Máster Universitario en
Nuevas Tecnologías en Informática

Asignatura “Visión Artificial”

Introducción al entorno de prácticas: Python - OpenCV

Facultad de Informática
Universidad de Murcia
Curso 2018/19

Características de Python

- **Open Source.**
- **Alto nivel:**
  - Simple y minimalista (y por tanto, fácil de aprender).
  - Tipado dinámico.
  - Manejo memoria automático.
- **Portable:**
  - Linux, Windows, Macintosh, …
  - P. e., disponible también para Raspberry Pi
- **Interpretado** (no compilado).
- **Orientado a objeto.**
- **Extensible con C/C+** fácilmente.
- Extenso conjunto de **librerías disponibles**:
  - GUI, documentación, threads, bases de datos, web, XML, HTTP, juegos, multimedia, criptografía, proceso imagen, cómputo científico, visión por computador, machine learning...

Python 2.x vs Python 3

- Python está en transición de la versión 2 (hasta 2020) a la versión 3 (recomendada).
- Usaremos ya la versión de Python 3, en concreto la **versión 3.7**:
  - Utilizada y testeada con OpenCV 3.X.
- Hay algunas (pequeñas) **diferencias sintácticas** entre Python 2 y Python 3:
  - Existen herramientas automatizadas de traducción.
- Usaremos la implementación **Anaconda**:
  - Mantenimiento de paquetes con mayor facilidad.

Variables en Python. Comentarios.

- Una variable contiene información de cualquier tipo.
  - Ejemplos de asignación en Python:

```
x = 1.343  # a number

greeting = 'hi'  # a string

arr = [1, 1, 2, 3, 5, 8]  # a list
```

  - El carácter '#' es para comenzar un comentario en Python, que siempre termina al final de la línea.

Tipos de datos en Python

- Al contrario que en otros lenguajes, como C, C++ o Java, una variable puede cambiar su tipo durante la ejecución (**tipado dinámico**):

```
x = 2

x = [3, 4, 5]

x = 'hi'
```

  - Python se encarga internamente en todo momento de mantener la información del tipo actual de cada variable.
  - Siempre podemos consultar el tipo con `type(var)`.

Operaciones

- Operaciones clásicas sobre datos, incluídas cadenas (*strings*):

```
2 + 4

3.2 * 6

(8.7 - 3.3) / 4

'hi' * 10

'hello, ' + 'world!'
```

```
x = 4
x += 2  # x = x + 2

y = 'hi'
y *= 4   # y = y * 4
```

Operadores de comparación

- Clásicos, pero también extendidos:

```
x == 1 # check for equality

y != 'hello' # check for inequality

2 > 2  # False

2 >= 2 # True

2.1 < z < 5.4 # chained inequalities
```

Operaciones booleanas

- Python usa las palabras clave **and**, **or** y **not** (en lugar de sus homólogos C/C++ &&, ||, !):

```
x = 1
y = 'hello'

x == 1 or y == 'hi'
x > 0 and y != 'hi'

y = 0
x or y
x and y
not y
```

Control de flujo

- La instrucción `if` (posiblemente acompañada por un `else` y/o uno o varios `elif`) es la más básica para controlar el flujo de un programa:

```
if x:
    print('hello1')

if not x:
    print('hello2')

if not x and y:
    print('hello3')
```

```
if x:
    print('hello')
else:
    print('hi!')
```

Evaluación booleana de valores numéricos

- El valor numérico 0 es falso, el resto siempre evalúan como verdaderos:

```
x = -5
y = 5
z = 0

if x and y and z:
    print('hello1')# will not print

if (x and y) or z:
    print('hello2')# will print
```

Bucles *while*

- La instrucción `while` repite una tarea mientras su condición evalúe a cierto:

```
x = 7.0
lower = 0.0
upper = x
guess = (upper + lower) / 2
while (abs(x - guess * guess) > 0.1):
    if guess * guess > x:
        upper = guess
    else:
        lower = guess
    guess = (upper + lower) / 2
```

Bucles *for*

- La instrucción `for` repite una tarea, pero con un estilo diferente a C/C++. Se itera sobre elementos en un "objeto":

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in arr:
    print(number * 2)
```

- Así, el bucle típico sobre una serie de números se suele montar usando la construcción de una lista. P.e., en el caso anterior se podría sustituiría `arr` por `range(1,10)`.
- **Ojo:** diferencia evaluación "perezosa" python 3 *vs.* "ansiosa" python2:

`range(1,10)` *vs.* `list(range(1,10))`

Funciones

- Muy usadas para estructurar el código de un programa:

```
def root(x, tol=0.1):
    lower = 0.0
    upper = x
    guess = (upper + lower) / 2
    while (abs(x - guess * guess) > tol):
        if guess * guess > x:
            upper = guess
        else:
            lower = guess
        guess = (upper + lower) / 2
    return guess
```

- Podemos hacer llamadas root(3), root(113,0.01), etc., para desencadenar la consiguiente búsqueda binaria.

Funciones

- Otro ejemplo (observar el uso de `enumerate` para iterar sobre una lista, generando pares `(índice,valor)`):

```
def polyval(p, x):
    val = 0
    for i, coeff in enumerate(p):
        val += coeff * (x ** i)
    return val

print(polyval([1, 2, 0, 1], 4)) # prints '73'
```

Indentación y espaciado

- Python usa la indentación (“espacio en blanco”) para agrupar sentencias:
  - Cada bloque de código se indenta con la misma cantidad de espacio (habitualmente 4 espacios).
  - Bloques de código: cuerpo `if/elif/else`, cuerpo bucle `for`, cuerpo bucle `while`, definición de función, etc.
  - Ejemplo de indentación errónea:

```
a = 10
b = 2
while a > 1:
  print(a + b )
    a = a - 1  # wrong indentation --> error!
```

Indentación y espaciado

- Más ejemplos (2 incorrectos y 1 correcto):

```
def func_incorrect1(x, y, z):
if x < y:     # wrong indentation --> error!
  return z
return 0

def func_incorrect2(x, y, z):
  if x < y:
  return z  # wrong indentation --> error!
  return 0

def func_correct(x, y, z):
  if x < y:
    return z
  return 0
```

Ejecutar un *script* Python

- Para iniciar un intérprete de python, teclear en un terminal `python`, o mejor aún, `ipython`. También puede estar integrado en el IDE (spyder, PyCharm). Se sale con Ctrl-D.
- Si hemos escrito un *script* de Python llamado `filename.py`, se puede ejecutar así:

```
$ python filename.py
```

- Aunque en Linux se puede hacer al *script* directamente ejecutable colocando esta línea arriba del todo:

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

- También debe haberse hecho previamente ejecutable el archivo, ejecutando el comando:

```
$ chmod +x hello.py
```

Estructuras de datos

- Python posee (entre otros no tan utilizados) los siguientes **tipos de datos agregados**:
  - **Listas**.
  - **Tuplas**.
  - **Cadenas**.
  - **Diccionarios**.
- Todos son muy sencillos de usar, a la par que potentes y versátiles.

Listas

- Almacenan una secuencia de datos que soporta la operación de indexado.
- Vimos ya una en el ejemplo anterior de la evaluación polinomial.
- Observar la **posibilidad de indexar con negativos (**desde el final):

```
l = [2, 4, 6, 8]

print(l[3])# prints '8'
print(l[-1])# prints '8'

print(l[4])# error
l.append(10)
print(l[4])# prints '10'
```

Listas

- Se pueden concaternar listas con +, y acceder a ellas mediante otros operadores como `pop`:

```
l1 = [2, 4, 6, 8, 10]
l2 = [3, 5]
l3 = l1 + l2
# addition: l3 is now [2, 4, 6, 8, 10, 3, 5]

l3.pop() # removes 5 from l3
```

Listas

- También se pueden manipular mediante *slices* (“rodajas”).
- Formato `[comienzo:final:paso]` (cualquiera puede omitirse):

```
l = [2, 4, 6, 8]

print(l[0:2])# prints [2, 4]
l[1:3] = [7, 7] # l is now [2, 7, 7, 8]
print(l[2:])# prints [7, 8]

l *= 2
print(l[3:6])# prints [8, 2, 7]
```

Listas

- Hay muchas funciones *built-in* que trabajan con listas:

```
l = [2, 4, 6, 8, 10, 12]

len(l) # 6: number of elements
max(l) # 12
min(l) # 2

# more advanced if you are interested:
filter(lambda x : x % 4, l)
```

Listas

- Las listas no tienen por qué ser homogéneas. ¡Incluso se pueden anidar!
- Esto hace de ellas una estructura extremadamente versátil y potente:

```
l = [2, [3, 5, 6], 'orange', 6, 'blue']
print(l[2])# prints 'orange'
```

- Serán la base del tipo de datos array n-dimensional sobre el que se fundamenta el paquete NumPy.

Iteraciones sobre listas

- Operadores `for` (para iterar sobre la lista) e `in` (para buscar elementos en ella):

```
l = [2, 4, 'orange', 6, 'blue']

for elmt in l:
    print(elmt)

if 'blue' in l and 'red' not in l:
    print('hi')# this will be printed
```

Iteraciones sobre listas

- Como vimos, usar `enumerate` es el modo más conveniente de iterar sobre una lista manteniendo el índice:

```
squares = [0, 1, 4, 9, 16, 25]

for i, val in enumerate(squares):
    print(i, val)

# cleaner and more concise than:
i = 0
for val in squares:
    print(i, val)
    i = i + 1
```

Listas por comprensión

- Se pueden formar nuevas listas a partir de la manipulación de otras (incluso de forma anidada):

```
vals = [1, 2, 3, 5, 7, 9, 10]

# Only include doubles for values divisible by 5
double_vals5 = [2 * v for v in vals if v % 5 == 0]
```

```
x_pts = [-1, 0, 2]
y_pts = [2, 4]

xy_pts = [[x, y] for x in x_pts for y in y_pts]

# [[-1, 2], [-1, 4], [0, 2], [0, 4], [2, 2], [2, 4]]
```

Tuplas

- Similares a las listas, pero no pueden ser modificadas (son **inmutables**):

```
p1 = ('start', 1.2, -3.0, 17.222)
p2 = ('end', -7.3, 0.0, -0.0001)

p1[3] = 17.2 # error!

print(p2[2])# prints '0.0'

# unpacking
type1, x1, y1, z1 = p1
type2, x2, y2, z2 = p2

print(x1 - x2)# prints '8.5'
```

Cadenas

- Propiedades similares a las de las listas:
  - Pueden indexarse, hacer *slices* de ellas, etc.
  - También manipularlas “aritméticamente”:

```
str = 'hello, world!'

print(str[1])# prints 'e'
print(str[-1])# prints '!'
print(str[7:12])# prints 'world'

str += '!!!1!'
```

Cadenas

- Hay también muchas funciones *built-in* para trabajar con ellas:

```
vec = '[12.4, 3, 4, 7.22]'

# strip away the brackets
vec = vec.lstrip('[')
vec = vec.rstrip(']')

# form an array by splitting on comma
nums = vec.split(',')

# go from string to floating point
nums = [float(n) for n in nums]
```

Cadenas

- Más ejemplos de uso de funciones *built-in*:

```
vec = '[12.4, 3, 4, 7.22]'

nums = [float(n) for n in vec.strip('[]').split(',')]
```

```
str = 'Hello, World!'

len(str) # 13
str = str.lower() # 'hello, world!'

str = ' '.join(['Hello', 'World', '!'])
# Hello World !
```

- Formateo de cadenas muy potente (muchos más ejemplos en https://pyformat.info/):

```
'{1} {0}'.format('one', 'two')
```

→ `two one`

```
'{} {}'.format(1, 2)
```

→ `1 2`

Diccionarios

- Los diccionarios son asociaciones (*mappings*) de un conjunto de claves a un conjunto de valores.
- También llamados ***arrays* asociativos**.
- Pares clave-valor:

$$K = \{ \text{ keys } \},\ V = \{ \text{ values } \}.\ D : K \to V\text{:}$$

$$k \overset{D}{\longmapsto} v_k \in V$$

Diccionarios

- Ejemplo de construcción de un diccionario:

```
import math

p = (1.2, -40.0, 2*math.pi)

point = {} # form an empty dictionary

point['x'] = p[0]
point['y'] = p[1]
point['z'] = p[2]
point['r'] = math.sqrt(sum([v ** 2 for v in p]))
point['theta'] = math.acos(point['z'] / point['r'])
point['phi'] = math.atan(point['y'] / point['x'])
```

Diccionarios

- Construcción alternativa, **sintaxis más limpia**:

```
import math

p = (1.2, -40.0, 2*math.pi)

# Create dictionary with keys
point = {'x': p[0], 'y': p[1], 'z': p[2],
         'r': math.sqrt(sum([v ** 2 for v in p]))}
point['theta'] = math.acos(point['z'] / point['r'])
point['phi'] = math.atan(point['y'] / point['x'])
```

Diccionarios

- Acceso, borrado y sobreescritura de claves:

```
# access
magnitude = point['r']
x = point['rho'] # error!

# overwrite
point['r'] = 5.13
point['r'] = 6.23

# remove key-value pair
del point['theta']
```

Diccionarios

- También pueden usarse `for` e `in` con ellos:

```
# print all keys
for key in point:
    print key

# check if a key is there
if 'theta' not in point:
    print('missing theta!')
```

Más sobre funciones

- En python las funciones son objetos normales:
  - Y por tanto pueden asignarse a variables y usarse como argumentos a otra función, por ejemplo:

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

def cube(x):
    return x ** 3

def operate(f, y):
    return f(y)

print(operate(square, 4))# prints '16'
print(operate(cube, 4))# prints '64'
```

Funciones lambda

- Se trata de funciones anónimas:
  - Se definen en el mismo lugar en que se usan, sin darles un nombre explícito:

```
def operate(f, y):
    return f(y)

print(operate(lambda x : x ** 2, 4)) # prints '16'
print(operate(lambda x : x ** 3, 4)) # prints '64'

square_plus_cube = lambda x : x ** 2 + x ** 3
print operate(square_plus_cube, 4) # prints '80'
```

Funciones lambda

- Ejemplo interesante de uso con función `sorted`:

```
id_dept_pairs = [(8283, 'Aero/Astro'),
                 (3456, 'CS'),
                 (7888, 'Math')]

# Sort by id number
print(sorted(id_dept_pairs,
             key=lambda pair: pair[0]))
# [(3456, 'CS'), (7888, 'Math'),
#  (8283, 'Aero/Astro')]

# Sort by department alphabetically
print(sorted(id_dept_pairs,
             key=lambda pair: pair[1]))
# [(8283, 'Aero/Astro'), (3456, 'CS'),
#  (7888, 'Math')]
```

Parámetros a funciones

- Declaración de función muy genérica:

```
def func(p1, p2=7, *args, **kwargs)
```

- Ejemplos de llamadas válidas:

```
func(1)                 # p1=1, p2=7
func(1, 2)              # p1=1, p2=2
func(1, p2=2)           # p1=1, p2=2 (equivalente)
func(1, 2, 3, 4)        # p1=1, p2=2,
                        # args[0]=3, args[1]=4
func(1,a=3,b=4)         # p1=1, p2=7, kwargs['a']=3
                        # kwargs['b']=4
```

Mutable *vs.* inmutable

- Algunos tipos no pueden cambiar su contenido (p.e. strings) mientras que otros sí (p.e. listas):

```
x = 'foo'
y = x
print(x)# foo
y += 'bar'
print(x)# foo

x = [1, 2, 3]
y = x
print(x)# [1, 2, 3]
y += [3, 2, 1]
print(x)# [1, 2, 3, 3, 2, 1]
```

- **Nota:** Interesante para comprobar si dos variables referencian en realidad al mismo objeto: `id(var)`.

Mutable *vs.* inmutable

- La (in)mutabilidad afecta principalmente al paso de parámetros a una función:

```
x = 'foo'
print(x)# foo
func(x)
print(x)# foo

def func(val):
    val += [3, 2, 1]

x = [1, 2, 3]
print(x)# [1, 2, 3]
func(x)
print(x)# [1, 2, 3, 3, 2, 1]
```

Excepciones

- Si una instrucción (o bloque de ellas) pueden generar una excepción, ésta abortará el programa, a no ser que se capture, así:

```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Excepciones

- Se puede también ser más selectivo en la captura de excepciones, obtener información de ellas, etc.:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print("I/O error({0}): {1}".format(e.errno, e.strerror))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Excepciones

- Otro ejemplo (usando la instrucción `pass`, que esencialmente sirve para *"no hacer nada"*), para ignorar varios tipos de excepción:

```
except (RuntimeError, TypeError, NameError):
    pass
```

Cláusula `import`

- Se usa para cargar un módulo (librería, u otro módulo fuente):

```
import math
# code is in lambda2.py
import lambda2

print(math.pi)

print(lambda2.operate(lambda2.square_plus_cube, 4))
```

Cláusula `import`

- También puede cambiarse el nombre de la misma al importar la librería (para abreviar):

```
import math
import lambda2 as l2

print(math.pi)

print(l2.operate(l2.square_plus_cube, 4))
```

Cláusula `import`

- O incluso importar todo directamente en el espacio de nombres actual, evitando la necesidad de la cualificación previa:

```
import math
from lambda2 import *

print(math.pi)

print(operate(square_plus_cube, 4))
```

- Aunque no está muy recomendado (se “llena demasiado” el espacio de nombres).

Lectura de archivos

- Supongamos que tenemos un fichero de texto con compuestos químicos:

```
salt: NaCl
sugar: C6H12O6
ethanol: CH3CH2OH
ammonia: NH3
```

- Y que queremos leer este archivo y almacenar su información en un diccionario.

Lectura de archivos

```
f = open('compounds.txt', 'r')
```

- El parámetro `'r'` especifica que queremos leer el archivo.
- A partir de ahora, `f` será un objeto fichero.
- Se puede leer todo de una sola vez, o línea a línea:

```
f = open('compounds.txt', 'r')
contents = f.read()
print(contents)
```

```
f = open('compounds.txt', 'r')
for i, line in enumerate(f):
    print('(Line #' + str(i + 1) + ') ' + line)
```

Lectura de archivos

- Lectura en un diccionario (algo verbosa):

```
f = open('compounds.txt', 'r')
compounds = {}
for line in f:
    split_line = line.split(':')
    name = split_line[0]
    formula = split_line[1]
    formula = formula.strip()
    compounds[name] = formula

f.close()
```

Lectura de archivos

- Lectura en un diccionario (más "*phytonic*"):
  - La sentencia `with` cierra el fichero automáticamente:

```
compounds = {}
with open('compounds.txt', 'r') as f:
    for line in f:
        compounds[line.split(':')[0]] = \
          line.split(':')[1].strip()
```

  - Incluso *one-liner*:

```
compounds = dict([(line.split(':')[0],
                   line.split(':')[1].strip()) \
                   for line in open('compounds.txt',
                                    'r')])
```

Escritura en archivos

- Supongamos que tenemos estos datos en un diccionario:

```
d1 = {'title': "The eyes says it all",
      'sub': 'aww', 'comments': 595}

d2 = {'title': "From typical youtube upload " + \
               "to serendipity in 30 seconds",
      'sub': 'AskReddit', 'comments': 6494}

d3 = {'title': "Use a decent host or don't " + \
               "even try at all...",
      'sub': 'AdviceAnimals', 'comments': 95}

data = [d1, d2, d3]
```

Escritura en archivos

- Pasamos el parámetro `'w'`, y formateamos a voluntad:

```
from reddit_data import *

with open('reddit2.txt', 'w') as f:
  for i, point in enumerate(data):
    data = 'Post #%d\n' % i
    data += '\t%s\n' % point['title'][0:20]
    data += '\t\t%s (%d)\n' % (point['sub'],
                               point['comments'])
    f.write(data)
```

Orientación a objetos

- Clases con variables de instancia, métodos.
- Pueden variar sobre la marcha (*"on the fly"*).
- Primer parámetro implícito en métodos (`self`):

```
class Stock():
    def __init__(self, name, symbol, prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

google = Stock('Google', 'GOOG')
apple = Stock('Apple', 'APPL', [500.43, 570.60])

print(google.symbol)
print(max(apple.prices))
```

Documentación de fuentes

- Para módulos, simplemente se usa una cadena de múltiples líneas (entre dos líneas con `'''`).
- Ídem para clases, funciones, etc., justo a continuación de la cabecera.
- El propio texto dentro de la cadena puede ser formateado para ser procesado por herramientas de generación de documentación.
- Ejemplo:

  http://sphinxcontrib-napoleon.readthedocs.org/en/latest/example_google.html

NumPy

- Extensión de Python, con soporte para grandes arrays multidimensionales:
  - Matrices, vectores, tensores, etc., con multitud de funciones sobre ellos.
  - Basada en el objeto de tipo `ndarray`:
    - Encapsulan arrays n-dimensionales de datos homogéneos.
    - Implementados en código compilado desde C (no interpretado por Python) → Eficiencia.
    - Usado por la mayoría de paquetes científicos (también OpenCV).
    - Habitualmente creamos objetos `ndarray` mediante funciones como `np.array`, `np.zeros`, `np.ones`, etc.

NumPy

- Creación y acceso a elementos:

```
import numpy as np

normal_arr = [[1.2, 2.3], [-3.1, 4.77]]
ndarr = np.array(normal_arr)

ndarr.shape # (2, 2)
```

```
import numpy as np

identity10 = np.eye(10)
ones4x2 = np.ones((4, 2))
```

```
import numpy as np

A = np.ones(4)
A[0, 0] += 2
A12 = A[1, 2]

first_row = A[0,:]
last_col = A[:,-1]
```

NumPy

- Varias formas de crear arrays:

```
import numpy as np

#  lists
arr = np.array([[1, 2, 3], [4, 5, 6]])

#In [1]: arr
#Out[1]:
#array([[1, 2, 3],
#       [4, 5, 6]])

#  sequences
np.arange(0, 10, 0.1)
np.linspace(0, 2 * np.pi, 100)

#  zeros & ones
np.zeros((5, 5))
np.ones((5, 5))

#  random
np.random.random(size=(3, 4))
np.random.normal(loc=10., scale=3., size=(3, 4, 5))
```

NumPy

- Ejemplo de resolución de un sistema de ecuaciones:

```
import numpy as np

list_matrix = [[1, 3, 4], [2, 3, 5], [5, 7, 9]]
A = np.array(list_matrix)
b = np.array([4, 4, 4])

# Solve for Ax = b
x = np.linalg.solve(A, b)
```

NumPy

- Ejemplo sencillo aplicado a *machine learning*:
  - Clasificador lineal:

```
import numpy as np

def svm_classify(w, b, x):
    return np.dot(w, x) - b > 0

w = [-1.3, 4.555, 7]
b = 9.0
points = [[8.11, 3.42, 11.2], [-4.9, 4.557, 7.08]]
labels = [svm_classify(w, b, p) for p in points]
```

NumPy

- E/S de arrays desde/a ficheros:

```
import numpy as np

# create an array, write to file, read from file
arr = np.array([[1, 2, 3], [4, 5, 6]])

# save to a text file
# creates a space delimited file by default
np.savetxt(fname='array_out.txt', X=arr)

# load text file
loaded_arr = np.loadtxt(fname='array_out.txt')

np.all(arr == loaded_arr)  # True
```

- (Posibilidad de controlar tipos de datos, comentarios, cabeceras, etc., ver documentación).

NumPy

- Atributos de un array:

```
import numpy as np

arr = np.arange(10).reshape((2, 5))

arr.ndim    #  2 number of dimensions
arr.shape   #  (2, 5) shape of the array
arr.size    #  10 number of elements
arr.T       #  transpose
arr.dtype   #  data type of elements in the array
```

NumPy

- Operaciones sobre arrays:

```
import numpy as np

arr1 = np.arange(10).reshape((2, 5))
arr2 = np.random.random((2, 5))

# elementwise for basic and boolean operations
#   +, -, *, /, **, np.log, <, >=, ==
#   arrays are upcast, resulting in float or boolean arrays
arr1 + arr2  # elementwise sum
arr1 * arr2  # elementwise multiplication

# operations in place
arr1 += arr2

# matrix product
np.dot(arr1, arr2)

# similarly numpy ufunc's operate elementwise
np.sin(arr1)
np.sqrt(arr1)
```

New in python 3.5:

```
arr1 @ arr2
```

NumPy

- *Slicing* e *indexing*:

```
import numpy as np

arr = np.arange(20).reshape((4, 5))

# slicing (like lists for each dimension)
arr[0:4, 3:5]  # all rows and last two columns
arr[:4,  3:5]  # equivalent - can leave off start if 0
arr[:, 3:]     # equivalent - can leave off end if size of axis
arr[slice(None), slice(3, None)]  # equivalent - can use slice()

# integer indices
arr[[1, 2], :]  # rows one and two, all columns
arr[np.array([1, 2]), :]  # equivalent

# boolean indices
arr[[False, True, True, False], :]  # equivalent
arr[np.array([False, True, True, False]), :]  # equivalent
```

- ¡Particularmente potente es el indexado booleano!

NumPy

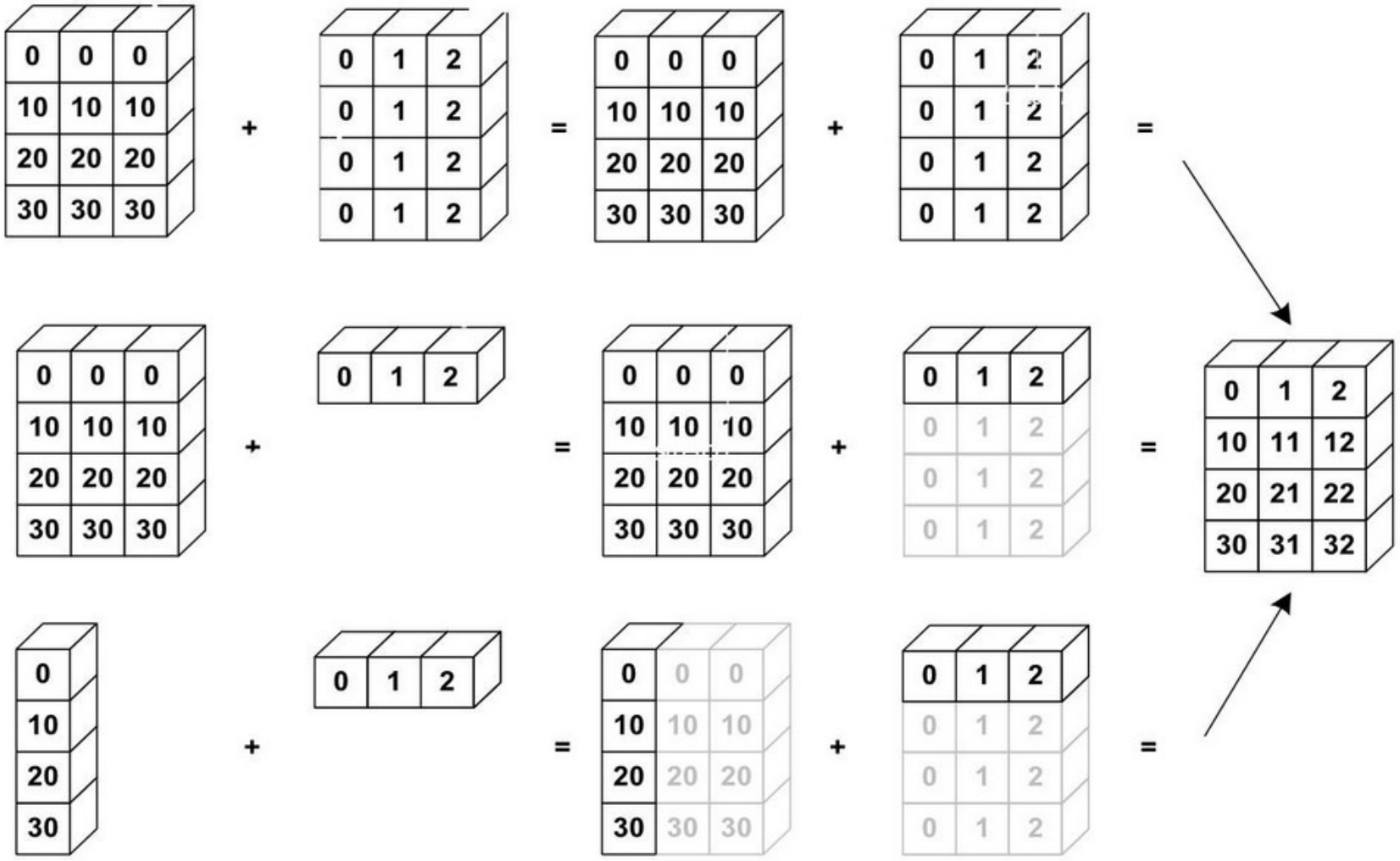
- *Broadcasting*:

```
import numpy as np

# multiplication by a scalar
arr = np.random.random((4, 5))
arr * 5  # multiply each element of the array by 5

# scales the first column by 0.
# scales the second column by 1.
# etc.
arr * np.arange(5)
```

- Permite trabajar con arrays de diferentes tamaños.

NumPy

- Ejemplo gráfico de *broadcasting*:



Matplotlib

- Librería estándar de gráficos matemáticos 2D/3D
  - Funciones, histogramas, diagramas de barras, scatter plots, ...
- Salida:
  - Arrays de píxeles
    - Ventanas de programa
    - Archivos en formatos jpg, ppm, etc.
  - Vectorial:
    - Archivos en formatos pdf, svg, etc.
- Ejemplos: http://matplotlib.org/gallery.html

SciPy

- Conjunto de librerías estándar de cómputo científico.
- Principalmente usaremos álgebra lineal (módulo `scipy.linalg`):
  - Resolución de sistemas de ecuaciones lineales, inversas, determinantes, normas, etc.
  - Descomposiciones matriciales:
    - QR, SVD, LU, Valores y vectores propios, Cholesky, etc.
- También submódulos de estadística, optimización, interpolación, integración, álgebra lineal *sparse*, etc.

OpenCV para Python

- **Procesamiento de imagen**:
  - Espacios de color, umbralizado, filtros, transformaciones geométricas, morfología, contornos, histogramas, etc.
- Detección de ***features***, y descriptores asociados:
  - Canny, Harris, SIFT, SURF, FAST, BRIEF, ORB, ...
- **Machine learning**:
  - Alternativa más potente: `scikit-learn`.
- **Geometría proyectiva**:
  - Estimación de homografías, triangulación, calibración, geometría epipolar, etc.
- **Otros**:
  - Filtros de *Kalman*, *clustering*, procesamiento de caras, etc.

Entorno de trabajo

- **Spyder (ver. 3.2):**
  - IDE con facilidades de edición / depuración.
  - Integra `Ipython` (intérprete enriquecido).
  - Inspector de variables.
  - Ayuda rápida.
  - Modelo de código:
    - Corrector de sintaxis.
    - Compleción de código.
  - Análisis de eficiencia (*profiling*).
- Posible alternativa: **PyCharm** (Free Community)

Ejercicio recomendado para empezar

- Esquema típico de una aplicación de visión, `ejemplosimple.py`, que ilustra:
  - **Modularización**:
    `utilscv.py`, `videoinput.py`
  - **Lectura** vídeos, cámara e imágenes con OpenCV.
  - **GUI** (ventanas, zoom, salvar imagen, etc.).
  - **Eventos**: teclado / ratón, sliders, etc.
  - Típico **procesamiento de imagen** mono/multicanal.
  - Típico **procesamiento de *features*** (listas de puntos).
  - **Dibujo** sobre imágenes (líneas, texto, etc.).
  - Uso elemental del paquete de **álgebra lineal**.
  - **Clases Python** (ejemplo `VideoInput`).

Referencias (I)

- Curso Stanford Scientific Python:
  - https://web.stanford.edu/~schmit/cme193/
- Tutorial oficial de Python:
  - https://docs.python.org/3.7/tutorial/index.html
- NumPy para usuarios de Matlab:
  - http://mathesaurus.sourceforge.net/matlab-numpy.html
- Documentación NumPy / SciPy:
  - http://docs.scipy.org/

Referencias (II)

- Apuntes de Python, E. Aranda, UCLM: ↵
- Documentación OpenCV:
  - https://docs.opencv.org/3.4.0/

    (Buscar siempre funciones Python, no C++)
  - Particularmente interesante para nosotros:
    - Tutorial de uso de la OpenCV desde Python: ↵

Y por supuesto... stackoverflow :-)