# 2. Data Structures

Subject: Programming with Python (IT3008)

Faculty: Ms. Twinkle Kosambia
Assistant Professor, Computer Science Department,
**Asha M. Tarsadia Institute of Computer Science**

Website: utu.ac.in

Syllabus of Programming with Python:
https://app.utu.ac.in/utuformaccess/utusyllabus.aspx?CF=7&CM=177&SY=1

# Programming with Python

Subject code : IT3008

Credits : 5

Theory marks : 60+ 40= 100

Practical marks : 100 (CIE)

# Reference books

**Text book:**

1. Allex Martelli, Anna Ravenscroft and Steve Holden, “Python in Nutshell”, 3rd Edition, O’Reilly Publication.

**Reference books:**

1. Magnus Lie Hetland, "Beginning Python From Novice to Professional", Third Edition, Apress, 2017.

2. David Beazley, Brian K. Jones, "Python Cookbook", 3rd edition, O’Reilly Publication, 2016.

3. Brett Slatkin, "Effective Python: 59 Specific Ways to Write Better Python", Novatec, 2016.

4. Mark Lutz "Learning Python", 4th Edition, O’Reilly Publication, 2016.

# Course outcomes

| Unit No. | Unit Name | Course Outcomes | | | | | |
|---|---|---|---|---|---|---|---|
| | | CO1 | CO2 | CO3 | CO4 | CO5 | CO6 |
| 1 | Introduction to Python | ✓ | | | | | |
| 2 | Data Structures | | ✓ | | | | |
| 3 | Control Structure and Functions | | | ✓ | | | |
| 4 | Object Oriented Programming | | | | ✓ | | |
| 5 | Exception Handling and Regular Expression | | | | | ✓ | |
| 6 | File Handling | | | | | | ✓ |

# Unit – II Data Structures

Common sequence operations: Indexing, Slicing, Adding Sequences, Multiplication, Membership, Length, Minimum, and Maximum.

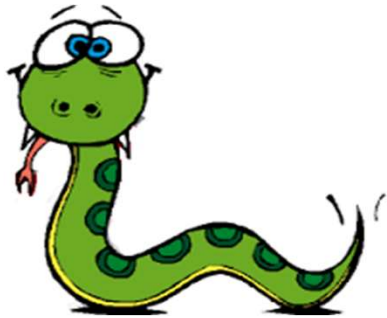
Using lists as Stacks, Using lists as Queues, List Comprehensions, Nested List comprehensions, The del statement, Tuples and Sequences, Sets, Dictionaries, Comparing Sequences and Basic string operations.

# Data Types in Python

**Python has five standard data types**

1) Numbers
2) String
3) List
4) Tuple
5) Dictionary





# Python Data Type : Numbers

- **Three subtypes of Number data type:**

**1)Int :**

- Accepts positive as well as negative values.
- can store a number of unlimited size.

e.g. x=10

y=-1000

**2)Float**

- Negative as well as positive

e.g. x=1.24

Y=-1.34566

z=23E3=>23*10^3


UKA TARSADIA university

## 3)Complex Number

- Number can be written in form a+bj/a+bJ

a=>real part

b=>Imaginary part

- Must end in `j` or `J`
- Typing in the imaginary part first will return the complex number in the order `Re+ImJ`
- Examples: `3+4j, 3.0+4.0j, 2J`

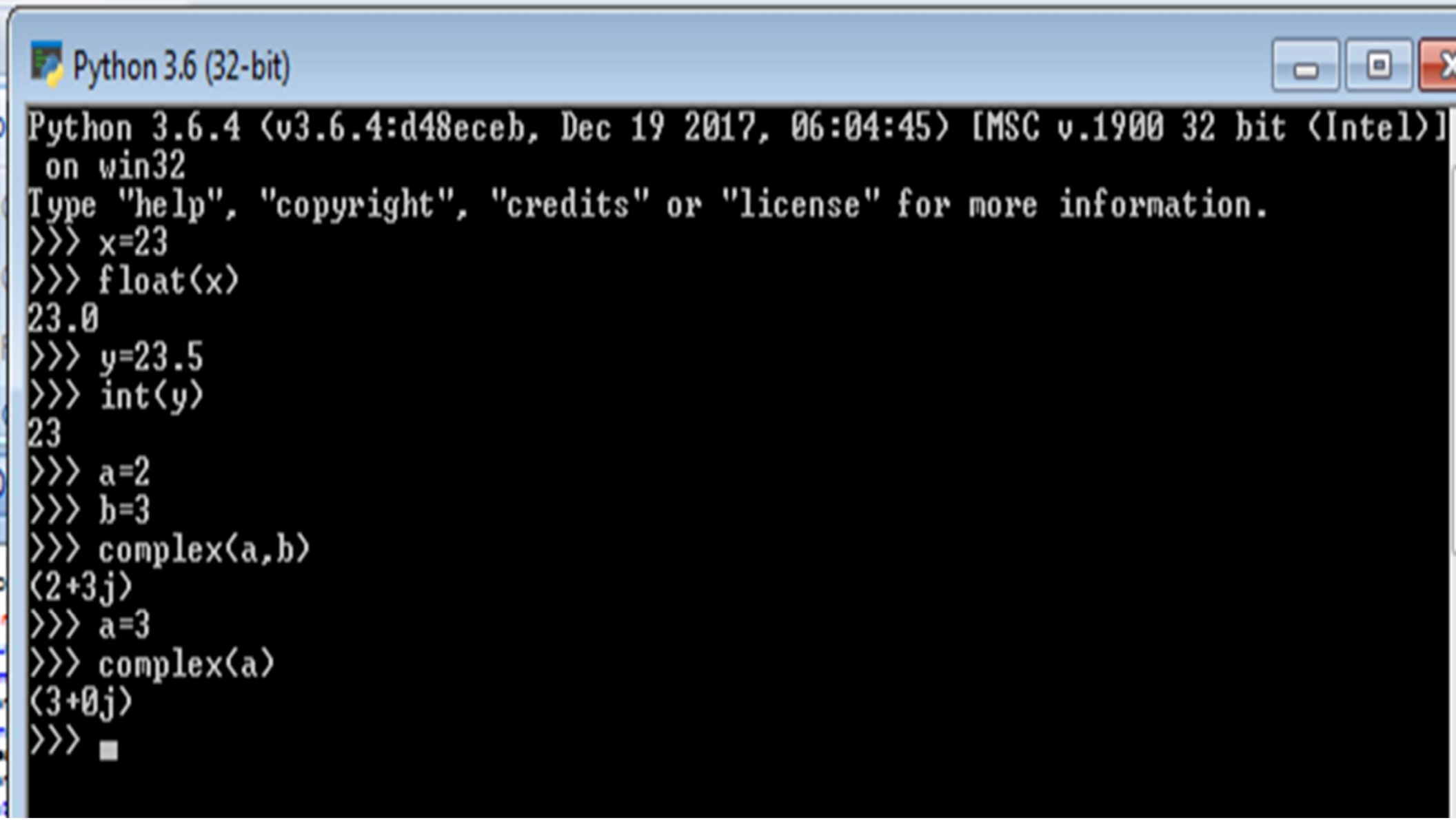
y=1-2j

# Python Data Type : Numbers

```
# Integers Numbers
year = 2010
year = int("2010")

# Floating Point Numbers
pi = 3.14159265
pi = float("3.14159265")

# Fixed Point Numbers
from decimal import Decimal
price = Decimal("0.02")
```

# Type Conversion/Coersion

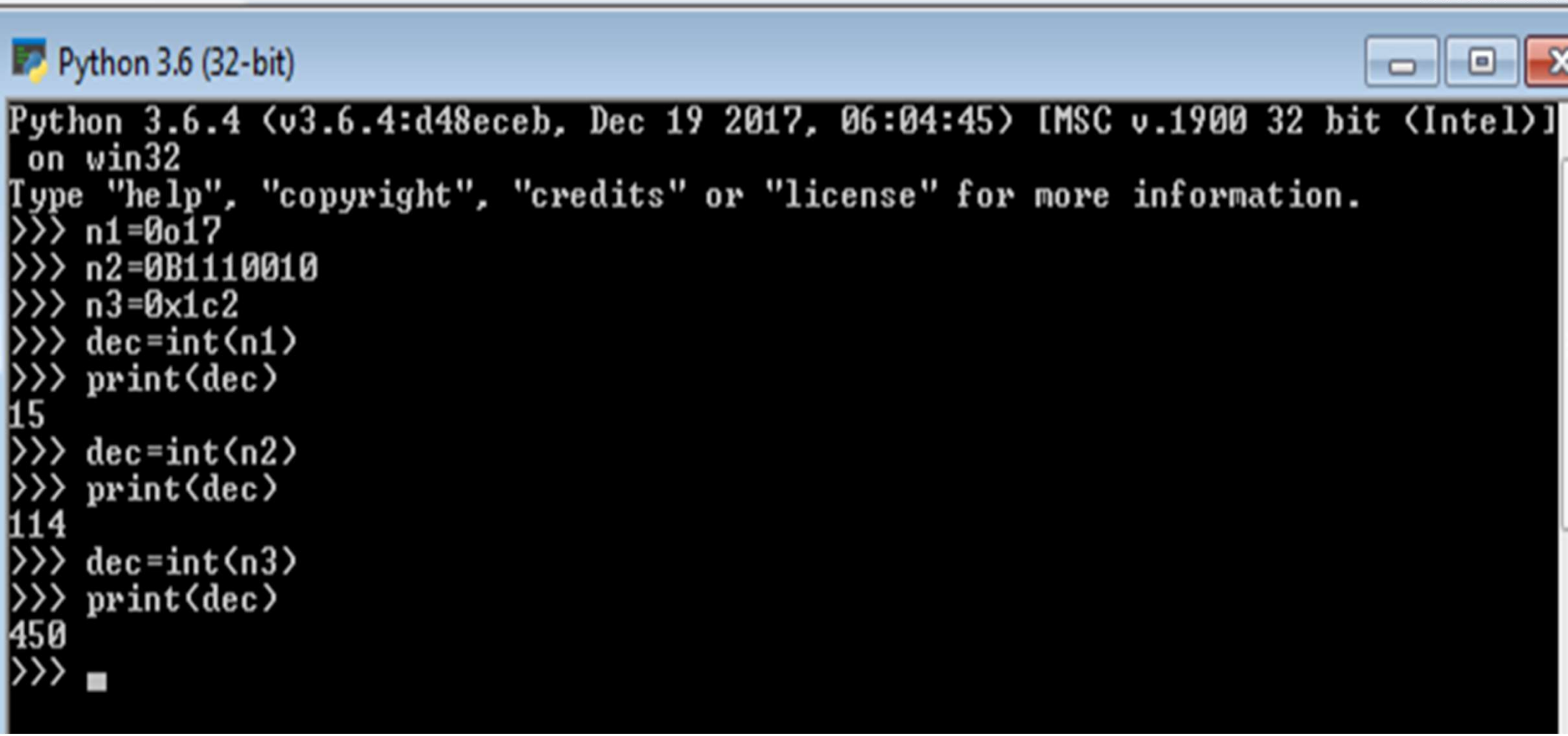
```
X=15.64
Int(x)          # Display 15
X=15
float(x)        #Display 15.0
a=10
Complex(a)    #display 10+0j
a=2
b=3
Complex(a,b)   #display 2+3j
```

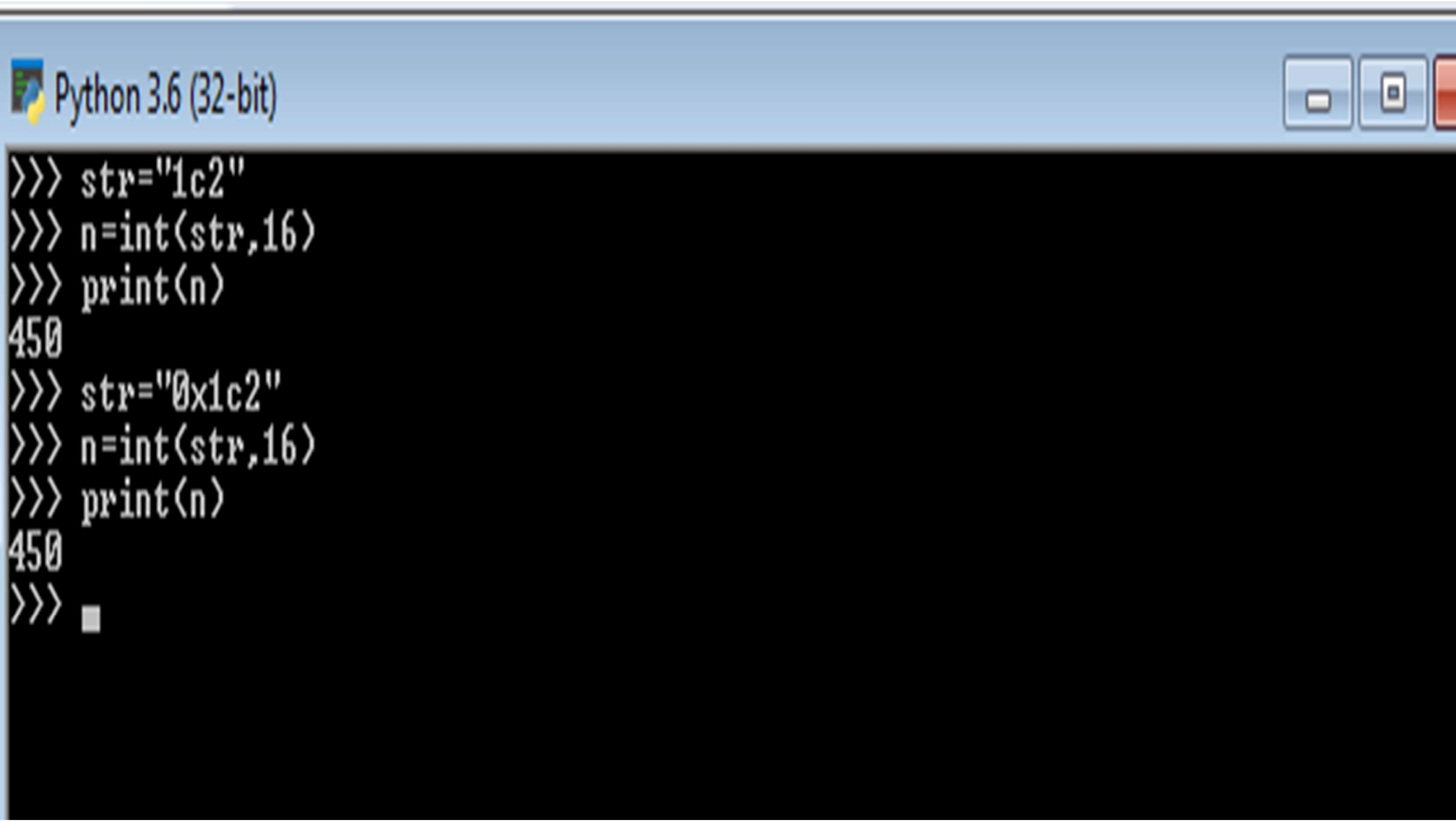

Python 3.6 (32-bit)
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x=23
>>> float(x)
23.0
>>> y=23.5
>>> int(y)
23
>>> a=2
>>> b=3
>>> complex(a,b)
(2+3j)
>>> a=3
>>> complex(a)
(3+0j)
>>>


# Python Data Type : Numbers

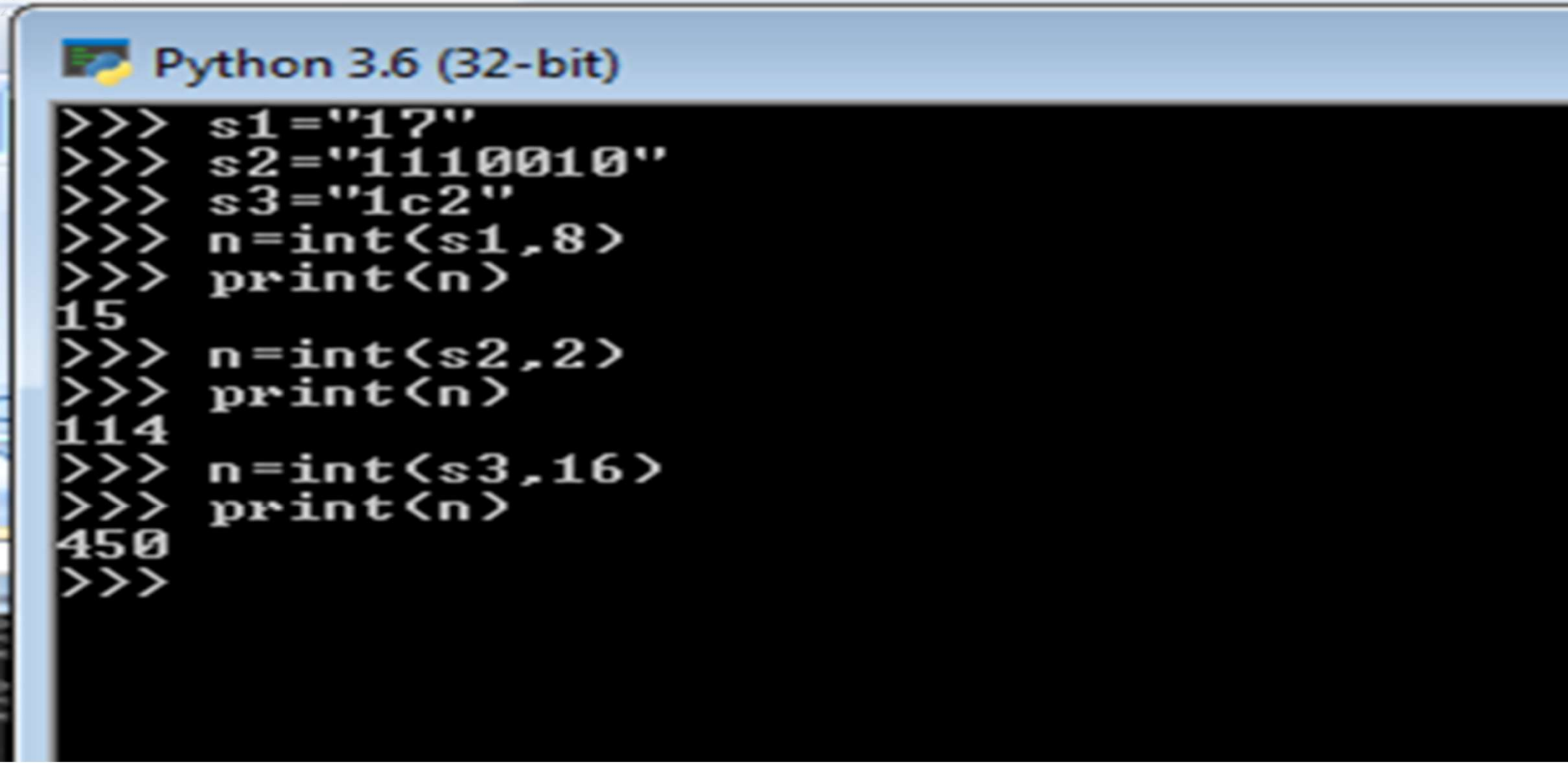

- **Other numeric types:**
- **Binary numbers**
  - Examples: `0b1011100,0B100110`
  - Must start with a leading `0b or 0B`
- **Octal constants**
  - Examples: `0O177, -0o1234`
  - Must start with a leading `0O or 0o`
- **Hex constants**
  - Examples: `0x9ff, 0X7AE`
  - Must start with a leading `0x` or `0X`

# Conversion from Octal, Binary and Hexadecimal to Decimal

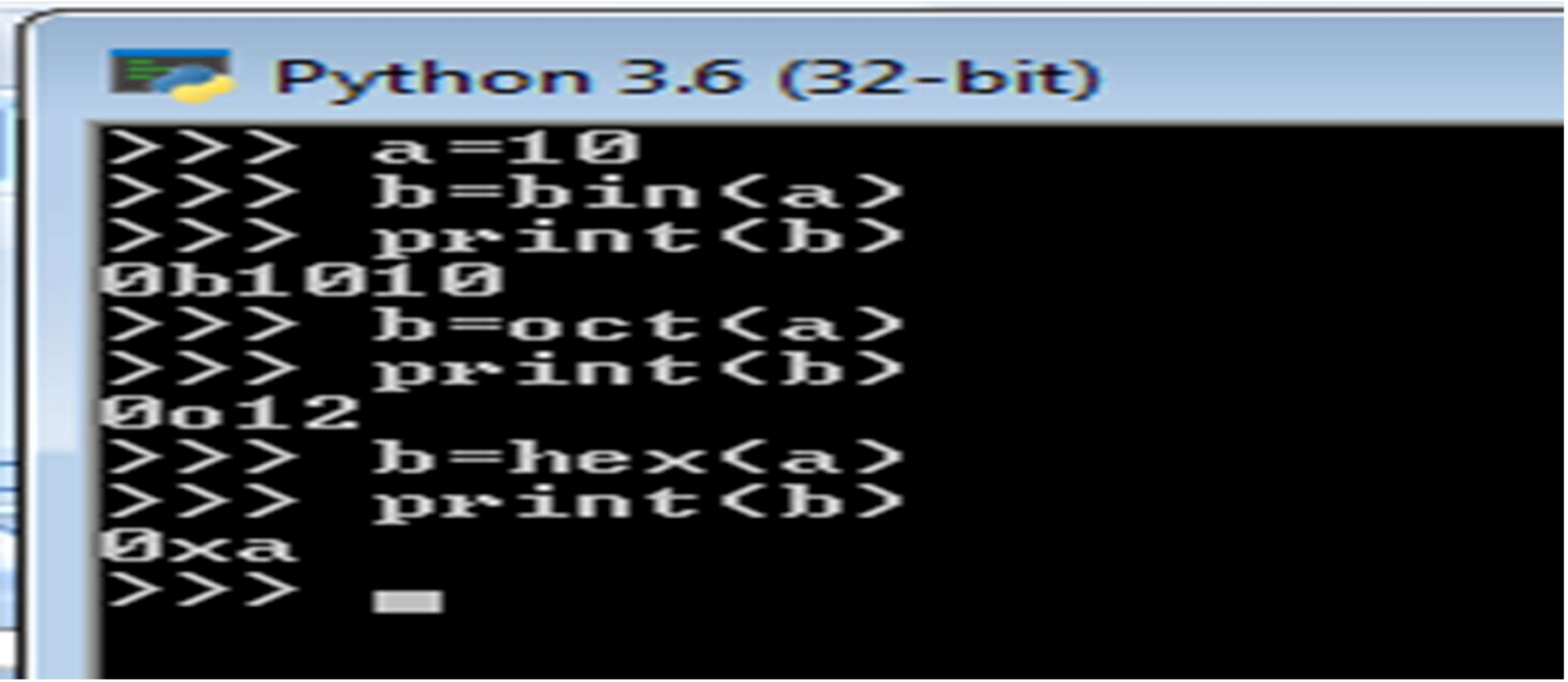



Python 3.6 (32-bit)
>>> str="1c2"
>>> n=int(str,16)
>>> print(n)
450
>>> str="0x1c2"
>>> n=int(str,16)
>>> print(n)
450
>>>

# Conversion using int() function



# How to get binary, Octal and hexadecimal?



# Python Data Type : String




# Python Data Type : String

- **String** is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

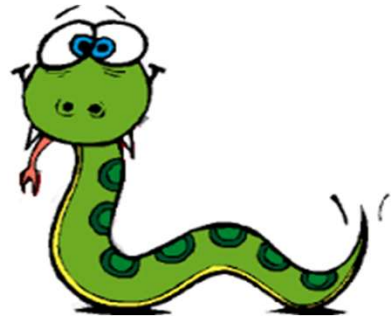
  **s = "This is a string"**

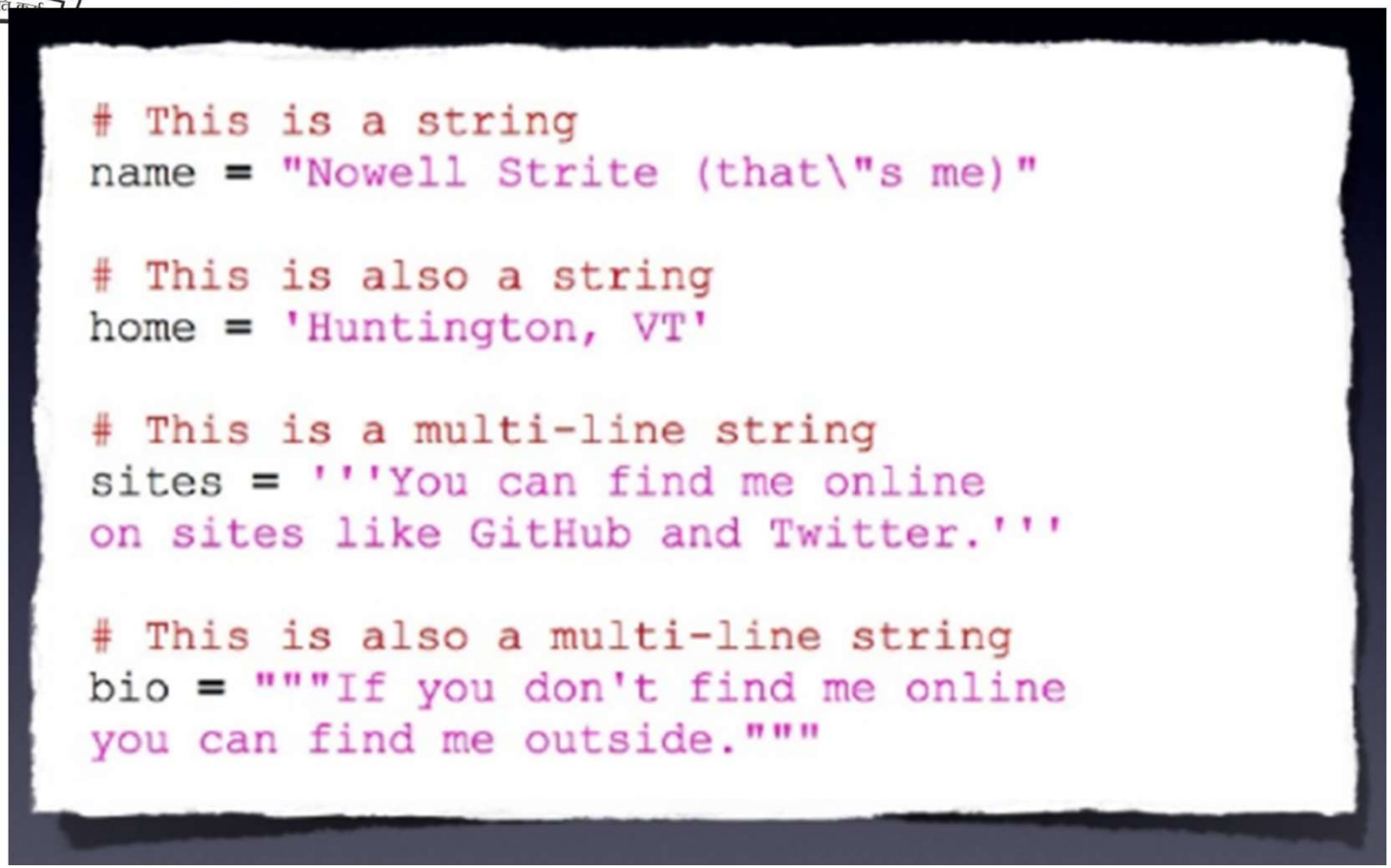
- Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

  **len(str)=>calculate length of string.**

  **a= len(str)**







```
# This is a string
name = "Nowell Strite (that\"s me)"

# This is also a string
home = 'Huntington, VT'

# This is a multi-line string
sites = '''You can find me online
on sites like GitHub and Twitter.'''

# This is also a multi-line string
bio = """If you don't find me online
you can find me outside."""
```

```
>>> name="Nowell Strite (that's me)"
>>> name
"Nowell Strite (that's me)"
>>> name="Nowell Strite (that"s me)"
SyntaxError: invalid syntax
>>> name="Nowell Strite (that\"s me)"
>>> name
'Nowell Strite (that"s me)'
>>>
```

# Operations on string

## Concatenation, Repetition & Replace

- Strings are concatenated with the + sign:

```
>>> 'abc'+'def'
'abcdef'
```

- Strings are repeated with the * sign:

```
>>> 'abc'*3
'abcabcabc'
```

- String replace:

```
>>> s='I like cmp99'
>>> n= s.replace('like', 'love')
>>> print(n)
I love cmp99
```

# Operations on string

## Indexing and slicing

- Python starts indexing at `0`. A string `s` will have indexes running from `0` to `len(s)-1` (where `len(s)` is the length of `s`) in integer quantities.
- Python also supports negative indexes. For example, `s[-1]` means extract the first element of `s` from the end (same as `s[len(s)-1]`)

```
>>> s[-1]
'g'
>>> s[-2]
'n'
```

- `s[i]` fetches the ith element in s

```
>>> s = 'string'
>>> s[1]  # note that Python considers 't' the first element
't'       # of our string s
```

- `s[i:j]` #fetches elements i (inclusive) through j (not inclusive)

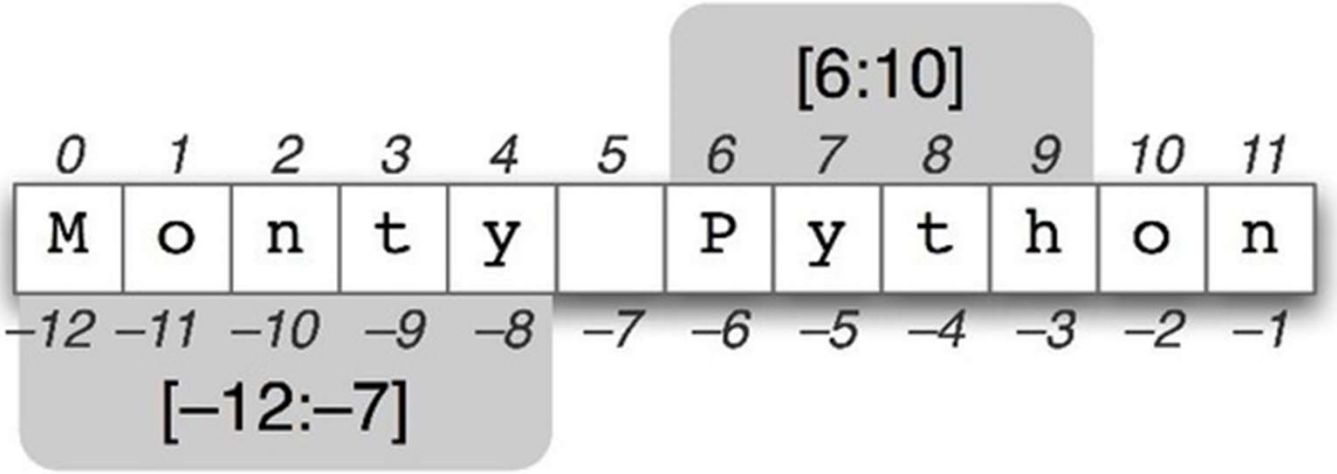
```
>>> s[1:4]
'tri'
```

- `s[:j]` # fetches all elements up to, but not including j

```
>>> s[:3]
'str'
```

- `s[i:]` #fetches all elements from i onward (inclusive)

```
>>> s[2:]
'ring'
```


[6:10]
0 1 2 3 4 5 6 7 8 9 10 11
M o n t y P y t h o n
−12 −11 −10 −9 −8 −7 −6 −5 −4 −3 −2 −1
[−12:−7]


# Operations on string


- **Indexing and slicing, contd.**
- `s[i:j:k]` extracts every `k`th element starting with index `i` (inclusive) and ending with index `j` (not inclusive)

```
>>> s[0:5:2], s[1:5:2]
'srn', 'ti'
```

# Operations on string

- **Upper case/ Lower case:**

```
>>> s='Abc'
>>> print(s.upper())
ABC
>>> print(s.lower())
abc
>>> print(s.capitalize())
Abc
```

- **Join: add any character into string**

```
>>> print(":".join('python'))
p:y:t:h:o:n
>>> print('a'.join("python"))
payatahaoan
```

- Reverse:

```
>>>print(''.join(reverse('python')))
Nohtyp
```

- Split:

```
>>> w='Guru career cmp99'
>>> print (w.split('r'))
Gu u ca ee cmp99
```

# String functions in python

- Python has lots of built-in methods that you can use on strings, we are going to cover some frequently used methods for string like
  - **len()**
  - count()
  - capitalize(), lower(), upper()
  - istitle(), islower(), isupper()
  - find(), rfind(), replace()
  - index(), rindex()
  - Methods for validations like
    - isalpha(), isalnum(), isdecimal(), isdigit()
  - strip(), lstrip(), rstrip()
  - Etc..

# Length

- The len() function returns the number of items in an object (such as a list, tuple, or string).

```
my_string = "Hello, World!"
length_of_string = len(my_string)
print(length_of_string)  # Output: 13
```

# Count:

The count() method returns the number of occurrences of a specified value in a string or list.

```
my_string = "Hello, Hello, World!"
count_of_hello = my_string.count("Hello")
print(count_of_hello)  # Output: 2
```

# Capitalize, Lower, Upper:

These string methods change the case of the letters in a string.

```
my_string = "hello world"
capitalized_string = my_string.capitalize()
print(capitalized_string)  # Output: Hello world

lowercased_string = my_string.lower()
print(lowercased_string)  # Output: hello world

uppercased_string = my_string.upper()
print(uppercased_string)  # Output: HELLO WORLD
```

# Istitle, Islower, Isupper:

These methods check if the string's words are in title case, lowercase, or uppercase, respectively.

```
my_string = "Title Case"
print(my_string.istitle())  # Output: True

print(my_string.islower())  # Output: False

print(my_string.isupper())  # Output: False
```

# Find, Rfind, Replace:
# These methods help in searching for substrings or replacing them in a string.

```
my_string = "Hello, World!"
index_of_comma = my_string.find(",")
print(index_of_comma)  # Output: 5

last_index_of_l = my_string.rfind("l")
print(last_index_of_l)  # Output: 10

new_string = my_string.replace("World", "Python")
print(new_string)  # Output: Hello, Python!
```

# Index, Rindex:

These methods return the index of the first or last occurrence of a substring in a string.

```
my_string = "Hello, World!"
index_of_l = my_string.index("l")
print(index_of_l)  # Output: 2

last_index_of_o = my_string.rindex("o")
print(last_index_of_o)  # Output: 8
```

# Validation Methods:
# These methods check if the string meets certain criteria.

```
alpha_string = "abc"
print(alpha_string.isalpha())  # Output: True

alnum_string = "abc123"
print(alnum_string.isalnum())  # Output: True

digit_string = "123"
print(digit_string.isdigit())  # Output: True

decimal_string = "12.34"
print(decimal_string.isdecimal())  # Output: False
```

- The isdecimal() method checks if all characters in a string are decimal characters (0-9). However, the string '123.5' contains a decimal point, so it is not considered a decimal string. Therefore, the isdecimal() method will return False for this string.

```
1. x = '123.5'
2. r = x.isdecimal()
3. print(r)  # Output: False
```

In this example, r will be False because the string '123.5' is not composed of only decimal digits. If you want to check if a string represents a decimal number (including the possibility of a decimal point), you might consider using the isdigit() method instead.

# Using in as an Operator

- The in keyword can also be used to check to see if one string is "in" another string
- The in expression is a logical expression and returns True or False and can be used in an if statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print 'Found it!'
...
Found it!
>>>
```

# Comparison of Strings

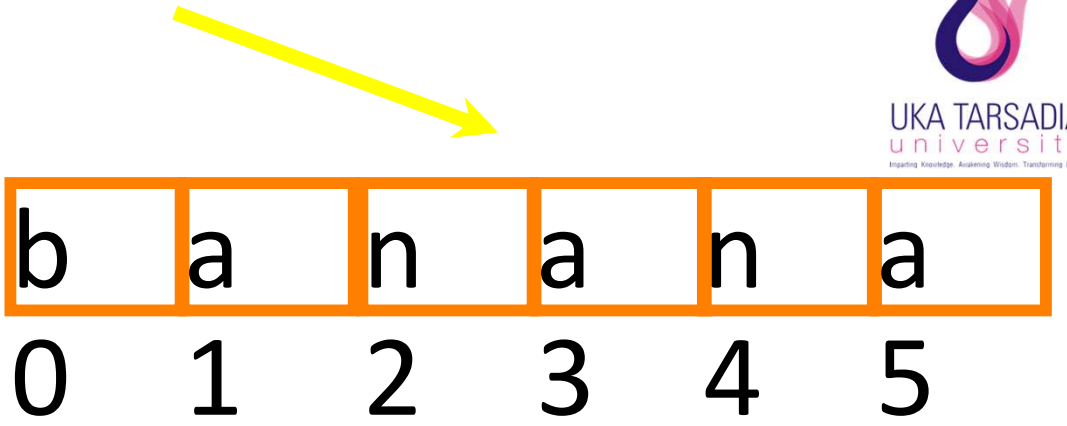
```
>>> x='boy'
>>> y='box'
>>> print(x>y)
True
>>> x='boY'
>>> y='box'
>>> print(x>y)
False
>>> x='boY'
>>> y='box'
>>> print(x<y)
True
>>>
```

# Remove spaces on left side and right side of string

```
>>> str='              string     '
>>> print(str.lstrip())
string
>>> print(str.rstrip())
     string
>>>
```

# Searching a String

- We use the find() function to search for a substring within another string
- find() finds the first occurance of the substring
- If the substring is not found, find() returns -1
- Remember that string position starts at zero




```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print pos
2
>>> aa = fruit.find('z')
>>> print aa
-1
```

# Find Substring

**Syntax: mainstring.find(substring,beginning,ending)**

```
>>> str="This is SFIT"
>>> n=str.find('is',0,len(str))
>>> if n==-1:
    print("String not found")
else:
    print("String found at position",n+1)
String found at position 3
```

# Data structures in python

- There are four built-in data structures in Python - *list, dictionary, tuple and set*.

| Name | Type | Description |
|---|---|---|
| List | list | **Ordered** Sequence of objects, will be represented with **square** brackets **[ ]**<br>Example : [ 18, "Amitcs", True, 102.3 ] |
| Dictionary | dict | **Unordered key : value** pair of objects , will be represented with **curly** brackets **{ }**<br>Example : { "college": "Amtics", "code": "054" } |
| Tuple | tup | **Ordered immutable** sequence of objects, will be represented with **round** brackets **( )**<br>Example : ( 18, "Amtics", True, 102.3 ) |
| Set | set | **Unordered** collection of **unique** objects, will be represented with the **curly** brackets **{ }**<br>Example : { 18, "Amtics", True, 102.3 } |

# Python Data Type : List

["Python", 11, "C", "R"]

Python Lists

# Python Data Type : List

- List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type.
- Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].
- **>>> a = [1, 2.2, 'python']**
- **Examples:**

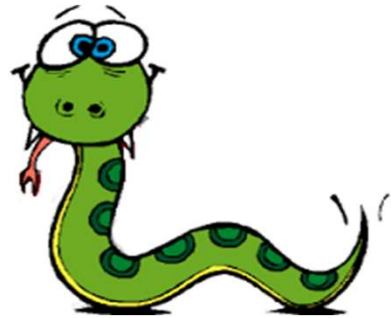
```
L1 = [0,1,2,3], L2 = ['zero', 'one'],
L3 = [0,1,[2,3],'three',['four, one']],
L4 = []
```

# Python Data Type : List

- We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts from 0 in Python

```
>>>a = [5,10,15,20,25,30,35,40]
>>>print("a[2] = ", a[2])
15
>>>print("a[0:3] = ", a[0:3])
[5, 10, 15]
>>>print("a[5:] = ", a[5:])
[30, 35, 40]
```



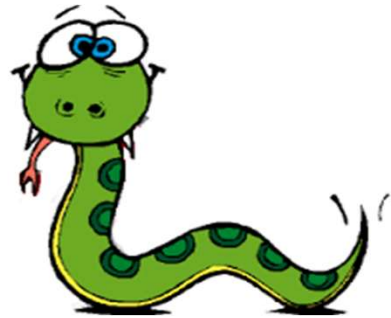


# Python Data Type : List

- Lists are mutable, meaning, value of elements of a list can be altered.

```
>>> a = [1,2,3]
>>> a[2]=4
>>> print(a)
[1, 2, 4]
```





# Operations on Lists (1)

- **Some basic operations on lists:**

- **Indexing:** `L1[i], L2[i][j]`

```
>>> l1=[1,2,3,[4,5]]
>>> l1[3][0]
4
```

- **Slicing:** `L3[i:j]`
- **Concatenation:**

```
>>> L1 = [0,1,2]; L2 = [3,4,5]
>>> L1+L2
[0,1,2,3,4,5]
```

- **Repetition:**

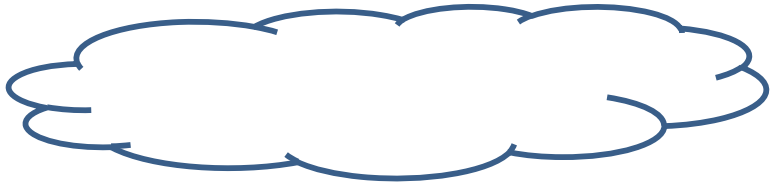
```
>>> L1*3
[0,1,2,0,1,2,0,1,2]
```

- **Appending:**

```
>>> L1.append(3)
[0,1,2,3]
```

- **Insert**

```
>>>L1.insert(0,4)( index,obj)
 [4,0,1,2,3]
```



- **Sorting:**

```
>>> L3 = [2,1,4,3]
>>> L3.sort()
[1,2,3,4]
```

# Operations on Lists (2)

- **More list operations:**

- **Reversal:**

  >>> L4 = [4,3,2,1]

  >>> L4.reverse()

  >>> L4

  [1,2,3,4]

- **Length**:

  >>>len(L4)

  4

- **Shrinking**:

  >>> del L4[2]  => [1,2,4]

  >>> L4.remove(4)

```
L4 = [1, 2, 3, 4, 5]
L4.remove(4)
print(L4)
```

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

removing element

## Index and slice assignment:

```
>>> l1=[1,2,4]
>>> l1[1]=4
>>> l1
[1, 4, 4]
>>> l1[1:4] = [4,5,6]
>>> l1
[1, 4, 5, 6]
```

- **Making a list of integers:**

```
>>> range(4)
[0,1,2,3]
>>> range(1,5)
[1,2,3,4]
```

- 
```
>>> list(range(1,7,2))
[1, 3, 5]
```

- **Min & Max Value:**

```
>>>print(max(L4))
 4
>>> print (min(L4))
1
```

**Assign values:**

```
>>> lst=[1,2,3,4]
>>> lst[1:3]=11,12
>>> print(lst)
[1,11,12,4]
```

**Membership in list:**

```
>>> x=[10,20,30,40,50]
>>> print(10 in x)
True
```

**Cloning a list:**

```
>>> x=[1,2,3]
>>> y=x.copy()
>>> print(y)
[1, 2, 3]
```

**Nested List:**

```
>>> l1=[1,2,3,[4,5]]
>>> l1[3][0]
4
```

# Two-Dimensional Lists

- Two-dimensional list: a list that contains other lists as its elements
  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data
- To process data in a two-dimensional list need to use two indexes
- Typically use nested loops to process

# Nested list as matrices


# Nested list as matrices


```
>>> l1=[[11,12,13],[13,14,15],[15,16,17]]
>>> l1[0][0]
11
>>> l1[1][2]
15
>>>
```


UKA TARSADIA university
Imparting Knowledge. Awakening Wisdom. Transforming Lives.

# Using Lists as Stacks

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out").
- To add an item to the top of the stack, use append().
- To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

# Using Lists as Queues

- It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose.
- While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```


UKA TARSADIA university

# List Comprehensions

- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

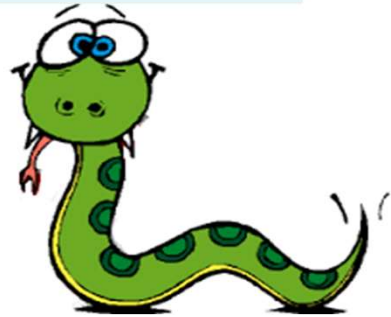
- For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

# Python Data Type : Tuple

( [1, 2], (3, 4), “Tuple” )

Python tuples



# Python Data Type : Tuple

- Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. **Tuples once created cannot be modified.**
- Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.
- It is defined within parentheses **()** where items are separated by commas.

**>>> t = (5,'program', 1+3j)**

**Examples:**

```
t1 = (0,1,2,3), t2 = ('zero', 'one'),
t3 = (0,1,(2,3),'three',('four,one'))
t4 = ()
t5 = (50,)
```

# Python Data Type : Tuple

```
>>>t5 = (50,)
>>>t5
(50,)
>>>t5=(50)
>>>t5
50
```

**The parentheses are optional but is a good practice to write it.**

```
>>>tup3='a','b','c','d';
>>>tup3
('a','b','c','d')
```

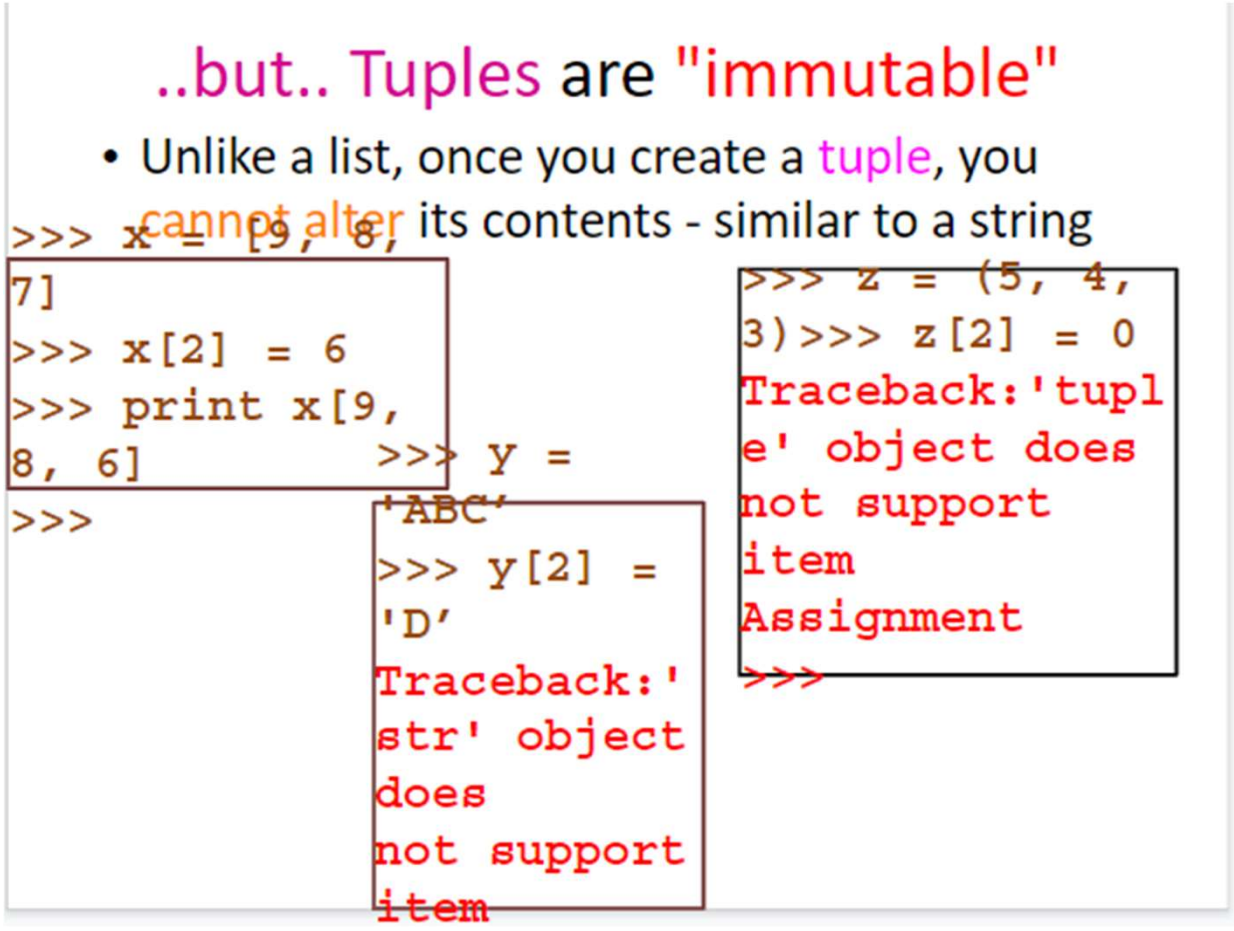
# Python Data Type : Tuple

- **Tuple: an immutable sequence**
  - Very similar to a list
  - Once it is created it cannot be changed
  - **Format: tuple_name = (item1, item2)**
  - Tuples support operations as lists
    - **Subscript indexing for retrieving elements**
    - **Methods such as index**
    - **Built in functions such as len, min, max**
    - **Slicing expressions**
    - **The in, +, and * operators**

# Python Data Type : Tuple

- Elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

```
>>> t1=(1,2,3,4,[3,"str"])
>>> t1[4][1]=4
>>> t1
(1, 2, 3, 4, [3, 4])
>>>
```

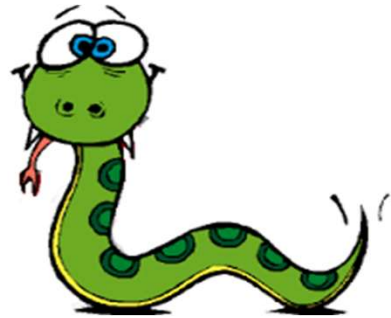

..but.. Tuples are "immutable"
• Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print x[9, 8, 6]
>>>
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback: 'str' object does not support item
>>> z = (5, 4, 3)>>> z[2] = 0
Traceback:'tuple' object does not support item Assignment
>>>

# Python Data Type : Tuple

- We can use the slicing operator [] to extract items but **we cannot change its value.**

```
>>>t = (5,'program', 1+3j)
>>>print("t[1] = ", t[1])
program
>>>print("t[0:3] = ", t[0:3])
(5, 'program', (1+3j))
>>>t[0]=10      error
```





# Operations on tuples (1)

- **Packing : place value into tuple**

```
>>> t = ('guru',20,'education')
>>> print(t)
('guru', 20, 'education')
```

- **Unpacking: extract value into variables**

```
>>> (company,empid,profile)=t
>>> print (company)
guru
>>> print(empid)
20
>>> print(profile)
education
```

# Packing Unpacking

```
>>> t1=(1,2,3,4)
>>>a,b,c,d=(1,2,3,4)
>>> (a,b,c)=(1,2,3,4)[0:3]
>>> print(a,b,c)
1 2 3
>>>
```

**Packing:**

Packing involves putting values into a tuple. This is typically done when you have multiple values that you want to group together.

```
# Packing
my_tuple = 1, 2, 3, 'hello'
print(my_tuple)
# Output: (1, 2, 3, 'hello')
```

```
# Packing with parentheses
my_tuple = (1, 2, 3, 'hello')
print(my_tuple)
# Output: (1, 2, 3, 'hello')
```

**Unpacking:**

Unpacking involves extracting values from a tuple and assigning them to separate variables.

```
# Unpacking
a, b, c, d = my_tuple
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
print(d)  # Output: 'hello'
```

```
# Unpacking with *
first, *rest = my_tuple
print(first)  # Output: 1
print(rest)   # Output: [2, 3, 'hello']
```

- **Deleting :**

```
>>> del(t[2])      #element deletion not supported but can
   delete the whole tuple.
>>> del(t)
```

- **Comparing : comparison starts from 1st element**

```
>>>a=(5,6)
>>>b=(1,4)
>>>if(a>b):
            print('a is bigger')
else:
               print('b is bigger')
a is bigger
```

```
>>>a=(5,4)
>>>b=(6,4)
>>>if(a>b):
            print('a is bigger')
else:
               print('b is bigger')
b is bigger
```


# Tuples are Comparable


- The comparison operators work with tuples and other sequences If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') < ('Adams', 'Sam')
False
```


# Operations on tuples (2)


- Concatenation ,
- repetition,
- length,
- max, min,
- count,
- index
- All are same as list

# Concatenation, Repetition, index

```
>>> t1=(1,2,3)
>>> t2=(3,'priya',4)
>>> t3=t1+t2
>>> print(t3)
(1, 2, 3, 3, 'priya', 4)
>>> print(t1*3)
>>> l=(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> print(l.index(2))
1
```

# Length, max, min, index, count

```
>>> print(len(t1))
3
>>> print(max(t1))
3
>>> print(min(t1))
1
>>> print(t1[2])
3
>>> my_tuple = ('a','p','p','l','e',)
>>> my_tuple.count('p')
2
>>>
```


- **In operator**

```
>>> l=(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> print(2 in l)
True
```

- **Iterating through a tuple**

```
>>> for name in ('John','Kate'):
        print("Hello",name)
Hello John
Hello Kate
```


# Tuples (cont'd.)


- Tuples **do not support** the methods:
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`


# Tuples (cont'd.)


- **Advantages for using tuples over lists:**
  - **Processing tuples is faster than processing lists**
  - **Tuples are safe**
  - **Some operations in Python require use of tuples**

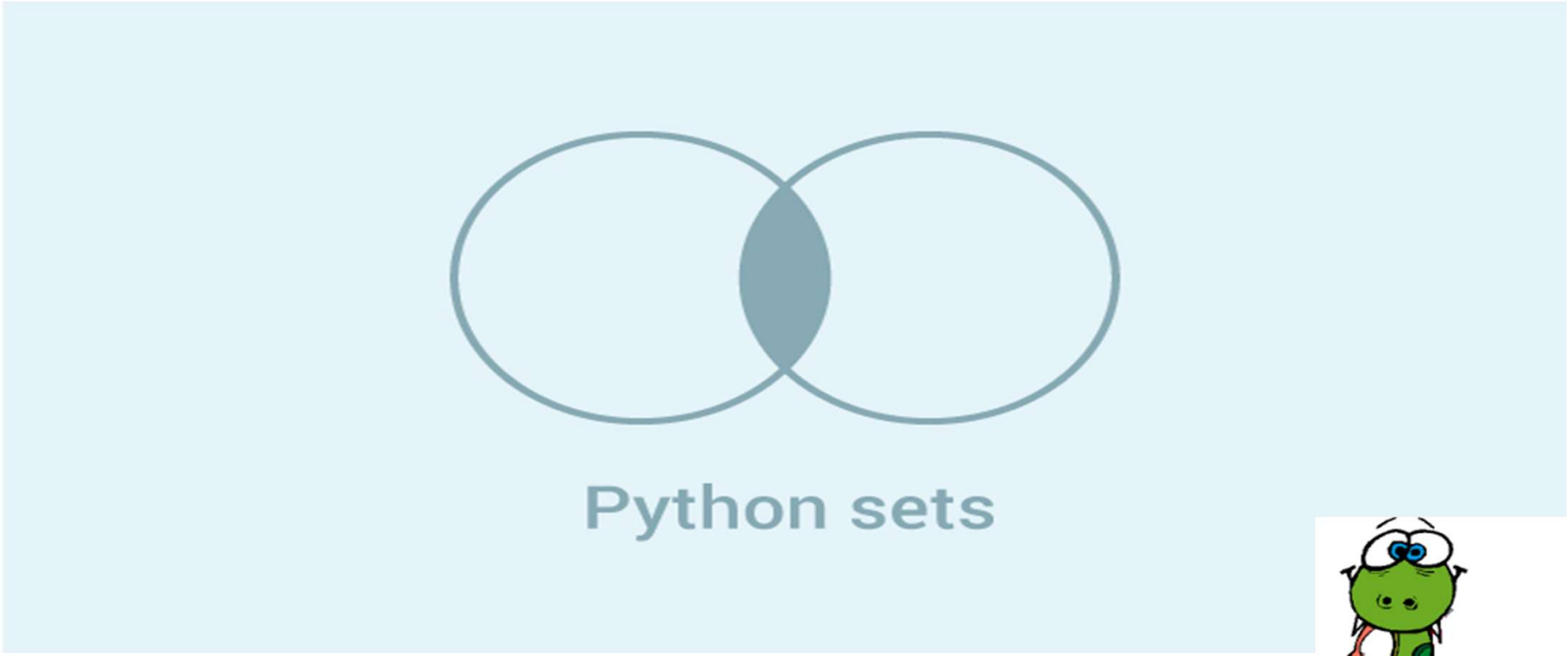
- **`list()` function: converts tuple to list**

```
>>> t1=(1,2,"abc")
>>> list(t1)
[1, 2, 'abc']
```

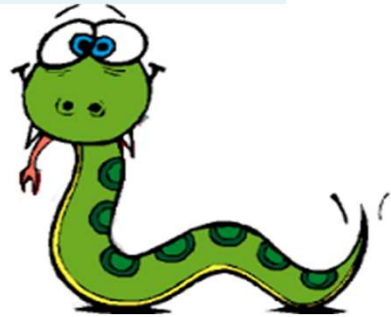
- **`tuple()` function: converts list to tuple**

```
>>> l1=[1,2,3,4]
>>> tuple(l1)
(1, 2, 3, 4)
```

# Python Data Type : Set





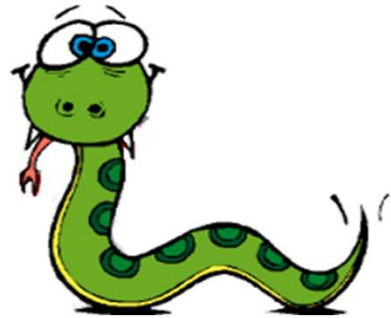



# Python Data Type : Set

- **Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.**

```
>>>a = {5,2,3,1,4}
>>>print("a = ", a)
a= {1, 2, 3, 4, 5}
>>>print(type(a))
<class 'set'>
```



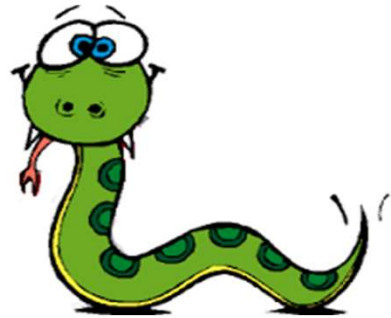


# Python Data Type : Set

- We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
>>>a = {1,2,2,3,3,3}
>>>print(a)
{1, 2, 3}
```





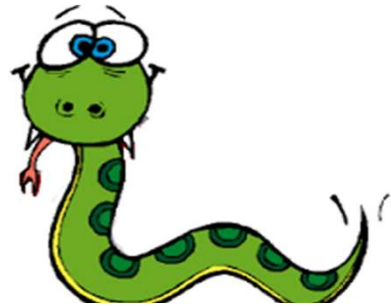
# Python Data Type : Set

- Since, set are unordered collection, **indexing has no meaning. Hence the slicing operator [] does not work.**

```
>>>a = {1,2,3}
>>>print(a[1])        error
```

- **Traceback (most recent call last): File "<string>", line 301, in runcode File "<interactive input>", line 1, in <module> TypeError: 'set' object does not support indexing**






# Sets, as in Mathematical Sets


- in mathematics, a set is a collection of objects, potentially of many different types
- in a set, no two elements are identical. That is, a set consists of elements each of which is unique compared to the other elements
- there is no order to the elements of a set.
- a set with no elements is the empty set.

# Creating a set

**Set can be created in one of two ways:**

- **constructor: `set(iterable)` where the argument is iterable.**

**iterable (Optional)** - a sequence (string, tuple, etc.) or collection (set, dictionary, etc.) or an iterator object to be converted into a set.

```
my_set = set('abc')
my_set →  {'a', 'b', 'c'}
```

- **shortcut: { }, braces where the elements have no colons (to distinguish them from dicts)**

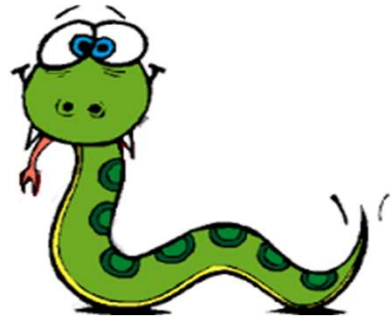
```
my_set = {'a','b','c'}
```

# Python Data Type : Set

- We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}
print(a)
```

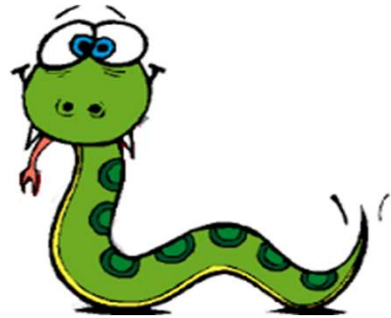




# Python Data Type : Set

- Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

```
>>>a = {1,2,3}
>>>print(a[1])
Traceback (most recent call last): File
"<string>", line 301, in runcode File
"<interactive input>", line 1, in
<module> TypeError: 'set' object does
not support indexing
```




# Diverse elements


- A set can consist of a mixture of different types of elements

```
my_set = {'a',1,3.14159,True}
```

- as long as the single argument can be iterated through, you can make a set of it.


# no duplicates


- duplicates are automatically removed

```
my_set = set("aabbccdd")
print(my_set)
```

→ `{'a', 'c', 'b', 'd'}`


# Example


```
>>> null_set = set()          # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4}         # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}       # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")       # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'}          # order not maintained!
```


# common operators


**Most data structures respond to these:**

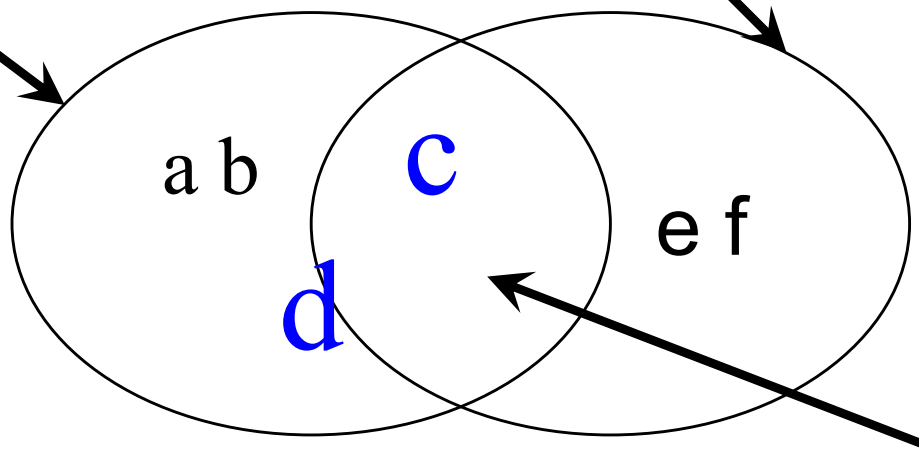
- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - boolean indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`


# Set operators


- The set data structure provides some special operators that correspond to the operators you learned in middle school.
- These are various combinations of set contents.
- These operations have both a method name and a shortcut binary operator.

# method: intersection, op: &

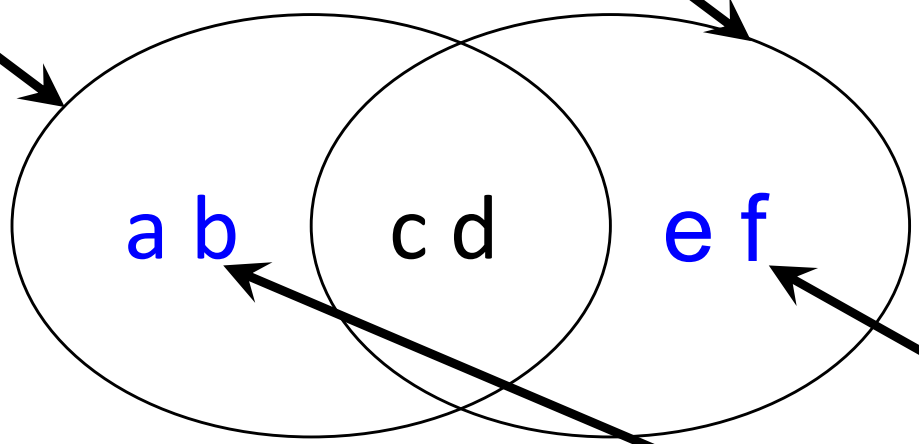
```
a_set=set("abcd")  b_set=set("cdef")
```




```
a_set & b_set →  {'c', 'd'}
b_set.intersection(a_set) →  {'c', 'd'}
```

# method:difference op: -

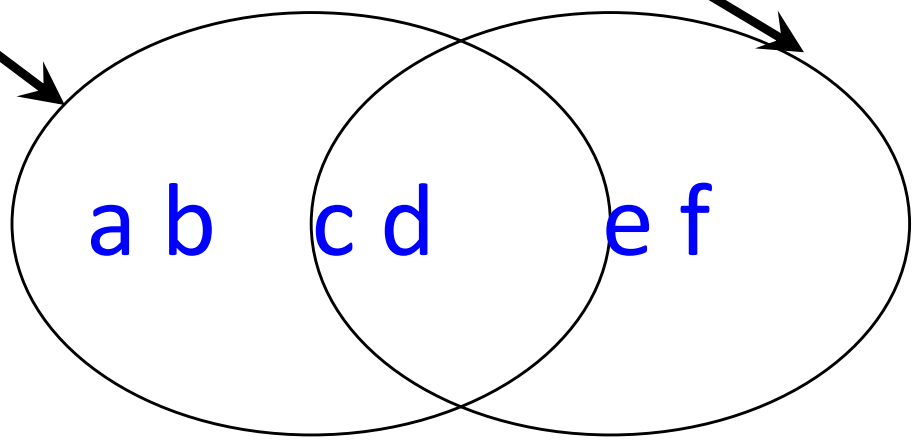
```
a_set=set("abcd")  b_set=set("cdef")
```




```
a_set - b_set →  {'a', 'b'}
b_set.difference(a_set) →  {'e',
'f'}
```

# method: union, op: |

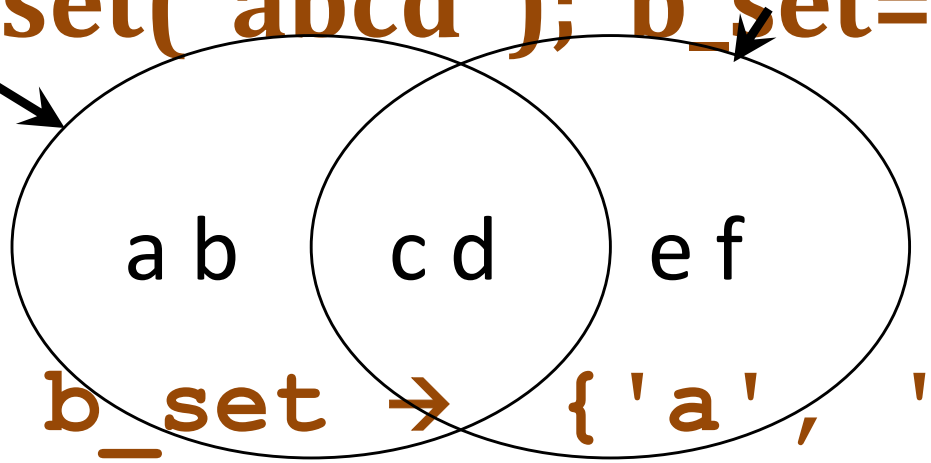
## a_set=set("abcd") b_set=set("cdef")




```
a_set | b_set →  {'a', 'b', 'c', 'd', 'e', 'f'}
b_set.union(a_set) →  {'a', 'b', 'c', 'd', 'e',
'f'}
```

# method:symmetric_difference, op: ^

**a_set=set("abcd"); b_set=set("cdef")**

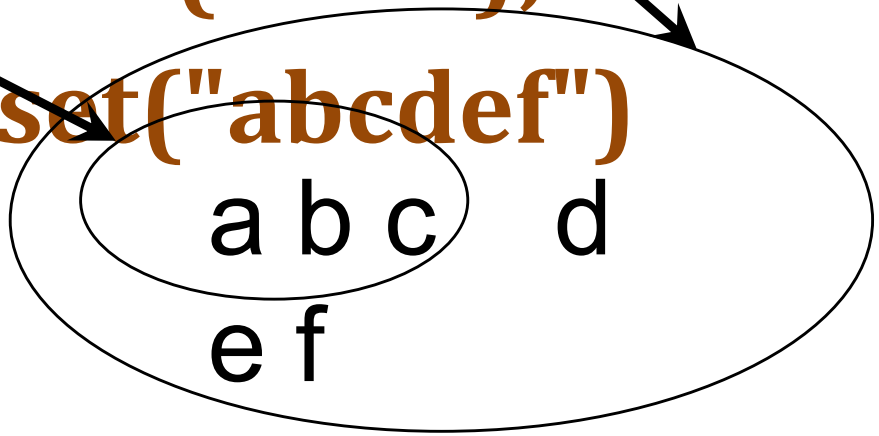



```
a_set ^ b_set → {'a', 'b', 'e',
'f'}
b_set.symmetric_difference(a_set) →
{'a', 'b', 'e', 'f'}
```

**method: issubset, op: <=**
**method: issuperset, op: >=**



**small_set=set("abc");**
**big_set=set("abcdef")**




```
small_set <= big_set →  True
big_set >= small_set →  True
```

# Other Set Ops

- `my_set.add("g")`
  - adds to the set, no effect if item is in set already

```
my_set.update([])
>>> thisset = {"apple", "banana", "cherry"}
>>> thisset.update(["orange", "mango", "grapes"])
>>> print(thisset)
{'mango', 'orange', 'banana', 'apple', 'cherry', 'grapes'}
```

- `my_set.clear()`
  - empties the set


# Other Set Ops

`my_set.remove("g")` versus
`my_set.discard("g")`

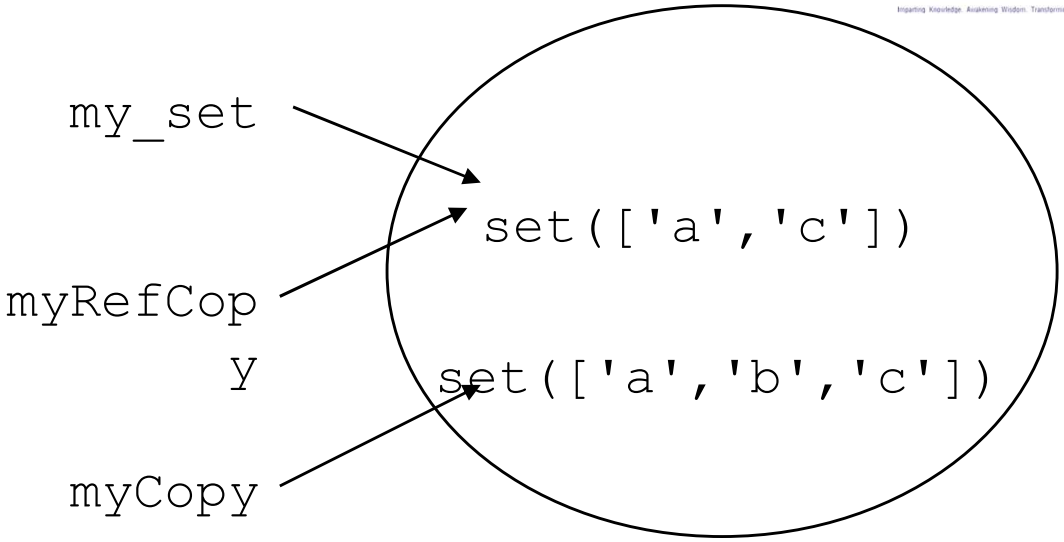
- `remove` throws an error if "g" isn't there.
- `discard` doesn't care. Both remove "g" from the set

`my_set.copy()`

- returns a shallow copy of `my_set`

# Copy vs. assignment

```
my_set=set {'a', 'b', 'c'}
my_copy=my_set.copy()
my_ref_copy=my_set
my_set.remove('b')
```




- copy() method returns a shallow copy of the set
- If we use "=" to copy a set to another set, when we modify in the copied set, the changes are also reflected in the original set

# Copy vs. assignment

```
>>> numbers = {1, 2, 3, 4}
>>> new_numbers = numbers
>>> new_numbers.add('5')
>>> print('numbers: ', numbers)

numbers:  {1, 2, 3, 4, '5'}
>>> print('new_numbers: ',new_numbers)
new_numbers:  {1, 2, 3, 4, '5'}
```

**if you need the original set unchanged when the new set is modified, you can use copy() method.**

# Copy vs. assignment

**The syntax of copy() is:**

```
set.copy()
```

**e.g.**

```
>>> numbers = {1, 2, 3, 4}
>>> new_numbers = numbers.copy()
>>> new_numbers.add('5')
>>> print('numbers: ', numbers)
numbers:  {1, 2, 3, 4}
>>> print('new_numbers: ', new_numbers)
new_numbers:  {1, 2, 3, 4, '5'}
```

# Python Data Type : Set

| Operation | Equivalent | Result |
|---|---|---|
| len(s) | | number of elements in set *s* (cardinality) |
| x in s | | test *x* for membership in *s* |
| x not in s | | test *x* for non-membership in *s* |
| s.issubset(t) | s <= t | test whether every element in *s* is in *t* |
| s.issuperset(t) | s >= t | test whether every element in *t* is in *s* |
| s.union(t) | s \| t | new set with elements from both *s* and *t* |
| s.intersection(t) | s & t | new set with elements common to *s* and *t* |
| s.difference(t) | s - t | new set with elements in *s* but not in *t* |
| s.symmetric_difference(t) | s ^ t | new set with elements in either *s* or *t* but not both |
| s.copy() | | new set with a shallow copy of *s* |

# Example of Set Operations

```
>>>s={1,2,3,4,5}
>>>t={1,4,7,8}
>>>c=len(s)
>>>print(c)
5
>>>print(s.union(t))
{1, 2, 3, 4, 5, 7, 8}
>>>print(s|t)
{1, 2, 3, 4, 5, 7, 8}
```

```
>>>print(s)
{1, 2, 3, 4, 5}
>>>print(s.intersection(t))
{1, 4}
>>>print(s)
{1, 2, 3, 4, 5}
>>>print(s&t)
{1, 4}
```

```
>>>print(s.difference(t))
{2, 3, 5}
>>>print(s)
{1, 2, 3, 4, 5}
>>>print(s-t)
{2, 3, 5}
>>> print(s.symmetric_difference(t))
{2, 3, 5, 7, 8}
>>>print(s)
{1, 2, 3, 4, 5}
>>>print(s^t)
{2, 3, 5, 7, 8}
```

# Program Exercise on Set

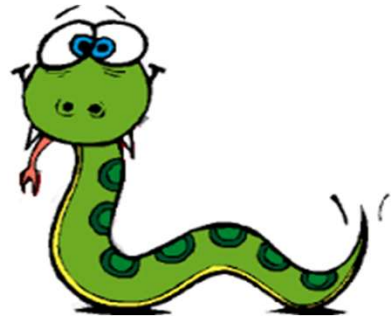
- Write a program to solve below problem .
- Among 60 students in a class ,45 passed in the first semester examination, 30 passed in second semester examinations. If 12 did not pass in either exam. How many passed in both examination

# Python Data Type : Dictionary

# Python Data Type : Dictionary

- **Dictionary** is an unordered collection of key-value pairs.
- It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.






# What is a dictionary?


- In data structure terms, a dictionary is better termed an *associative array, associative list* or a *map*.
- You can think if it as a list of pairs, where the first element of the pair, the ***key***, is used to retrieve the second element, the ***value***.
- Thus we ***map a key to a value***


# Key Value pairs


- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key

# Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate **key:value** pairs
- An item has a **key** and the corresponding value expressed as a pair, **key: value.**

```
>>>contacts= {'bill': '353-1234',
 'rich': '269-1234', 'jane': '352-1234'}
>>>print (contacts)
{'jane': '352-1234',
 'bill': '353-1234',
 'rich': '369-1234'}
```

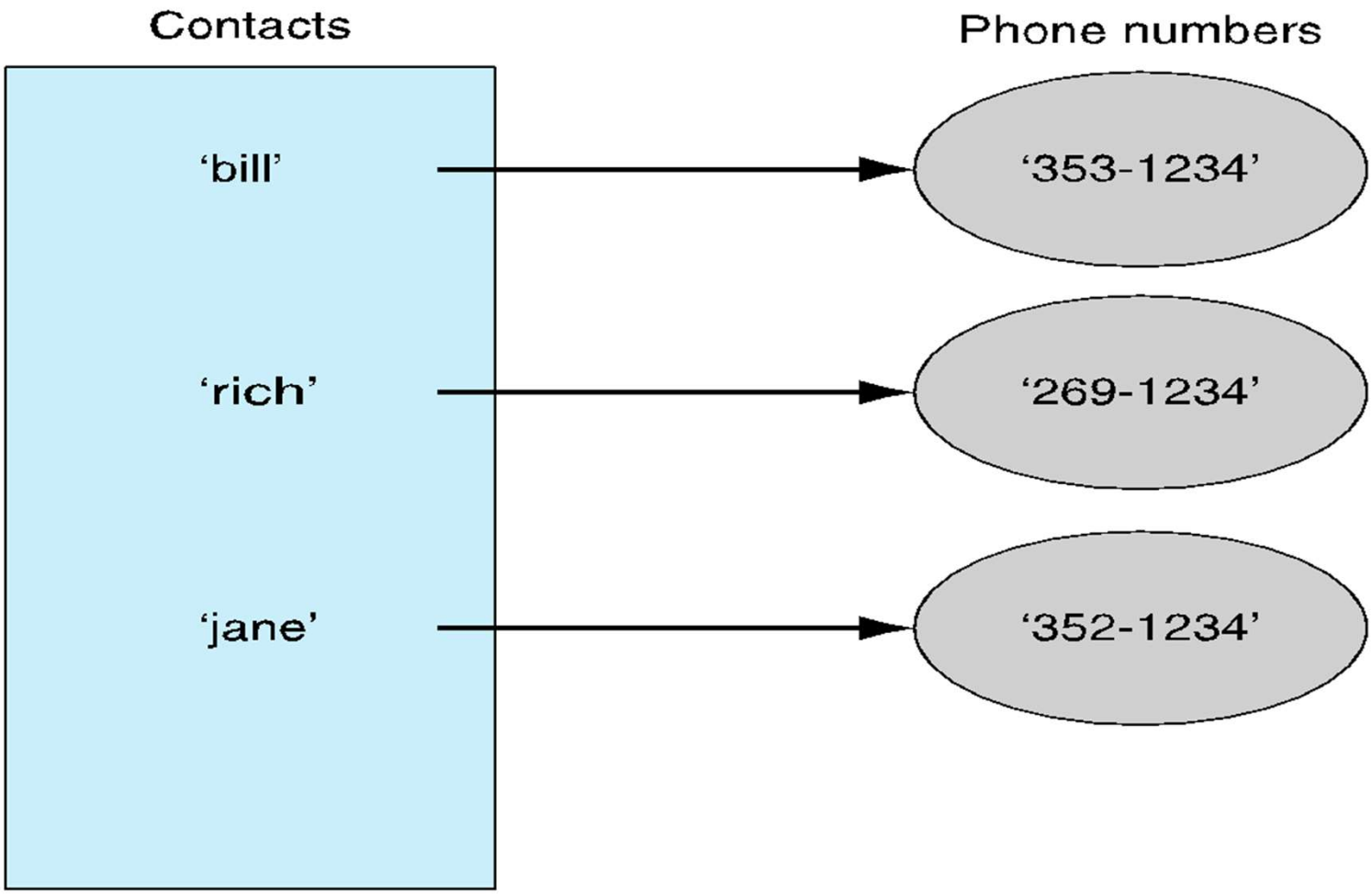



**FIGURE 9.1** Phone contact list: names and phone numbers.

# keys and values

- Key must be immutable
  - **strings, integers, tuples are fine**
  - **lists are NOT.**
- Value can be anything.

```
my_dict = {}
my_dict = {1: 'apple', 2: 'ball'}
my_dict = dict({1:'apple', 2:'ball'})…….Constructor
my_dict ={(1,2):”abc”, “string”:24}
```

# collections but not a sequence

- dictionaries are collections but they are not sequences such as lists, strings or tuples.
  - **there is no order to the elements of a dictionary**
  - **in fact, the order (for example, when printed) might change as elements are added or deleted.**
- So how to access dictionary elements?

# how to access dictionary elements?

```
>>> my_dict = {'name':'Jack', 'age': 26}
>>> print(my_dict['name'])
Jack
>>> print(my_dict.get('age'))
26
>>>print(my_dict['abc'])
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    print(my_dict['abc'])
KeyError: 'abc‘
>>> print(my_dict.get('abc'))
None
```

# Check Yourself: Dictionaries

- What is the output on line 2?

**A. 2**

**B. '2'**

**C. 6**

**D. KeyError**

```
1 x = {'a' :'b', 'b' :2, '2' : 6}
2 x[2]
```

# Access dictionary elements

**Access requires `[ ]`, but the *key* is the index!**

```
>>>my_dict={}
```

- an empty dictionary

```
>>>my_dict['bill']=25
```

- added the pair 'bill':25

```
>>>print(my_dict['bill'])
25
>>> my_dict
{'bill': 25}
```

# Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
>>> my_dict = {'bill':3, 'rich':10}
>>> print(my_dict['bill'])
3
>>> my_dict['bill'] = 100
>>> print(my_dict['bill'])
100
```

# Dictionary keys can be any immutable object

```
>>>demo = {2: ['a','b','c'],(2,4): 27, 'x':
  {1:2.5,'a':3}}
>>>demo
    {'x': {'a':3, 1:2.5}, 2: ['a','b','c'],
  (2,4): 27}
>>>demo[2]
    ['a', 'b', 'c']
>>>demo[(2,4)]
    27
```

# Dictionary keys can be any immutable object

```
>>>demo ['x']
    {'a':3, 1: 2.5}
>>>demo['x'][1]
    2.5
>>>demo={2:['a','b','c'],(2,4):27,'x':{1
  :2.5,'a':3}}
>>> print(demo[2][1])
b
```


# again, common operators


Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

# Operators on Dictionary

- Let d1={‘tim’:67, ’xyz’:12}
- Copy: `>>> x=d1.copy()`

```
>>>print(x)
{'tim': 67, 'xyz': 12}
```

- Update:

```
>>> d1.update({'xyz':34})
>>> print(d1)
{'tim': 67, 'xyz': 34}
```

Length: `>>> print(len(d1))`

```
2
```

# fewer methods

Only 9 methods in total. Here are some

**`key in my_dict`**

does the key exist in the dictionary

```
>>> d1={'tim':67, 'xyz':12}
>>> 'tim' in d1
True
```

**`>>>d1.clear()` – empty the dictionary**

```
>>>d1
{}
```


# fewer methods


`my_dict.update(yourDict)` – for each key in `yourDict`, updates `my_dict` with that key/value pair

```
>>> car = {"brand": "Ford", "model": "Mustang",
"year": 1964}
>>> car.update({"color": "White"})
>>> print(car)
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color':
'White'}
```

# fewer methods

`my_dict.copy` - shallow copy

`my_dict.pop(key)` – remove key, return value

```
>>> squares = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> print(squares.pop(4))
16
>>> print(squares)
{1: 1, 2: 4, 3: 9, 5: 25}
>>>
```

# Dictionary content methods

- `my_dict.items()` – all the key/value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values

```
>>> squares = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> squares.items()
dict_items([(1, 1), (2, 4), (3, 9), (5, 25)])
>>> squares.keys()
dict_keys([1, 2, 3, 5])
>>> squares.values()
dict_values([1, 4, 9, 25])
>>>
```

# Dictionary content methods

- There return what is called a *dictionary view*.
- The order of the views correspond are dynamically updated with changes are iterable.
- How to delete dictionary??

```
>>> del squares
>>> squares
Traceback (most recent call last): File "<pyshell#47>", line 1, in <module> squares
NameError: name 'squares' is not defined
```


# Views are iterable


```
for key in my_dict:
    print(key)
```

- prints all the keys

```
for key,value in my_dict.items():
    print (key,value)
```

- prints all the key/value pairs

```
for value in my_dict.values():
    print (value)
```

- prints all the values

# Check Yourself: Matching

## Question

1. s[0]['GPA']
2. s[3]['Name']
3. s[1]['name']

Given the following Python code, match the Python Expression to it's answer

```
s= [
 { 'Name':'bob','GPA':3.4 },
 { 'Name':'sue','GPA':2.8 },
 { 'Name':'kent','GPA':4.0 }
]
```

## Answers

A. 3.4
B. KeyError
C. IndexError
D. 'sue'

# Python Data Type : Dictionary

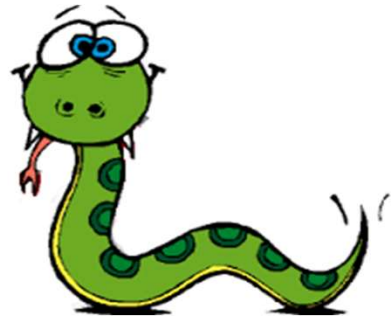
- In Python, dictionaries are defined within braces {} with each item being a pair in the form {key:value}
- Key and value can be of any type.

```
d = {1:'value','key':2}
 type(d)              #<class 'dict'>
print("d[1] = ", d[1])
print("d['key'] = ", d['key'])
```

We use key to retrieve the respective value.

But not the other way around.





# Recall Python Data Types

| | List | Tuple | Set | String | Dictionary |
|---|---|---|---|---|---|
| Symbol | [] | () | {} | “/””””/’ | {} ,Key Value pair |
| Ordered/ Unordered | Ordered | Ordered | Unordered | -- | Unordered |
| Mutable/ Immutable | Mutable | Immutable | -- | Immutable | Mutable |
| Slicing Operator | Yes | Yes | No | Yes | No |
| Example | a=[1,2.5, “data”] | a=(1,2.5, “data”) | a={1,2,3} | ‘Hello’ | {‘abc’:12, ‘xyz’:34} |

**Which of the following function convert a String to a tuple in python?**

1. repr(x)
2. eval(str)
3. tuple(s)
4. list(s)

**Suppose you have the following datasets:**

x = ['leopard', 'panther', 'pallas cat']

y = [1, 2, 3, 5, 8, 13]

z = [(1, 1), (4, 4), (9, 9), (16, 16)]

In order, what types of data are these?

1. List, array, list of arrays
2. List, list, list of tuples
3. Array, array, array of tuples
4. Ambiguous/could be multiple types

**Suppose you run the following code:**

n = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]

print n[2]

What gets printed?

1. [6, 5, 4]
2. 2
3. 7
4. [3, 2, 1]

**What Will Be The Output Of The Following Code Snippet?**

```
a=[1,2,3,4,5,6,7,8,9]
a[::2]=10,20,30,40,50,60
print(a)
```

- **Looking at the below code, write down the final values of A0, A1, ...A6.**

1. A0 = dict(zip(('a','b','c','d','e'),(1,2,3,4,5)))
2. A1 = range(10)
3. A2 = sorted([i **for** i **in** A1 **if** i **in** A0])
4. A3 = sorted([A0[s] **for** s **in** A0])
5. A4 = [i **for** i **in** A1 **if** i **in** A3]
6. A5 = {i:i*i **for** i **in** A1}
7. A6 = [[i,i*i] **for** i **in** A1]


# Summary: Tuples vs. Lists


- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and methods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

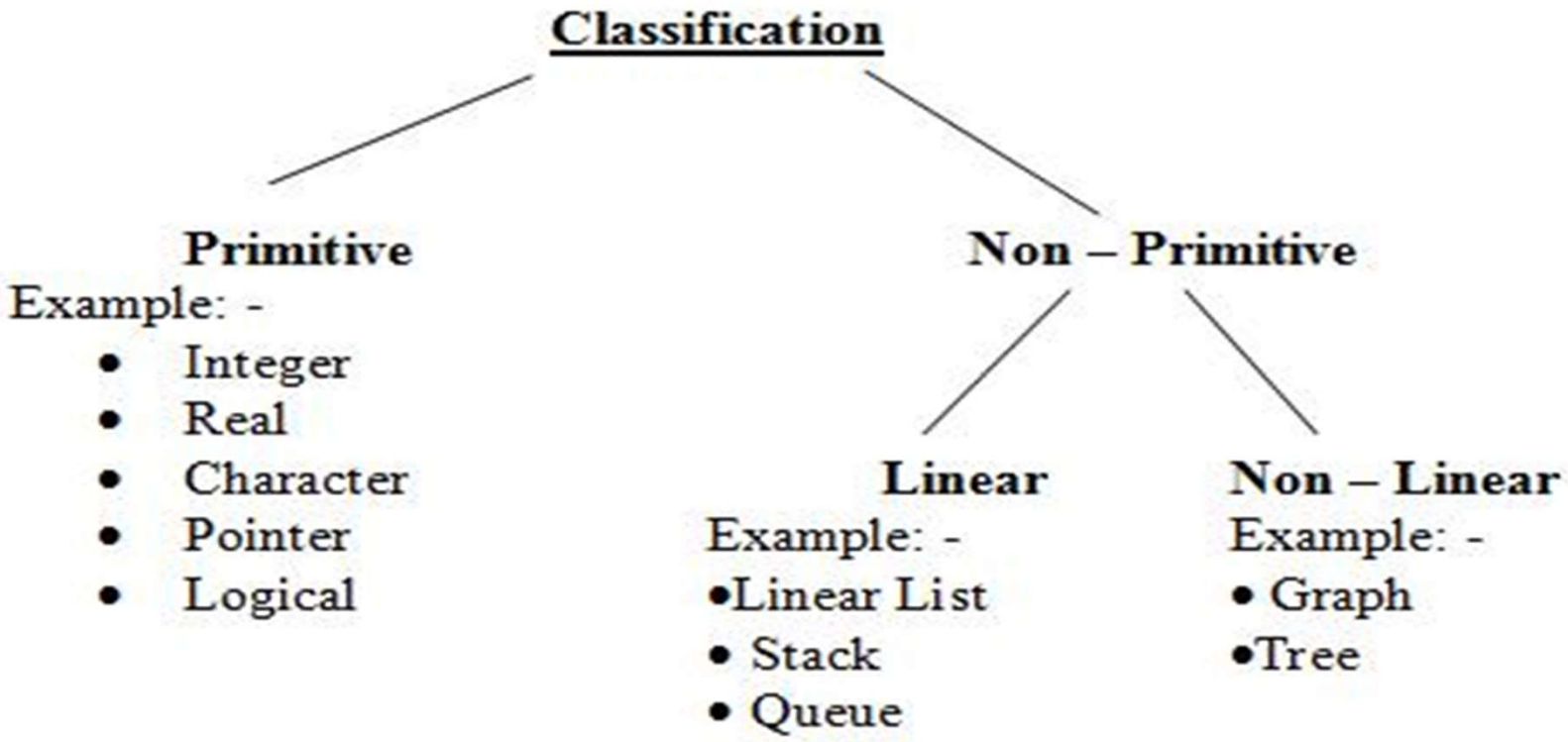
```
li = list(tu)
tu = tuple(li)
```


# Data Structure : Definition

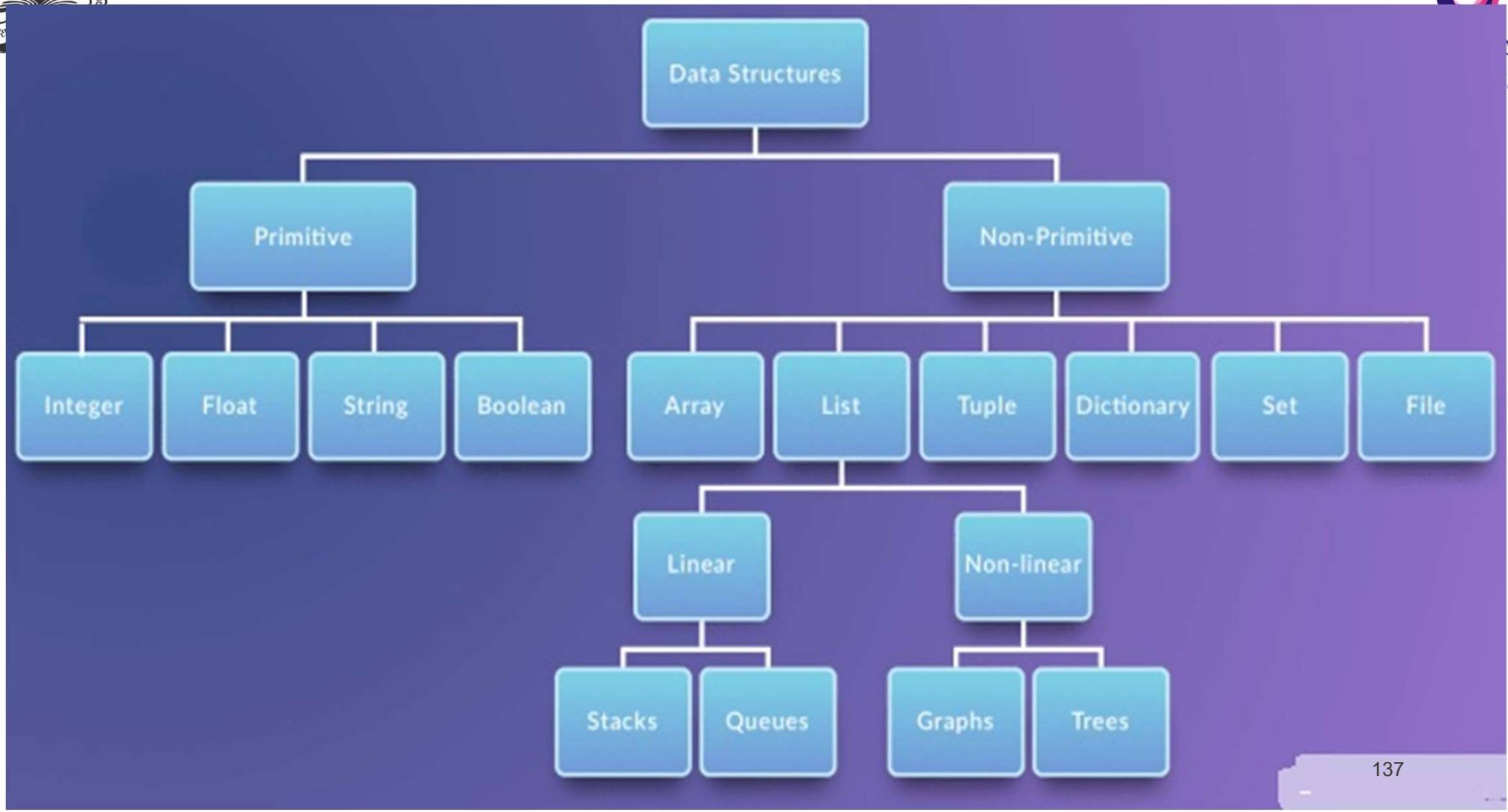

- Data structures are a way of organizing and storing data so that they can be accessed and worked with efficiently.

- Data structures define the relationship between the data, and the operations that can be performed on the data.

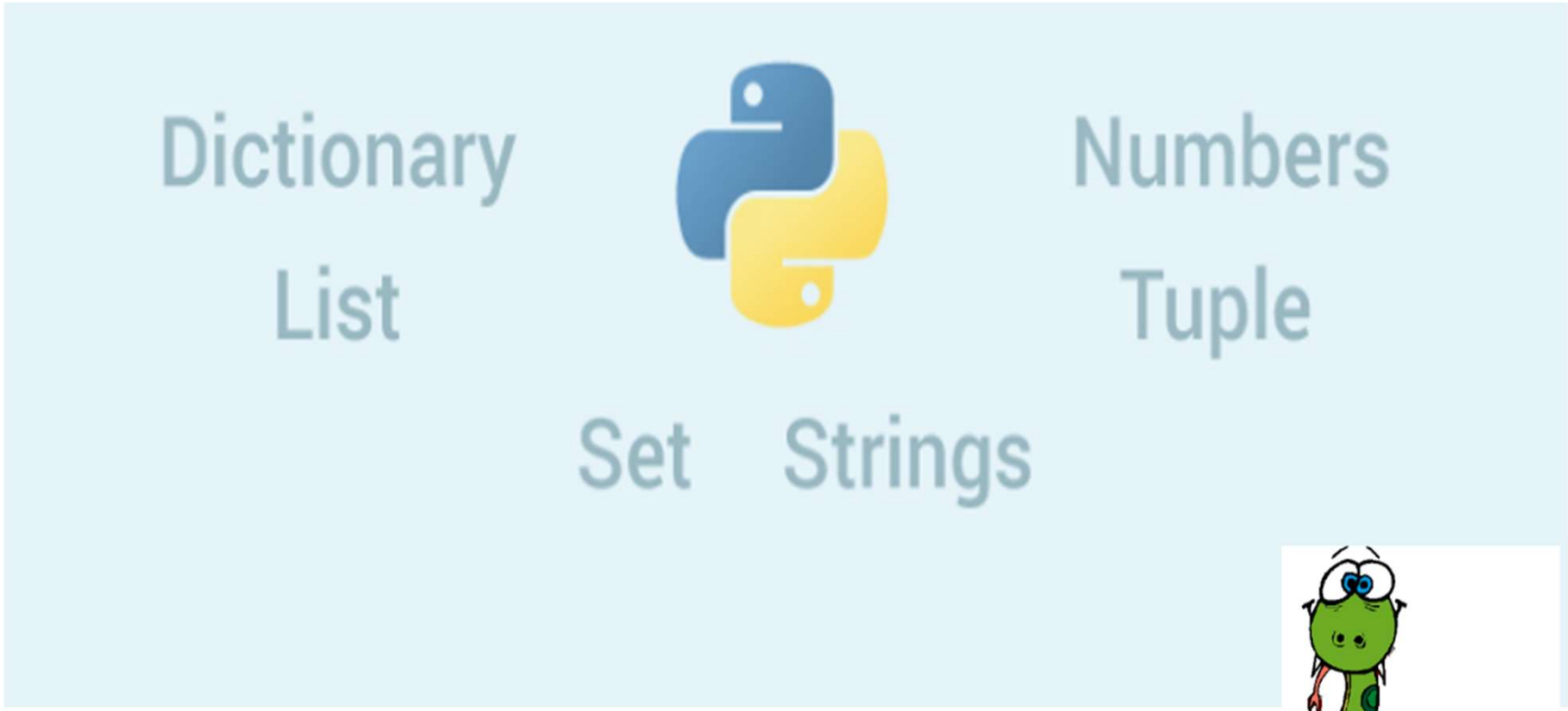
# Standard Classification of Data Structure



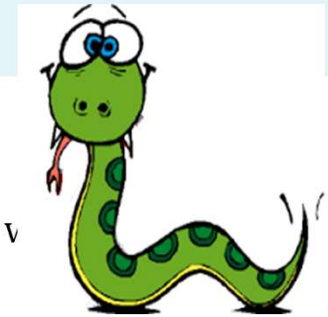
# Python Classification of Data Structure



# Non-Primitive Data Structure in Python



# Python List:

# Python List :

- List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type.
- Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].
- >>> a = [1, 2.2, 'python']

# Python Data Type : List

- We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python

```
a = [5,10,15,20,25,30,35,40]

print("a[2] = ", a[2])

print("a[0:3] = ", a[0:3])

print("a[5:] = ", a[5:])
```


# Python List:


- Lists are mutable, meaning, value of elements of a list can be altered.

a = [1,2,3]

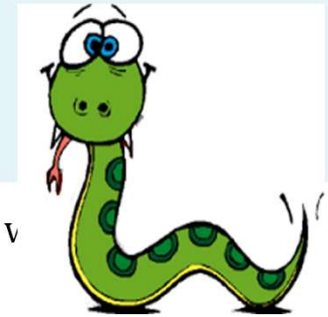
a[2] = 4

a [1, 2, 4]

# Python Tuple :

( [1, 2], (3, 4), “Tuple” )

Python tuples



# Python Data Type : Tuple

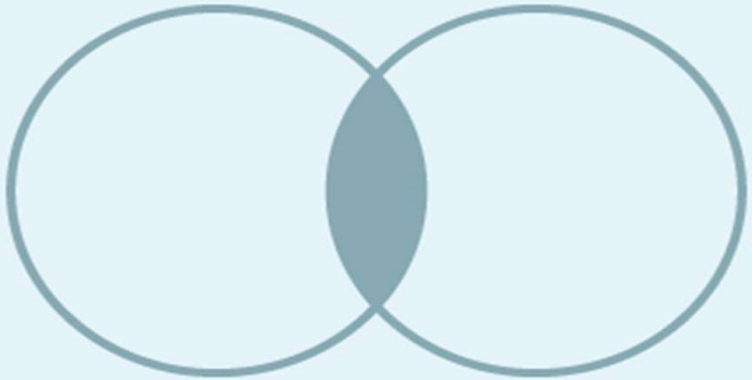
- Tuple is an ordered sequence of items same as list.The only difference is that tuples are immutable. Tuples once created cannot be modified.
- Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.
- It is defined within parentheses () where items are separated by commas.
- >>> t = (5,'program', 1+3j)

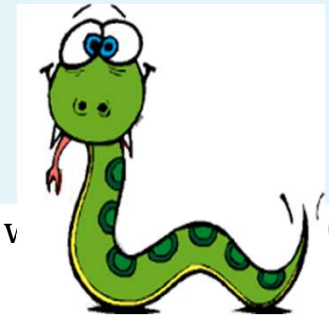
# Python Tuple:

- We can use the slicing operator [] to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

print("t[1] = ", t[1])

print("t[0:3] = ", t[0:3])

t[0]=10
```

# Python Set :



# Python Set:

- Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.
- a = {5,2,3,1,4}
- print("a = ", a)
- print(type(a))

# Python Data Type : Set

- We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}

print(a)
```

# Python Data Type : Set

- Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

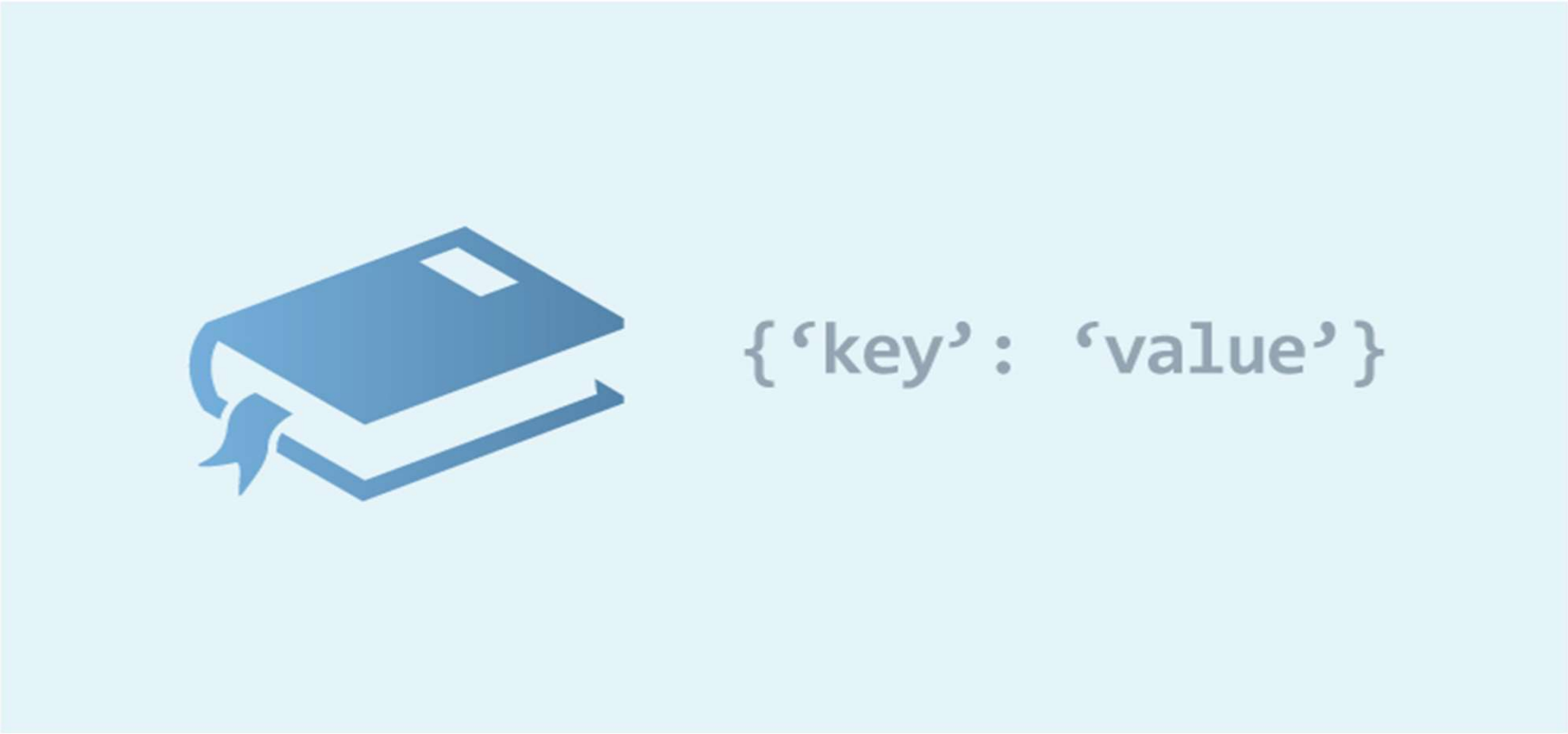
```
a = {1,2,3}

print(a[1])
```

- Traceback (most recent call last): File "<string>", line 301, in runcode File "<interactive input>", line 1, in <module> TypeError: 'set' object does not support indexing

# Python Set :

| Operation | Equivalent | Result |
|---|---|---|
| len(s) | | number of elements in set *s* (cardinality) |
| x in s | | test *x* for membership in *s* |
| x not in s | | test *x* for non-membership in *s* |
| s.issubset(t) | s <= t | test whether every element in *s* is in *t* |
| s.issuperset(t) | s >= t | test whether every element in *t* is in *s* |
| s.union(t) | s \| t | new set with elements from both *s* and *t* |
| s.intersection(t) | s & t | new set with elements common to *s* and *t* |
| s.difference(t) | s - t | new set with elements in *s* but not in *t* |
| s.symmetric_difference(t) | s ^ t | new set with elements in either *s* or *t* but not both |
| s.copy() | | new set with a shallow copy of *s* |

# Python Dictionary :



# Python Dictionary :

- Dictionary is an unordered collection of key-value pairs.
- It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

# Python Data Type : Dictionary

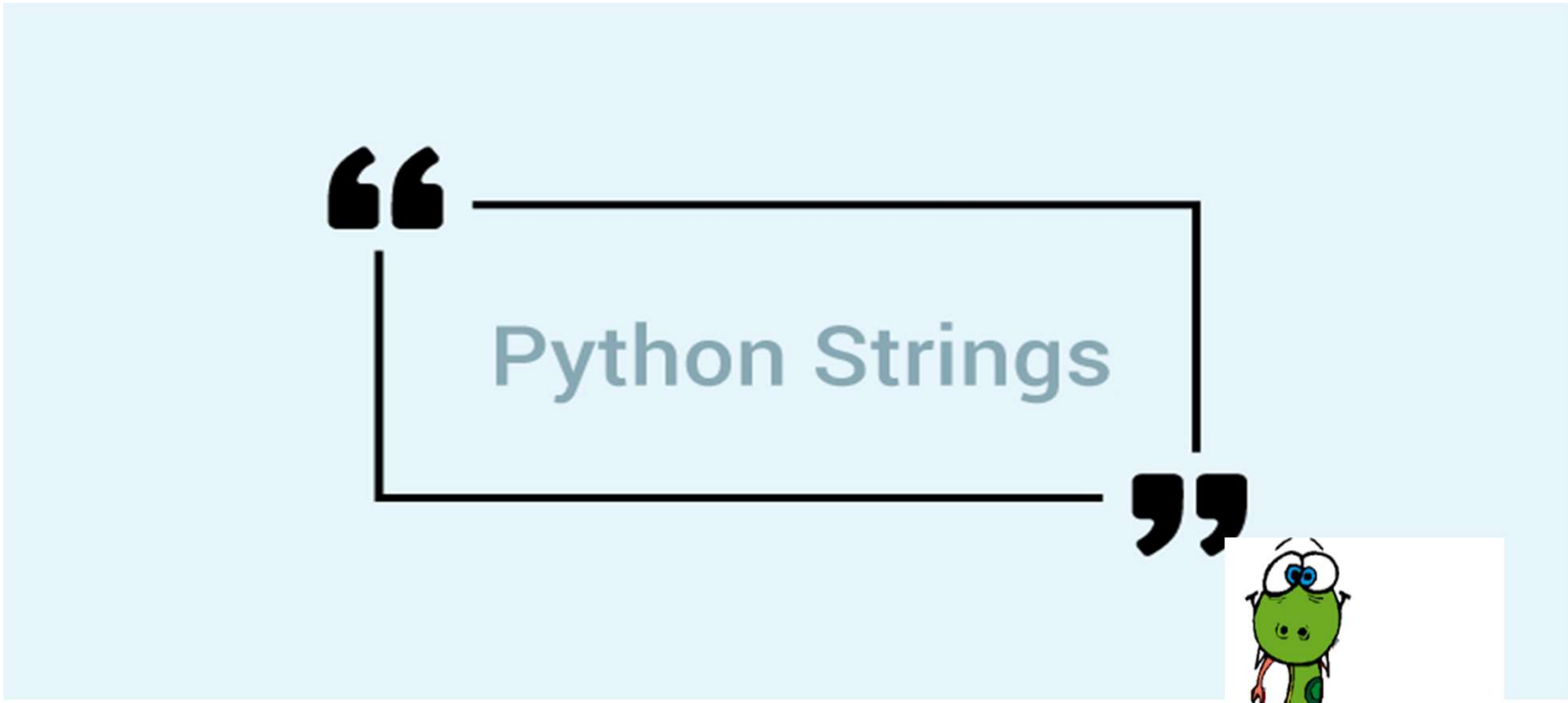
- In Python, dictionaries are defined within braces {} with each item being a pair in the form key : value.
- Key and value can be of any type.

```
d = {1:'value','key':2}
type(d)          #<class 'dict'>
print("d[1] = ", d[1])
print("d['key'] = ", d['key'])
```

We use key to retrieve the respective value.

But not the other way around.

# Python String :



# Python String :

- String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

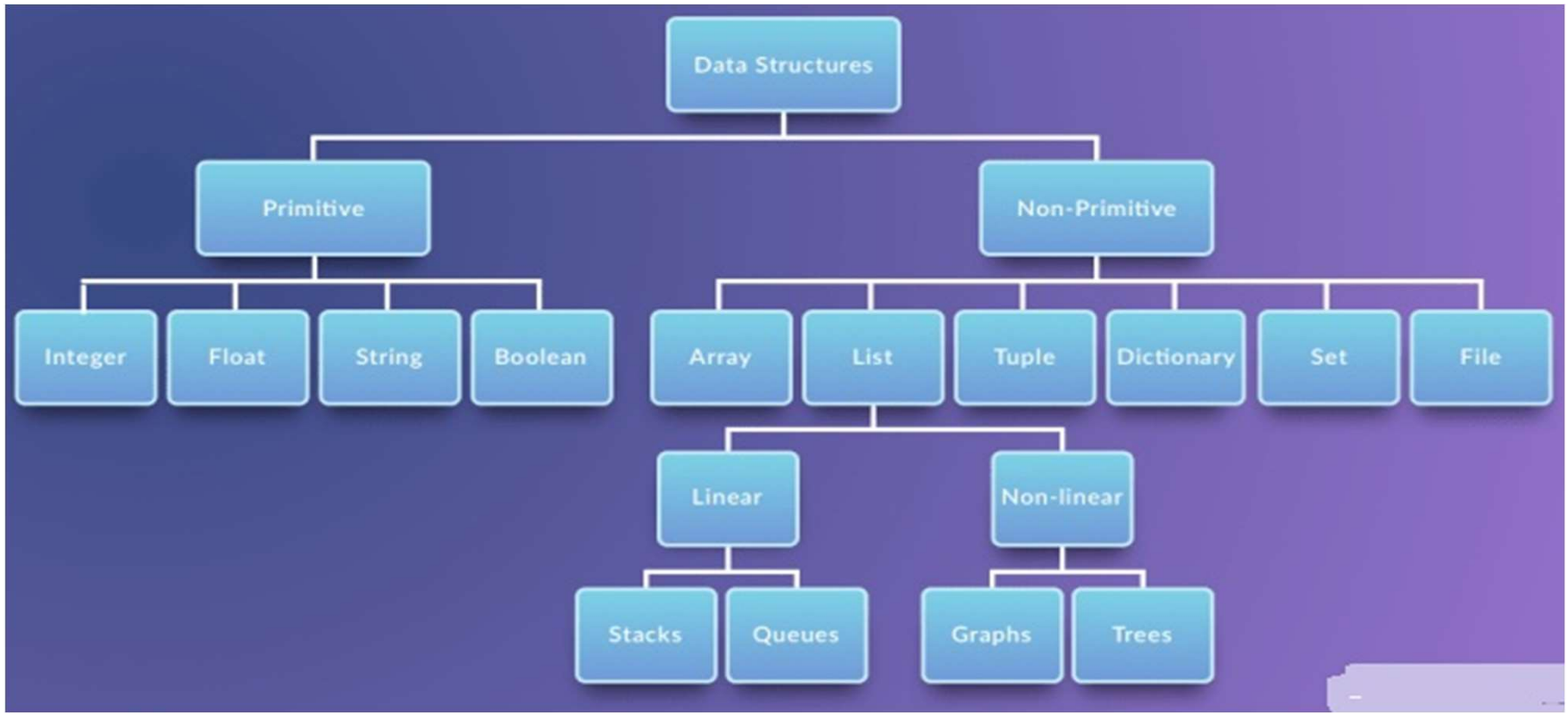
```
s = “This is a string”
```

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable

# Recall Python Non-Primitive Data Structure

| | List | Tuple | Set | String | Dictionary |
|---|---|---|---|---|---|
| Symbol | [] | () | {} | | {} ,Key Value pair |
| Ordered/ Unordered | Ordered | Ordered | Unordered | -- | Unordered |
| Mutable/ Immutable | Mutable | Immutable | -- | Immutable | -- |
| Slicing Operator | Yes | Yes | No | Yes | No |
| Example | a=[1,2.5, “data”] | a=(1,2.5, “data”) | a={1,2,3} | | |

# Python Classification of Data Structure



# More about List

| Sr.No | Function | Description |
|---|---|---|
| 1 | list.**append**(*x*) | Add an item to the end of the list; equivalent to a[len(a):] = [x]. |
| 2 | list.**extend**(*L*) | Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L. |
| 3 | list.**insert**(*i*, *x*) | Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x) |
| 4 | list.**remove**(*x*) | Remove the first item from the list whose value is *x*. It is an error if there is no such item. |
| 5 | list.**pop**([*i*]) | Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. |

# More about List

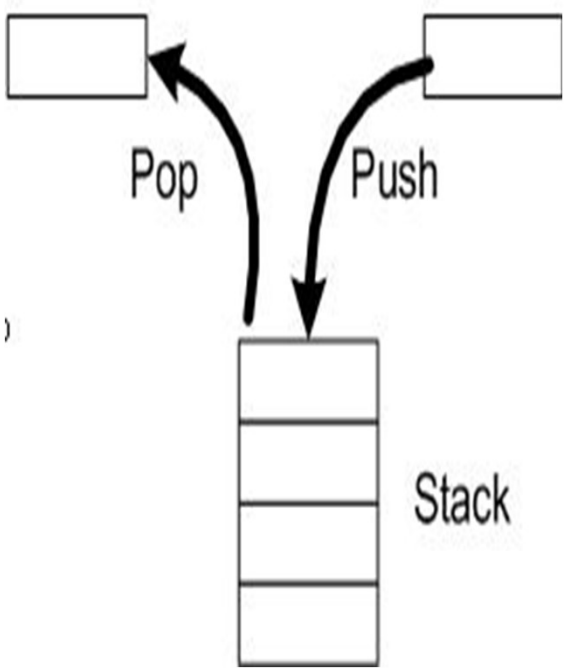
| Sr.No | Function | Description |
|---|---|---|
| 1 | list.**index**(*x*) | Return the index in the list of the first item whose value is *x*. It is an error if there is no such item |
| 2 | list.**count**(*x*) | Return the number of times *x* appears in the list. |
| 3 | list.**sort**() | Sort the items of the list, in place |
| 4 | list.**reverse**() | Reverse the elements of the list, in place. |



# Implementation of Data Structures

# STACK- Definition /Principle/Concept

- Stack is last in first out Structure (LIFO) Structure .
- Stack is linear data structure.
- It used to store the data in such a way that element inserted into the stack will be removed at last.
- A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top.*



# Real Life Example of Stack principle

**This concept is used for evaluating expressions and syntax parsing, scheduling algortihms/routines, etc.**

# Stack Implementation Using List

- The list methods make it very easy to use a list as a stack,.
- To add an item to the top of the stack, use **append()**.
- To retrieve an item from the top of the stack, use **pop()**

without an explicit index.

# Stack Implementation Using List

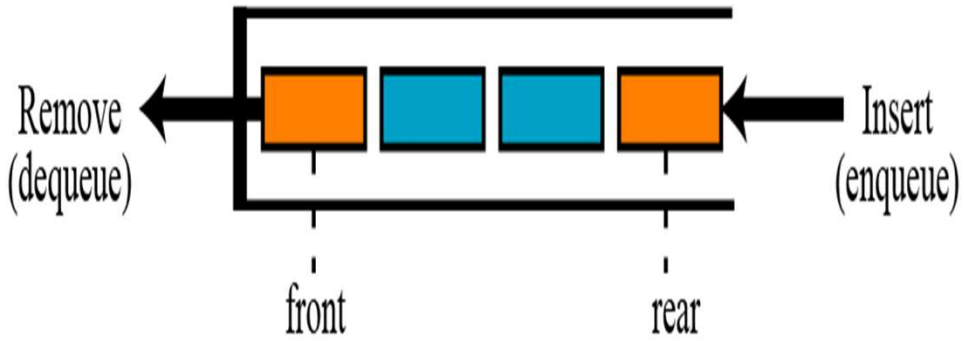
```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

# Queue- Definition /Principle/Concept

- Queue is first in and first out Structure (FIFO) Structure .
- Queue is linear data structure.
- It used to store the data in such a way that element inserted into the queue at one end will be removed from other end .
- Data can only be inserted at one end, called the rear, and deleted from the other end, called the front.

# Queue Examples

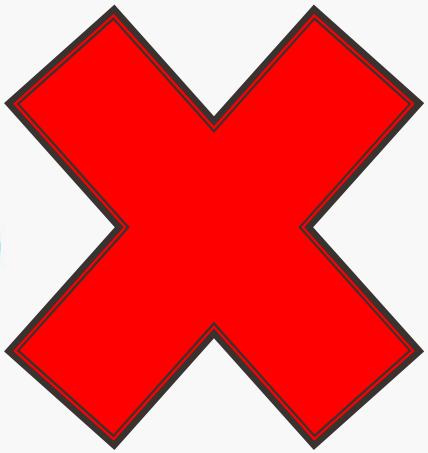

A queue of people




A computer queue

# Queue Implementation in Python : List

```
# How to use Python's list as a FIFO queue:

q = []

q.append('eat')
q.append('sleep')
q.append('code')

>>> q
['eat', 'sleep', 'code']

# Careful: This is slow!
>>> q.pop(0)
'eat'
```



It's possible to use a regular `list` as a queue but this is **not ideal from a performance perspective**. Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring *O(n)* time.

Therefore I would **not recommend** you use a `list` as a makeshift queue in Python (unless you're dealing with a small number of elements only).

# Queue Implementation in Python

- **collections** — Container datatypes

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

| | |
|---|---|
| `namedtuple()` | factory function for creating tuple subclasses with named fields |
| `deque` | list-like container with fast appends and pops on either end |
| `Counter` | dict subclass for counting hashable objects |
| `OrderedDict` | dict subclass that remembers the order entries were added |
| `defaultdict` | dict subclass that calls a factory function to supply missing values |
| `UserDict` | wrapper around dictionary objects for easier dict subclassing |
| `UserList` | wrapper around list objects for easier list subclassing |
| `UserString` | wrapper around string objects for easier string subclassing |

# Queue Implementation in Python

*class* collections.**deque**([*iterable*[, *maxlen*]])

- Returns a new deque object initialized left-to-right (using **append()**) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

- Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

- Though **list** objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

- If *maxlen* is not specified or is *None*, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length.

# Queue Implementation in Python

Deque objects support the following methods:

**append**(*x*)

Add *x* to the right side of the deque.

**appendleft**(*x*)

Add *x* to the left side of the deque.

**clear**()

Remove all elements from the deque leaving it with length 0.

**extend**(*iterable*)

Extend the right side of the deque by appending elements from the iterable argument.

**extendleft**(*iterable*)

Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

**pop**()

Remove and return an element from the right side of the deque. If no elements are present, raises an `IndexError`.

**popleft**()

Remove and return an element from the left side of the deque. If no elements are present, raises an `IndexError`.

**remove**(*value*)

Removed the first occurrence of *value*. If not found, raises a `ValueError`.

**rotate**(*n*)

Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left. Rotating one step to the right is equivalent to: `d.appendleft(d.pop())`.

Deque objects also provide one read-only attribute:

**maxlen**

Maximum size of a deque or *None* if unbounded.

# Queue Implementation in Python : Using deque

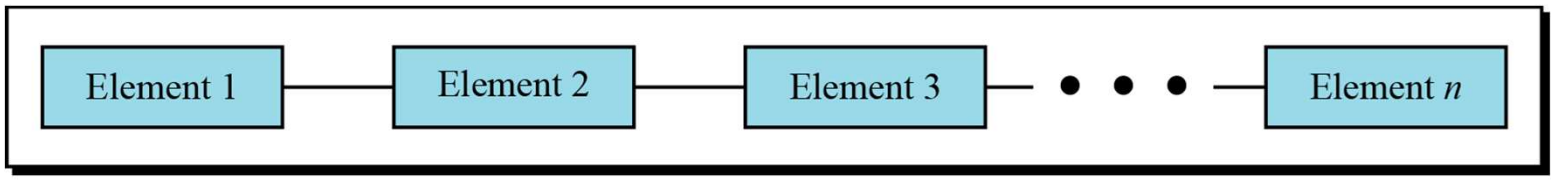
```
# How to use collections.deque as a FIFO queue:

from collections import deque
q = deque()

q.append('eat')
q.append('sleep')
q.append('code')

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'

>>> q.popleft()
IndexError: "pop from an empty deque"
```

# Queue Implementation in Python : Using deque

```
>>> print(q)
deque(['sdf', 'erty'])
>>> q.appendleft('sneha')
>>> print (q)
deque(['sneha', 'sdf', 'erty'])
>>> q.extend(['ram','avneesh'])
>>> print(q)
deque(['sneha', 'sdf', 'erty', 'ram', 'avneesh'])
>>> q.rotate(1)
>>> print(q)
deque(['avneesh', 'sneha', 'sdf', 'erty', 'ram'])
>>> q.rotate(2)
>>> print(q)
deque(['erty', 'ram', 'avneesh', 'sneha', 'sdf'])
```

# General Linked List

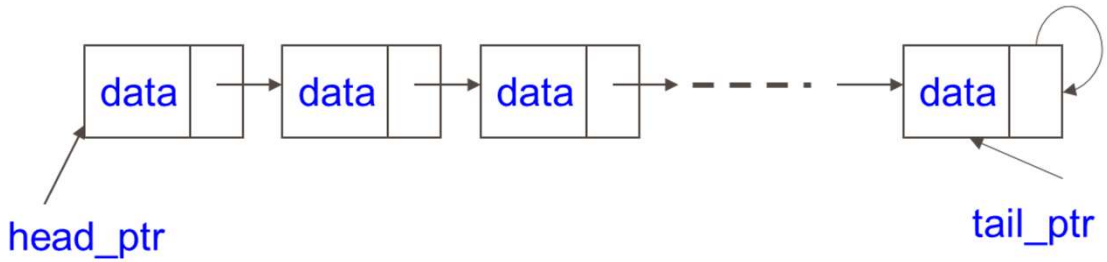
Stacks and queues defined in the two previous sections are **restricted linear lists**. A **general linear list** is a list in which operations, such as insertion and deletion, can be done anywhere in the list—at the beginning, in the middle or at the end. Figure shows a general linear list.




General linear list

# Definition of Linked Lists

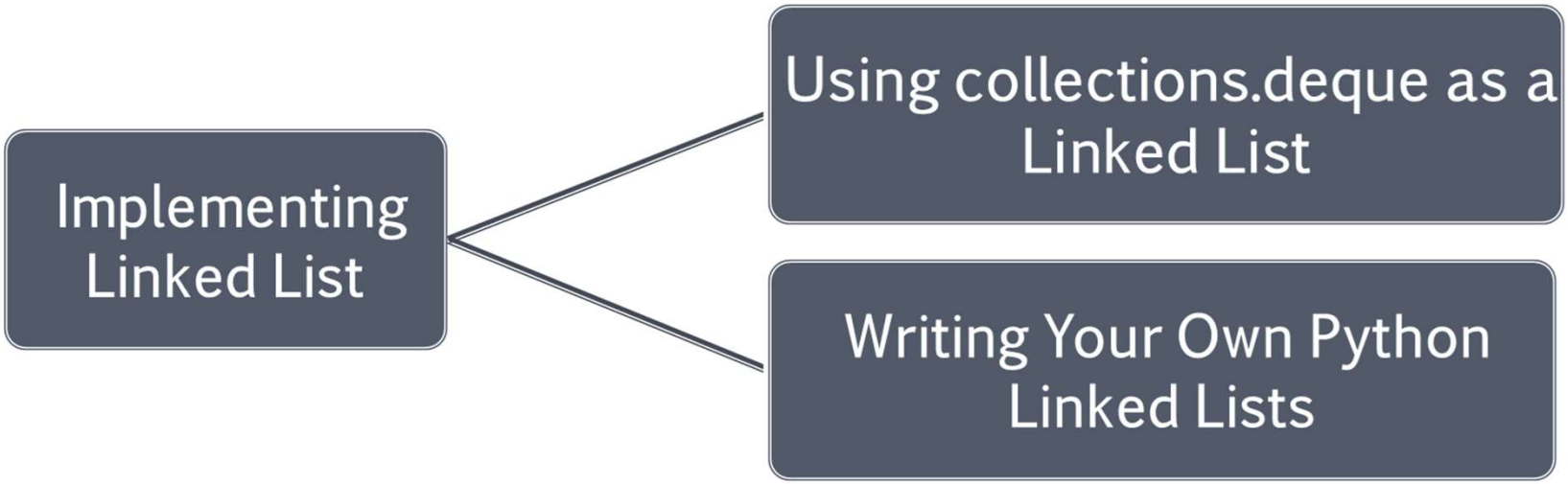
- A linked list is a sequence of elements (objects) where every element is linked to the next.
- Graphically:




LINKED LISTS
IN PYTHON

# Does Python have a built-in or “native” linked list data structure?

- As of Python 3.6 ,doesn’t provide a dedicated linked list. Theirs is nothing like Java’s linked list built into Python or into the Python standard library.
- Python does however includes the collections.deque class which provides a double-ended queue and is implemented as a doubly linked list internally.




Python does however include the `collections.deque` class which provides a double-ended queue and is implemented as a doubly-linked list internally.

# Option1 : **Using collections.deque as a Linked List**

**Step 1: Import required module and create object**

```
>>> import collections
>>> lst = collections.deque()
```

**Step 2: Insert data into linked list**

**Case 1 :( Insert first and Insertlast)**

```
# Inserting elements at the front  or back takes O(1) time:
>>> lst.append('B')        #insertlast
>>> lst.append('C')
>>> lst.appendleft('A')    #insertfirst
>>> lst
deque(['A', 'B', 'C'])    #output
```

# Option1: **Using collections.deque as a Linked List**

### Case 2: Insert data at arbitrary Position

# However, inserting elements at arbitrary indexes takes O(n) time:

```
>>> lst.insert(2, 'X')
>>> lst
deque(['A', 'B', 'X', 'C'])    #output
```

### Step 3: Delete an element

### Case 1 :( Delete first and Delete last)

# Removing elements at the front or back takes O(1) time:

```
>>> lst.pop()   >>> 'C‘            #Output
>>> lst.popleft() >>> 'A‘ >>> lst  deque(['B', 'X'])   #output
```

# Option1 : **Using collections.deque as a Linked List**

## Case 2: Delete an element at arbitrary

# Removing elements at arbitrary indexes or by key takes O(n) time again:

```
>>> lst.remove('B')
```

# Searching for elements takes

# O(n) time:

```
>>> lst.index('X')
1
```

Option 2: **Writing Your Own Python Linked Lists**

```
class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None

class SLinkedList:
    def __init__(self):
        self.headval = None

list1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3
```

```
class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None

class SLinkedList:
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

# Link first Node to second node
list.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3

list.listprint()
```

```
class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None

class SLinkedList:
    def __init__(self):
        self.headval = None

# Print the linked list
    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval
    def AtBegining(self,newdata):
        NewNode = Node(newdata)

# Update the new nodes next val to existing node
        NewNode.nextval = self.headval
        self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtBegining("Sun")

list.listprint()
```

# The Node

```
class Node(object):

    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node

    def get_data(self):
        return self.data

    def get_next(self):
        return self.next_node

    def set_next(self, new_next):
        self.next_node = new_next
```

- node initializes with a single datum and its pointer is set to None by default
- few convenience methods: one that returns the stored data,
- another that returns the next node (the node to which the object node points), and
- finally a method to reset the pointer to a new node.

# The Linked List

- **Head of the List**
  - When the list is first initialized it has no nodes, so the head is set to None.

```
class LinkedList(object):
    def __init__(self, head=None):
        self.head = head
```

- **Insert**
  - initializes a new node with the given data, and adds it to the list.

```
def insert(self, data):
    new_node = Node(data)
    new_node.set_next(self.head)
    self.head = new_node
```

# The Linked List

- **Search**
  - Search is actually very similar to size, but instead of traversing the whole list of nodes it checks at each stop to see whether the current node has the requested data and if so, returns the node holding that data. If the method goes through the entire list but still hasn't found the data, it raises a value error and notifies the user that the data is not in the list..

```
def search(self, data):
    current = self.head
    found = False
    while current and found is False:
        if current.get_data() == data:
            found = True
        else:
            current = current.get_next()
    if current is None:
        raise ValueError("Data not in list")
    return current
```

3)

# Option 2: **Writing Your Own Python Linked Lists**

```
class ListNode:
    """
    A node in a singly-linked list.
    """
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

    def __repr__(self):
        return repr(self.data)
```

# Option 2: **Writing Your Own Python Linked Lists**

```
class SinglyLinkedList:
    def __init__(self):
        """
        Create a new singly-linked list.
        Takes O(1) time.
        """
        self.head = None

    def __repr__(self):
        """
        Return a string representation of the list.
        Takes O(n) time.
        """
        nodes = []
        curr = self.head
        while curr:
            nodes.append(repr(curr))
            curr = curr.next
        return '[' + ', '.join(nodes) + ']'
```

# Option 2: **Writing Your Own Python Linked Lists**

```
    def prepend(self, data):
        """
        Insert a new element at the beginning of the list.
        Takes O(1) time.
        """
        self.head = ListNode(data=data, next=self.head)

    def append(self, data):
        """
        Insert a new element at the end of the list.
        Takes O(n) time.
        """
        if not self.head:
            self.head = ListNode(data=data)
            return
        curr = self.head
        while curr.next:
            curr = curr.next
        curr.next = ListNode(data=data)
```

# Option 2: **Writing Your Own Python Linked Lists**

```
def append(self, data):
    """
    Insert a new element at the end of the list.
    Takes O(n) time.
    """
    if not self.head:
        self.head = ListNode(data=data)
        return
    curr = self.head
    while curr.next:
        curr = curr.next
    curr.next = ListNode(data=data)
```

# Option 2: **Writing Your Own Python Linked Lists**

```
def remove(self, key):
    """
    Remove the first occurrence of `key` in the list.
    Takes O(n) time.
    """
    # Find the element and keep a
    # reference to the element preceding it
    curr = self.head
    prev = None
    while curr and curr.data != key:
        prev = curr
        curr = curr.next
    # Unlink it from the list
    if prev is None:
        self.head = curr.next
    elif curr:
        prev.next = curr.next
        curr.next = None
```

# Option 2: **Writing Your Own Python Linked Lists**

```
def reverse(self):
    """
    Reverse the list in-place.
    Takes O(n) time.
    """
    curr = self.head
    prev_node = None
    next_node = None
    while curr:
        next_node = curr.next
        curr.next = prev_node
        prev_node = curr
        curr = next_node
    self.head = prev_node
```

# Option 2: **Writing Your Own Python Linked Lists**

```
>>> lst = SinglyLinkedList()
>>> lst
[]

>>> lst.prepend(23)
>>> lst.prepend('a')
>>> lst.prepend(42)
>>> lst.prepend('X')
>>> lst.append('the')
>>> lst.append('end')

>>> lst
['X', 42, 'a', 23, 'the', 'end']

>>> lst.find('X')
'X'
>>> lst.find('y')
None

>>> lst.reverse()
>>> lst
['end', 'the', 23, 'a', 42, 'X']

>>> lst.remove(42)
```


# Key Performance Characteristics


- **Inserting and removing elements at the front and back of a deque is a fast *O(1)* operation.** However, inserting or removing in the middle takes *O(n)*time because we don't have access to the previous-element or next-element linked list pointers. That's abstracted away by the deque interface.

- **Storage is *O(n)*—but not every element gets its own list node.** The dequeclass uses blocks that hold multiple elements at once and then these blocks are linked together as a doubly-linked list. As of CPython 3.6 the block size is 64 elements. This incurs some space overhead but retains the general performance characteristics given a large enough number of elements

- **In-place reversal**: In Python 3.2+ the elements in a deque instance can be reversed in-place with the reverse() method. This takes *O(n)* time and no extra space


## Suggested Practical in Syllabus


- Menu Driven Program for data structure using built in function for linked list, stack and queue.
- Stack application – Infix to Postfix ,Postfix Evaluation ,Parenthesis checking
- Queue application
- Implementation aspects of data structure in C,Java and Python as a Case Study.