Python Introduction

Machine perception

2022/2023

1 Basic Python

(a) **Libraries**: Libraries can be imported using the keyword `import`. After importing a library, its functions can be used by prepending them with the library name (e.g. `np.array()`).

```
import cv2 # import entire library
import numpy as np # import library as a specifit, shorter name
from matplotlib import pyplot as plt # import a specific module from a library
from PIL import Image
```

(b) **Lists and tuples**: Tuples are immutable while lists are mutable and support sorting and other list operations. Both are indexed using brackets (`[]`).

```
a = (1,2,3) # tuple
b = [1,2,3] # list
```

(c) **Sets**: A set is an unordered, unchangeable, and unindexed collection. Checking whether an element is in set is fast. Sets are written with curly brackets.

```
a = {1, 2, 'a'}
b = set()
b.add(2)
b.add('b')
print(2 in a and 2 in b) # prints True
print(a.intersection(b)) # prints {2}
```

(d) **Dictionaries**: Dictionaries are python's associative arrays, whose values are accessed using keys. Integers, tuples and strings can be used as dictionary keys. The dictionary values are accessed using using brackets (`[]`).

```
c = {'d': 15, 'e': 'f'}
print(c['d']) # prints 15
```

(e) **Indentation**: Python uses indentation to separate blocks of code, such as functions, loops and if statements. Indentation can be either tabs or spaces, but the indentation needs to be consistent.

(f) **For loops**: The core of the for loop must be indented in python. Python can iterate over many different objects (they are called *iterables*) like tuples and lists. Explicit indexing (as in C or Java) is not needed.

```
a = [1,5,4,2,3,4,5,6,7,8,1]
for x in a: # iterates over elements directly
    print(x) # this must be indented
```

If you need an index for some reason, the simplest way of getting both at the same time is using `enumerate`

```
a = [1,5,4,2,3,4,5,6,7,8,1]
for i, x in enumerate(a): # iterates over elements using an enumerator object
    print(i,x) # displays both the index and the corresponding value
```

(g) **Function definition**: Functions are defined by the keyword `def`. Function arguments are defined in parentheses. Python will accept any type of arguments and only produce an error at runtime if argument types are incompatible.

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

(h) **Unpacking**: Python supports unpacking a tuple into named variables. If you know the number of tuple elements you can assign a variable name to each one. This is especially useful when a function returns more than one value.

```
def myfunction(a,b):
    return a+b, a*b, a**b

my_sum, my_product, my_power = myfunction(2,3) # unpack the result immediately
```

(i) **Indexing**: Elements of iterable objects (such as tuples or lists) can be accessed using brackets. The basic functionality of accessing one element (e.g. `a[2]`) is actually shorthand for more complex indexing using colon `:`. The full indexing syntax is `iterable[start:end:step]`, where `start` and `end` signify the first and the last index. These indexing statements create a new array. Negative indices are supported for all of the indexing arguments and they wrap around the array. Accessing the last element is performed by `a[-1]`. Note that the wrap around is only defined once (i.e. for an array of length n, the valid indices are on interval `[-n,n-1]`).

```
a = [0,1,2,3,4,5]
a[2] # 2
a[3:] # [3,4,5]
a[-2:] # [4,5]
```

(j) **For loops in one line**: Sometimes it is cleaner to do some simple operations in one line. Ternary operators are also supported in the same way:

```
a = np.arange(10) # list of all integers from 0 to 9
[x**2 for x in a] # squaring every element of the list
```

```
[x**2 if x % 2 == 0 else x for x in a] # squaring only even elements
[x for x in a if x % 2 == 0 and x % 3 == 0] # extracting elements divisible by 2 and 3
```

2 Numpy and array iteration

(a) **Numpy**: Numpy is a computational library. It supports multidimensional arrays and functions to manipulate them. The arrays are defined by their type and their size.

```
a = np.array([[1,2],[3,4]]) # create a numpy array from nested lists
print(a.shape, a.dtype) # (2,2) int32
print(a[1,1]) # 4

z = np.zeros((2,5)) # creates a new array of zeros
c = np.ones_like(z) # creates an array of ones the size of z (equivalent to np.ones(z.shape))
```

Shape para el tamaño.

(b) **Data types**: The default integer data type in numpy is `np.int32`, while for floating point numbers it is `np.float64`. Type conversion can be performed using the function `np.astype()`.

(c) **Array indexing**:

Arrays are indexed by their dimensions using brackets (`[]`). Multidimensional arrays can be indexed using multiple indices in sequence. Each of them uses the standard python indexing syntax and can contain more complex indexing patterns. For example `I[0:100, 0::2]` returns the first 100 rows and every second column from a 2-D array.

(d) **Iterating over multidimensional arrays**:

This can be done with nested `for` loops but due to increasing time complexity should be avoided if at all possible.

```
# Open an image and convert it to Numpy array.
I = Image.open('image.jpg').convert('RGB')
I = np.asarray(I)

for i in np.arange(I.shape[0]):
    for j in np.arange(I.shape[1]):
        pixel_value = I[i,j]
```

(e) **Reshaping arrays**:

Sometimes, the spatial component of the data is not as important. In these cases, arrays can be unrolled into 1-D vectors. Arrays can also be reshaped using the function `np.reshape()` as long as the number of elements remains the same. The transpose of an array can be accessed using `np.transpose()`.

```
a = np.random.rand(4,6) # shape: (4,6)
b = a.reshape(-1) # shape: (24,)
c = a.reshape((12,2)) # shape: (12,2)
d = a.transpose() # shape (6,4)
a.T # equivalent to d
np.transpose(a) # equivalent to d
```

(f) **Accessing specific elements of an array**:

Sometimes it can be useful to only access a small subset of array elements based on some condition. This can be done by creating a boolean array that will serve as an index for the wanted elements. The seleced elements can be extracted or replaced in the same way.

```
a = np.random.randint(0,100, (10,10)) # array of random integers from interval [0,100]
mask = a%13==0 # boolean mask of the conditional
res = a[mask] # array of all array elements marked by the mask array (all elements divisible ↩
    by 13)

a[mask]=a[mask]**3 # cubing the selected elements and replacing them in the original array
```

(g) **Transposing an array**:

```
k = np.array([1, 2, 3], dtype=float)
print(k) # [1. 2. 3.]
print(k.T) # The same result, [1. 2. 3.]
k = np.expand_dims(k, 0) # array needs to be 2D to support transposing
print(k) # [[1. 2. 3.]]
print(k.T) # [[1.], [2.], [3.]]
```

3 Plotting with pyplot

(a) **Subplots**:

Subplots are used to display multiple images in the same windows. The first two arguments define the grid shape (e.g. plt.subplot(2,3,x) defines a 2 row, 3 column grid) and the last one is used to focus on a specific subplot cell. This means that after calling `plt.subplot` with a specific index, the next `plt.imshow` call will be displayed in the currently active grid cell. Note that the cells are linearly indexed and the indices start with `1`.

```
# Open an image and convert it to Numpy array.
I = Image.open('image.jpg').convert('RGB')
I = np.asarray(I)

plt.clf() # clears the figure (is necessary if plotting many images in a sequence)

plt.subplot(1,2,1) # creates two columns and focuses on the left one
plt.imshow(I)
plt.title('RGB image') # displays a title for the subplot cell

plt.subplot(1,2,2)
plt.imshow(I[...,0])
plt.title('blue channel')

plt.show()
```