Programming for Artificial Intelligence

Muhammad Adeel Nisar

Assistant Professor – Department of IT,
Faculty of Computing and Information Technology,
University of the Punjab, Lahore

Contents

- Operators (arithmetic, assignment, comparison, logical, bitwise)
- Control Structures (selection: if-elif-else, repetition: for, while)
- Data structures (lists, tuples, set, dictionary)
- Functions
- File handling

Operators

- Arithmetic Operators (+, -, *, /, %, **, //)
- Assignment Operators (+=, -=, *=, /=, %=, **=, //=)
- Comparison Operators (<, >, <=, >=, ==, !=)
- Conditional Operators (and, or, not)

Lists

- Items separated by commas and enclosed within square brackets ([])
  - Similar to arrays in C.
- **Indexing starts at 0**
- Items can be of different data type
- Items can deleted: *del*

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print(list) # Prints complete list
print(list[0]) # Prints first element of the list
print(list[1:3]) # Prints elements starting from 2nd till 3rd
print(list[2:]) # Prints elements starting from 3rd element
print(tinylist * 2) # Prints list two times
print(list + tinylist) # Prints concatenated lists
```

Tuple

- Items separated by commas and enclosed within parentheses
- Indexing starts at 0
- **Cannot be updated**, i.e. read-only lists

```
# Creates a tuple
tuple = (1,2,3)
# Prints tuple
print(tuple)
# Prints first element of tuple
print(tuple[0])
# Not allowed:
tuple[0] = 1
```

Dictionary

- Kind of hash table type
- Consist of **key-value pairs**
  - Keys can be almost any Python type
  - Values can be any arbitrary Python object
- Dictionaries are enclosed by curly braces ({ })

```
python_dict = {'name': 'john','code':6734, 'dept': 'sales'}

# Prints complete dictionary
print(python_dict)
# Prints all the keys
print(python_dict.keys())
# Prints all the values
print(python_dict.values())
# Prints value for given key
print(python_dict['code'])
```

Functions

- The keyword *def* introduces a function definition
- Followed by:
  - function name
  - parenthesized list of parameters
- Function body must be intended

```
def print_square (number):
        print(number * number)

print_square(2)
```

Example procedure (there is no return value)
In fact Python returns *None*

Functions – Return value

- *Return* statement to return a value of the function

```
def square (number):
    return number * number
a = square(2)
```

- Python does not support *overloading* a function

Functions - Argument

- Default argument values

```
def func(mandatory_var, default_arg_var=2):
        print(mandatory_var, default_arg_var)

func(5,5)
func(5)
```

- Keyword arguments

```
# allowed:
func(mandatory_var = 5, default_arg_var = 5)
func(default_arg_var = 5, mandatory_var = 5)
func(mandatory_var = 5)
func(5, default_arg_var = 5)
# not allowed:
func(default_arg_var = 5)
func(5, mandatory_var = 5)
```

Array and list indexing (1/4)

The elements of arrays and lists can be accessed using the **brackets** [...] notation.

Examples:

```
>>> import numpy as np
>>> T = np.arange(5) # Creates array([0, 1, 2, 3, 4])
>>> T[0] # First element of T
0
>>> T[2] # 3rd element of T
2
>>> T[-1] # Last element of T
4
>>> T[-2] # Last-but-one element of T
3
```

Array and list indexing (2/4)

It is possible to access specific elements of an array using **slice indexing.** Given an array or list T:

```
T[start:end:increment]
```

returns all elements of T between the indices `start` (included) and `end` (excluded) by intervals of `increment`.

Note 1: `start`, `end` and `increment` are all optional (integer) arguments. If not specified, `increment` is equal to 1 by default.

Note 2: this syntax still works if T is a multidimensional array

Array and list indexing (3/4)

Slice indexing examples:

```
>>> import numpy as np
>>> T = np.arange(5) # Creates array([0, 1, 2, 3, 4])
>>> T[1:4] # Elements of T between the 2nd and 4th
array([1,2,3])
>>> T[:3] # All elements of T until the 3rd one
array([0,1,2])
>>> T[2:] # All elements of T from the 3rd to the end
array([2,3,4])
>>> T[1:4:2] # All second elements of T between the 2nd and 4th
array([1,3])
>>> T[:] # Returns all elements of T; equivalent to T[::] and T
array([0,1,2,3,4])
```

Array and list indexing (4/4)

Extracting a sub-array can also be performed using **list indexing**. The list can contain either boolean or integers.

Examples:

```
>>> import numpy as np
>>> T = np.array([12,5,-3,7,24])
>>> b = [True,False,False,True,True] # List of booleans
>>> T[b]
array([12,7,24])
>>> idx1 = [3,2,0,4,1] # List of integers with same length than T
>>> T[idx1]
array([7,-3,12,24,5])
>>> idx2 = [2,3,0] # List of integers shorter than the length of T
>>> T[idx2]
array([-3,7,12])
```

Exercise 1: Shuffling in Unison

Write a Python function `shuffleInUnison` which shuffles (i.e. re-orders) the elements of two arrays of same length using the **same** random permutation:

```
shuffledT1, shuffledT2 =
shuffleInUnison(T1,T2)
```

With:

- **[input]** T1 and T2: arrays assumed to have same length
- **[output]** `shuffledT1` and `shuffledT2`: randomly shuffled input arrays/lists using the same permutation

Tip: for random related functions, use the `numpy.random` package

Multidimensional array manipulation (1/3)

Some useful options to initialize a multidimensional array:

- Direct initialization with `numpy.array`

```
>>> import numpy as np
>>> T = np.array([[1,2],[3,4]]) # 2x2 array containing the values 1, 2, 3, 4
```

- Initialization with numpy functions

```
>>> import numpy as np
>>> T0 = np.zeros((2,2),dtype=int) # 2x2 array to zero with integer type
>>> T1 = np.ones((100,50,200),dtype=float) # 3D array to one with float type
>>> TRand = np.random.rand(10,20) # 2x2 array of random float (default) values
```

Multidimensional array manipulation (2/3)

Multidimensional array **indexing** and **slicing** works the same way than for the 1D case. In particular, for a N-dimensional array T it is possible to use the following syntax to obtain a specific slice of T:

```
T[start1:end1:incr1,start2:end2:incr2,…,startN:endN:incrN]
```

Examples:

```
>>> import numpy as np
>>> T = np.random.rand(10,5,20,10,dtype=float) # 4D array of random float values
>>> s1 = T[-2,3,15,0] # Float element at position (8,3,15,0)
>>> s2 = T[:,3,:,:] # 3D slice of shape (10,20,10)
>>> s3 = T[5,0:5:2,9,:] # 2D slice of shape (3,10)
>>> s4 = T[1:4,:,:,-1] # 3D slice of shape (3,5,20)
```

Multidimensional array manipulation (3/3)

Numpy arrays are **mutable**: it is possible to change their values after initialization.

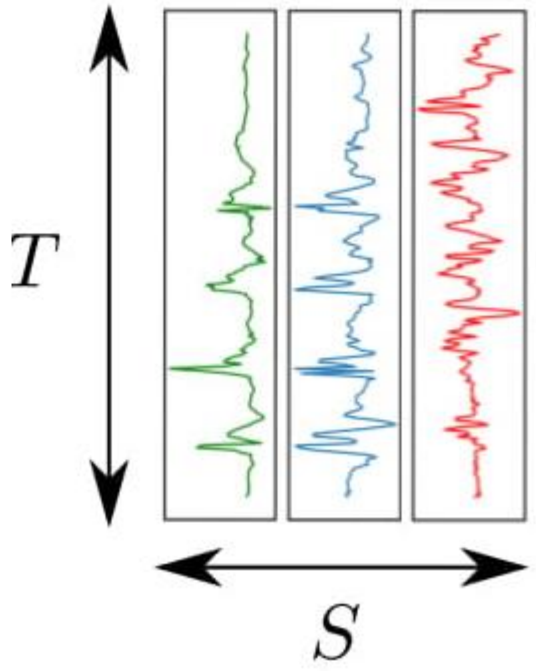
Examples:

```
>>> import numpy as np
>>> T = np.zeros((2,2)) # 2x2 array of random float values
>>> T[0,0] = 1; print(T)
array([[1.,0.],
       [0.,0.]])
>>> T[:,1] += 3; print(T) # Increment by 3 the 2nd column of T
array([[1.,3.],
       [0.,3.]])
>>> T[1,:] = -2; print(T) # Set the 2nd line of T to –2
array([[1.,3.],
       [-2.,-2.]])
```

Exercise 2: Standard Normalization (1/2)

Multimodal time-series data records can be represented as a 2D array of size T x S with:

- T: number of timestamps (i.e. duration of the data record)
- S: number of sensor channels



It is usually needed to perform pre-processing operations on the raw data records. **Standard normalization** is one of them:

$$X \leftarrow \frac{X - \mu}{\sigma}$$

with $\mu = mean(X)$ and $\sigma = std(X)$