Práctica 1. Introducción al procesamiento de imágenes y vídeos en Python

---

Objetivos de aprendizaje

- Familiarizarse con el entorno de programación que va a utilizarse a lo largo de toda la asignatura:
  - **Python**: Lenguaje de programación.
  - **Anaconda**: Distribución de Python que utilizaremos para gestionar paquetes y librerías
  - **Visual Studio Code**: entorno de desarrollo que utilizaremos para programar y ejecutar código, así como depurarlo para encontrar errores
  - **OpenCV y matplotlib**: Librerías de Python útiles para la visión por computador.
- Aprender a cargar y mostrar imágenes, utilizando las librerías correspondientes, así como modificaciones básicas (transformar a escala de grises, umbralizar)
- Aprender a realizar operaciones convolucionales, como suavizado de la imagen, detección de contornos y reconocimiento de patrones por correlación.
- Aprender a insertar texto y dibujos vectoriales sobre las imágenes
- Entender cómo funciona la operación de correlación en la identificación de patrones.
- Extender estos procesamientos a vídeo, realizando operaciones de las ya vistas por cada frame y guardando un vídeo con las modificaciones implementadas.

Desarrollo de la práctica

Lo primero que haremos será descargar los ficheros de la práctica proporcionados y descomprimirlos en una carpeta que será nuestra carpeta de trabajo para toda la práctica. Abriremos Visual Studio Code (VSCode) y abriremos la carpeta de trabajo con los ficheros descargados. A continuación, seguiremos las indicaciones de los próximos apartados.

1. Mi primer programa de Python

La primera parte va a ser crear un programa de Python que llamaremos `p1.py` y en el que copiaremos el contenido que se muestra a continuación:

```
# Importamos las librerias que vayamos a utilizar
import sys
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt

# Esta es la función donde se ejecuta el programa principal
```

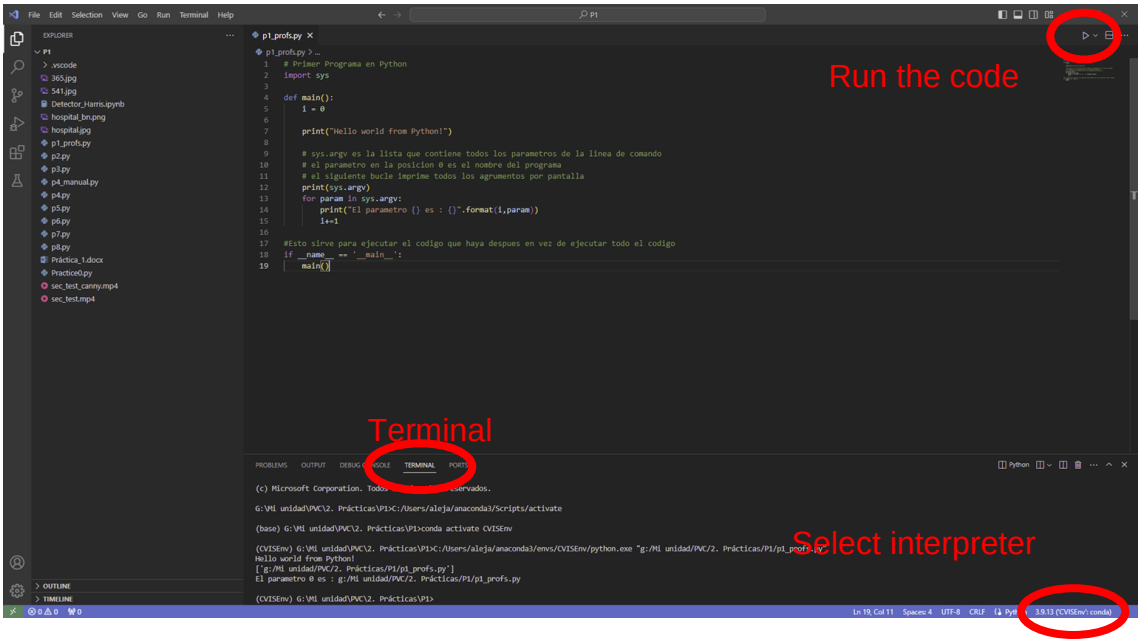
```python
def main():
    i = 0

    print("Hello world from Python!")

    # sys.argv es la lista que contiene todos los parámetros de la
línea de comando
    # el parámetro en la posición 0 es el nombre del programa
    # el siguiente bucle imprime todos los argumentos por pantalla
    for param in sys.argv:
        print("El parametro {} es : {}".format(i,param))
        i+=1

#Esto sirve para ejecutar el código que haya después en vez de
ejecutar todo el código (en este caso ejecuta la función main)
if __name__ == '__main__':
    main()
```

Antes de ejecutarlo tendremos que asegurarnos que estamos utilizando el intérprete adecuado (`VxCEnv`) haciendo click abajo a la derecha en VSCode (ver figura).

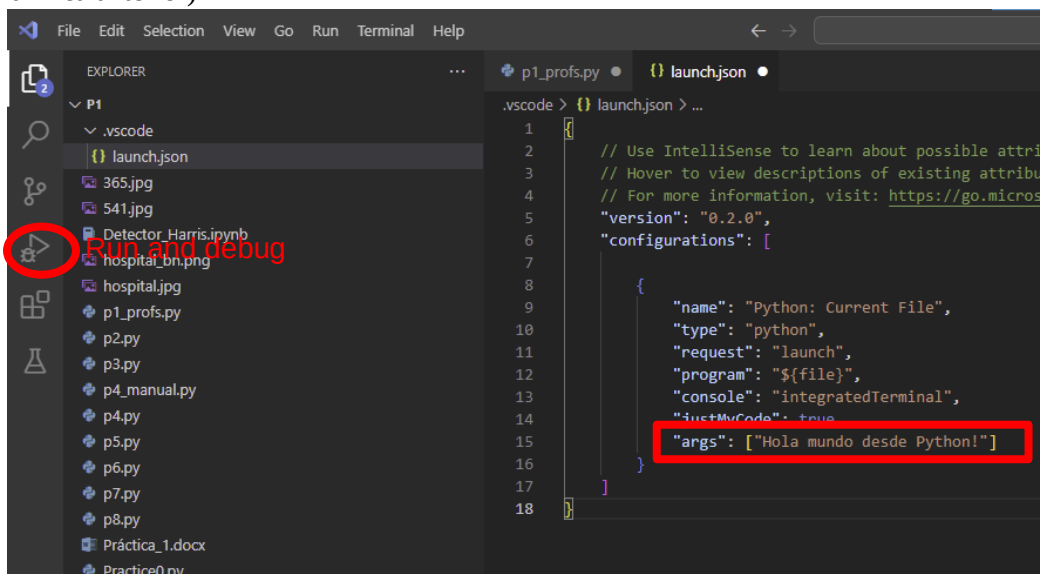


Para ejecutar código de Python, arriba a la derecha tenéis una ejecución rápida del código que tenéis abierto. Tras la ejecución debería aparecer abajo, en la pestaña “Terminal” el siguiente output del programa (fijarse en los `prints` en el código):

- `Hello world from Python!`
- `El parametro 0 es:` *(y aquí el directorio completo donde está p1.py)*

Por defecto, el argumento de entrada número 0 es el nombre completo del fichero. Si queremos añadir parámetros de entrada a un programa:

1. Ir a Run → Add configuration → Python File. Esto crea un fichero editable `launch.json` en la carpeta oculta `.vscode`.
2. Añadir `"args": ["Hola mundo desde Python"]` (ojo: no dejarse la coma en la línea anterior)



3. Ir a la pestaña Run and Debug (ver figura) y hacer click en el botón de Run superior (o hacer click en F5)

De esta forma podéis introducir parámetros de entrada en el programa, de igual manera que si en una ventana de comandos hubiésemos escrito la orden:

```
python p1.py "Hola mundo desde Python!"
```

2. Cargar y mostrar imágenes

Ahora cargaremos la imagen incluida en el fichero de la práctica "`stanfordPersRoom.png`" y la mostraremos por pantalla. Crear el programa `p2.py`, partiendo del anterior, e introducir código para cargar la imagen:

```
img = cv.imread('stanfordPersRoom.png')
```

Para mostrarla por pantalla. Aquí podemos hacerlo con dos librerías distintas:

1. Con **OpenCV**:

```
cv.imshow("Color Image", img)
cv.waitKey(0)
```

   Donde "`Color Image`" es el nombre de la ventana, y `cv.waitKey(x)` es una función que permite mostrar la imagen y quedarse esperando `x` milisegundos hasta que pulsemos una tecla (si ponemos 0 se queda esperando que pulsemos la tecla indefinidamente).

2. Con **Matplotlib.pyplot**:

```
plt.figure(1)
```

```
plt.imshow(img)
plt.show()
```

Donde la función `figure` la utilizamos para crear una ventana (y con la numeración nos permite crear varias simultáneamente) y `show` es la función que muestra la ventana por pantalla (`imshow` únicamente no hace nada visible).

NOTA: Si has utilizado `plt` *es posible que los colores se te vean "irreales". Esto se debe a que hemos cargado la imagen con OpenCV y por defecto ésta almacena los datos en formato BGR y no RGB. Antes de visualizar con* `plt`*, tendremos que convertirla de BGR a RGB:*
`imgColor = cv.cvtColor(img, cv.COLOR_BGR2RGB)`

Por lo general, preferimos el uso de Matplotlib puesto que nos da mayor versatilidad para Python (por ejemplo nos permite añadir anotaciones vectoriales como texto o flechas). Cuando lo probéis, familiarizaos con los controles de la pantalla de visualización, y observad que podéis ver rápidamente la posición y valor de un pixel concreto.

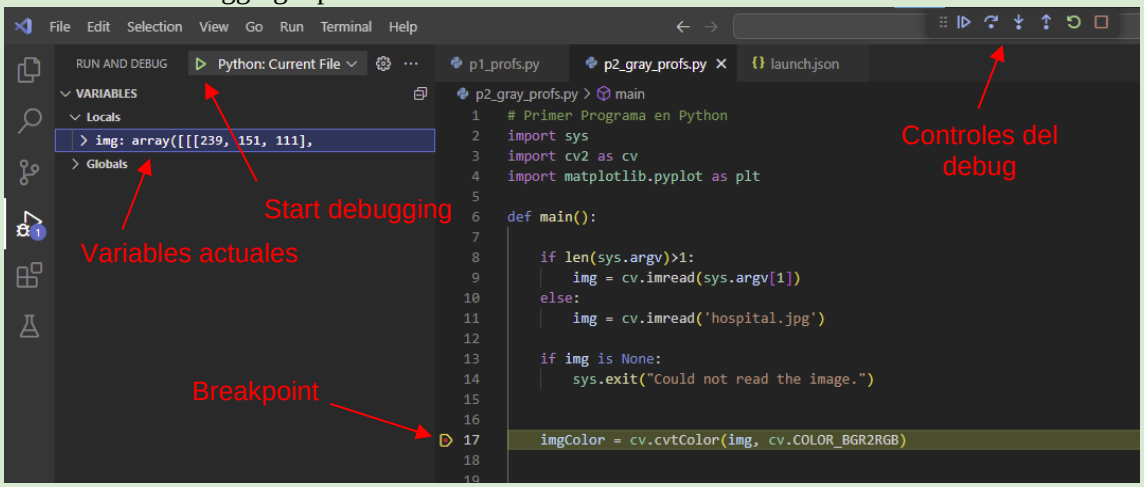
Ahora podéis probar a hacer algunas modificaciones básicas en la imagen:

- Convertir a escala de grises u a otra escala de colores con `cvtColor`.
  Tutorial: https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html
- Binarizar las imágenes (*simple thresholding*), de manera que los píxeles pasan a ser blanco o negro en función de si están por encima o por debajo de un valor umbral (*threshold*).
  Tutorial: https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html

Una vez hayáis hecho una modificación a la imagen, podéis guardarla en la carpeta de trabajo:

```
cv.imwrite("stanfordPersRoom_modified.png", img_modified)
```

TIP: Es interesante que aprendáis a depurar (*debuggear*). Podéis marcar *Breakpoints* en las líneas de código en las que queráis detener la ejecución y consultar el valor de las variables o incluso escribir código en la terminal. Para ello, haced click en la pestaña *Run and debug* y darle arriba a Start debugging o pulsar F5.



Además también os señalará más rápido los errores en la línea que se han cometido, o si hacéis uso de los controles de debug os permitirán ir ejecutando el programa línea a línea y averiguar dónde está el problema.

3. Insertar anotaciones

Es conveniente que aprendáis a insertar anotaciones sobre las imágenes (puntos, líneas, texto, etc.). Además, también aprenderemos a interactuar con las imágenes (por ejemplo, para hacer click en un determinado punto de la imagen).

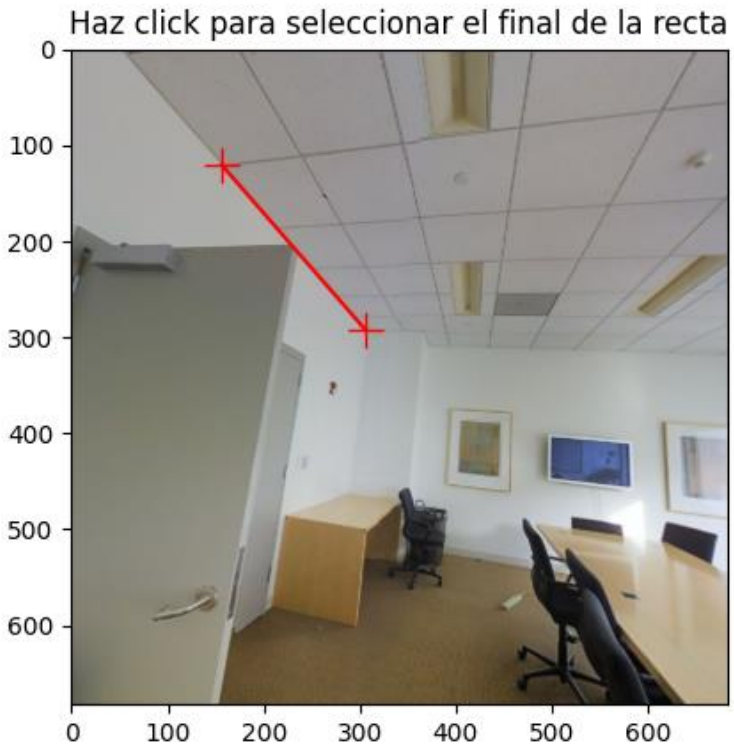
Para este apartado, os damos el código (`p3.py`) y os vamos a pedir que lo miréis antes de lanzarlo. En particular, fijaos que aparecen nuevas funciones de la librería matplotlib.pyplot: title, draw, waitforbuttonpress, plot, close, text. Aseguraos que entendéis el objetivo y funcionamiento de cada una de ellas.

Tenéis también a vuestra disposición la documentación de la librería: https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

Cuando lo lancéis, por orden, se ejecutarán las siguientes tareas (se pasa de una a otra pulsando con el ratón, tal y como pone en el título de la figura):

1. Muestra una imagen.
2. Sitúa un punto A en la imagen (con coordenadas del mismo proporcionadas dentro del código)
3. Pone texto acompañando al punto.
4. Vuelve a mostrar la imagen para que hagas click en un punto.
5. Inserta el punto clicado en la imagen.
6. Vuelve a mostrar la imagen vacía, y esta vez haremos click en dos puntos para definir una recta.

¿Cuál es la diferencia entre `plt.draw()` y `plt.show()` ?



4. Operaciones convolucionales

Vamos a observar, con el ejemplo de la imagen del hospital, qué es lo que sucede cuando le aplicamos distintas operaciones convolucionales. Para ello, puedes partir del código de la segunda parte (`p2.py`) y probar las siguientes operaciones convolucionales:

4.1. Filtro gaussiano

La convolución la haremos con un filtro (*kernel*) Gaussiano, que, en OpenCV se puede definir como sigue:

```
kernel_size = 31
gaussian_kernel = cv.getGaussianKernel(kernel_size,0)
gaussian_kernel_2d = np.outer(gaussian_kernel, gaussian_kernel.T)
```

Donde el kernel tiene que tener dimensiones impares (en este caso 31) y ha hecho falta hacer el producto exterior para generar el kernel 2D. Lo puedes visualizar con la siguiente línea (donde se ha hecho 10 veces más grande para poder verlo mejor):

```
gaussian_kernel_2d_big =
cv.resize(gaussian_kernel_2d*255,(kernel_size*10,kernel_size*10),
interpolation=cv.INTER_NEAREST)
plt.imshow(gaussian_kernel_2d_big,cmap ='gray')
```

Para aplicar una convolución, podemos utilizar la función de OpenCV `filter2D`:

```
img_convolved = cv.filter2D(img, -1, gaussian_kernel_2d)
```

También se puede implementar la convolución a mano utilizando numpy. Para ello completa la función `convolucion` del fichero `convolucion.py` incluido entre los ficheros de la práctica y después la puedes importar y usar en tu código mediante

```
from convolucion import convolve #introducir esto al inicio
...
img_convolved = convolve(imgGray/255,gaussian_kernel_2d)
```

Observa el efecto de la convolución con un kernel gaussiano. Compara el funcionamiento de ambas maneras de realizar la convolución (con la librería OpenCV y a mano).

4.2. Filtros de Sobel

Estos filtros los vamos a definir manualmente, para direcciones *x* e *y*:

```
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
```

Antes de aplicarlos (puedes usar de nuevo la función `filter2D`), tendrás que convertir la imagen a escala de grises. Observa y razona el resultado de la convolución con cada uno de estos filtros.

Finalmente haz la operación de suma ponderada de ambas imágenes resultantes (la de aplicar cada uno de los filtros de arriba) que puedes hacer con el siguiente código:

```
img_sobel = cv.addWeighted(img_sobel_x, 0.5, img_sobel_y, 0.5,0)
```

Donde `img_sobel_x` e `img_sobel_y` son las resultantes de aplicar la convolución anterior con los filtros `sobel_x` y `sobel_y` respectivamente. Compara el resultado con el obtenido anteriormente con cada uno de los filtros por separado.

4.3. Filtro de Canny

Ejecuta ahora el filtro de Canny, por ejemplo, con estos parámetros y visualízalo:

```
low_threshold = 50
ratio = 3
kernel_size = 5
img_canny = cv.Canny(img, low_threshold, low_threshold*ratio,
kernel_size)
```

Compara el resultado con el obtenido con el filtro de Sobel.

5. Transformaciones geométricas

OpenCV permite realizar transformaciones geométricas a imágenes mediante la función `warpAffine`. Previamente se debe definir la matriz de transformación correspondiente. Utiliza la función `rotate_img` para rotar una imagen de ejemplo,

```
def rotate_img(img, angle):
   """ Function to rotate an image [img] by [angle] degrees """
   center = (img.shape[1]/2, img.shape[0]/2)
   r_mat = cv.getRotationMatrix2D(center, angle, 1.0)
   rotated_img = cv.warpAffine(img, r_mat,
      (img.shape[1],img.shape[0]))
   return rotated_img
```

6. Correlación para búsqueda de patrones

Además de las convoluciones existen otras operaciones de vecindad conocidas como correlaciones que pueden utilizarse para búsqueda de patrones.

(Opcional) Completa el fichero `correlacion_valor.py` para calcular la correlación de suma de diferencias al cuadrado "Sum of Squared Differences" (SSD) y la correlación cruzada normalizada "Zero Mean Normalized Cross Correlation" (ZNCC). Este ejemplo permite ver el valor de la correlación entre sub-imágenes parecidas y diferentes.

El fichero `correlacion_busqueda.py` muestra la sub-ventana más parecida a la seleccionada. Comprueba lo que pasa con patrones similares pero a distinta escala.

Ahora modifica el fichero correlacion_busqueda.py para comparar la búsqueda de una sub-ventana en una imagen y en la misma imagen rotada 45 grados.

7. Cargar un vídeo y procesarlo

Vamos a terminar la práctica procesando un vídeo (sec_test.mp4). Para ello, se os da un esqueleto de programa (`p7.py`), cuyas líneas importantes se detallan a continuación:

```
cap = cv.VideoCapture('.\sec_test.mp4')
fps = cap.get(cv.CAP_PROP_FPS)
width  = cap.get(cv.CAP_PROP_FRAME_WIDTH)
height = cap.get(cv.CAP_PROP_FRAME_HEIGHT)

output_file = 'sec_test_processed.mp4'
fourcc = cv.VideoWriter_fourcc(*'mp4v')
fps_save = fps
output_video = cv.VideoWriter(output_file, -1, fps_save,
(int(width),int(height)))
```

Donde `cap` es el objeto de captura de vídeo, del cual extraemos sus frames-per-second (`fps`), y dimensiones (`width`,`height`). A continuación creamos el objeto VideoWriter que utilizaremos para guardar los frames procesados.

A continuación, comienza el bucle, en el que se comienza leyendo un frame (que es un objeto de tipo matriz como los que estábamos utilizando hasta ahora para las imágenes), que puede mostrarse por pantalla o, incluso, procesarlo con alguno de los métodos que hemos visto anteriormente. Al final del bucle, el fotograma procesado se guarda en el objeto de vídeo de salida y, tras acabar el vídeo, se cierra el objeto:

```
while cap.isOpened():
   ret, frame = cap.read()
        '''
        ( Aquí se puede mostrar por pantalla el vídeo, y procesarlo y
     generar el frame de output )
        '''
   output_video.write(output_frame)
   cap.release()
   output_video.release()
```

El código adjunto (`p7.py`) incluye también una temporización del bucle, para que el vídeo se reproduzca en tiempo real. Se queda esperando (con la función `cv.waitKey`) el tiempo necesario hasta que se cumpla el tiempo que venía dado por los `fps` del vídeo original. Si se pulsa la tecla ‘s’ se sale del bucle por adelantado.