Algunas cosas que quedaron de Python

Cátedra Visión por Computadoras

Función `range`

range

- **`range(stop)`**
- **`range(start, stop[, step])`**
- Devuelve un objeto que produce una secuencia de enteros desde start (incluido) hasta stop (excluido) con un paso dado por step
- **`start`** por defecto está en 0 y lo podemos obviar usando la primera forma
- **`range(4)`** produce **`0, 1, 2, 3`**. Con estos indices podemos recorrer una lista de 4 elementos
- **`range(i, j)`** produce **`i, i+1, i+2, ..., j-1`**
- Cuando el **`step`** está dado, especifica el incremento (positivo) o decremento (negativo)
- **`range(4, 0, -1)`** produce **`4, 3, 2, 1`**
- **`range`** no devuelve una lista o tupla con los elementos, sino que devuelve un objeto conocido como **`iterador`**

Como recorrer un arreglo bidimensional

Al estilo de C/C++ ???

```
#! /usr/bin/env python (doble_for_indices.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

h, w = img.shape

thr = 200

for i in range(h):
    for j in range(w):
        if(img[i,j] > thr):
            img[i,j] = 255
        else:
            img[i,j] = 0

cv2.imwrite('resultado.png', img)
```

Como recorrer un arreglo bidimensional

Al estilo de python ???

```
#! /usr/bin/env python (doble_for.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200

for row in img:
    for col in row:
        print(col)
        if(col > thr):
            col = 255
        else:
            col = 0

cv2.imwrite('resultado2.png', img)
```

Como recorrer un arreglo bidimensional

Al estilo de python ???

```
#! /usr/bin/env python (doble_for.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200

for row in img:
    for col in row:
        print(col)
        if(col > thr):
            col = 255
        else:
            col = 0

cv2.imwrite('resultado2.png', img)
```

¿Por qué no anda?

Sentencia `id`

- Dijimos que en python todas las variables eran punteros
- Entonces para saber si dos variables apuntan al mismo objeto usamos **`id`**

```
#! /usr/bin/env python (doble_for_con_id.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200

for row in img:
    for col in row:
        print("type(col) = ",type(col))
        print("id(col)", id(col))
        print("col =", col)

        col = 0
        micol = col

        print("col =", col)
        print("id(col)", id(col))
        print("id(micol)",id(micol))
        break
    break
```

Sentencia `is`

- Para saber si dos variables apuntan al mismo objeto usamos **`is`**

```
#! /usr/bin/env python (ejemplo_is.py)
# -*- coding: utf-8 -*-

a = [1, 2, 3]
b = [1, 2, 3]
c = a

if(a is b):
    print("a apunta al mismo objeto que b")
else:
    print("a no apunta al mismo objeto que b")

if(a is c):
    print("a apunta al mismo objeto que c")
else:
    print("a no apunta al mismo objeto que c")

if(a == b):
    print("a es igual que b")
else:
    print("a no es igual que b")

if(a == c):
    print("a es igual que c")
else:
    print("a no es igual que c")
```

Que pasa si el objeto es inmutable?

```
a = 5
b = 5
print(a is b)
```

True

Que pasa si el objeto es inmutable?

```
a = 5
b = 5
print(a is b)
```

True

Entonces si modificamos a, se modifica b?

```
a = 4
print(b)
print(a is b)
```

Que pasa si el objeto es inmutable?

```
a = 5
b = 5
print(a is b)
```

True

Entonces si modificamos a, se modifica b?

```
a = 4
print(b)
print(a is b)
```

5
False

Que pasa si el objeto es inmutable?

```
a = 5
b = 5
print(a is b)
```

True

Entonces si modificamos a, se modifica b?

```
a = 4
print(b)
print(a is b)
```

5
False

Y esto?

```
a = 257
b = 257
print(a is b)
```

Que pasa si el objeto es inmutable?

```
a = 5
b = 5
print(a is b)
```

True

Entonces si modificamos a, se modifica b?

```
a = 4
print(b)
print(a is b)
```

5
False

Y esto?

```
a = 257
b = 257
print(a is b)
```

Utilizar el comando **`is`** *sólo con objetos mutables*

Función `enumerate`

- `enumerate(iterable[, start])`
- Separa indice de valor
  `for index, val in enumerate(iterable):`
- `iterable` debe ser un objeto que soporte ser iterado (listas, tuplas, iteradores)
- `enumerate` produce pares conteniendo una cuenta (desde `start`, que por defecto vale 0) y un valor producido por el argumento `iterable`
- Es útil para obtener listas indexadas:
  `enum = list(enumerate(s))`
  `enum` ← `[(0, s[0]), (1, s[1]), (2, s[2]), ...]`

Volviendo al arreglo bidimensional

Al estilo de python, ahora sí...

```
#! /usr/bin/env python (doble_for_enumerate.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200

for i, row in enumerate(img):
    for j, col in enumerate(row):
        if(col >= thr):
            img[i,j] = 255
        else:
            img[i,j] = 0

cv2.imwrite('resultado3.png', img)
```

Umbralizado usando numpy

. . . y ahora usando numpy

```
#! /usr/bin/env python (umbralizado_numpy.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200

img[img >= thr] = 255
img[img < thr] = 0

cv2.imwrite('resultado_numpy.png', img)
```

Umbralizado usando OpenCV

Función `threshold`

- Sirve para aplicar un umbralizado (fijo o adaptable) a todos los elementos de un arreglo
- `retval, dst = cv2.threshold(src, thresh, maxval, type[, dst])`
- `src` es el arreglo a umbralizar
- `threshold` es el umbral a aplicar
- `maxval` es el valor a guardar en cada componente del arreglo en caso que sea mayor que el umbral
- `type` es el tipo de umbralización, por defecto binaria
  - `cv2.THRESH_BINARY`
  - `cv2.THRESH_BINARY_INV`
  - `cv2.THRESH_TRUNC`
  - `cv2.THRESH_TOZERO`
  - `cv2.THRESH_TOZERO_INV`
- Existen dos umbralizados que calculan el umbral óptimo en forma automática:
  - `cv2.THRESH_OTSU`
  - `cv2.THRESH_TRIANGLE`

Umbralizado usando OpenCV

Función `threshold` continuación

- **`retval, dst = cv2.threshold(src, thresh, maxval, type[, dst])`**
- Si el arreglo **`dst`** existe (debe ser del tipo y tamaño que el arreglo de entrada **`src`**), lo pasamos como argumento y la imagen umbralizada se guarda ahí
- Si no pasamos **`dst`** como argumento, la función devuelve un nuevo arreglo con el resultado de la umbralización
- **`retval`** nos devuelve el valor del umbral utilizado por la función, esto sirve para el caso en donde dejemos que el umbral lo calcule la función

Umbralizado usando OpenCV

```python
#! /usr/bin/env python (umbralizado_opencv.py)
# -*- coding: utf-8 -*-

import cv2

img = cv2.imread('siempre_verde.png', 0)

thr = 200
maxval = 255

retval, dst = cv2.threshold(img, thr, maxval, cv2.THRESH_BINARY)

cv2.imwrite('resultado_opencv.png', dst)

retval, dst = cv2.threshold(img, int(), maxval, cv2.THRESH_OTSU)
print("Umbral Otsu= ", retval)

cv2.imwrite('resultado_opencv_otsu.png', dst)
```