

DS 440 Data Mining

Lecture 2: Python Basics

1 Python data types

This notebook discusses the various data types available by default in Python. Additional data types like array and dataframes are possible from additional packages like Numpy and Pandas etc.

1.1: String data types ¶

String is defined in double or single quotes in python

```
In [1]: s = 'Hello World'
print(f'Variable s has value: {s}')
print(f'Type of s is: {type(s)}')  ## type(s) gives the class to which
```

```
Variable s has value: Hello World
Type of s is: <class 'str'>
```

```
In [4]: x = 4
print(f'This is x')
```

```
This is x
```

```
In [5]: print(f'This is {x}')
```

```
This is 4
```

print function expects string as arguments. print('Variable s has value: '+s) basically concatenates two strings using the + operator and prints the resultant string. For example

```
In [7]: 'This'+ ' is the 2nd class'
```

```
Out[7]: 'This is the 2nd class'
```

```
In [2]: 'this'+' '+'is'
```

```
Out[2]: 'this is'
```

Same example with double quotes

```
In [11]: print("Albert's")
```

```
Albert's
```

```
In [12]: s = 'Albert'
         type(s)
```

```
Out[12]: str
```

```
In [3]: s = "Hello World"
         print(f'Variable s has value: {s}')
         print(f'Type of s is: {type(s)}')
```

```
Variable s has value: Hello World
Type of s is: <class 'str'>
```

Single and double quotes

Suppose we want to print the string: Sam's car

Then double quotes is required to define this string so that python is able to differentiate between outer quotes and ' in Sam's car. Hence following would give an error

```
In [4]: s = 'Sam's car'
         print(s)
```

```
File "<ipython-input-4-3f15ab3535f9>", line 1
    s = 'Sam's car'
        ^
SyntaxError: invalid syntax
```

This is the correct way to do it.

```
In [5]: s = "Sam's car"
        print(s)
```

Sam's car

One more example

```
In [13]: print('Hi')
```

"Hi"

```
In [14]: print("'Hi'")
```

'Hi'

1.2: Numerical Types

Here we show how to define different kind of numerical data types

```
In [15]: a = 1
        type(a)
```

Out[15]: int

```
In [16]: b = 1.1
        type(b)
```

Out[16]: float

```
In [18]: a = 2.5
        b = 2.5
        type(a+b)
```

Out[18]: float

```
In [16]: a1 = float(a)
        print(a1)
```

1.0

```
In [21]: a = 2.999  
b = int(a)  
b
```

Out[21]: 2

a = 2.4 b = round(a) b

Casting float as integer removes the decimal part

```
In [19]: a = 2.1  
b = int(a)  
b
```

Out[19]: 2

```
In [24]: import numpy as np
```

```
In [26]: np.floor(2.3)
```

Out[26]: 2.0

We can also do complex data types

```
In [27]: c = complex(2+1.1j)  
type(c)
```

Out[27]: complex

```
In [32]: print(c)  
a = 'this'
```

(2+1.1j)

Separating the real and imaginary components

```
In [21]: print(f'The real part is: {c.real}')  
print(f'The imaginary part is: {c.imag}')
```

The real part is: 2.0
The imaginary part is: 1.1

1.3: Sequence Types

Defining data objects in python which are structured as a collection and have the notion of indexing

```
In [34]: a = [1,2,3,4,'erau']  
         type(a)
```

```
Out[34]: list
```

```
In [39]: a[-2]
```

```
Out[39]: 4
```

```
In [ ]: a.
```

Many other operations like appending is possible for the list object

```
In [40]: a.append(5)  
         a
```

```
Out[40]: [1, 2, 3, 4, 'erau', 5]
```

```
In [27]: #a.remove(2)  
         #a
```

For accessing elements in the list:

1. 0 index denotes the first element in the list. Indexing in python always starts from 0 (unless specified otherwise)
2. -1 is the index of the last element, -2 is the index of the second last element and it goes like that from the back

```
In [28]: print(a[0],a[2],a[-1],a[-2])
```

```
1 4 5 5
```

Like list, we also have a data object called as tuple. It uses () instead of [] (as in list)

```
In [41]: b = (1,2,3)
         type(b)
```

```
Out[41]: tuple
```

Re-assigning elements in the list and tuple object.

With list it works fine when we try to assign the element at index 0 to 9. Right now it is 1 (for list a)

```
In [42]: a
```

```
Out[42]: [1, 2, 3, 4, 'erau', 5]
```

```
In [43]: a[0] = 9
         a
```

```
Out[43]: [9, 2, 3, 4, 'erau', 5]
```

However, tuple is an immutable object, so reassigning values is not possible with the tuple object

```
In [44]: b
```

```
Out[44]: (1, 2, 3)
```

```
In [45]: b[0] = 9
         b
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
Input In [45], in <cell line: 1>()
----> 1 b[0] = 9
      2 b

TypeError: 'tuple' object does not support item assignment
```

```
In [46]: type(b)
```

```
Out[46]: tuple
```

```
In [47]: c = list(b)
c
```

```
Out[47]: [1, 2, 3]
```

```
In [48]: c = (1,2,3,'Tuesday')
c
```

```
Out[48]: (1, 2, 3, 'Tuesday')
```

```
In [49]: list(c)
```

```
Out[49]: [1, 2, 3, 'Tuesday']
```

```
In [ ]:
```

Using range function to generate the sequence

```
In [33]: list(range(10))
```

```
Out[33]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [50]: range(10)
```

```
Out[50]: range(0, 10)
```

```
In [ ]:
```

Testing various ways of generating a list object. For better understanding, try to think why these commands lead to the following output

```
In [51]: c1 = list(range(10));print(c1)
c2 = list(range(0,10));print(c2)
c3 = list(range(1,10));print(c3)
c4 = list(range(10,20));print(c4)
c5 = list(range(10,5));print(c5)
c6 = list(range(-2,10));print(c6)
c7 = list(range(-2,10))[2:10];print(c7)
c8 = list(range(-2,10))[:-10];print(c8)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[]
[-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7]
[-2, -1]
```

```
In [53]: list(range(-2,10))[2:5]
```

```
Out[53]: [0, 1, 2]
```

```
In [54]: list(range(2,10))
```

```
Out[54]: [2, 3, 4, 5, 6, 7, 8, 9]
```

Another way of generating a list using range and using a for loop to iterate over the range object

```
In [56]: b = (1,2,3)
```

```
In [57]: d = [i for i in b]
d
```

```
Out[57]: [1, 2, 3]
```

For example we can use this method to compute a list of log of number from 1 to 9. Here I am using the numpy package as numpy includes the log functionality


```
In [58]: import numpy as np
d = [np.sin(i) for i in np.arange(1,10)]
d
```

```
Out[58]: [0.8414709848078965,
0.9092974268256817,
0.1411200080598672,
-0.7568024953079282,
-0.9589242746631385,
-0.27941549819892586,
0.6569865987187891,
0.9893582466233818,
0.4121184852417566]
```

```
In [ ]:
```

1.4: Mapping type

These kind of objects basically have a key value pair. For example the dictionary object in python

```
In [61]: Dict = {
    'Tom': [1,2,3],
    'Sam': (5,6,7),
    'Eric': list(range(10)),
    1: [1,2]
}
Dict
```

```
Out[61]: {'Tom': [1, 2, 3],
'Sam': (5, 6, 7),
'Eric': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
1: [1, 2]}
```

```
In [62]: Dict['Tom']
```

```
Out[62]: [1, 2, 3]
```

Here for example 'Tom' would be a key and [1,2,3] would be an object. Please note we have used a mixture of strings and an integer as key here. This is valid for dictionaries

We cannot use indexing here because dictionary is just a collection of key value mapping pairs

```
In [38]: Dict[0]
```

```
-----  
-----  
KeyError                                Traceback (most recent call  
last)  
Input In [38], in <cell line: 1>()  
----> 1 Dict[0]  
  
KeyError: 0
```

For accessing the elements we have to index the dictionary object with a key value

```
In [39]: Dict['Tom']
```

```
Out[39]: [1, 2, 3]
```

```
In [40]: Dict[1]
```

```
Out[40]: [1, 2]
```

1.5. Set types

Set is again a collection. Here also we don't usually have indexing. Uses {} to define

```
In [52]: a = {10,1,2,3}  
         type(a)
```

```
Out[52]: set
```

```
In [53]: a[0]
```

```
-----  
-----  
TypeError                                Traceback (most recent call  
last)  
Input In [53], in <cell line: 1>()  
----> 1 a[0]  
  
TypeError: 'set' object is not subscriptable
```

Editing the set object is possible. For example adding one more integer to the object a

```
In [56]: a.add(11)
print(a)

{1, 2, 3, 10, 11}
```

We can generate some sort of ordering from a set by using other functions like

```
In [57]: print(sorted(a))  ## gives ascending order

[1, 2, 3, 10, 11]
```

frozensets are immutable analogs to sets

```
In [58]: b = frozenset(a)
type(b)
```

Out[58]: frozenset

```
In [59]: b.add(10)
print(b)
```

```
-----
AttributeError                                Traceback (most recent call
last)
Input In [59], in <cell line: 1>()
----> 1 b.add(10)
      2 print(b)

AttributeError: 'frozenset' object has no attribute 'add'
```

1.6. Boolean Types

This type of variable only allows two states: True or False

```
In [60]: a = bool(-2)
a
```

Out[60]: True

```
In [61]: print(bool(100),bool(10),bool(-30),bool(0))
```

True True True False

An int, float or complex number set to zero returns False. An integer, float or complex number set to any other number, positive or negative, returns True.

```
In [62]: print(bool(complex(10j)))  
print(bool(0))  
print(bool(complex(0+0j)))
```

True
False
False

```
In [63]: print(str(1>3)+' ', '+str(1<3)+' ', '+str(1==3))
```

False, True, False

```
In [ ]:
```

2. Control Flow

Here we will discuss ways to reorder the the default execution ordering

2.1 Conditional Statements

```
In [66]: if 5<6:  
        print('You are good in math')
```

You are good in math

```
In [67]: if 5>6:  
        print('You are good in math')
```

```
In [68]: if 5>6:  
        print('You are good in math')  
else:  
        print('Sorry you need to improve')
```

Sorry you need to improve

```
In [69]: if 5>6:
          print('You are good in math')
        elif 0<1:
          print('You are good but need to improve')
        else:
          print('You need to improve')
```

You are good but need to improve

2.2 Loops

1. For loops

```
In [70]: for i in range(3):
          print(i)
```

0
1
2

```
In [71]: for j in [1,2,3]:
          print(j)
```

1
2
3

```
In [72]: dictt = {
          2:'I',
          0:'Hi',
          4:'Prashant',
          3:'am',
          }
```

```
In [73]: for i in dictt:
          print(i,dictt[i])
```

2 I
0 Hi
4 Prashant
3 am

```
In [74]: for i in sorted(dictt):  
         print(i,dictt[i])
```

```
0 Hi  
2 I  
3 am  
4 Prashant
```

```
In [ ]:
```

2. While loop

```
In [75]: a = 0  
while(a<5):  
    print(a)  
    a=a+1
```

```
0  
1  
2  
3  
4
```

Special Iterable objects

```
In [76]: for a in 'abcde':  
         print(a)
```

```
a  
b  
c  
d  
e
```

```
In [77]: for i in ['Hi','how','are','you']:  
         print(i)
```

```
Hi  
how  
are  
you
```

There is no do-while loop in python

3. Break, continue and pass statements

1. Break statement is used to move out of the loop.
2. Continue statement is used to stop the current iteration and move forward with the next iteration.
3. Pass statement can be used to write a placeholder definition of a loop.

For your understanding try to explain the code in the following two cells.

```
In [141]: a = 0
while(a<5):
    print(a)
    if a>2:
        break
    a = a+1
```

```
0
1
2
3
```

```
In [142]: for i in range(5):
            if i == 3:
                continue
            print(i)
```

```
0
1
2
4
```

```
In [143]: for i in range(10):
```

```
Input In [143]
for i in range(10):
    ^
```

IndentationError: expected an indented block

```
In [144]: for i in range(10):
            pass
```

2.3 Conditional expression

These are expressions for checking conditions

== statement

```
In [145]: a = 1;b = 2  
a == b
```

Out[145]: False

'in' statement

```
In [146]: c = [1,2,3]  
a in c
```

Out[146]: True

'is' statement

```
In [147]: a = 1  
b = 1  
a == b
```

Out[147]: True

```
In [148]: a = [1,12,3]  
d = [1,12,3]  
a is d
```

Out[148]: False

```
In [149]: a = [1,12,3]  
d = a  
a is d
```

Out[149]: True

>=, != statements


```
In [150]: a = 10  
          b = 5  
          a >= b
```

Out[150]: True

Checking not equal to

```
In [108]: a != b
```

Out[108]: True

3 Functions

For defining functions we use the following syntax. `def` is the keyword used to define functions in python. Here `summation` is the function name and `a` and `b` are parameters. `Return` is a keyword used to return a value to the calling program

```
In [151]: def summation(a,b):  ## summation is the function name  
          summ = a+b  
          return summ        ## return statement is optional
```

Now passing 1 and 2 as parameters computes the sum of the two numbers

```
In [152]: summation(1,2)
```

Out[152]: 3

Based on the definition of the function, it always requires two parameters (denoted by `a` and `b` in the definition). So calling a function without any parameters would produce an error.

In [153]: `summation()`

```
-----
-----
TypeError                                Traceback (most recent call
last)
Input In [153], in <cell line: 1>()
----> 1 summation()

TypeError: summation() missing 2 required positional arguments: 'a' a
nd 'b'
```

We can have a definition of function as follows where we can assign default values of 10 and 20 to a and b so that no error is produced when no parameter is passed.

In [154]: `type(summation)`

Out[154]: `function`

```
In [155]: def summation_1(a=10,b=20):  ## providing default parameters
          summ = a+b
          return summ
```

```
In [156]: print(summation_1(1,2))
          print(summation_1())  ## when no parameters passed it just assumes a an
          3
          30
```

3.1 Lambda functions

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression. For example following is a lambda function to add 10 to a number

```
In [167]: x = lambda a: a + 10
          print(x(10))
```

20

Two inputs at ones

```
In [169]: y = lambda a,b: a**b
          print(y(2,3))
```

8

3.2 Be careful while changing values in the function

Value of variable changed inside a function are not reflected outside

```
In [117]: def change_values(a):
          a = a*2
          print('Value of a after change inside the function is: '+str(a))
```

```
In [118]: a = 10  ## original value of a
          print('Original value of a: '+str(a))
          change_values(a)  ## a function changing the value of 'a' internally
          print('However value of a outside the function is still: '+str(a)) ##
```

Original value of a: 10

Value of a after change inside the function is: 20

However value of a outside the function is still: 10

However, if a list is changed it retains the change outside the function as well. So you should be careful while changing values inside a function.

```
In [119]: def changelist(l):
          l.append(1)
          print('printing the list after changing it within the function '+s
```

```
In [120]: l = [-1,-2,-3]
          print('Original list is: '+str(l))
          changelist(l)
          print('Printing the list after function call is finished: '+str(l)) #
```

Original list is: [-1, -2, -3]

printing the list after changing it within the function [-1, -2, -3, 1]

Printing the list after function call is finished: [-1, -2, -3, 1]

3.3 Global variable

Variable defined outside the function is accessible from inside.

```
In [157]: a = 10  ## Global variable
def add_num(b):
    b = b+a  ## a is still accessible inside.
    print(b)
add_num(10)

20
```

3.4 Variable number of parameters

If we want to define a function which takes variable number of parameters. For example 1 parameter or 2 or even more, we use the following syntax. The following function gives the product of numbers passed as an argument.

```
In [158]: def mult(*data):
        prod = 1
        for j in data:
            prod = prod*j
        return prod
```

```
In [159]: mult(2)  ## There is just one number so product will return the number
```

```
Out[159]: 2
```

```
In [160]: mult(2,2,10)  ## Will produce product of 2 and 2
```

```
Out[160]: 40
```

```
In [ ]:
```

Now if you want to have a function definition which can handle variable number of arguments both as usual arguments and as key value pairs, use the following following syntax

```
In [161]: def myfunc(*args, **kwargs):  
          for a in args:  
              print(a)  
          for x in kwargs:  
              print(x,kwargs[x])  
  
          ## kwargs here handle the key value pairs
```

```
In [162]: myfunc('Toni',age=30,birth_place='London')  
  
Toni  
age 30  
birth_place London
```

```
In [163]: myfunc('Toni','Sam',age1=30,birth_place1='London',age2=30,birth_place2  
  
Toni  
Sam  
age1 30  
birth_place1 London  
age2 30  
birth_place2 Sydney
```

3.5 Returning multiple things

You can return multiple things from a function if required

```
In [164]: def mult_sum(*data):  
          prod = 1  
          for j in data:  
              prod = prod*j  
  
          summ = 0  
          for j in data:  
              summ = summ+j  
          return prod,summ ## can return many things together
```

```
In [ ]:
```

```
In [165]: mult_sum(2,20.4) # 2*20.4 = 40.8 and 2 + 20.4 is 22.4
```

```
Out[165]: (40.8, 22.4)
```

```
In [166]: mult_sum(1),mult_sum(2,20)
```

```
Out[166]: ((1, 1), (40, 22))
```

4 Input and Output in Python

Following code creates a file MyFile.txt in the same folder as this notebook. 'w' mode here indicates it is opened for writing new stuff. Other available options are explained in the lecture 2 slides

Write stuff

```
In [170]: file1 = open("MyFile.txt","w")
```

Now writing numbers 0 to 9 to this text file

```
In [171]: for j in range(10):  
            file1.write(str(j)+'\n')  ## \n enables to write each number in a  
            #file1.write(str(j))  
file1.close()
```

Read stuff

```
In [174]: with open("MyFile.txt") as f:  
            content = f.readlines()  
            for j in content:  
                print(j[:-1])
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Another way for reading stuff from this file

```
In [176]: file1 = open("MyFile.txt","r")
content = file1.readlines()
for j in content:
    print(j[:-1])
file1.close()
```

```
0
1
2
3
4
5
6
7
8
9
```

Append stuff

'a' mode denotes append. using this we can append more content into the file

```
In [177]: file1 = open("MyFile.txt","a")
for j in range(10,20):
    file1.write(str(j)+'\n')
    #file1.write(str(j))
file1.close()
```

Finally printing full contents of the file

```
In [178]: file1 = open("MyFile.txt","r")
content = file1.readlines()
for j in content:
    print(j[:-1])
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Error and Exception Handling

Usually if some error occurs while execution, the program stops right there


```
In [179]: for i in range(10):  
          print(i)  
          print(i+'sam')
```

0

```
-----  
-----  
TypeError                                Traceback (most recent call  
last)  
Input In [179], in <cell line: 1>()  
      1 for i in range(10):  
      2     print(i)  
----> 3     print(i+'sam')
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

However we can define an exception where a program uses 'try' and 'except' clause like this. Here in the try statement the program is trying to add integers with a string 'sam' so it will produce an error. However, the program won't stop and just execute the except clause instead. More details are in lecture 2 slides

```
In [180]: for i in range(10):  
          try:  
              print(i+'sam')  
          except:  
              print('sorry')
```

sorry
sorry
sorry
sorry
sorry
sorry
sorry
sorry
sorry
sorry

Here we will try to convert a string 's' to an integer, since that would usually produce an error. Here the program will not stop because of the error and will execute the except clause.

```
In [181]: a = 's'
          try:
              print(int(a))
          except:
              print('Check the data type')
```

Check the data type

There are different types of exceptions (error classes) in python.

```
In [7]: a = 'a'
        try:
            print(int(a))
        except ValueError:
            print('Check the datatype')
```

Check the datatype

In []:

In []:

In []:

In []: