para TCP y
# socket.SOCK_DGRAM para sockets UPD

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 5000))
```

Para que la anterior conexión tenga éxito hemos de tener un servicio escuchando en ese puerto. Ese servicio puede estar también desarrollado por nosotros usando Python y la librería `socket`. En el caso del servidor de escucha, crearemos un socket de forma similar pero ahora indicaremos al que puerto queremos atarlo y usaremos el método `listen()` del objeto socket para ponerlo en modo escucha. Cuando ejecutamos un proceso escrito de esta forma, el método `listen()` bloquea al proceso servidor hasta que se recibe una conexión por parte de un cliente. Una vez recibida esta, ejecutamos el método `accept()` para aceptarla y empezar a comunicarnos por ella. Veamos un ejemplo sencillo:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("localhost", 5000))
s.listen()

print("Me quedo esperando a clientes ...")

cliente_socket, direccion_cliente = s.accept()
print("Acepto una conexión de "+str(direccion_cliente))
```

Tras aceptar la conexión obtenemos, además de la dirección del cliente que nos acaba de contactar, la referencia al objeto socket por donde hemos de contestarle para iniciar la conversación. Para enviar datos, tanto el cliente como el servidor usarán el método `send()`. El otro lado, el que recibe, que puede igualmente ser el cliente o el servidor, debe usar el método `recv()` indicando eso si, el tamaño máximo de bytes que quiere recibir. Completamos los ejemplos anteriores tanto del cliente como del servidor respectivamente:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 5000))

datos = "Hola caracola !!"
s.send(datos.encode())

s.close()
```

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("localhost", 5000))
s.listen()

print("Me quedo esperando a clientes ...")

cliente_socket, direccion_cliente = s.accept()
print("Acepto una conexión de "+str(direccion_cliente))

datos_recibidos = cliente_socket.recv(1024)
print("Datos recibidos del cliente: ", datos_recibidos.decode())
cliente_socket.close()

# Si cerramos el socket principal del servidor
s.close()
```

Una vez se haya terminado la transmisión de datos el socket debe cerrarse en ambos extremos usando el método `close()` en ambos. Si cerramos también en el servidor, el primer socket que hemos abierto (el que hemos puesto en modo escucha) ya no podremos recibir nuevas conexiones de otros clientes. Esto se suele hacer solo cuando vayamos a cerrar o matar el servidor. En caso contrario se debe cerrar el socket del cliente únicamente. Vamos ahora a modifica el servidor para dejarlo a la escucha de múltiples clientes que quieran conectarse a el.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("localhost", 5000))
s.listen()

print("Me quedo esperando a clientes ...")

while True:
    cliente_socket, direccion_cliente = s.accept()
    print("\nAcepto una conexión de "+str(direccion_cliente))

    datos_recibidos = cliente_socket.recv(1024)
    print("Datos recibidos del cliente: " + str(datos_recibidos.decode()))
    cliente_socket.close()
    print ("Cierro la conexión de este cliente y sigo esperando a otros ...")

s.close()
```

Para terminar es importante remarcar un error común al iniciarse en la programación de procesos de tipo cliente/servidor. En estos casos ambos procesos, los clientes y los servidores, pueden invocar al método `send()` para enviar datos al otro lado y ambos pueden invocar al método `recv()` para recibirlos del otro extremo. Es muy común caer en el error de que solo los clientes enviarán datos y los servidores los recibirán. Normalmente, ambos realizan ambas tareas para tener una comunicación completamente bidireccional.

8.2. Monitorización de la red

Una de las herramientas más versátiles para la monitorización de la red es, sin duda alguna, `nmap`. En Python disponemos de una potente librería del mismo nombre que nos aporta un montón de funciones relacionadas con esta labor: escanear puertos, nodos activos en una red, servicios funcionando en una máquina, etc. Para instalar esta librería lo más recomendable es utilizar `pip`, el gestor de librerías y módulos mencionado en capítulos anteriores.

```
$ pip install python-nmap
```

A continuación se muestra un sencillo ejemplo de como usamos esta librería para hacer un escaneo de puertos de un nodo específico de nuestra red:

```
import nmap

def scan_ports(target):
    # Creamos un objeto de la clase PortScanner
    nm = nmap.PortScanner()

    # Realizamos el escaneo de puertos para el objetivo especificado
    nm.scan(target, arguments='-p 1-1000') # Escaneamos los puertos del 1 al 1000

    # Imprimimos los resultados del escaneo
    for host in nm.all_hosts():
        print(f"Host: {host}")
        print("State:", nm[host].state())

        # Iteramos sobre los protocolos (tcp, udp) y sus puertos
        for proto in nm[host].all_protocols():
            print("Protocol:", proto)
            ports = nm[host][proto].keys()
            for port in ports:
                print(f"Port: {port}\tState: {nm[host][proto][port]['state']}")



# Ejemplo de uso:

target_ip = '127.0.0.1' # Dirección IP del objetivo
scan_ports(target_ip)
```

Este es solo un ejemplo básico, de `python-nmap`. Esta librería ofrece muchas más funcionalidades para realizar escaneos más detallados y complejos. Puedes consultar la documentación oficial que encontrarás en su web oficial[1] para obtener más información sobre las capacidades de esta librería.

[1]Página oficial python-nmap: https://xael.org/pages/python-nmap-en.html

Actividades

1. Realiza un servidor de eco. Un servidor de eco es un programa servidor que recibe conexiones de clientes que le envían una frase leída desde el teclado y les contesta con la misma frase.

2. Realiza un barrido o escaneo por tu propia subred cercana y contabiliza cuantos servicios de SSH están funcionando en el puerto 22 y cuantos están funcionando en otros puertos.