PYTHON PROGRAMMING (CS0452)
UNIT 2

Numeric Types

In Python, numbers are not a single object type, but a category of similar type. Python not only provides the usual numeric types, but also provides more advanced numeric programming support and objects for advanced work.

This includes:

- Integer and floating-point objects
- Complex number objects
- Decimal() : fixed-precision objects
- Fraction() : rational number objects
- Sets : collection with numeric operations
- Booleans : True and False
- Built - in functions and modules : round, math, random, etc.
- Expressions; unlimited integer precision; bitwise operations; hex, octal and binary formats
- Third - party extensions : vectors, libraries, visualization, plotting, etc.

1. Integers and floating-point literals

eg. 1234567, -89343, 9999999999999, 1.234, 3.14e-10, 5E21

Integers in **Python 2.x:** (2 types)
* Are either normal (32 bit) or long (unlimited precision).
* Typically, an integer may end with an 'l' or 'L' indicating a long int.
* But Python automatically converts values exceeding 32 bits to long, hence no need of using 'l' or 'L'.

Integers in **Python 3.x:** (single type)
* Normal and long integers are merged into long integer types.

2. Hexa, Octal and Binary literals

* Hexadecimal starts with a leading 0x or 0X followed by hexa digits

* Octals start with a leading 0o or 0O (zero and lower-/upper-case letter o) followed by octal digits

* Binary starts with a leading 0b or 0B followed by a sequence of 0's and 1's

Related Built-in functions:
hex(n), oct(n), bin(n)
where 'n' is an integer, return value is the string representation in the 3 bases.

3. Complex numbers

In Python, complex literals are written as:

rp + ip 'j'

or

rp + ip 'J'

These complex numbers are implemented internally as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers.

To create a complex number (using built-in function)
Syntax:

```
>>>complex(rp,ip)
rp + ipj
```

eg.

```
>>>complex(2, 3)
2 + 3j
>>>
```

Try these...

1. (2+3j)*(2-3j)
2. (4j)+(2-5j)

4. Coding other numeric types

For advanced/specialized roles such as,

(A) decimal module - for decimal fixed point and floating point arithmetic

Provides better precision in the result
Decimal instances can be constructed from integers, strings, or tuples.

Note: To create a decimal from float, first convert it to a string.

Syntax:

```
from decimal import Decimal

Decimal(expn)
```

eg(i):

```
>>>Decimal(1/3)
?
>>>Decimal(str(1/5))
?
```

eg(ii):

```
>>>Decimal(str(22/7))
?
```

All arithmetic operations can be performed using Decimal.

Why use Decimal()?

eg.
Consider the following expression:
>>> 0.1 + 0.1 +0.1 - 0.3
?

>>>Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3')
?

>>>Decimal(0.1)+Decimal(0.1)+Decimal(0.1)-Decimal(0.3)
?

Setting precision:

1. Option 1

```
from decimal import Decimal
x = Decimal(16.0/7)
output = round(x,2)
print output
```

2. Option 2

```
from decimal import Decimal, ROUND_HALF_UP
value = Decimal(16.0/7)
output = Decimal(value.quantize(Decimal('.01'),rounding=ROUND_HALF_UP))


# Other options for rounding: ROUND_05UP, ROUND_DOWN, ROUND_HALF_DOWN,
ROUND_HALF_UP, ROUND_CEILING, ROUND_FLOOR, ROUND_HALF_EVEN, ROUND_UP
```

3. Option 3

```
from decimal import getcontext, Decimal
getcontext().prec = 3
output = Decimal(16.0)/Decimal(7)
print output
```

(B) The fractions module - provides support for Rational number arithmetic

A fraction instance can be created by a pair of integers, from another rational number or from a string.

Syntax:

from fractions import Fraction

(i) Fraction(numerator, denominator)

(ii) Fraction(other_fraction)

(iii) Fraction(string)

eg (i):
>>>Fraction(16, -10)
Fraction(-8, 5)

>>>print(Fraction(16,-10)
?

eg (ii):
>>>Fraction(123)
Fraction(123, 1)

eg (iii):
>>>Fraction('2/5')
Fraction(2, 5)

All arithmetic operations can be performed using Fraction.

Built - in Numeric tools

(a) Operators
eg: *+, -, *, /, >>, <<, **, &, |, ! etc..*

(b) Functions
eg: *pow, abs, round, int, hex, bin, sum, min, max etc..*

(c) Modules
eg: *math, random etc..*

(d) Constants
eg: *math.pi, math.e*

Numbers in Action

egs:
>>>a=10
>>>a+2, a-2
(12, 8)
returns a tuple!

*>>>2+4.0, 2 **a*
(6.0, 1024)

Numeric Display Formats

egs.
>>>num = 1/3.0
>>>num
?

>>>'%e' %num
3.333333e-01

>>>'%4.2g' %num
0.33

>>>'%4.2f' %num
0.33

>>>'{0:4.2g}'.format(num)
0.33

Comparisons (Always returns a Boolean value)

(a) Normal
egs.
>>> 2 > 1
True

>>> 2.0 >= 1
True

(b) Chained
Python permits us to chain multiple comparisons together!

egs.
>>>x = 2
>>>y = 3
>>>z = 4

>>> x < y < z
True

>>> x > y and y < z
False

>>> x < y > z
?

>>> 1 == 2.0 < 3
?

Floor Vs Truncate

Floor = next lower integer
Truncate = drop decimal digits

Sets

- An unordered collection of unique and immutable objects
- Supports operations corresponding to mathematical set theory

Properties:

- An item appears only once in a set, no matter how many times it is added

Defining a set:

Syntax

var = set(iterable_object)

egs:
>>>A = set('abc')
>>>B = set(range(5))
>>>C = set([1,3,5,7]) # creating a set thru a list
>>>D = {2, 4, 6, 8}

verify the type...

>>>type(D)
<class 'set'>

Set Operations:

I. Using Operators:

1. Difference

eg:
>>> x – y

2. Union

eg:
>>> x | y

3. **Intersection**

eg:

>>> x & y

4. **Symmetric difference (XOR) (elements that are in one of the sets but not both)**

eg:

>>> x ^ y

5. **Superset (Boolean type)**

eg:

>>> x > y # Is x superset of y?

6. **Subset (Boolean type)**

eg:

>>> x < y # Is x subset of y?

7. **Membership (Boolean type)**

eg:

```
>>> x = set('abcde')
>>> 'e' in x
True
>>> 'g' in x
False
```

II. Using built-in functions:

eg:

```
>>>x = set('abcde')
>>>y = set('defgh')
```

1. **Union**

>>>x.union(y)

>>>x.update(y)

 # in-place union operation, set 'x' is replaced by the union of 'x' and 'y'

2. **Intersection**

>>>x.intersection(y)

3. **Subset**

>>>x.issubset(y)

4. **Superset**

>>>x.issuperset(y)

Overruling the immutable object concept

5. **Insert one item**

>>>x.add('g')

6. **Delete one item**

>>>x.remove('d')

Indexing and slicing

using for loop..

eg:

```
>>>x = set('hai')
>>>for element in x: print(element *2)
. . .
aa
hh
ii
>>>
```

Strings

1. **Initializing strings** (just like assigning any other data type value)

```
>>>s1 = 'Hello'
>>>s2 = ' There!'
>>>s3 = s1 + s2          #s3 is a concatenated string
>>>print s3
Hello There!

>>>s4 = s1 + ', how r u?'
>>>print s4
Hello, how r u?

>>>print s1*2           # string repetition
HelloHello
```

2. **Reading from the user**

```
>>>name = raw_input('Enter a string..')
>>>print name
```

3. **Accessing characters from a string (indexing)**

```
>>>colour = 'Blue'
>>>colour[1]
'l'
```

4. **Length of a string**

Syntax:
var = len(str)

eg:
>>>colour = 'Blue'
>>>print len(colour)
4

5. **Looping thru a string**

```
# using while loop
colour = 'Purple'
index = 0
while index < len(colour):
      letter = colour[index]
      print letter
      index =index + 1


#using for loop
colour = 'Purple'
for letter in colour:
     print letter
```

Eg: To count the occurrance of a char in a string

```
word = 'purple'
count = 0
for letter in word:
   if letter == 'p' or letter == 'P':
      count += 1
print 'No. of occurrance of letter p : ', count, 'in word', word
```

6. **String Comparison**

use '==', '!=', '<', '>' operators

eg:

```
if color == ' purple':
     print ('Wow!! Colours are matching :)')
elif color < 'purple':
     print ('Your color', color,'comes alphabetically, before purple')
else:
     print ('Your color', color,'comes alphabetically, after purple')
```

7. **String Slicing**

 # using indexing
eg:
let, s = 'Monty Python'

What will be the output of the following statements?

print s[0:4]
print s[6:7]
print s[6:20]
print s[:2]
print s[8:]
print s[:]
print s[1:12:3]
print s[::2]
print s[::-1]
print s[5:1:-1]
print s[-5:]
print s[:-5]

Note:
1. Indexing s[i] fetches components at offsets:

- The first item is at offset 0(zero)
 s[0] fetches the first item

- Negative indexes mean to count backward from the end or right
 eg: s[-2] fetches the second item from the end (like s[len(s)-2])

2. Indexing s[i:j] extracts contiguous sections of sequences:

- The upper bound is non-inclusive
 s[1:3] fetches items at offsets 1 up to but not including 3
 s[:3] fetches items at offsets 0 up to but not including 3

- Slice boundaries default to 0 and the sequence length, if omitted
 s[:] fetches items at offsets 0 thru the end
s[1:] fetches items at offset 1 thru the end
s[:-1] fetches items at offset 0 thru the end but not including the last item

3. Indexing s[i:j:k] accepts a step 'k', which defaults to +1
 - Allows for skipping items and reversing order

Exercise:
Check for palindrome

8. **Using 'in' as an operator**

```
>>>colour = 'purple'
>>>'r' in colour          #is 'r' in colour
True
>>>'b' in colour
False
```

for substring search

```
>>>subst = 'urp' in colour
>>>if subst == True:
        print 'Found'
    else:
        print 'Not Found'
```

9. **String Library (importing from string package)**

#ref- https://docs.python.org/2/library/stdtypes.html#string

-methods

capitalize()	*center()*	*count()*	*decode()*	*encode()*
endswith()	*expandtabs()*	*find()*	*format()*	*index()*
isalnum()	*isalpha()*	*isdigit()*	*islower()*	*isspace()*
istitle()	*isupper()*	*join()*	*ljust()*	*lower()*
lstrip()	*partition()*	*replace()*	*rfind()*	*rindex()*
rjust()	*rpartition()*	*rsplit()*	*rstrip()*	*split()*
splitlines()	*startswith()*	*strip()*	*swapcase()*	*title()*
translate()	*upper()*	*zfill()*		

eg:

```
s = 'i\'ll be there for you :)'

print(s.capitalize())
print(s.find('for you'))
print(s.isalpha())
print(s.isalnum())
print(s.replace('for','with'))
print(s.strip())
print(s.rstrip())
print(s.rjust(50))
word_list = s.split(" ")
print(word_list)
print(s.swapcase())
```

defining multiple line string using 3 single quotes '''
eg:
```
s1 = '''This is the first line
This string has a second line too
And also a third line'''

print(s1)


print(s1.splitlines())  # returns a list containing one line of s1 as one element

if s.startswith(' ') and s.endswith(' '):
   print('String begins and ends with blanks!')
else:
   print('No blanks at the beginning and end!')
```

join() in strings:

syntax:

```
output_String = 'delimiter'.join(seq/element)
```

Note: 'delimiter' works only with 'seq' not with 'element'

eg: (1)
```
s1 = 'abc'
s2 = ''

for ele in s1[::-1]:
     s2 += ''.join(ele)

print(s2)
```

eg:(2)
```
s1 = 'this is a string'
s2 = ''
s2 = '-'.join(s1.split())
```

10. **Usage of format specification**
eg:
```
>>> 'That is %d %s bird!' %(1,'dead')'
'That is 1 dead bird!

>>>'%d %s %g you', %(1,'spam',4.45)
'1 spam 4.45 you'
```

>>>'%s -- %s -- %s' %(42, 3.141, [1,2,3]) #everything as string!
'42 -- 3.141 --[1, 2, 3]'

11. **Strings are immutable!!**

You can't change an existing string, rather create a new string

eg:
>>>greet = 'Hello, World!'
>>>greet[0] = 'M'
#Results in TypeError!!

#Instead,
>>>greet1 = 'M' + greet[1:]
>>>greet1
Mello, World!

Exercises:

I. Check if given 2 strings are anagrams

[Two strings are anagrams if they are written using the same letters (ignoring spaces, punctuations and cases).
eg: 'Mary and Army', 'Silent' and 'Listen' are anagrams]

II. Print all the permutations of a given string.
eg: Input: 'xyz'
    Output: 'xyz', 'xzy', 'yxz', 'yzx', 'zxy', 'zyx'

III. Encryption (plain text to cypher text)

Write a function called rotate_word that takes a string and an integer as parameters, and that returns a new string that contains the letters from the original string "rotated" by the given amount.

[Hint: You might want to use the built-in functions '*ord( )*', which converts a character to a numeric code, and '*chr( )*', which converts numeric codes to characters.
eg: *ord('a')* returns 97 (Unicode of 'a')
     chr(104) returns 'h']

Solution for Ex.III:

```
def rot_letter(letter,n):
  if letter.isupper():
    start = ord('A')
```

```
    elif letter.islower():
        start = ord('a')
    else:
        return letter
    c = ord(letter) - start
    i = (c + n) % 26 + start
    return chr(i)


def rot_word(word,n):
    res = ''
    for letter in word:
        res += rot_letter(letter,n)
    return res
```

Lists

1. **Intro:**
Recall... 'strings are sequence of characters.'

"A list in Python is a sequence of anything!"

Python has 2 sequence structures: tuples and lists.

Both tuples and lists contain zero or more elements.

Unlike strings, the elements can be of different types.

Why both Lists and Tuples??

Tuples are immutable; when you assign elements to a tuple, they can't be changed

Lists are mutable; you can insert and delete elements anytime.

2. **Creating a list with [] or ()**

(a) A list is made from zero or more elements, separated by commas, and surrounded by square brackets:

eg.

```
>>>empty_list = []

>>>working_days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

(b) An empty list can be created using the list() function:
>>>next_empty_list = list()

(c) **The Pythonic Way!! (Comprehensions)**
Comprehension is a compact way of creating Python data structure from one or more iterators

eg(1):
>>>num_list = list(range(1,10))
>>>num_list

eg(2):
>>>alist = list(ch for ch in input())
hello!
>>>alist
['h', 'e', 'l', 'l', 'o', '!']

3. **Lists are mutable (elements are changeable)**

eg:
>>>num_list
[1, 2, 3, 4]

>>>num_list[1] = 8

>>>num_list
[1, 8, 3, 4]

4. **Indexing a list**
- Any integer expression can be used as an index
- If an index has a negative value, it counts from backwards (from the end of the list)
  eg:
  >>>lst = list(range(1,5))
  >>>lst[-1]
  4

5. **Traversing a list**
Most common way: using loops

To read from list:
eg: *for item in new_list:*
        print item

To write and update a list:
eg: *numbers = [1, 2, 3, 4, 5]*
for i in range(len(numbers)):
*numbers[i] = numbers[i] * 2*

List of iterable items:
eg:
>>>nu_list = list('hello')
>>>nu_list

what will be the contents of the list below?
>>>n_list = list(3333)

A list can have another list as its element
eg:
>>>numbers = [1, 2, 3, 4, 5]
>>>courses = ['PP', 'CD', "OS", 'SS']
>>>new_list = [numbers, courses, '5th sem']

>>>new_list
[[1, 2, 3, 4, 5], ['PP', 'CD', "OS", 'SS'], '5th sem']
>>>len(new_list)
?

Accessing the list elements: thru indexing (single index/multiple index)
>>>new_list[0]

>>>new_list[1][0]

6. **List Operations**
(a) **'+' for concatenation**
eg:
>>>list_a = ['a','b','c']
>>>list_no = [1, 2, 3, 4]
>>>list_new = list_a + list_no
['a', 'b', 'c', 1, 2, 3, 4]

(b) **'*' for repetition**
eg:
*>>>list_b = list_a * 3*

(c) **Membership (using 'in' operator)**
eg:
>>>'a' in list_a # is 'a' an element in list_a?
True

Note: Works only for single list, not for nested list

(d) **List Slicing**
What is the output of the following?
>>>list_b = [1,2,3,4,5,6,7,8,9]
>>>list_b[2:5]
>>>list_b[:4]

>>>list_b[4:]

>>>list_b[-1]

>>>list_b[:-1]

>>>list_b[::-1]

Using slicing to update multiple elements in a list

>>>list_b[1:3] = ['x', 'y']

>>>list_b
['a', 'x', 'y', 'a', 'b', 'c', 'a', 'b', 'c']

(e) **List Methods**
Can be classified as:
- Growing methods
- Searching methods
- Sorting methods
- Reversing methods
- Shrinking methods
- Copy method

- Growing methods
(i) **append(x)** - adds a new element 'x' at the end of the list

>>>alist = ['a', 'b', 'c']
>>>alist.append('d')
>>>alist
['a', 'b', 'c', 'd']

What is the contents of 'alist' after this statement:
>>>alist.append('e','f')

(ii) **extend(list)** - adds a list to another list
>>>alist.extend(list_no)
>>>alist
['a', 'b', 'c', 'd', 1, 2, 3, 4]

(iii) **insert(i, x)** - inserts an new element 'x' at specified index 'i'
>>>alist.insert(4,'e')
>>>alist
['a' ,'b', 'c', 'd', 'e', 1, 2, 3, 4]

Exercise:
I. Write a function to add all the elements in a given list of numbers. Display the sum.

II. Write a function that takes a list of numbers and returns the cumulative sum, i.e., a new list where the 'i'th element is the sum of the first 'i+1' elements from the original list.

eg: the cumulative sum of [1, 2, 3, 4] is [1, 3, 6, 10]

- Searching methods
(i) **index(x)** - returns the index value of 'x' in the list
eg:
>>>alist = [1, 2, 3, 4, 5]
>>>alist.index(3)
2

(ii) **count(x)** - returns the number of occurance of 'x' in the list
eg:
>>>alist.append(3)
>>>alist
[1, 2, 3, 4, 5, 3]
>>>alist.count(3)
2

- Sorting Methods
(a) **sort()** - Orders a list 'in place'. Default ordering - ascending order

eg:
>>>num = [4, 2, 7, 3, 9, 1]
>>>num.sort()
>>>num
[1, 2, 3, 4, 7, 9]

To change the order of sorting, use keyword 'reverse' as an argument to sort()
eg:
>>>num = [4, 2, 7, 3, 9, 1]
>>>num.sort(reverse = True)
[9, 7, 4, 3, 2, 1]

To sort a string list:
eg:
>>>alist = ['Hello', 'world', 'Whatsup?', 'bye', 'Done']
>>>alist.sort()
>>>alist
?
['Done', 'Hello', 'Whatsup?', 'bye', 'world']

For proper sorting the 'case' must be ignored!
eg:
>>>alist = ['Hello', 'world', 'Whatsup?', 'bye', 'Done']
>>>alist.sort(key = str.lower) # normalize all the elements to lowercase
>>>alist
['bye', 'Done', 'Hello', 'Whatsup?', 'world']

eg:
>>>alist = ['Hello', 'world', 'Whatsup?', 'bye', 'Done']
>>>alist.sort(key = str.lower, reverse = True)
>>>alist
?

Note: *'key = str.upper'* can also be used, which normalizes the elements to uppercase before sorting

(b) **sorted(list)** - returns the sorted 'list', but does not replace order in original list
eg:
>>>num = [4, 6, 2]
>>>sorted(num)
[2, 4, 6]
>>>num
[4, 6, 2]

eg:
>>>num = [4, 6, 2]
>>>sorted(num, reverse = True)
?

Using string elements in sorted():
eg:
>>> L = ['abc', 'ABD', 'aBe']
>>>sorted(L)
?

>>>sorted(L, key = str.lower)
?

>>>sorted(L, key = str.lower, reverse = True)
?

Using list comprehension to generate a new list:
eg:
>>> L = ['abc', 'ABD', 'aBe']
>>>sorted([x.lower() for x in L], reverse = True)
['abe', 'abd', 'abc']

- Reverse methods
(a) **reverse()** - reverses the list elements 'in place'
eg:
>>>L = list(range(1,5))
>>>L
[1, 2, 3, 4]
>>>L.reverse()
>>> L
[4, 3, 2, 1]

(b) **reversed(list)** - returns the reverse of 'list' as iterator
eg:
>>>L = list(range(1:5))
>>>L
[1, 2, 3, 4]
>>>list(reversed(L))
[4, 3, 2, 1]

-Shrinking methods
(a) **remove(x)** - removes the first occurance of element 'x' from the list
eg:
>>>num = [4, 7, 2, 6, 3, 9]
>>>num.remove(2)
>>>num
[4, 7, 6, 3, 9]

>>>num.remove(10)
??

Predict the contents of nu_list...
>>>nu_list = [1, 2, 4, 3, 2, 1]
>>>nu_list.remove(1)
?

(b) **pop(i)** - removes and returns the element in index position 'i'
eg:
>>>num = [4, 7, 2, 6, 3, 9]

>>>num.pop(3)
6
>>>num
[4, 7, 2, 3, 9]

(c) **pop()** - removes the last element from the list
eg:
>>>num = [4, 7, 2, 6, 3, 9]
>>>num.pop()
9
>>>num
[4, 7, 2, 6, 3]

(d) **clear()** - removes all the elements from a list.
eg:
>>>num = [4, 7, 2, 6, 3, 9]
>>>num.clear()
>>>num
?

(e) **del** command
syntax(1): *del list[index]*
deletes/ removes element in 'index' position from the list.

syntax(2): *del list[index1:index2]*
deletes/removes element(s) from position 'index1' upto position 'index2', excluding element in 'index2' from the list.

eg:
>>>num = [4, 7, 2, 6, 3, 9]
>>>del num[0]
>>>num
[7, 2, 6, 3, 9]

>>>num = [4, 7, 2, 6, 3, 9]
>>>del num[2:4]
?

- Copy method (module copy)
copy.copy(list) - returns a copy of the 'list'.
#available from v 3.3 onwards!
eg:
>>>L = list(range(1,5))
>>>L
[1, 2, 3, 4]
>>>import copy
>>>L1 = copy.copy(L)
>>>L1
[1, 2, 3, 4]

Exercise:
Develop a program to implement
(a) FIFO (queue)
(b) LIFO (stack)
(c) Priority Queue

Accessing inputs thru text files...

Syntax:

Handle = open(filename, mode)

 # default mode = 'r'

To read lines from the file:

```
for eachline in Handle:          # eachline is a string containing one line of text from file
    print(eachline)
```