Lab Manual for Tools and Technologies for Data Science

Lab-02

Introduction to Python and NumPy

Table of Contents

Lab 2: Introduction to Python and NumPy

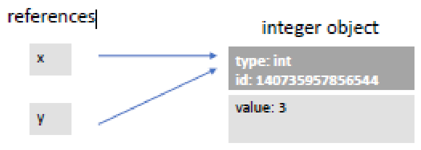
1. Objective

- Define functions and pass parameters to functions with examples
- Controlling program flow
- Learn Python data types and sequences
- Iterate over larges set of data using for and while loops
- Differentiate between set and dictionaries with the help of
- Learn NumPy package and its features
- Use Simple array vs NumPy array
- Apply NumPy array manipulations with respect to data science
- Concatenate and split data with real world examples.
- Explore NumPy package applications in data science
- Use fundamental Python objects for data science algorithms

2. Python Language Fundamentals

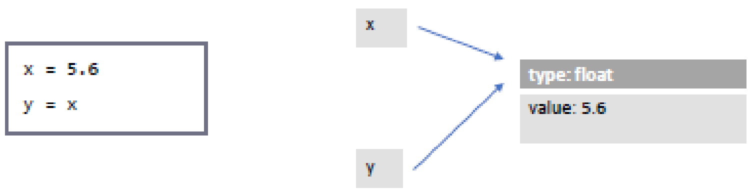
This section provides you the overview of the python language fundamentals that will be discussed and implemented in this lab.

Python is an object-oriented language Every piece of data in the program is an Object. Reference = symbol in a program that refers to a particular object. A single Python object can have multiple references (alias). In Python Variable = reference to an object. When you assign an object to a variable it becomes a reference to that object



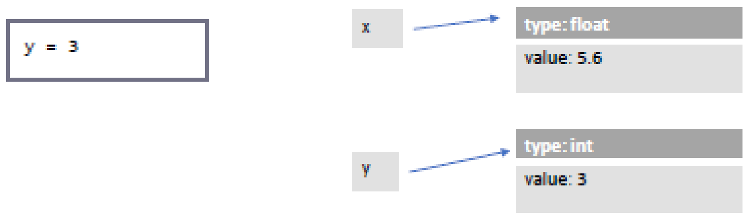
Defining a variable:

- ❖ No need to specify its data type
- ❖ Just assign a value to a new variable name e.g. a =3
  - Example



```
x = 5.6
y = x
```

- If you assign y to a new value…



```
y = 3
```

2.1 Data Types

- Basic data types
  - int, float, bool, str
  - None
  - All of these objects are immutable
- Composite data types
  - tuple immutable list of objects)
  - list, set, dict (mutable collections of objects)

2.2 int,float

- ❖ Available operations
  - +, --, *, /, // (integer division), % reminder, **

Example:

```
x = 9
y = 5
r1 = x // y# r1 = 1
r2 = x % y# r2 = 4
r3 = x / y# r3 = 1.8
r4 = x ** 2# r4 = 81
```

2.3 bool

- ❖ Can assume the values True, False
- ❖ Boolean operators: and, or, not

Example:

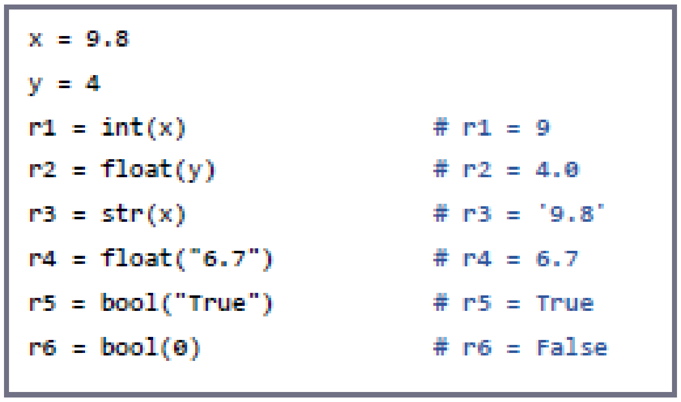
```
is_sunny = True
is_hot = False
is_rainy = notis_sunny# is_rainy = False
bad_weather = not(is_sunny oris_hot)  # bad_weather = False
temperature1 = 30
temperature2 = 35
growing = temperature2 >temperature1# growing = True
```

2.4 String

```
string1 = "Python's nice"# with double quotes
string2 = 'He said " yes"'# with single quotes
print(string1)
print(string2)
```

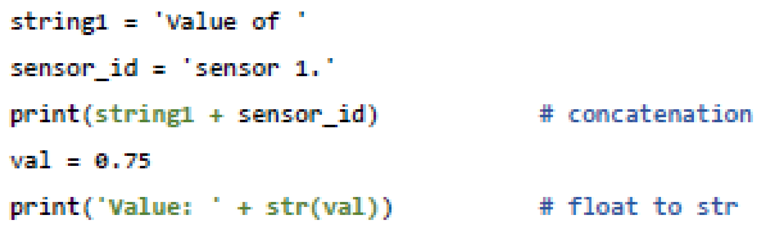
Note: Definition with single or double quotes is equivalent

2.5 Conversion between types



```
x = 9.8
y = 4
r1 = int(x)                # r1 = 9
r2 = float(y)              # r2 = 4.0
r3 = str(x)                # r3 = '9.8'
r4 = float("6.7")          # r4 = 6.7
r5 = bool("True")          # r5 = True
r6 = bool(0)               # r6 = False
```

2.6 Strings: Concatenation



```
string1 = 'Value of '
sensor_id = 'sensor 1.'
print(string1 + sensor_id)          # concatenation
val = 0.75
print('Value: ' + str(val))         # float to str
```

2.7 Sub-strings

- str[start:stop]
  - The start index is included, while stop index is excluded
  - Index of characters starts from 0

- x = **'This is python class'**
  print(x[0])
  print(x[1])
  print(x[0:2])
  # Negative Index also works which is cool
- *Negative indices: count characters from the end*
- *-1 = last character*
  print(x[-4:-2])
- fname = **'John'**
  sname = **'wick'**
  print(fname + sname)
  print(fname *3)
  print(**'John' in** fname)

Note: Strings are immutable
str1 = "example"
str1[0] = "E"# will cause an **error**

2.8 None type

- Specifies that a reference does not contain data.

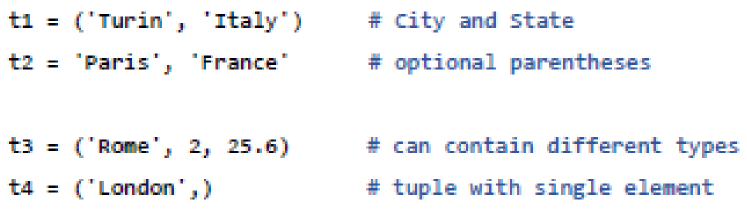
-
  ```
  def add_numbers(x,y,z=None):
      if(z == None):
          return x+y
      else:
          return x+y+z

  print(add_numbers(1,2,5))
  ```

Useful to:

- Represent "missing data" in a list or a table
- Initialize an empty variable that will be assigned later on
- e.g. when computing min/max

2.9 Tuple

I. Immutable list of variables
   - x = (1,'a',2,'b')   #Tuple class
   - print(type(x))



```
t1 = ('Turin', 'Italy')     # City and State
t2 = 'Paris', 'France'      # optional parentheses

t3 = ('Rome', 2, 25.6)      # can contain different types
t4 = ('London',)            # tuple with single element
```

II. Assigning a tuple to a set of variables

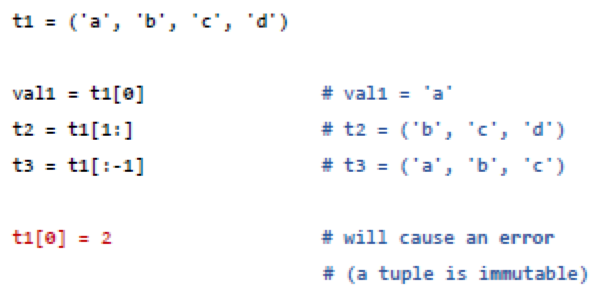
```
t1 = (2, 4, 6)
a, b, c = t1
print(a)      # 2
print(b)     # 4
print(c)     # 6
```

III. Swapping elements with tuples

```
a = 1
b = 2
a, b = b, a
print(a)
print(b)
```

IV. Accessing elements of a tuple

```
t [start:stop]
```



```
t1 = ('a', 'b', 'c', 'd')

val1 = t1[0]                # val1 = 'a'
t2 = t1[1:]                 # t2 = ('b', 'c', 'd')
t3 = t1[:-1]                # t3 = ('a', 'b', 'c')

t1[0] = 2                   # will cause an error
                            # (a tuple is immutable)
```

2.10 List

Mutable sequence of heterogeneous elements
Each element is a reference to a Python object
x = [1,'a',2,'b']   //List Class
print(type(x))
Adding elements and concatenating lists
l1 = [2, 4, 6]
l2 = [10, 12]
l1.**append**(8)   # append an element to l1
l3 = **l1 + l2**   # concatenate 2 lists
print(l1)
print(l3)
Other Methods:
list1.count(element):
Number of occurrences of element
list1.extend(l2):
Extend list1 with another list l2
list1.insert(index, element):
Insert element at position
list1.pop(index):
Remove element by position

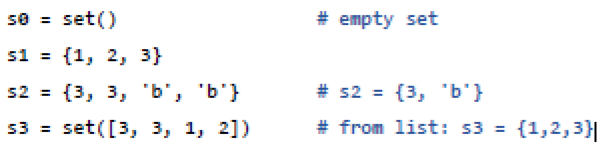
Note: Iterate over list elements

```
for item in x:
    print(item)

i=0
while (i != len(x)):
    print(x[i])
    i = i+1
```

```
for x in range(10):
    if x == 3:
        continue # go immediately to the next iteration
    if x == 5:
        break # quit the loop entirely
    print(x)
```

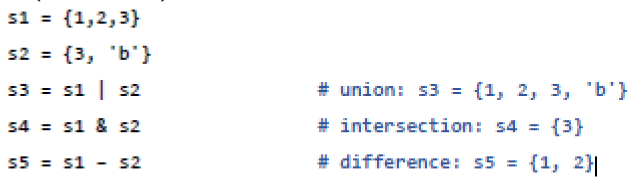
3. Set

Unordered collection of unique elements.



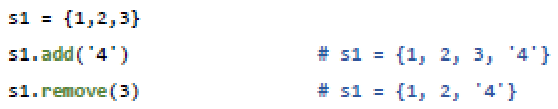
```
s0 = set()                  # empty set
s1 = {1, 2, 3}
s2 = {3, 3, 'b', 'b'}       # s2 = {3, 'b'}
s3 = set([3, 3, 1, 2])      # from list: s3 = {1,2,3}
```

Operators between two sets
| (union), & (intersection), --(difference)



```
s1 = {1,2,3}
s2 = {3, 'b'}
s3 = s1 | s2                # union: s3 = {1, 2, 3, 'b'}
s4 = s1 & s2                # intersection: s4 = {3}
s5 = s1 - s2                # difference: s5 = {1, 2}
```

Add/remove elements



```
s1 = {1,2,3}
s1.add('4')                 # s1 = {1, 2, 3, '4'}
s1.remove(3)                # s1 = {1, 2, '4'}
```

Set example: removing list duplicates

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list) # 6
item_set = set(item_list) # {1, 2, 3}
print(item_set)
```

4. Dictionary

Collection of key value pairs
Allows fast access of elements by key
Keys are unique:

```
x={'ITU':'itu@edu.pk','UCP': 'ucp@edu.pk'}
print(x['ITU'])

x={'ITU':'itu@edu.pk','UCP': 'ucp@edu.pk'}
for name, email in x.items():
    print(name)
    print(email)
```

4.1 Access by Key:

```
d1 = {'key1' : 'val1', 5 : 6.7}
val1 = d1['key1']           # val1 = 'val1'
val2 = d1[5]                # val2 = 6.7
val3 = d1['not_valid']      # Get an error if key does not exist
```

4.2 Reading keys and values

```
d1 = {'key1' : 1, 'key2' : 2}
keys = list(d1.keys())      # keys = ['key1', 'key2']
values = list(d1.values())  # values = [1, 2]
```

4.3 Adding updating values

```
d1 = {'key1' : 'val1', 5 : 6.7}
d0['key1'] = 3              # Update existing value
d0['newkey'] = 3            # Add a new key
```

4.4 Deleting a Key:

```
d2 = {'key1':1, 'key2':2}
del d2['key1']              # d1 = {'key2':2}
```

4.5 Iterating keys and values

```
d1 = {'Cola' : 0.99, 'Apples' : 1.5, 'Salt' : 0.4}
cost = 0
for k, v in d1.items():
    cost += v
    print(f"{k}: {cost}")
```

5. NumPy (Numerical Python)

techniques for effectively loading, storing and manipulating in-memory data in python. Datasets can come from a wide range of sources and a wide range of formats, including collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers. No matter what the data are, the first step in making them analyzable will be to transform them into arrays of numbers. For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. We'll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package and the Pandas package. This Lab will cover NumPy in detail. NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interest you.

Features
- Multidimensional arrays
- Slicing/indexing
- Math and logic operations

Applications
- Computation with vectors and matrices
- Provides fundamental Python objects for data science algorithms

array is the main object provided by NumPy

Characteristics
- Fixed Type
- All its elements have the same type

Multidimensional
- Allows representing vectors, matrices and n dimensional arrays

Simple Array in python //Mix items

```
import array
L = list(range(10))
A = array.array('i', L)
print(A)
NumPy Array
np.array([1, 4, 2, 5, 3])
```

NumPy advantages:
- Higher flexibility of indexing methods and operations
- Higher efficiency of operations

Characteristics of NumPy arrays:
- Fixed type (no overhead)
- Contiguous memory addresses (faster indexing)
- E.g. my_numpy_array = np.array([0.67, 0.45, 0.33])

NumPy data types:
- NumPy defines its own data types
- Numerical types
  - int8, int16, int32, int64
  - uint8, ... , uint64
  - float16, float32, float64
  - Boolean values
  - bool

`Note:` Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are upcast to floating point):

Creating arrays from Scratch

```
#Create a length 7 int-array filled with zeros
np.zeros(7, dtype=int)
# Create a 3x5 floating-point array filled with 1s
np.ones((3, 5), dtype=float)
# Create a 3x5 array filled with 2.9
np.full((3, 5), 2.9)
# Create a 3x3 identity matrix
np.eye(3)
```

6. NumPy Arrays Manipulations:

Attributes of arrays

Determining the size, shape, memory consumption, and data types of arrays

Indexing of arrays

Getting and setting the value of individual array elements

Slicing of arrays

Getting and setting smaller subarrays within a larger array

Reshaping of arrays

Changing the shape of a given array

Joining and splitting of arrays

Combining multiple arrays into one, and splitting one array into many

6.1 A**ttributes of a NumPy array**

Consider the array

x = np.array([[2, 3, 4],[5,6,7]])

x.ndim: number of dimensions of the array

Out: 2

x.shape : tuple with the array shape

Out: (2,3)

x.size : array size (product of the shape

Out: 2*3=6

Moreover, let's start by defining three random arrays: a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
import numpy as np
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=5) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = (np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array

print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

Other are dtype, itemsize and nbytes.

6.2 **Array Slicing and Indexing**

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character.
The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values start=0, stop=*size of dimension*, step=1. We'll take a look at accessing subarrays in one dimension and in multiple dimensions. X[:5] //first five
A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array:

```
x1 = np.random.randint(10, size=5)
print(x1)
print(x1[::-1])
```

Multidimensional arrays

```
x2[:2, :3] # two rows, three columns,                    x2[:3, ::2] # all rows, every other column
```

Finally, subarray dimensions can even be reversed together: `x2[::-1, ::-1]`

6.3 **Reshaping of Array**

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape() method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
[4 5 6]
[7 8 9]]
```

Change dimension

```
x = np.array([1, 4, 3])
print(x[:, np.newaxis])
```

6.4 **Array Concatenation and Splitting**

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines np.concatenate, np.vstack, and np.hstack. np.concatenate takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
print(np.concatenate([x, y]))
```

More than two arrays at once

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

7. Evaluation Task (Unseen) [Expected time = 30mins for tasks]

I will give you unseen task depending upon the progress of the class.

7.1 Practice Task 1

```
def do_math(?, ?,?):
  if (wish=='add'):
    return a+b
  else:
    return a-b
do_math(1, 2)
```

7.2 Practice Task 2

```
x = 'This is python class'
```

What is the output of this? x[:3], x[3:]

7.3 Practice Task 3

What would be an appropriate slice to get the name "Christopher" from the string "Dr. Christopher Brooks"?

```
x = 'Dr. Christopher Brooks'

print(x[????])
```

8. Evaluation criteria

The evaluation criteria for this lab will be based on the completion of the following tasks. Each task is assigned the marks percentage which will be evaluated by the instructor in the lab whether the student has finished the complete/partial task(s).

Table 3: Evaluation of the Labs

Sr. No.	**Description**	**Marks**
1	Problem Modeling	20
2	Procedures and Tools	10
3	Practice tasks and Testing	35
4	Evaluation Tasks (Unseen)	20
5	Comments	5
6	Good Programming Practices	10