Parul® University

Python Programming

PIET

CSE Dept.

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-2

Python Data Structure

What is Data Structure?

- Data structure identifies how data or values are stored in memory

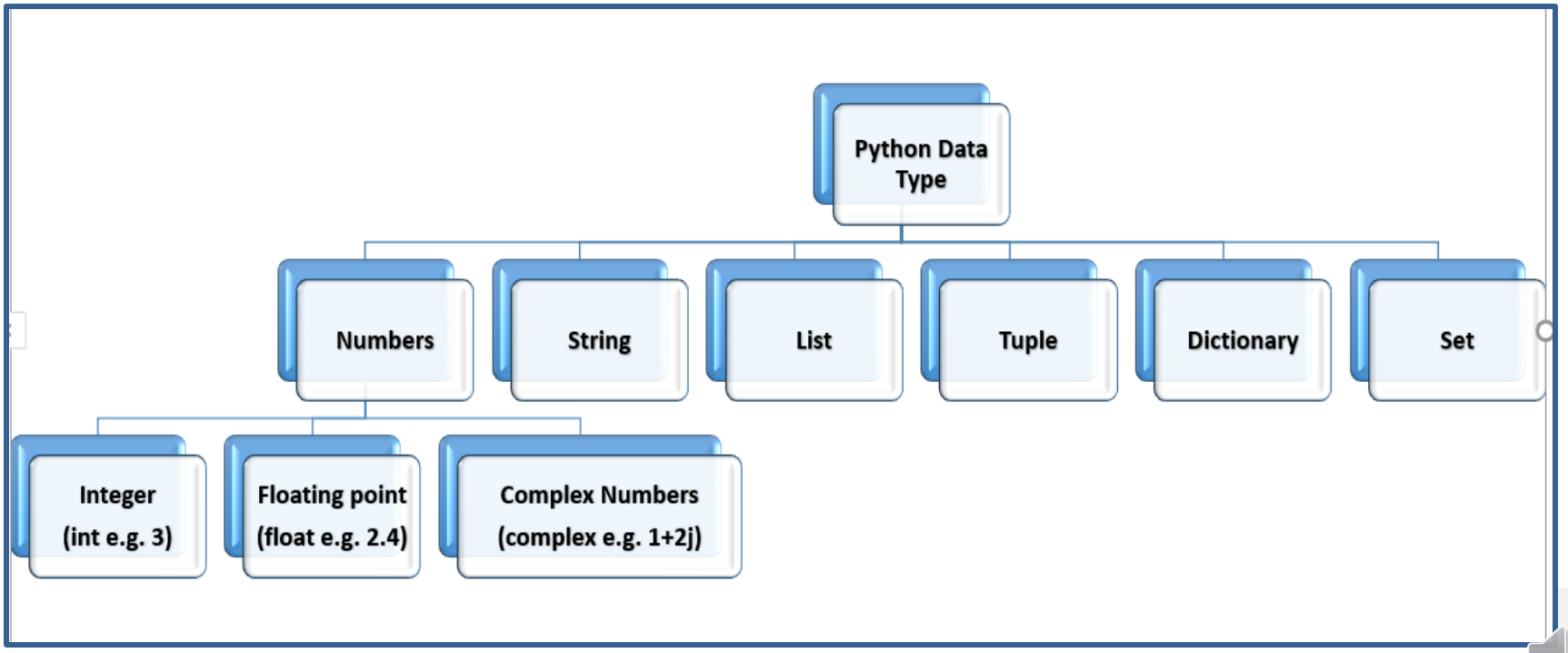


Figure 3.1 Python Data Structure

String

- Sequence of characters
- Keyword for string datatype : str
- Uses single or double quotes : 'Hello' or "Hello"
- When a string contains numbers, it is still a string : ‘123’
- Convert numbers in a string into a number using int()
- immutable : Value of sting can not be changed

```
>>> st1 = 'Hello'
>>> st2 = "World"
>>> print(st1 + st2)
HelloWorld
>>> st1 = '1'
>>> print(1 + st1)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(1 + st1)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> st1 = int('1')
>>> print(1 + st1)
2
```

input() reads string only

```
>>> x = input('Enter Data')
Enter Data2
>>> x + 1
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    x + 1
TypeError: must be str, not int
```

```
>>> x = int(input('Enter Data'))
Enter Data2
>>> x + 1
3
```

More on String

- Get length of string
  - **len()**
- String slicing using colon operator
  - **st[start : end]**
- Count the occurrence of character
  - **st.count('character)**
- String uses index
  - String = 'World' is indexed as follows

W	o	r	l	d
0	1	2	3	4

```
>>> st1 = 'World'
>>> letter = st1[2]
>>> letter
'r'
>>> print(len(st1))
 5
>>> print(st1[2:4])
 rl
>>> 'Hello'.count('l')
 2
```

String Library

- List of common function provided by string library
- Explore more using : dir(string_object)

- capitalize
- center
- count
- endswith
- find
- Index
- Isalnum
- Isalpha
- Isdigit
- Islower
- isupper
- join
- ljust
- Lower
- Lstrip
- replace
- rjust
- rsplit
- Rstrip
- Startswith
- Swapcase
- upper

```
>>> st1 = 'hello'
>>> st1.capitalize()
'Hello'
>>> st1.endswith('f')
False
>>> st1.find('l')
2
>>> st1.replace('e','a')
'hallo'
>>> ' hello '.strip()
'hello'
```

List

- Collection of many values in a single variable
- Mutable : value of list variable can be changed
- Uses square brackets : []
- Example:

list_of_number = [1, 2.3, 3, 4, 0]

Friend_list = [‘kanu’, ‘manu’, ‘tanu’]

City_list = [‘Baroda’, ‘Anand’, 123]

Image source : Google

Parul®
University

DIGITAL LEARNING CONTENT

Exploring List

- List is an ordered collection
- Access any element using index
- Get number of elements : len()
- List concatenation : '+'
- List slicing using colon

nita	12	pavan	13	14
0	1	2	3	4

```
>>> my_list = list() #create an empty list
>>> my_list = []  #create an empty list
>>> my_list = ['nita' , 12, 'pavan' , 13, 14]
```

```
>>> my_list[2]
'pavan'
>>> len(my_list)
5
>>> your_list = [2,3]
>>> my_list + your_list
['nita', 12, 'pavan', 13, 14, 2, 3]
>>> my_list[:3]
['nita', 12, 'pavan']
>>> my_list[2:]
['pavan', 13, 14]
>>> my list[2:3]
['pavan']
```

Parul® University

DIGITAL LEARNING CONTENT

List Methods

Methods	Description
append	Add element in the list as it is
count	Count the occurrence of an element in list
extend	Add values of object as elements of the list
index	Get the index of an element
sort	Sort the elements of list
insert	Add element at specified index
pop	Retrieve the last element of list
remove	Remove the given element from list
reverse	Reverse the sequence of elements in list
clear	Empty the list by removing all elements

```
>>> list1 = list()
>>> list1.append(2) #add element to list
>>> list1
 [2]
>>> list1.append([3,4]) #add nested list in list1
>>> list1
[2, [3, 4]]
>>> list1.extend([3,4]) # add elements of list
>>> list1
[2, [3, 4], 3, 4]
 >>> list1.index(3)
 2
>>> list1.count(3)
1
```

Playing with List

```
>>> my_list = [4 , 5 , 6 , 3 , 2]
>>> my_list.pop()
 2
>>> my_list.pop() #returns the last element
 3
>>> my_list
[4, 5, 6]
>>> my_list = [4 , 2 , 6 , 3 , 1]
>>> my_list.pop() #removes the last element
1
>>> my_list.remove(6) # removes the given value
>>> my_list
[4, 2, 3]
>>> my_list.sort() #arrange elements in ascending
>>> my_list
[2, 3, 4]
>>> my_list.reverse()
>>> my_list
[4, 3, 2]
```

Playing with List

```
>>> num_list = [2 , 5 , 7 , 8 , 3]
>>> 5 in num_list
 True
>>> 1 not in num_list
 True
>>> max(num_list)
 8
>>> min(num_list)
 2
>>> sum(num_list)
 25
>>> avg = sum(num_list)/len(num_list)
>>> avg
5.0
```

List Comprehension

- Use logical statement to create list

```
>>> num_list = [i for i in range(10)]
>>> num_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Tuple

- Same as list
- Immutable : values can not be changed
- Use round brackets

```
>>> my_tuple = (1 , 2 , 3)
>>> my_tuple
(1, 2, 3)
>>> my_tuple[1] = 4
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    my_tuple[1] = 4
TypeError: 'tuple' object does not support item assignment
```

Exploring Tuple

- Tuple has two method
  - Count
  - Index

```
>>> t1 = (2 , 3 , 2 , 3 , 4 , 5)
>>> t1.count(3) #count of element 3
2
>>> t1.index(4) #index of element 4
4
```

- Tuple as assignment

```
>>> (x , y) = (2 , 3) # x and y are variable
>>> x
2
>>> y
3
>>> x , y = 2 , 3 #No need to put '()'
>>> x
2
```

Dictionary

- Unordered collection of data in key : value form
- Indexed by unique key
- Uses curly braces : { }

Bag of items

Image source : Google

Creating Dictionary

```
>>> purse = dict() #create an empty dictionary
>>> purse['money'] = 12
>>> purse['calculator'] = 1
>>> purse['perfume'] = 2
>>> purse['tissue'] = 10
>>> purse
{'money': 12, 'calculator': 1, 'perfume': 2, 'tissue': 10}
```

OR

```
>>> name = {1:'maya' , 2:'Sachin' , 3:'happy'}
>>> name
{1: 'maya', 2: 'Sachin', 3: 'happy'}
```

Keys as Index

```
>>> purse
{'money': 12, 'tissues': 75, 'candy': 3}
>>> purse = {'money': 12, 'tissues': 75, 'candy': 3}
>>> purse['money']
12
>>> purse['candy']+1
4
```

12	75	3
money	tissues	candy

Dictionary Methods

- **get()**
  - give the value at given key if key is there, otherwise create given and assign default value
- **keys()** : list of keys
- **values()** : list of values
- **Items()** : list of (key, value)

```
>>> name = {1:'maya' , 2:'sachin'}
>>> name.get(1,0)
 'maya'
>>> name[3] = name.get(3,0)
>>> name
{1: 'maya', 2: 'sachin', 3: 0}
>>> name.keys()
 dict_keys([1, 2, 3])
>>> name.values()
dict_values(['maya', 'sachin', 0])
>>> name.items()
dict_items([(1, 'maya'), (2, 'sachin'), (3, 0)])
```

Counting Pattern

```
string1 = 'twinkle twinkle little little star'
my_string = string1.lower().split() #converts string into list of words
my_dict = {}
for item in my_string:

  my_dict[item] = my_string.count(item)

print(my_dict)
```

```
{'twinkle': 2, 'little': 2, 'star': 1}
```

Set

- Unordered collection of unique and immutable objects
- set itself is mutable
- Uses curly braces : {}

```
>>> a = set('hello world')
>>> a
{'w', 'l', 'e', 'h', 'r', 'd', ' ', 'o'}
>>> a[1]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a[1]
TypeError: 'set' object does not support indexing
```

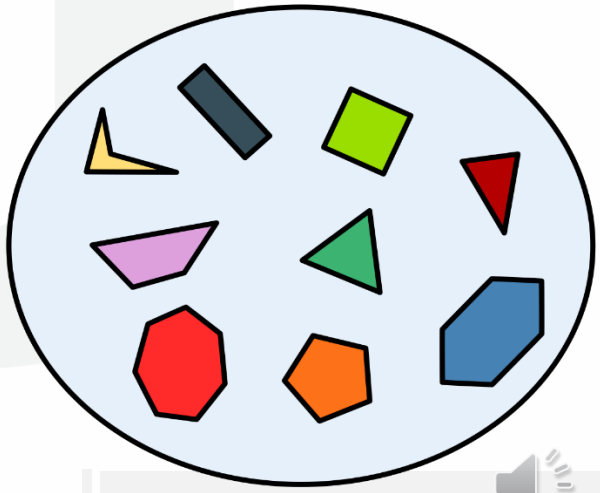


Image source : Google

Exploring Set

- **add()** : add any single element in set
- **update()** : add multiple elements passed in the form of tuples, list, string or other set in set
- **discard()/ remove()** : remove element from set

```
>>> a = set()
>>> a.add(1)
>>> a
{1}
>>> a.update([2,3])
>>> a
{1, 2, 3}
>>> a.discard(2)
>>> a
{1, 3}
>>> a.remove(1)
>>> a
{3}
```

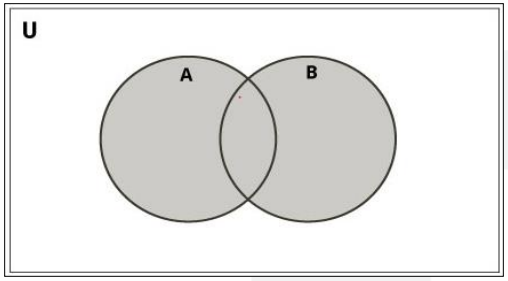
Frozenset

- Frozensets are like sets except that they cannot be changed
- They are immutable

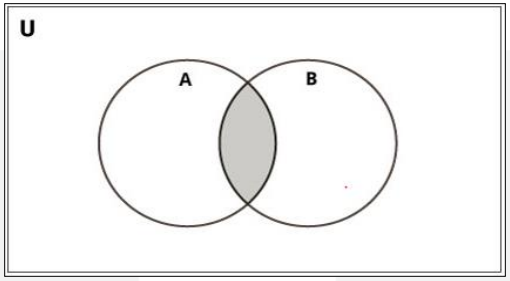
```
>>> a = frozenset('python')
>>> a.add('a')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    a.add('a')
AttributeError: 'frozenset' object has no attribute 'add'
```

Image source : Google

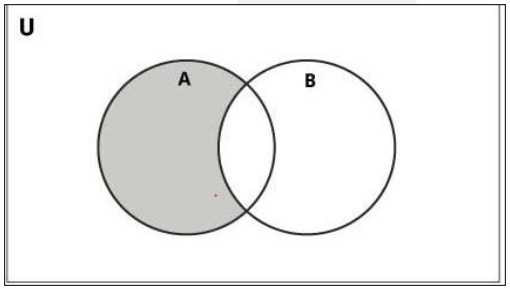
Set Operation



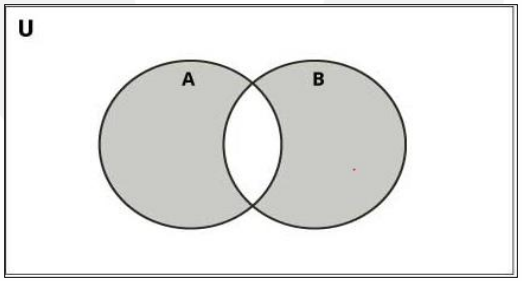
union (|)



intersection (&)



difference (-)



symmetric_difference (^)

Image source : Google

Set Operation

```
>>> A = {1,2,3}
>>> B = {3,4,5}
>>> A|B # OR A.union(B)
{1, 2, 3, 4, 5}
>>> A & B # OR A.intersection(B)
{3}
>>> A - B #OR A.difference(B)
{1, 2}
>>> A ^ B #OR A.symmetric_difference(B)
{1, 2, 4, 5}
```

Stack and Queue

Stack: Stores items in last in first out(LIFO) manner.

The operations of adding and removing the elements is known as **PUSH** and **POP**.

PUSH into a Stack: to add element on top of the stack append() will be used.

POP from a Stack: to remove element on top of the stack pop() will be used.

Stack and Queue

Example:

```
stack=[1,2,3]
print(stack)
stack.append(4)
stack.append(5)
print(stack)
print(stack.pop())
print(stack)
```

Output:

```
[1, 2, 3]
[1, 2, 3, 4, 5]
5
[1, 2, 3, 4]
```

Stack and Queue

Queue: Stores items on first in first out (FIFO) manner.
Example:

```
from collections import deque
queue=deque([1,2,3])
print(queue)
queue.append(4)
queue.append(5)
print(queue)
print(queue.popleft())
print(queue)
```

Output:

```
deque([1, 2, 3])
deque([1, 2, 3, 4, 5])
1
deque([2, 3, 4, 5])
```

Hash Tables

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function.

Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

Hash Tables

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements:

- The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

Example : dict= {‘name’:’shree’, ‘age’=25}
print(dict[‘name’])
Output: shree

Searching Algorithms

Searching is the process of looking for a particular value in a collection.
Built-in Python methods:

If we want to know the position of `x` in a list, the `index` method can be used.

```
>>> nums=[3,1,4,2,5]
>>>nums.index(4)
>>>2
```

Parul®
University

DIGITAL LEARNING CONTENT

Searching Algorithms

Linear Search:

search through the list of items one by one until the target value is found.

`index` operations implement linear searching algorithms

Searching Algorithms

Binary Search:

If the data is sorted, there is an even better searching strategy – one you probably already know!
Binary means two, and at each step we are diving the remaining group of numbers into two parts.
The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value `x`.

If `x` is smaller than the middle item, `high` is moved so that the search is confined to the lower half.

If `x` is larger than the middle item, `low` is moved to narrow the search to the upper half.

Searching Algorithms

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:          # There is a range to search
        mid = (low + high)//2 # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        elif x < item:          # x is in lower half of range
            high = mid - 1      #  move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       #  move bottom marker up
    return -1                   # No range left to search,
                                # x is not there
```

Sorting in Python

Bubble Sort

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

```
def bubblesort(list):

# Swap the elements to arrange in order
    for iter_num in range(len(list)-1,0,-1):
        for idx in range(iter_num):
            if list[idx]>list[idx+1]:
                temp = list[idx]
                list[idx] = list[idx+1]
                list[idx+1] = temp
list = [1,2,31,5,6,11,12,27]
bubblesort(list)
print(list)
```

```
[1, 2, 5, 6, 11, 12, 27, 31]
```

Sorting in Python

Insertion Sort

Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them.

Then we pick the third element and find its proper position among the previous two sorted elements.

This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

Sorting in Python

```
def insertion_sort(InputList):
    for i in range(1, len(InputList)):
        j = i-1
        nxt_element = InputList[i]
# Compare the current element with next one
    while (InputList[j] > nxt_element) and (j >= 0):
        InputList[j+1] = InputList[j]
        j=j-1
    InputList[j+1] = nxt_element
list = [1,2,31,45,0,11,21,27]
insertion_sort(list)
print(list)
```

```
[1, 2, 31, 45, 0, 11, 21, 27]
```

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in