UD6 – Python para sysadmins

ÍNDICE

1. INTRODUCCIÓN 3

2. PREPARACIÓN DEL SISTEMA 4

2.1. Instalación de Python en Windows 4

2.2. Instalación de Python en GNU-Linux 5

2.3. Entornos virtuales 6

2.4. Entornos de desarrollo 7

2.5. Funcionamiento de Python 8

3. RESUMEN DEL LENGUAJE 9

3.1. Sintaxis 9

3.2. Funciones y ayuda 10

3.3. Variables y tipos de datos 12

3.4. Estructuras de control 15

3.5. Colecciones: listas, tuplas y diccionarios 17

3.6. Funciones 20

3.7. Excepciones 21

3.8. Ficheros: json, csv y xml 22

4. ORIENTACIÓN A OBJETOS 27

4.1. Clases y objetos 27

4.2. Herencia 29

5. MÓDULOS Y PAQUETES 30

5.1. Módulos 30

5.2. Paquetes 32

6. MÓDULOS DE SISTEMAS 33

6.1. os 33

6.2. subprocess 34

6.3. shutil 36

6.4. sys 37

6.5. re 38

7. PERSISTENCIA 40

7.1. Shelves 40

7.2. sqlite3 42

8. ENTORNO GRÁFICO 44

9. DISTRIBUCIÓN DE PROGRAMAS 48

10. PRÓXIMOS PASOS 50

1. INTRODUCCIÓN

Un administrador de sistemas, o **sysadmin**, necesita automatizar muchas tareas y procesos de administración. Tradicionalmente, para automatizar estas tareas, se han utilizado los lenguajes de script disponibles en los sistemas operativos, como por ejemplo Bash o PowerShell.

La ventaja que ofrece **Python** para un administrador de sistemas es que es un lenguaje **multiplataforma**, es decir, que permite escribir programas o scripts que se ejecuten de la misma forma en distintos sistemas. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo para la programación **orientada a objetos**. La elegante sintaxis de Python y su **tipado dinámico**, junto con su naturaleza **interpretada**, hacen de éste un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas.

Algunas de las ventajas que ofrece Python a los administradores son las siguientes:

- El trabajo con ficheros es muy sencillo, tanto con ficheros TXT, JSON, XML, etc..
- El trabajo con expresiones regulares también es muy sencillo.
- Existen distintos módulos que ofrecen funciones para trabajar a nivel de sistema operativo, como “os”, “shutil”, “subprocess”, etc..
- Es muy sencillo crear scripts parametrizados. Normalmente cuando creamos un script, los datos de entrada no se introducen por el teclado, normalmente se introducen como parámetro en la terminal.
- Tenemos distintos módulos como “Fabric”, que nos permite ejecutar instrucciones en servidores remotos.

Python permite escribir programas compactos y legibles. Los programas en Python son típicamente más cortos que sus programas equivalentes en C, C++ o Java por varios motivos:

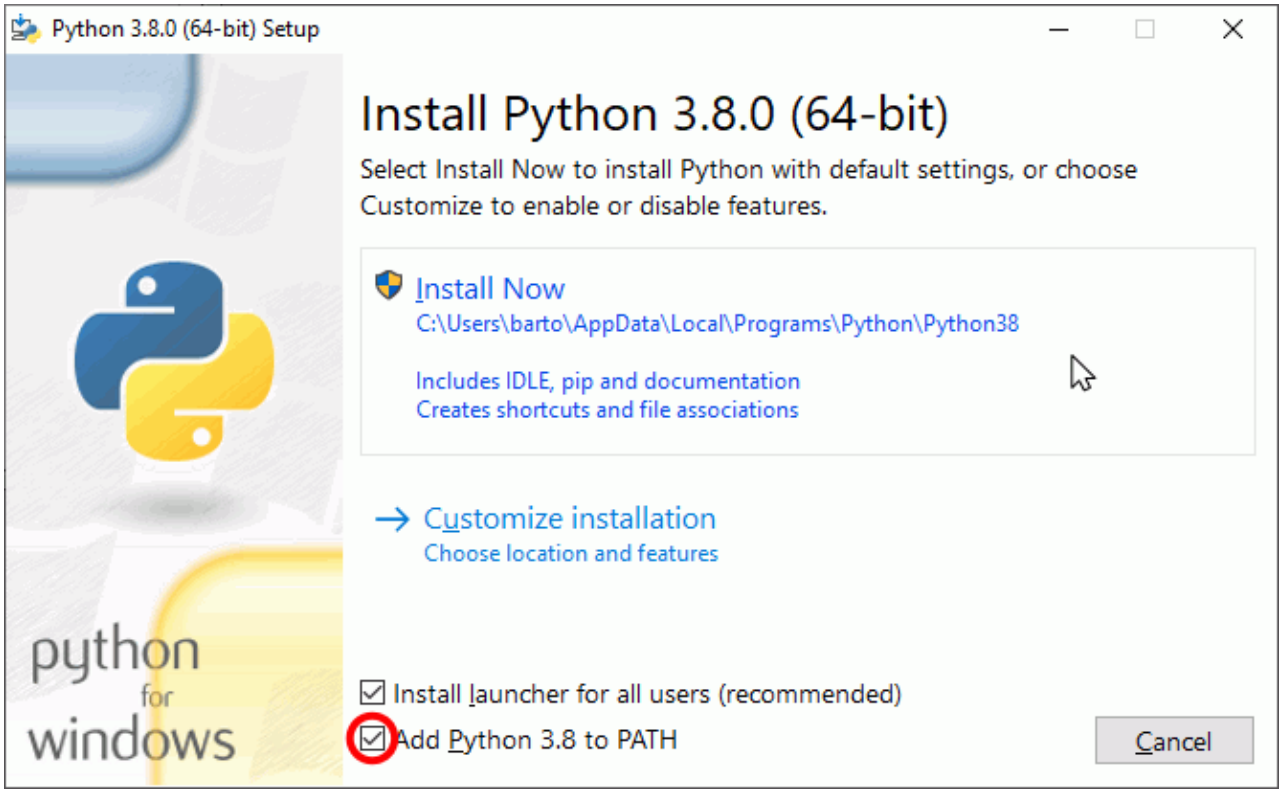
- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción.
- La agrupación de instrucciones se hace por sangría en vez de llaves de apertura y cierre.
- No es necesario declarar variables ni argumentos.

2. PREPARACIÓN DEL SISTEMA

2.1. Instalación de Python en Windows

Podemos descargar Python desde su página oficial https://www.python.org/, una vez descargado el instalador, haga doble clic en él para iniciar la instalación.

La primera pantalla permite seleccionar las opciones de instalación haciendo clic en "Customize installation", aunque en principio, no es necesario modificarlas. Tan sólo se aconseja marcar la casilla "**Add Python to PATH**" para poder ejecutar programas desde la línea de comandos. Para continuar, haga clic en "Install Now".



A lo largo de su historia, Python ha utilizado varios sistemas para la distribución de módulos:

- distutils: fue el primer sistema incluido en la biblioteca estándar en 1998 y sigue siendo la base de los sistemas posteriores. Desde Python 2.7, no se recomienda usarlo directamente.
- setuptools: fue el sistema introducido en 2004.
- pip: es el instalador "oficial" actual. Desde 2014 (Python 3.4), el módulo pip se instala junto con Python.

Aunque pip se instala con Python, pip se desarrolla de forma independiente, así que es conveniente actualizar pip cuando se publica una nueva versión.

Para saber la versión de pip instalada, ejecute en una ventana de terminal la orden:

```
pip --version
```

Para saber los paquetes instalados con pip, ejecute en una ventana de terminal la orden:

```
pip list
```

Para actualizar pip, ejecute en una ventana de terminal la orden:

```
python -m pip install --upgrade pip
```

2.2. Instalación de Python en GNU-Linux

Las últimas distribuciones de GNU/Linux suelen venir con python3 instalado, para comprobar la versión instalada podemos ejecutar:

```
python3 --version
```

Si no tenemos python3 instalado podemos instalarlo con el siguiente comando:

```
sudo apt install python3
```

Para tener el gestor de módulo pip, necesitaremos instalar el paquete “python3-pip” usando el siguiente comando:

```
sudo apt install python3-pip
```

Podemos verificar la versión de pip3 instalada con:

```
pip3 --version
```

2.3. Entornos virtuales

Un entorno virtual de Python **es un ambiente creado con el objetivo de aislar recursos** como librerías y entorno de ejecución, del sistema principal o de otros entornos virtuales.

```
sudo pip3 install virtualenv
```

Dentro del directorio donde vayamos a crear nuestro proyecyto:

```
virtualenv env --python=python3
```

La ejecución del comando anterior crea el directorio env/ con la siguiente estructura:

```
env/
  bin/
  include/
  lib/
    site-packages/
```

En el directorio “bin/” se encuentran los ejecutables necesarios para interactuar con el entorno virtual. En el directorio “include/” se encuentran algunos archivos de cabecera de C, cuya extensión es *.h, necesarios para compilar algunas librerías. En el directorio “lib/” se encuentra una copia de la instalación de Python así como un directorio llamado “site-packages/” donde se almacenan los paquetes Python instalados para el entorno virtual.

Activar entorno virtual:

```
source env/bin/activate
```

Una vez dentro del entorno virtual todas las librerías que instalemos solo estarán disponibles en este entorno, no afectando al resto del sistema:

```
(env)$ pip3 install Django
```

Desactivar entorno virtual:

```
deactivate
```

Si usamos git es recomendable desactivar la sincronización del entrono virtual añadiendo al fichero .gitignore el directorio env.

2.4. Entornos de desarrollo

Para programar en Python solo necesitamos un editor de textos, pero para ser más productivos es recomendable utilizar un buen IDE o un editor avanzado, algunos de los más utilizados son los siguientes:

Editores de texto:

- Atom: https://atom.io
- Sublime Text: https://www.sublimetext.com
- Visual Studio Code: https://code.visualstudio.com

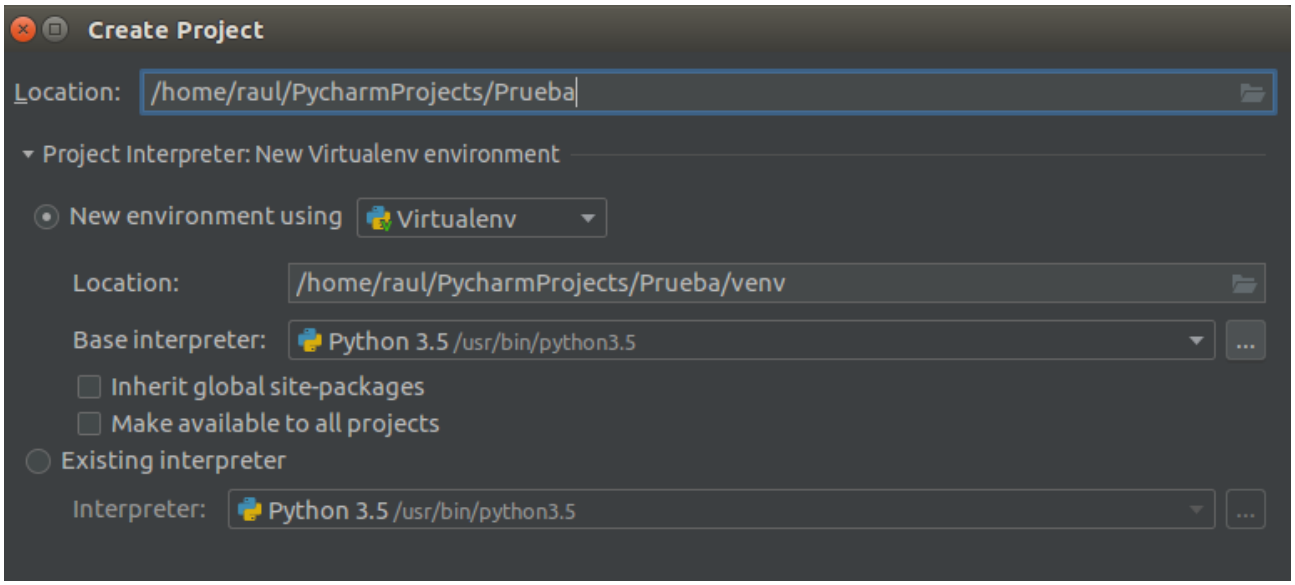
IDEs:

- PyCharm: https://www.jetbrains.com/pycharm
- Eclipse (con el plug-in Pydev): https://www.eclipse.org

El objetivo de estos apuntes no es aprender a utilizar un IDE y cada uno pude utilizar el que considere más adecuado o con el que se sienta más comodo. Por ejemplo, en Ubuntu, podemos instalar **Pycharm** como paquete snap:

```
sudo snap install pycharm-community --classic
```

Desde Pycharm ya podemos crear un nuevo proyecto seleccionando “File –> New Proyect” y establecer el entorno de trabajo deseado (nombre del proyecto, entrono virtual e interprete de python):



2.5. Funcionamiento de Python

Python ejecuta nuestro código línea por línea. Por cada línea de código estas son las acciones que se realizan:

1. Analizar (parse en inglés) el código comprobando que formato y la sintaxis es correcta, es decir, que cumplen las normas establecidas para el lenguaje de programación.
2. Traducir el código a bytecode (instrucciones que nuestra máquina puede ejecutar).
3. Enviar el código para su ejecución a la Python Virtual Machine(PVM), donde el código es ejecutado.

La **consola de Python** nos permite ejecutar código escrito en Python directamente, sin tener que escribir el código previamente en ningún fichero. Para entrar en la consola de Python, abre una consola (también conocida como terminal o shell) del sistema operativo en el que te encuentres y escribe "*python*". Esto nos mostrará la versión de Python instalada y dará comienzo a la consola de Python donde podemos ejecutar directamente las instrucciones.

El uso de la consola está recomendado para realizar pruebas, pero si queremos crear un programa, lo recomendable es escribir las instrucciones en uno o varios ficheros con extensión .py, utilizando cualquier editor de texto o IDE. Se puede ejecutar posteriormente el programa con "*python nombre_fichero.py*".

También podemos ejecutar directamente el fichero utilizando en la primera línea el shebang, donde se indica el ejecutable que vamos a utilizar, por ejemplo en Linux:

```
#!/usr/bin/python3
```

También podemos usar el programa env para preguntar al sistema por la ruta del interprete de python:

```
#!/usr/bin/env python
```

Por supuesto tenemos que dar permisos de ejecución al fichero y ya es posible ejecutarlo como cualquier otra aplicación del sistema:

```
chmod +x fichero.py
./fichero.py
```

3. RESUMEN DEL LENGUAJE

3.1. Sintaxis

La estructura de un programa python tienen las siguientes características principales:

- Las instrucciones acaban en un carácter de “salto de línea”.
- El punto y coma “;” se puede usar para separar varias sentencias en una misma línea, pero no se aconseja su uso.
- Una línea empieza en la primera posición, si tenemos instrucciones dentro de un bloque de una estructura de control de flujo habrá que hacer una identación.
- La identación se puede hacer con espacios y tabulaciones pero ambos tipos no se pueden mezclar. Se recomienda usar 4 espacios como identación.
- La barra invertida “\” al final de línea se emplea para dividir una línea muy larga en dos o más líneas.
- Las expresiones entre paréntesis “( )”, llaves “{ }” y corchetes “[ ]” separadas por comas “,” se pueden escribir ocupando varias líneas.
- Cuando el bloque a sangrar sólo ocupa una línea ésta puede escribirse después de los dos puntos.
- Se utiliza el carácter # para indicar los comentarios.

Ejemplo:

```
#!/usr/bin/env python3

# Sangrado con 4 espacios
edad = 23

if edad >= 18:
    print("Es mayor de edad")
else:
    print("Es menor de edad")
```

3.2. Funciones y ayuda

La documentación oficial de Python3 está en https://docs.python.org/3/index.html. Las siguientes funciones y algunos elementos comunes del lenguaje están definidos en el módulo built-in: https://docs.python.org/3/library/functions.html#built-in-funcs

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Algunos ejemplos de funciones:

- La entrada y salida de información se hacen con la función input y la función print
- Tenemos algunas funciones matemáticas como: abs, divmod, hex, max, min,…
- Hay funciones que trabajan con caracteres y cadenas: ascii, chr, format, repr,…
- Además tenemos funciones que crean o convierten a determinados tipos de datos: int, float, str, bool, range, dict, list, …

Un función fundamental cuando queremos obtener información sobre los distintos aspectos del lenguaje es `help`. Podemos usarla al entrar en una sesión interactiva ejecutando “python3” y dentro del idle:

```
>>> help( )
```

O pidiendo ayuda de un termino determinado, por ejemplo:

```
>>> help(print)
```

La lectura de datos en Python se realiza con la función **input()**, que permite leer por teclado información. Devuelve una cadena de caracteres y puede tener como argumento una cadena que se muestra en pantalla.

La función **print()** permite escribir en la salida estándar. Podemos indicar varios datos a imprimir, que por defecto serán separado por un espacio, y por defecto se termina con un carácter salto de línea \n. Podemos también imprimir varias cadenas de texto utilizando la concatenación.

Ejemplo de entada y salida:

```
#!/usr/bin/env python3
# Ejemplo de entrada y salida estándar
nombre = input ("¿Cómo te llamas? ")
print("Hola",nombre)
```

En la versión 3.6 de Python se introdujeron las llamadas **f-strings** , una forma más cómoda y directa para insertar variables y expresiones en cadenas. Permiten introducir cualquier variable o expresión dentro de un string incluyendo la variable entre llaves { }. El ejemplo anterior se podría haber realizado de la siguiente forma:

```
#!/usr/bin/env python3
# Ejemplo de entrada y salida estándar con f-string
nombre = input ("¿Cómo te llamas? ")
print(f"Hola {nombre}")
```

3.3. Variables y tipos de datos

Las variables permiten almacenar datos del programa. Estas serán de un tipo u otro en función de la información que se guarde en ellas.

El nombre de una variable se conoce como identificador, y deberá cumplir las siguientes reglas:

- Comenzar con una letra o un guión bajo.
- El resto del nombre estará formado por letras, números o guiones bajos.
- Los nombres de las variables son case sensitive, es decir, diferencia mayúsculas de minúsculas.
- Existen una serie de palabras reservadas que no se pueden utilizar (def, global, return, if, for,...).

Algunas de las **recomendaciones** respecto a los nombres de las variables están recogidas en la Guía oficial de Estilos PEP8 de Python https://www.python.org/dev/peps/pep-0008. Entre las más habituales encontramos las siguientes:

- Utilizar nombres descriptivos, en minúsculas y separados por guiones bajos si fuese necesario: longitud , mi_edad , velocidad_anterior ,...
- Escribir las constantes en mayúsculas: MI_CONSTANTE , NUMERO_PI , ...
- Antes y después del signo = , debe haber uno (y solo un) espacio en blanco.

Los **tipos de datos** se pueden resumir en esta tabla, con inmutable se indica si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución:

Tipo	Clase	Notas	Ejemplo
str	Cadena	Inmutable	'Cadena'
unicode	Cadena	Versión Unicode de str	u'Cadena'
list	Secuencia	Mutable, puede contener objetos de diversos tipos	[4.0, 'Cadena', True]
tuple	Secuencia	Inmutable, puede contener objetos de diversos tipos	(4.0, 'Cadena', True)
set	Conjunto	Mutable, sin orden, no contiene duplicados	set([4.0, 'Cadena', True])
frozenset	Conjunto	Inmutable, sin orden, no contiene duplicados	frozenset([4.0, 'Cadena', True])

dict	Mapping	Grupo de pares clave:valor	{'key1': 1.0, 'key2': False}
int	Número entero	Precisión fija, convertido en *long* en caso de overflow.	42
long	Número entero	Precisión arbitraria	42L ó 456966786151987643L
float	Número decimal	Coma flotante de doble precisión	3.1415927
complex	Número complejo	Parte real y parte imaginaria *j*.	(4.5 + 3j)
bool	Booleano	Valor booleano verdadero o falso	True o False

La función ***type()*** nos devuelve el tipo de dato de un objeto dado. Por ejemplo:

```
>>> type(5)
        <class 'int'>
>>> type(5.5)
        <class 'float'>
>>> type([1,2])
        <class 'list'>
```

La función ***isinstance()***, devuelve “True” si el objeto indicado es del tipo indicado, en caso contrario devuelve “False”.

```
>>> isinstance(5.5,float)
        True
```

Operadores aritméticos para tipos de datos numéricos:

- +: Suma dos números
- -: Resta dos números
- *: Multiplica dos números
- /: Divide dos números, el resultado es float.
- //: División entera
- %: Módulo o resto de la división
- **: Potencia

Asignaciones aumentadas, la variable se modifica a partir de su propio valor:

- a += b: igual que a = a + b
- a -= b: igual que a = a - b
- a *= b: igual que a = a * b
- a /= b: igual que a = a / b
- a **= b: igual que a = a ** b
- a //= b: igual que a = a // b
- a %= b: igual que a = a % b

Operadores a nivel de bit:

- *x | y: x OR y*
- *x ^ y: x XOR y*
- *x & y: a AND y*
- *x << n: Desplazamiento a la izquierda n bits.*
- *x >> n: Desplazamiento a la derecha n bits.*
- *~x: Devuelve los bits invertidos*

Conversión de tipos:

- *int(x): Convierte el valor a entero.*
- *float(x): Convierte el valor a float.*
- *str(x): Convierte el valor a string.*
- *bool(x): Convierte a booleano, el valor vacío o nulo es False, el resto True.*

Los operadores relacionales devuelven como resultado un booleano: True o False:

- *a>b: Mayor que: True si a es mayor que b*
- *a<b: Menor que: True si a es menor que b*
- *a == b: Igual: True si a y b son iguales*
- *a != b: Distinto: True si a y b son distintos*
- *a >= b: Mayor o igual: True si a es igual o mayor que b*
- *a <= b: Menor o igual: True si a es igual o menor que b*

Los operadores lógicos evalúan valores devolviendo también True o False:

- *a and b: True si a y b son True*
- *a or b: True si a o b son True*
- *not b: True si b es False*

También podemos consultar los **métodos** que acepta cada tipo de datos dentro de la consola de python, por ejemplo, para consultar los métodos disponibles para formatear cadenas podemos visualizar todos los métodos con “dir(strt)” y luego consultar con más detalle el método con “help(str.nombre_metodo)”.

3.4. Estructuras de control

Las estructuras de control se utilizan para ejecutar bloques de código en función de condiciones.

Condición if-else

Se evalúa la condición especificada en la sentencia **if** y en caso de cumplirse se ejecutará el bloque de código indentado. En caso de que el resultado de la condición sea False, el bloque especificado no se ejecutará. Mediante la palabra reservada **else** es posible especificar un bloque de código que se ejecute en caso de que la condición no se cumpla.

```
#!/usr/bin/env python3
# Ejemplo condición
edad = int(input ("¿Cuantos años tienes? "))
altura = int(input ("Dime tu altura en cm: "))
if ( edad >= 10 and altura >= 100 ):
   print(f"Con {edad} años y {altura}cm, puedes subir a la noria.")
else:
   print(f"Con {edad} años y {altura}cm, no puedes subir a la noria." )
```

Bucle while

La estructura while nos permite repetir un bloque de instrucciones mientras al evaluar una expresión lógica nos devuelve True. Puede tener una estructura else que se ejecutará al terminar el bucle.

```
#!/usr/bin/env python3
# Ejemplo while
contador = 0
while(contador < 5):
   # Se ejecutará mientras la variable contador sea menor a 5.
   contador = contador+1
   print("Iteración número",contador)
```

Bucle for

A diferencia de otros lenguajes de programación, en Python la sentencia FOR itera únicamente por secuencias (listas, tuplas, cadenas de caracteres,...)

```
#!/usr/bin/env python3
# Ejemplo for
notas = [4,8,2,7,2,9,3,5]
total = 0
for i in notas:
   total += i
print(f"Nota media {total/len(notas)}")
```

Función range()

La función range([inicio,] fin [, salto]) devuelve una secuencia de números. Es por ello que se utiliza de forma frecuente para iterar, si no se especifica el inicio y el salto, por defecto, se considera que valen 1:

```
#!/usr/bin/env python3
# Ejemplo for con range
print("Números impares del 1 al 20", end=': ')
for i in range(1, 20, 2):
   print(i, end=' ')
```

Control de bucles

En los bucles para detener una ejecución de forma voluntaria se utiliza la sentencia **break** y para saltar únicamente la iteración actual se utiliza la sentencia **continue**. También podemos hacer uso de la sentencia **pass** que continúa con la siguiente instrucción y se suele utilizar cuando se requiere por sintaxis una declaración pero no se quiere ejecutar ningún comando o código.

3.5. Colecciones: listas, tuplas y diccionarios

Las **listas** permiten guardar más de un elemento dentro de una variable, y además hacerlo en un orden concreto. Pueden contener un número ilimitado de elementos de cualquier tipo:

```
#!/usr/bin/env python3
# Lista vacía
lista_vacia = []

# Lista con valores
alumnos = ["Raúl", "María", "Pablo", "Carla"]

# Acceder a elementos
print(alumnos[0])    # muestra "Raúl"
print(alumnos[1])    # muestra "María"
print(alumnos[2])    # muestra "Pablo"
print(alumnos[-1])    # muestra "Carla"

# Cambiar un elemento
alumnos[0] = "Pepe"
```

Los **métodos** más utilizados con las listas son los siguientes:

- alumnos.append("Pepe"): Inserta “Pepe” al final de la lista.
- alumnos.insert(0,"Laura"): Inserta “Laura” en la posición 0.
- alumnos.remove("Laura"): Elimina la primera aparición de “Laura” de la lista.
- alumnos.pop(): Elimina el último elemento de la lista.
- alumnos.pop(3): Elimina el cuarto elemento de la lista.
- alumnos.clear(): Elimina todos los elementos de la lista.
- alumnos.index("María"): Devuelve el índice de la primera aparición de “María”.
- alumnos.sort(): Ordena la lista (los elementos deben ser comparables).
- sorted(alumnos): Devuelve una copia de la lista ‘alumnos’ ordenada.
- alumnos.reverse(): Ordena la lista en orden inverso.
- alumnos.copy(): Devuelve una copia de la lista.
- alumnos.extend(otra_lista): Fusiona las dos listas.

Las **tuplas** son listas inmutables. Es decir, una vez declaradas, no se pueden realizar modificaciones sobre ellas. El acceso a sus elementos se hace de igual que con listas. Para definir una tupla se escriben los elementos entre paréntesis. Una acción típica de las tuplas es “desempaquetar” (unpack) sus valores, es decir, asignarlos a variables directamente:

```
#!/usr/bin/env python3
# Ejemplo tuplas
mi_tupla = (38, "Raúl", 72.5)    # Creamos la tupla
edad, nombre, peso = mi_tupla    # Hacemos el unpack
print(edad)    # 38
print(nombre)    # "Raúl"
print(peso)    # 72 .5
```

Un **diccionario** es un conjunto de parejas clave-valor (key-value). Es decir, se puede acceder a cada elemento a partir de su clave. Las claves tienen que ser únicas y estar formadas por un string o un número. El valor de una clave puede ser cualquier tipo de dato, incluido otro diccionario o una tupla. Se definen de la siguiente manera:

```
#!/usr/bin/env python3
# Ejemplo diccionarios
estudiante = {
   "nombre": "Manuel García",
   "edad": 19,
   "nota_media": 7.25,
   "repetidor": False
}
# Acceder al valor de una clave
edad = estudiante["edad"] # devuelve el valor de 'edad'

# Insertar o actualizar un valor:
estudiante["edad"] = 20    # actualiza el valor de 'edad'
estudiante["suspensos"] = 2    # inserta una nueva clave – valor

# Eliminar un valor
del estudiante["repetidor"];
```

Algunos de los **métodos** más utilizados con diccionarios son los siguientes:

- diccionario.keys(): Devuelve todas las claves del diccionario.
- diccionario.values(): Devuelve todos los valores del diccionario.
- diccionario.pop(clave[,<default>]): Elimina la clave del diccionario y devuelve su valor asociado. Si no la encuentra y se indica un valor por defecto, devuelve el valor por defecto indicado.
- diccionario.clear(): Vacía el diccionario.
- clave in diccionario: Devuelve True si el diccionario contiene la clave o False en caso contrario.
- valor in diccionario.values(): Devuelve True si el diccionario contiene el valor o False en caso contrario.

La forma más habitual de recorrer un diccionario es mediante la sentencia for . Al recorrer un diccionario, por defecto se iterará sobre sus claves:

```
for clave in estudiante:
    print("El valor de", clave, "es", estudiante[clave])
```

En el siguiente ejemplo vamos a recorrer un diccionario que contiene otros diccionarios:

```
#!/usr/bin/env python3
estudiantes = {
    1: {
        "nombre": "Manuel García",
        "edad": 19,
    },
    2: {
        "nombre": "Lola Pérez",
        "edad": 17,
    },
    3: {
        "nombre": "Elena González",
        "edad": 21,
    }
}
```

```
# Imprimir nombre de los estudiantes mayores de edad
for clave in estudiantes:
    if estudiantes[clave]["edad"] >= 18:
        print(estudiantes[clave]["nombre"])
```

3.6. Funciones

Una función es un grupo de sentencias que realizan una tarea concreta. Esta forma de agrupar código es una forma de ordenar nuestra aplicación en pequeños bloques, facilitando así su lectura y permitiendo reutilizar el código que contienen.

Se escribe la palabra reservada **def** seguida del nombre de la función y sus parámetros entre paréntesis. Para llamar a una función solo hay que escribir el nombre de la función seguida de los parámetros, si los hubiera, entre paréntesis. Las funciones pueden devolver un valor utilizando la palabra **return** . Una vez devuelto un valor, la función finaliza su ejecución.

```
#!/usr/bin/env python3
# Ejemplo funciones

def suma(a, b):
    """Suma 2 números pasados como parámetros"""
    resultado = a + b
    return resultado

print(suma(4, 5))
```

Es posible recibir un número desconocido de parámetros añadiendo un * en la definición de la función, por ejemplo “def suma_todo(*args)”, de esta forma args es recibido como una lista de parámetros. También se puede establecer un valor por defecto a un parámetro si esté no se pasa a la función, por ejemplo “def calcula_iva (iva = 21)”.

El **ámbito de una variable** (scope) se refiere a la zona del programa dónde una variable "existe". Fuera del ámbito de una variable no podremos acceder a su valor ni manejarla. Los parámetros y variables definidos en una función no estarán accesibles fuera de la función. A este ámbito se le conoce como **ámbito local**. Es importante mencionar que una vez ejecutada una función, el valor de las variables locales no se almacena, por lo que la próxima vez que se llame a la función, ésta no recordará ningún valor de llamadas anteriores.

Por el contrario, las variables definidas fuera de una función sí que están accesibles desde dentro de la función. Se considera que están en el **ámbito global**. No obstante, no se podrán modificar dentro de la función a no ser que estén definidas con la palabra clave global.

3.7. Excepciones

Las excepciones son errores en la ejecución de un programa que hacen que el programa termine de forma inesperada. Normalmente ocurren debido a un uso indebido de los datos. La manera de controlar las excepciones es agrupando el código con:

- try : agrupa el bloque de código en el que se pueda dar una excepción.
- catch : contiene el código a ejecutar en caso de que la excepción haya sido lanzada.
- finally (opcional): permite ejecutar un bloque de código siempre.

```
try:
    numero = int(input("Introduce un número: "))
    dividendo = 150
    resultado = dividendo / numero
    print(resultado)
except ValueError:
    print("Número inválido")
except ZeroDivisionError:
    print("No se puede dividir entre 0")
finally:
    print("Ejecutando finally antes de salir")
```

Hay algunas excepciones que son bastante comunes a la hora de programar en Python y que deberíamos contemplar en nuestros programas:

- TypeError es lanzado cuando se intenta realizar una operación o una función sobre un objeto de un tipo inapropiado.
- ValueError es lanzado cuando el argumento de una función es de un tipo inapropiado.
- NameError es lanzado cuando no utiliza un objeto que no existe.
- IndexError es lanzado al intentar acceder a un índice que no existe en un array.
- KeyError es lanzado cuando no se encuentra la clave (key).
- ModuleNotFoundError es lanzado cuando no se encuentra el módulo indicado.

3.8. Ficheros: json, csv y xml.

Con la expresión **with-as** se puede leer y escribir en ficheros de forma óptima, cerrándolos y liberando la memoria al concluir el proceso de lectura. La expresión with-as es una llamada a la función open() y la variable es la conexión con el fichero. La función open puede tener varios argumentos. Los más importantes son

```
with open("FICHERO", mode="MODO", encoding="CODIFICACIÓN") as fichero:
    BLOQUE DE INSTRUCCIONES
```

- "FICHERO" es la ruta absoluta o relativa hasta el fichero.
- "MODO" indica si el fichero se abre para leer, escribir o ambas cosas y si se trata de un fichero de texto o binario. El modo predeterminado es lectura en modo texto.
- "CODIFICACIÓN" indica el juego de caracteres del fichero: "utf-8", "cp1252", etc.

Los modos de escritura son:

- "x": únicamente crear el fichero (da error si ya existe el fichero)
- "w": escribir (crea el fichero si no existe y borra el contenido anterior del fichero)
- "a": añadir (crea el fichero si no existe, no borra el contenido existente y escribe al final del fichero)

Se puede **escribir** en el fichero con la función **print()** añadiendo el argumento file=fichero, donde fichero es la variable utilizada en la expresión with-as o con el método **write** sobre el objeto fichero. La función print() añade un salto de línea al final de la cadena añadida al fichero, pero el método write no lo hace, por lo que habrá que añadirlo explícitamente.

Los ejemplos siguientes muestran la diferencia entre cada uno de los modos de escritura (usando la función print() o el método write):

Los modos de lectura son:

- "r": únicamente leer el fichero (da error si no existe el fichero, lee desde el principio y es posible desplazarse)
- "r+": leer y escribir (da error si no existe el fichero, lee desde el principio y es posible desplazarse, pero sólo escribe al final del fichero).

Se puede **leer** en el fichero con el método **read** sobre el objeto fichero. En el siguiente ejemplo añadimos una línea en un fichero y mostramos por pantalla el contenido del fichero.

```
#!/usr/bin/env python3
# Ejemplo ficheros
ruta = "/home/raul/notas.txt"

with open(ruta, mode="a", encoding="utf-8") as fichero:
    fichero.write("Prueba de escritura\n")

with open(ruta, mode="r", encoding="utf-8") as fichero:
    print(fichero.read())
```

El módulo **json** (https://docs.python.org/3/library/json.html) nos permite gestionar ficheros con formato **JSON (Javascript Object Notation)**. La correspondecia entre JSON y Python la podemos resumir en la siguiente tabla:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Para leer datos json podemos usar el método **load()**, por ejemplo para leer desde un fichero:

```
import json

ruta = "/home/raul/datos.json"
with open(ruta, mode="r", encoding="utf-8") as fichero:
    datos = json.load(fichero)
    for persona in datos:
        print(persona["nombre"] + " pesa " + str(persona["peso"]) + " kg")
```

Para escribir datos en json podemos usar el método **dump()**, por ejemplo para escribir en un fichero:

```
import json

nuevos_datos = {"nombre": "Juan", "edad": 55, "peso": 102.4}
ruta = "/home/raul/datos.json"
with open(ruta, mode="r", encoding="utf-8") as fichero:
    datos = json.load(fichero)
    datos.append(nuevos_datos)

with open(ruta, mode="w", encoding="utf-8") as fichero:
    json.dump(datos, fichero, indent=4)
```

El módulo **csv** (https://docs.python.org/3/library/csv.html) permite trabajar con ficheros **CSV (comma-separated values)**, son un tipo de ficheros donde las filas están estructuradas con campos separados por comas (o por otro carácter).

Para leer datos csv podemos usar el método **reader()**, que devuelve un objeto que iterará sobre líneas en el archivo csv dado, cada fila puede ser tratada como una lista, pudiendo acceder a cada campo por su índice:

```
import csv

ruta = "/home/raul/datos.csv"

with open(ruta, mode="r", encoding="utf-8") as fichero:
    datos = csv.reader(fichero)
    for fila in datos:
        print(fila[0], "pesa", fila[2], "kg")
```

Para escribir datos csv podemos usar el método **writer()**, que permitirá escribir filas con la función **writerow():**

```
import csv

ruta = "/home/raul/datos.csv"

with open(ruta, mode="a", encoding="utf-8") as fichero:
    datos = csv.writer(fichero)
    datos.writerow(["María", 33, 54.6])
```

El módulo **xml.etree.ElementTree** implementa una API simple y eficiente para analizar y crear datos XML (https://docs.python.org/3/library/xml.etree.elementtree.html).

Para analizar un archivo xml, le pasamos un identificador de archivo abierto a **parse(),** que leerá los datos, analizará el XML y devolverá un objeto ElementTree. Para leer todos los nodos en orden, se usa **iter()** sobre el objeto ElementTree. Cada objeto Element tiene cuatro atributos:

- **Tag**: el nombre de la etiqueta.
- **Text**: El texto guardado dentro de la etiqueta. Este atributo es None si la etiqueta está vacía.
- **Tail**: El texto de un elemento, que está a continuación de otro elemento.
- **Attrib**: Un diccionario python que contiene los nombres y valores de los atributos del elemento.

```
import xml.etree.ElementTree as ET

ruta = "/home/raul/datos.xml"
with open(ruta, mode='r', encoding="utf-8") as fichero:
    tree = ET.parse(fichero)

for nodo in tree.iter():
    print("Etiqueta: ", nodo.tag, "con contenido:", nodo.text)
```

Tenemos a nuestra disposición distintos métodos para realizar búsquedas:

- find(): Devuelve el primer Element cuya etiqueta corresponda al criterio de búsqueda que le proporcionamos. Podemos utilizarlo a partir de un objeto ElementTree, o a partir de un elemento Element.
- findall(): Devuelve una lista de objetos Element que coinciden con el criterio de búsqueda. Podemos también utilizarlo a partir de un objeto ElementTree, o a partir de un elemento Element.
- findtext(): Devuelve el contenido del atributo text del primer elemento que coincida con el criterio de búsqueda.

4. ORIENTACIÓN A OBJETOS

4.1. Clases y objetos

Python soporta la programación orientada a objetos. Esto quiere decir que podemos definir entidades agrupando (encapsulando) sus atributos y comportamiento (métodos) en clases. La definición de una clase en Python se hace de la siguiente manera:

```
# Ejemplo clase
class Persona:
    # atributos
    nombre = "Raúl"
    edad = 39
    peso = 72.5

    # métodos
    def come(self):
        print(self.nombre + " está comiendo.")
        self.peso += 1

    def corre(self, km):
        print(self.nombre + " está corriendo.")
        if km >= 5:
            self.peso -= 2
        else:
            self.peso -= 1
```

Una clase es como una plantilla o modelo para crear a partir de ella objetos. Esta plantilla contiene la información para definir cómo serán los objetos, es decir, qué atributos y métodos tendrán.

A partir de una clase se pueden crear tantos objetos como se desee. Los objetos de una clase se conocen como instancias. Cada objeto contiene los atributos y métodos de la clase y podrá asignar a esos atributos unos valores concretos, esto se conoce como el estado de un objeto.

```
p1 = Persona()    # p1 contiene un objeto de la clase Persona
print(p1.nombre)
print(p1.edad)
print(p1.peso)
p1.come()
p1.corre(10)
print(p1.peso)
```

Una función dentro de una clase se conoce como **método**. Las clases contienen el método especial __**init**__ conocido como **constructor** y que sirve para inicializar un objeto. Al crear un objeto siempre se llama al constructor. Una diferencia importante con otros lenguajes como Java es que solo se puede definir un único constructor.

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

El parámetro self de los métodos es una referencia a la propia instancia y se utiliza para acceder a las variables que pertenecen a la clase. Si no se define un constructor, la clase hereda uno que únicamente recibe el argumento self. Ahora en la creación del objeto es necesario indicar los argumentos del constructor:

```
p1 = Persona("Raúl", 39, 72.5)
```

Los atributos definidos dentro del constructor se conocen como **atributos de instancia**, por lo tanto, los atributos definidos dentro de la clase pero fuera del constructor se conocen como **atributos de clase**. La principal diferencia es que un atributo de clase puede ser accedido aunque no existan instancias de la clase. Además, si se modifica su valor, se modificará el valor en todas las instancias existentes de dicha clase.

A diferencia de otros lenguajes de Programación Orientados a Objetos, todos los métodos y atributos en Python son **públicos**. Es decir, no es posible definir una variable como private o protected. Existe una convención de añadir como prefijo un guión bajo (_) a los atributos que consideramos como protected y dos guiones bajos (__) a las variables que consideramos private.

4.2. Herencia

La herencia es una técnica de la Programación Orientada a Objetos en la que una clase (conocida como clase hija o subclase) hereda todos los métodos y propiedades de otra clase (conocida como padre o clase base). La subclase puede añadir funcionalidades, permitiendo reutilizar código.

```
class Profesor(Persona):
    def __init__(self, nombre, edad, peso, modulos, tutor=False):
        # Llamada al constructor del padre
        Persona.__init__(self, nombre, edad, peso)
        self.modulos = modulos
        self.tutor = tutor

p1 = Profesor("Raúl", 41, 75, ["Sistemas", "Programación", "BD"])
```

Python también soporta la **herencia múltiple**, es decir, heredad métodos y atributos de más de un padre. En caso de heredar atributos o métodos con el mismo nombre, Python dará prioridad al posicionado más a la izquierda en la declaración.

5. MÓDULOS Y PAQUETES

5.1. Módulos

Cada uno de los ficheros .py que nosotros creamos se llama módulo. Los elementos creados en un módulo (funciones, clases, …) se pueden importar para ser utilizados en otro módulo. El nombre que vamos a utilizar para importar un módulo es el nombre del fichero.

```
#!/usr/bin/env python3
def cuadrado(n):
    return (n ** 2)


def cubo(n):
    return (n ** 3)


if __name__ == "__main__":
    print(cuadrado(3))
    print(cubo(3))
```

El nombre que tiene el módulo cuando es ejecutado directamente es __main__, por lo tanto si ejecutamos el programa se ejecutará el contenido dentro del if, pero además este módulo se podrá importar, y al importarlo la parte del __main__ no se tendrá en cuenta. Para importar un módulo completo tenemos que utilizar la instrucción import como en el siguiente ejemplo:

```
import potencias
print(potencias.cuadrado(2))
```

Para acceder desde el módulo donde se realizó la importación, se realiza mediante el **namespace**, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra import. Pero es posible abreviar los namespaces mediante un alias:

```
import potencias as p
print(p.cuadrado(2))
```

Para no utilizar el namespace podemos indicar los elementos concretos que queremos importar de un módulo con **from:**

```
from potencias import cuadrado
print(cuadrado(2))
```

Algunos módulos estándar son:

- copy : para copiar variables como colecciones y objetos
- collections : para utilizar las colas y más utilidades (Siguiente Apartado)
- datetime : para manejar las fechas y las horas
- doctest y unittest : para crear pruebas y hacer testing
- html, xml y json : para manejar estructuras de datos web
- pickle : para trabajar con ficheros
- math : para operaciones matemáticas
- re : para expresiones regulares
- random : para generar contenidos aleatorios y seleccionar aleatoriamente
- socket : para sistemas cliente-servidor
- sqlite3 : gestionar una base de datos relacional mediante un fichero
- sys : para manejar algunas funciones del sistema
- threading : para dividir procesos en subprocesos
- tkinter : para generar interfaces gráficas

5.2. Paquetes

Para estructurar nuestros módulos podemos crear paquetes. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado **__init__.py**. Este archivo, no necesita contener ninguna instrucción. Los paquetes, a la vez, también pueden contener otros sub-paquetes.

Para cargar un módulo ubicado en un paquete lo haremos de la siguiente forma:

```
import mi_calculadora.potencias
```

También los podemos hacer de la siguiente manera:

```
from mi_calculadora import potencias
```

También es posible importar elementos concretos de un módulo:

```
from mi_calculadora.potencias import cuadrado
```

6. MÓDULOS DE SISTEMAS

Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que también pueden ser importados. En este punto vamos a repasar las funciones principales de módulos relacionados con el sistema operativo.

6.1. os

El módulo **os** (https://docs.python.org/3/library/os.html) nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular ficheros y directorios.

Descripción	Método
Saber si se puede acceder a un archivo o directorio	os.access(path, modo_de_acceso)
Conocer el directorio actual	os.getcwd()
Cambiar de directorio de trabajo	os.chdir(nuevo_path)
Cambiar al directorio de trabajo raíz	os.chroot()
Cambiar los permisos de un archivo o directorio	os.chmod(path, permisos)
Cambiar el propietario de un archivo o directorio	os.chown(path, permisos)
Crear un directorio	os.mkdir(path[, modo])
Crear directorios recursivamente	os.mkdirs(path[, modo])
Eliminar un archivo	os.remove(path)
Eliminar un directorio	os.rmdir(path)
Eliminar directorios recursivamente	os.removedirs(path)
Renombrar un archivo	os.rename(actual, nuevo)
Crear un enlace simbólico	os.symlink(path, nombre_destino)

```
#!/usr/bin/env python3
import os
os.chdir("/home/raul/")    # Cambia al directorio /home/raul/
print (os.getcwd())   # Muestra el directorio actual
```

El módulo os también nos provee del submódulo path, **os.path,** el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios.

Descripción	Método
Ruta absoluta	os.path.abspath(path)
Directorio base	os.path.basename(path)
Saber si un directorio existe	os.path.exists(path)
Conocer último acceso a un directorio	os.path.getatime(path)
Conocer tamaño del directorio	os.path.getsize(path)
Saber si una ruta es absoluta	os.path.isabs(path)
Saber si una ruta es un archivo	os.path.isfile(path)
Saber si una ruta es un directorio	os.path.isdir(path)
Saber si una ruta es un enlace simbólico	os.path.islink(path)
Saber si una ruta es un punto de montaje	os.path.ismount(path)

6.2. subprocess

El módulo **subprocess** (https://docs.python.org/3/library/subprocess.html) nos permite generar nuevos procesos, conectarse a sus tuberías de entrada / salida / error y obtener sus códigos de retorno. Este módulo tiene la intención de reemplazar varios módulos y funciones anteriores como os.system() y os.spawnv().

La forma más sencilla de ejecutar un comando o invocar un proceso es vía la función call(), desde Python 2.4 hasta 3.4, o **run()** a partir de Python 3.5. Cuando invocamos un comando que contiene espacios, especialmente en distribuciones GNU-Linux, es recomendable pasar una lista en lugar de una cadena para evitar que sea interpretado erróneamente.

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

Los canales de entrada y salida estándar para el proceso iniciado por run() se vinculan a la entrada y salida del padre. Eso significa que el programa que llama no puede capturar la salida del comando. Por este motivo se pasa PIPE para los argumentos stdout y stderr para capturar la salida para su posterior procesamiento. El parámetro universal_newlines controla cómo se usan las tuberías. Si es falso, las lecturas de tubería devuelven objetos de bytes y es posible que necesiten decodificarse (por ejemplo, line.decode (“utf-8”)) para obtener una cadena, si es verdadero, Python hace la decodificación automáticamente.

```
import subprocess
procesos = subprocess.run(["ps", "-aux"], stdout=subprocess.PIPE,\
                universal_newlines=True)
print(procesos.stdout)
```

Como vemos, subproccess.run() devuelve un objeto **subprocess.CompletedProcess** representando el resultado de un proceso terminado:

- args: son los argumentos usados para lanzar el proceso, en relación directa con el objeto pasado al argumento args de subprocess.run.
- returncode: es simplemente el estado de salida del proceso. Típicamente 0 indica que el proceso se ejecutó correctamente, aunque lógicamente es algo que es definido por el propio proceso.
- stdout: captura la salida estándar del subproceso o None si no se ha capturado. Es una cadena de bytes o una cadena de texto (str) si si se pasó True al argumento universal_newlines=True de subprocess.run(). Para poder capturar la salida es necesario redirigirla mediante una tubería al llamar a subprocess.run() mediante stdout=subperocess.PIPE
- stderr: exactamente lo mismo que el atributo anterior, solo que para stderr.
- check_returncode(): igual que cuando se define check como True en el constructor de subprocess.run(), este método cuando es llamado lanza una excepción CalledProcessError si el código de retorno del subproceso no es cero.

6.3. shutil

El módulo **shutil** (https://docs.python.org/3/library/shutil.html) ofrece una serie de operaciones de alto nivel en archivos y colecciones de archivos. En particular, se proporcionan funciones que admiten la copia y eliminación de archivos. Para operaciones en archivos individuales, se utiliza también el módulo os.

Descripción	Método
Copia un fichero completo o parte	shutil.copyfileobj(fsrc, fdst[, length])
Copia el contenido sin metadatos	shutil.copyfile(src, dst, *, follow_symlinks=True)
Copia los permisos de un archivo a uno destino	shutil.copymode(src, dst, *, follow_symlinks=True)
Copia los permisos, la fecha-hora del último acceso, la fecha-hora de la última modificación y los atributos de un archivo a un archivo destino	shutil.copystat(src, dst, *, follow_symlinks=True)
Copia un archivo (sólo datos y permisos)	shutil.copy(src, dst, *, follow_symlinks=True)
Copia archivos (datos, permisos y metadatos)	shutil.move(src, dst, copy_function=copy2)
Obtiene información del espacio total, usado y libre, en bytes	shutil.disk_usage(path)
Obtener la ruta de un archivo ejecutable	shutil.chown(path, user=None, group=None)
Saber si una ruta es un enlace simbólico	shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)

```
import shutil


disco = shutil.disk_usage("/home")
total_bytes, usado_bytes, libre_bytes = disco    # unpack


GiB = 2 ** 30    # GiB == gibibyte
GB = 10 ** 9   # GB == gigabyte


print('Total: {:6.2f} GB  {:6.2f} GiB'.format(total_bytes / GB, total_bytes / GiB))
print('Usado : {:6.2f} GB  {:6.2f} GiB'.format(usado_bytes / GB, usado_bytes / GiB))
print('Libre : {:6.2f} GB  {:6.2f} GiB'.format(libre_bytes / GB, libre_bytes / GiB))
```

6.4. sys

El módulo **sys** (https://docs.python.org/3/library/sys.html) proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete y a funciones que interactúan directamente con el intérprete.

Algunas variables definidas en el módulo:

Variable	Descripción
sys.argv	Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar python modulo.py arg1 arg2, retornará una lista: ['modulo.py', 'arg1', 'arg2']
sys.executable	Retorna el path absoluto del binario ejecutable del intérprete de Python
sys.platform	Retorna la plataforma sobre la cuál se está ejecutando el intérprete
sys.version	Retorna el número de versión de Python con información adicional

Y algunos métodos:

Método	Descripción
sys.exit()	Forzar la salida del intérprete
sys.getdefaultencoding()	Retorna la codificación de caracteres por defecto

Podemos enviar información (argumentos) a un programa cuando se ejecuta como un script, por ejemplo:

```
#!/usr/bin/env python3

import sys


print("Has instroducido", len(sys.argv), "argumentos.")

suma = 0

for i in range(1, len(sys.argv)):

   suma = suma + int(sys.argv[i])


print("La suma es ", suma)
```

6.5. re

El módulo **re** (https://docs.python.org/3/library/re.html) proporciona operaciones de coincidencia de expresiones regulares similares a las que se encuentran en Perl.

Los métodos básicos que proporciona re son los siguientes:

- re.search: busca un patrón en otra cadena.
- re.match: busca un patrón al principio de otra cadena.
- re.split: divide una cadena a partir de un patrón.
- re.sub: sustituye todas las coincidencias en una cadena.
- re.findall: busca todas las coincidencias en una cadena.

Cuando hay coincidencias, Python devuelve un Objeto de coincidencia (salvo en el método findall() que devuelve una lista). Este Objeto de coincidencia también tiene sus propios métodos que nos proporcionan información adicional sobre la coincidencia; éstos métodos son:

- group(): El cual devuelve el texto que coincide con la expresion regular.
- start(): El cual devuelve la posición inicial de la coincidencia.
- end(): El cual devuelve la posición final de la coincidencia.
- span(): El cual devuelve una tupla con la posición inicial y final de la coincidencia.

Los **metacaracteres** son caracteres especiales que son la esencia de las expresiones regulares. Algunos de los más utilizados son los siguientes:

- ^: inicio de línea.
- $: fin de línea.
- \A: inicio de texto.
- \Z: fin de texto.
- .:cualquier carácter en la línea.
- \w: un carácter alfanumérico (incluye "_").
- \W: un carácter no alfanumérico.
- \d: un carácter numérico.
- \D: un carácter no numérico.

- \s: cualquier espacio (lo mismo que [\t \n \r \f]).
- \S: un no espacio.
- *: cero o más, similar a {0,}.
- +: una o más, similar a {1,}.
- ?: cero o una, similar a {0,1}.
- {n}: exactamente n veces.
- {n,}: por lo menos n veces.
- {n,m}: por lo menos n pero no más de m veces.
- *?: cero o más, similar a {0,}?.
- +?: una o más, similar a {1,}?.
- ??: cero o una, similar a {0,1}?.
- {n}?: exactamente n veces.
- {n,}?: por lo menos n veces.
- {n,m}?: por lo menos n pero no más de m veces.
- |: alternativas.

Algunos ejemplos de las expresiones regulares más utilizadas:

- Validar formato e-mail: \b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,6}\b
- Validar una URL: ^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$
- Validar una dirección IP: ^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
- Validar una fecha dd/mm/yyyy: ^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)$

Para crear un objeto **patrón**, debemos importar el módulo re y utilizamos la función **compile()**:

```
import re

ip_destino = input("Introduce IP de destino: ")
patron_ip = re.compile('^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$')

test = patron_ip.match(ip_destino)
if test:
    print("IP válida")
else:
    print("IP inválida")
```

7. PERSISTENCIA

7.1. Shelves

El módulo **shelve** (https://docs.python.org/3/library/shelve.html) puede usarse como una opción simple de almacenamiento persistente para objetos de Python cuando no se requiere una base de datos relacional. Se accede al estante mediante llaves, igual que con un diccionario. Los valores son serializados y se escriben en una base de datos creada y gestionada por **dbm**.

La forma más simple de usar shelve es a través de la clase DbfilenameShelf. Utiliza dbm para almacenar los datos. La clase puede ser utilizada directamente, o llamando a **shelve.open(),** si la base de datos no existe la creará.

```
import shelve

with shelve.open('mis_datos.db') as s:
    s['1'] = {
        'Nombre': "Raúl",
        'edad': 39,
        'peso': 73.5
    }
```

Para volver a acceder a los datos, abre el estante y utilízalo como un diccionario.

```
import shelve
with shelve.open('mis_datos.db') as s:
    dato = s['1']
print(dato)
```

El módulo dbm no admite la escritura de múltiples aplicaciones a la misma base de datos al mismo tiempo, pero soporta clientes de solo lectura concurrentes. Si un cliente no va a modificar el estante, se abre la base de datos de solo lectura con flag='r'.

Para capturar automáticamente los cambios en los objetos volátiles almacenados en el estante, ábrelo con la escritura habilitada. El flag writeback causa que el estante recuerde todos los objetos recuperados de la base de datos usando un caché en memoria. Cada objeto de caché también se escribe de nuevo en la base de datos cuando el estante se cierra

```
import shelve
with shelve.open('mis_datos.db', writeback=True) as s:
    s['1']['peso'] = 74.2

with shelve.open('mis_datos.db', flag="r") as s:
    print(s['1']['peso'])
```

7.2. sqlite3

El módulo **sqlite3** (https://docs.python.org/3/library/sqlite3.html) implementa una interfaz compatible de Python DB-API 2.0 a SQLite, una base de datos relacional en proceso. SQLite está diseñada para integrarse en aplicaciones, en lugar de utilizar un programa de servidor de base de datos como MySQL, PostgreSQL u Oracle. Es rápida, rigurosamente probada y flexible, lo que la hace adecuada para crear prototipos y despliegue de producción para algunas aplicaciones.

Los tipos de datos en SQLITE se agrupan por afinidad en 5 tipos de datos básicos y son estos:

- text: variable de tipo texto que se almacena en formato utf, aquí tenemos agrupados los diferentes tipos de datos sql para las variables de texto: character(20), varchar(255), varying character(255), nchar(55), native character(70), nvarchar(100), text y clob.
- numeric: numeric, decimal(10,5), boolean, date y datetime
- integer: es un entero con signo que se almacena con un longitud que va en función del tipo de dato definido: integer, tinyint, smallint, mediumint, bigint, unsigned big int, int2 y int8
- real: es un dato de tipo float, sus diferentes versiones variarán en la precisión: real, double, double precision y float
- blob: los datos se almacean en el mismo formato en que se introducen.

Para asegurar la integridad de los datos también pueden existir campos declarados como **PRIMARY KEY** (clave primaria) o F**OREIGN KEY** (clave ajena).

Para utilizar SQLite3 en Python hay que importar el módulo sqlite3 y luego crear un objeto de conexión para conectarnos a la base de datos. Este nos permitirá ejecutar las sentencias SQL. Un objeto de conexión se crea utilizando la función **connect ()**, si no existe la base de datos se creará.

```
import sqlite3
con = sqlite3.connect('mis_datos.db')
```

Para ejecutar sentencias de SQLite en Python, se necesita un objeto cursor. Puedes crearlo utilizando el método **cursor()**, que es un método del objeto de conexión. Para ejecutar sentencias de SQLite3, primero se establece una conexión y luego se crea un objeto cursor utilizando el objeto de conexión de la siguiente manera:

```
cursorObj = con.cursor()
```

Ahora podemos usar el objeto cursor para llamar al método **execute()** para ejecutar cualquier sentencia SQL (INSERT, UPDATE, SELECT,..). El método **commit()** guarda todos los cambios que hacemos.

```
cursorObj.execute('CREATE TABLE Alumnos (id integer PRIMARY KEY,
                    nombre text, curso text, edad integer)')

con.commit()
```

Para obtener los datos de una base de datos, ejecutaremos la sentencia SELECT y luego usaremos el método **fetchall()** del objeto cursor para almacenar los valores en una variable que contiene cada fila obtenida como una lista.

```
import sqlite3
con = sqlite3.connect('mis_datos.db')
cursorObj = con.cursor()

cursorObj.execute('SELECT id, nombre FROM Alumnos WHERE edad >= 18')

filas = cursorObj.fetchall()

for fila in filas:
    print(fila)
```

8. ENTORNO GRÁFICO

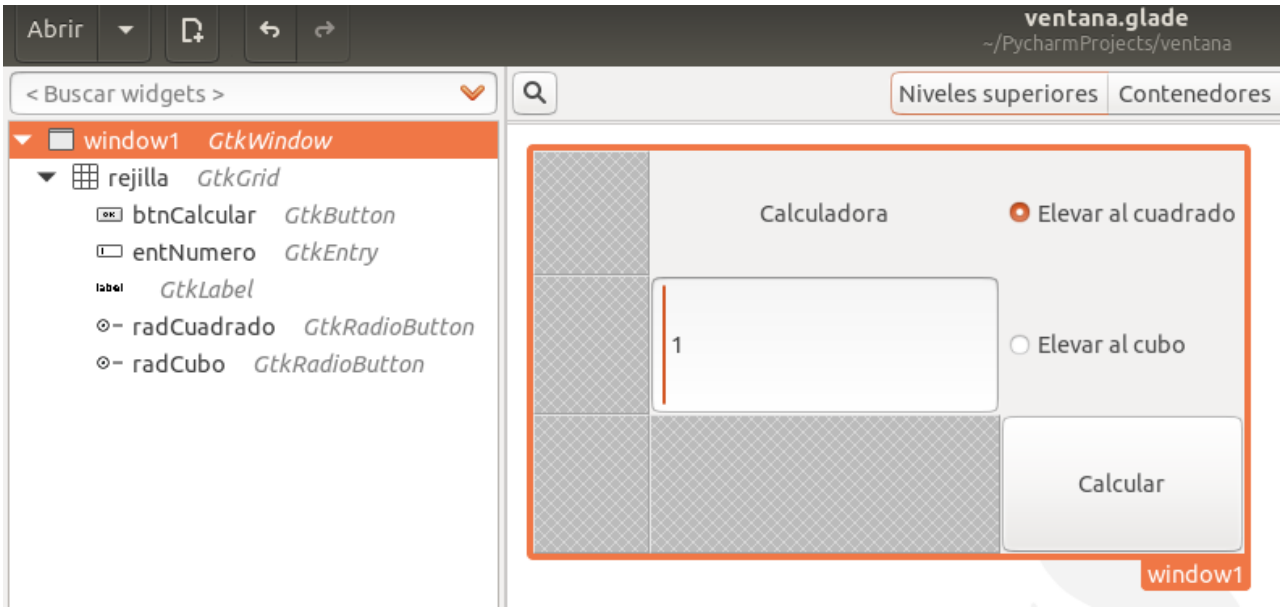
Existen múltiples y de muy variados tipos librerías para desarrollar aplicaciones gráficas. Entre las más populares se encuentran **Qt**, **GTK+** y **wxWidgets.** Python incluye en la librería estándar una cuarta llamada **Tcl/Tk**. Los módulos de Python correspondientes a estas librerías son los siguientes.

- **Tkinter** (https://docs.python.org/3/library/tkinter.html): Las compilaciones estándar de Python incluyen una interfaz orientada a objetos para el conjunto de widgets Tcl/Tk, llamada tkinter. Esta es probablemente la más fácil de instalar y usar, es totalmente portable para las plataformas Mac OS X, Windows y Linux. Es la generalmente recomendada para proyectos triviales y/o de aprendizaje, le faltan elementos gráficos y la apariencia no es nativa.
- **WxPython** (https://www.wxpython.org/): es una adaptación de la biblioteca gráfica wxWidgets para ser usada en el lenguaje de programación Python y de esta forma facilitar el desarrollo de aplicaciones gráficas. Es una biblioteca portable y gratuita escrita en C++ que proporciona una apariencia nativa en Mac OS X, Windows y Linux. Según el creador de Python hubiera sido la interfaz por defecto si no hubiese existido TK en primer lugar.
- **PyGObject** (https://pygobject.readthedocs.io/en/latest/): es un binding de Python que proporciona enlaces para bibliotecas basadas en GObject como GTK, GStreamer, WebKitGTK, GLib, GIO y muchas más. Es compatible con Linux, Windows y MacOS X y se puede usar de forma libre ya que tiene licencia LGPLv2.1 +. Si se desea escribir una aplicación Python para GNOME o una aplicación Python GUI usando GTK, entonces PyGObject es la más recomendada.
- **PyQt** (https://riverbankcomputing.com/software/pyqt/intro): es un binding de la biblioteca gráfica Qt para el lenguaje de programación Python. La biblioteca está desarrollada por la firma británica Riverbank Computing y está disponible para Windows, GNU/Linux y Mac OS X bajo diferentes licencias. Para aplicaciones libres usa licencia GPL y para aplicaciones cerradas debemos pagar por usarla.
- **PySide2** (https://wiki.qt.io/PySide): es el binding oficial de Qt distribuido bajo una licencia libre bajo el proyecto Qt for Python, que proporciona acceso al marco completo Qt5, está disponible para Windows, GNU/Linux y Mac OS X con licencias LGPLv3 / GPLv3. Pero hay que tener en cuenta la licencia de Qt.

Vamos a realizar como ejemplo de desarrollo de un programa sencillo con interfaz gráfica, una aplicación con **GTK+ 3** (https://python-gtk-3-tutorial.readthedocs.io/en/latest/) utilizando para diseñar el entrono gráfico **Glade** (https://glade.gnome.org/)**.**

```
sudo apt install python3-gi glade
sudo apt install python3-gi python3-gi-cairo gir1.2-gtk-3.0
```

Como ejemplo vamos a crear una ventana con id “ventana1” activando la señal “destroy” asociada al manipulador “cerrar”, una rejilla para distribuir el resto de elementos, una entrada con id “entNumero”, dos botones radio en grupo con ids “radCubo” y “radCuadrado” y un botón con id “btnCalcular” con la señal clicked asociada al manipulador “cick_btnCalcular”.



Las interfaces de usuario diseñadas en Glade se guardan como XML, con el objeto **GtkBuilder** de GTK+, las aplicaciones pueden cargarlas dinámicamente según sea necesario.

Vamos a dividir el programa en 2 ficheros, “potencias.py” que realizará la lógica del programa y “calculadora.py” que manejará el entrono gráfico que se encuentra en el fichero “ventana.glade”.

El contenido de potencias.py es el siguiente:

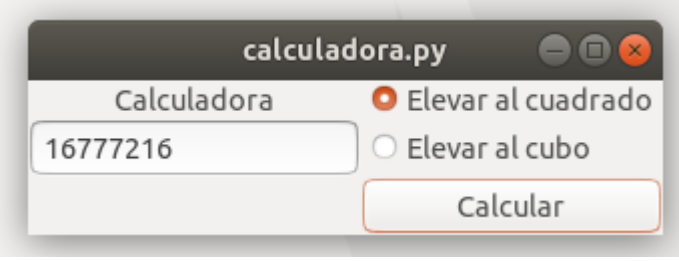
```
#!/usr/bin/env python3

def calcula_cuadrado(n):
    return (n ** 2)

def calcula_cubo(n):
    return (n ** 3)
```

El contenido de “calculadora.py” es el siguiente:

```
#!/usr/bin/env python3

import gi
import potencias

gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

class Eventos:
    def cerrar(self, *args):
        Gtk.main_quit()
```

```
        def click_btnCalcular(self, button):
            entrada = builder.get_object("entNumero")
            num = entrada.get_text()
            cubo = builder.get_object("radCubo")
            cuadrado = builder.get_object("radCuadrado")

            if cuadrado.get_active():
                resultado = potencias.calcula_cuadrado(int(num))
            elif cubo.get_active():
                resultado = potencias.calcula_cubo(int(num))

            entrada.set_text(str(resultado))

    builder = Gtk.Builder()
    builder.add_from_file("ventana.glade")
    builder.connect_signals(Eventos())

    window = builder.get_object("window1")
    window.show_all()

    Gtk.main()
```

La aplicación tendría el siguiente aspecto:



9. DISTRIBUCIÓN DE PROGRAMAS

Hoy en día existen distintas formas de distribuir programas y módulos Python:

- Subiéndolos al repositorio "PyPI - the Python Package Index".
- Paquetes deb, rpm, etc...
- Wheels.
- En un binario para Windows o para Linux junto con todas sus dependencias.
- En una carpeta con todas sus dependencias para Windows o para Linux.

En este punto se explican las 2 últimas posibilidades que permiten distribuir módulos Python "autosuficientes" que ni siquiera requieren un intérprete de Python instalado en la máquina para funcionar. Para generar los ejecutables existen distintas herramientas como py2exe, específico para Windows, y PyInstaller que sirve para Linux, Windows, Mac OS X y otros... Sin embargo, no es un compilador cruzado, para hacer una aplicación de un sistema operativo, se debe ejecutar PyInstaller en ese mismo sistema operativo.

PyInstaller se encuentra disponible PyPI, por lo tanto se puede instalar con pip:

```
pip install pyinstaller
```

Si existe una instalación de python 2.7 y otra de 3.X en el mismo equipo, instalar en la 3.X con :

```
pip3 install pyinstaller
```

Para generar la distribución utilizaremos el siguiente comando:

```
pyinstaller -n NombreApp archivo.py
```

PyInstaller analiza el archivo-.py y:

- Escribe el archivo de especificaciones archivo.spec en la misma carpeta que el archivo.py.
- Crea una carpeta build en la misma carpeta que el archivo.py si no existe.
- Escribe algunos archivos log y archivos de trabajo en la carpeta build.
- Crea una carpeta dist en la misma carpeta que el archivo.py si no existe.
- Crea el “archivo” ejecutable en la carpeta dist.

En la carpeta dist encontrará la aplicación y dependecias para distribuir a los usuarios.

Algunos parámetros de uso de pyinstaler:

- -D, --onedir: Crea un paquete de una carpeta conteniendo un archivo ejecutable (opción por defecto).
- -F, --onefile: Crea un paquete de un único archivo ejecutable.
- --specpath DIR: Carpeta para almacenar el archivo de especificaciones .spec generado (predeterminado: directorio actual).
- -n NAME, --name NAME: Nombre para asignar al paquete de aplicación y al archivo de especificaciones spec (predeterminado: nombre de archivo .py).
- -c, --consolse, --nowindowed: Abre una ventana de consola para E/S estándar (predeterminado).
- -w, --windowed, --noconsole: Windows y Mac OS X, no abrirán una ventana de consola para E/S estándar. Esta opción se ignora en los sistemas * NIX.

10. PRÓXIMOS PASOS

Python es uno de los lenguajes de programación con mayor potencial, y ahora que ya dispones de una sólida base de los fundamentos básicos, podrás comenzar a explorar otros mundos como:

- Desarrollo de aplicaciones web
- Internet of Things
- Análisis de datos (Data Science, Machine Learning, Big Data,...)
- Aplicaciones embebidas
- Desarrollo de aplicaciones de escritorio con GTK, Qt, Kivy
- …

Para cada uno de los distintos campos existen multitud de módulos y frameworks que podemos estudiar y utilizar para realizar nuestros proyectos.