SCIENTIFIC COMPUTING WITH PYTHON

Data Structure & File Handling

Course Coordinator: Dr. R. Mariyal Jebasty
Assistant Professor,
Department of Physics
Wavoo Wajeeha College of Arts & Science
Kayalpatnam.

Course Instructors

1. Mrs. Pushpa, Assistant Professor in Physics, Wavoo Wajeeha Women's College of Arts & Science, Kayalpatnam.
2. Dr. S. Usharani , Assistant Professor in Physics, Wavoo Wajeeha Women's College of Arts & Science, Kayalpatnam.

List Comprehensions

- Lists are ordered sequences that can hold a variety of object types.

- They use [] brackets and commas to separate objects in the list.
    - **[1,2,3,4,5]**

- Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.

- Height in Cm
- Weight in Kgs
- Age in Years

```
Ramesh_height = 150

Suresh_height = 145

Sudesh_height = 165
```

```
Ramesh_weight = 56

Suresh_weight = 60

Sudesh_weight = 65
```

```
Ramesh_age = 23

Suresh_age = 46

Sudesh_age = 58
```

- Height in Cm
- Weight in Kgs
- Age in Years

.

.

- Some info

Ramesh_height = 150

Suresh_height = 145

Sudesh_height = 165

Ramesh_weight = 56

Suresh_weight = 60

Sudesh_weight = 65

How Many Variables?

Ramesh_age = 23

Suresh_age = 46

Sudesh_age = 58

- Height in Cm
- Weight in Kgs
- Age in Years

.

.

- Some info

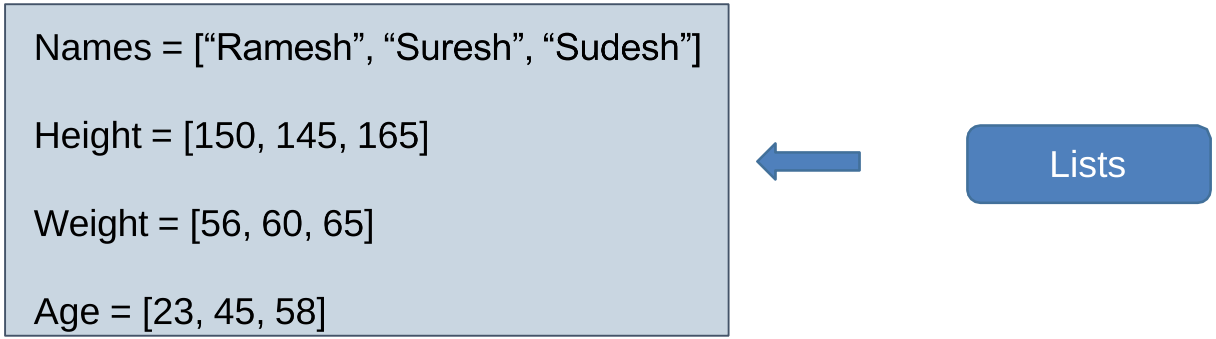
```
Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]
```

- Height in Cm
- Weight in Kgs
- Age in Years

.

.

- Some info



```
Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]
```

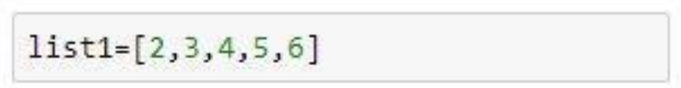
Lists

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.

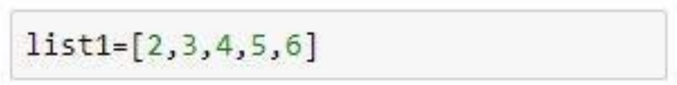


```
list1=[2,3,4,5,6]
```

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.

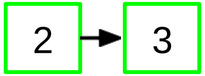
2

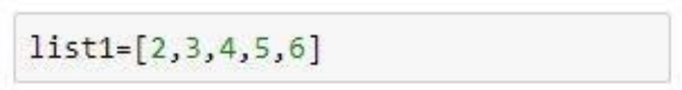


```
list1=[2,3,4,5,6]
```

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.

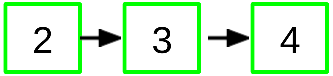


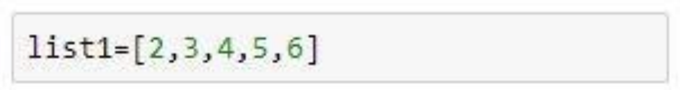


```
list1=[2,3,4,5,6]
```

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.

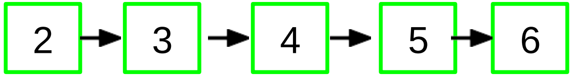




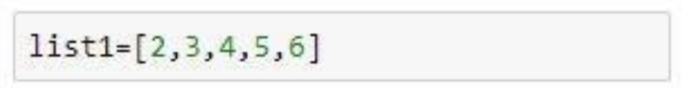
```
list1=[2,3,4,5,6]
```

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.



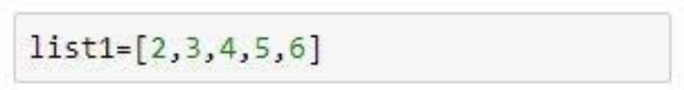
2 → 3 → 4 → 5 → 6



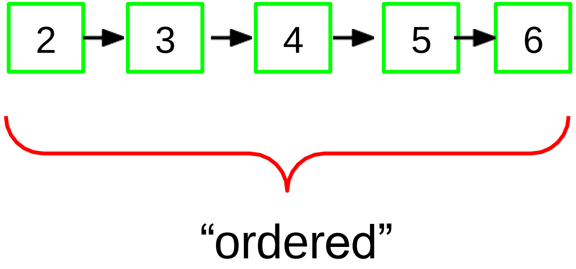
```
list1=[2,3,4,5,6]
```

What is a List?

A list is an **ordered** data structure with elements separated by comma and enclosed within square brackets.



```
list1=[2,3,4,5,6]
```



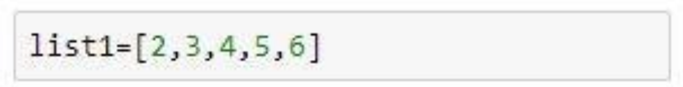
2 → 3 → 4 → 5 → 6

“ordered”

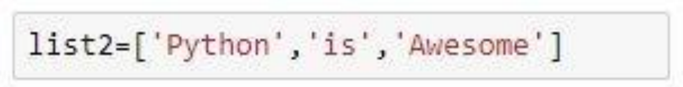
What is a List?

- A list is an ordered data structure with elements separated by comma and enclosed within square brackets.

- Some examples of List -



```
list1=[2,3,4,5,6]
```



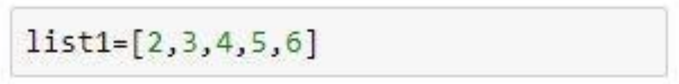
```
list2=['Python','is','Awesome']
```

- Single Data type

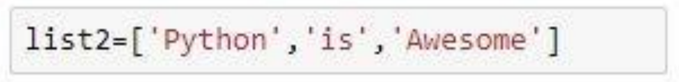
What is a List?

- A list is an ordered data structure with elements separated by comma and enclosed within square brackets.

- Some examples of List -

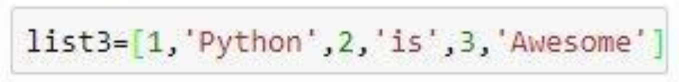


```
list1=[2,3,4,5,6]
```



```
list2=['Python','is','Awesome']
```

- Single Data type

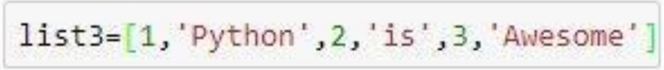


```
list3=[1,'Python',2,'is',3,'Awesome']
```

Mixed Data type

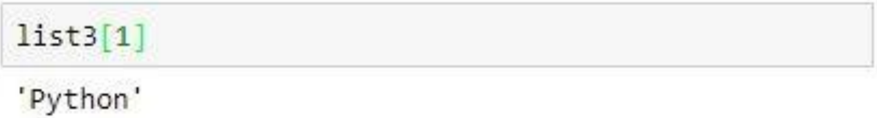
Extracting values from a List

0 1 2 3 4 5 Index



```
list3=[1,'Python',2,'is',3,'Awesome']
```

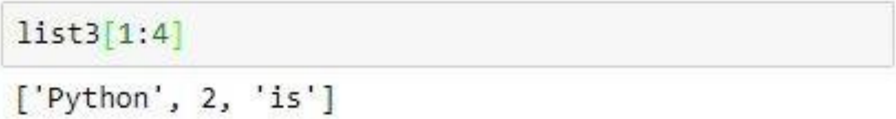
To extract a single element



```
list3[1]
```

```
'Python'
```

To extract a sequence of elements

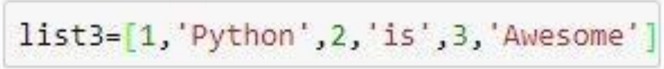


```
list3[1:4]
```

```
['Python', 2, 'is']
```

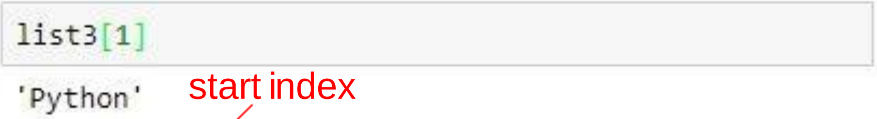
Extracting values from a List

0 1 2 3 4 5 ← Index



```
list3=[1,'Python',2,'is',3,'Awesome']
```

To extract a single element →

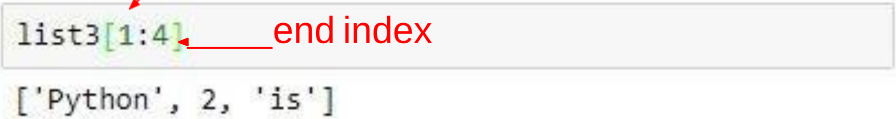


```
list3[1]
```

```
'Python'
```

start index

To extract a sequence of elements →

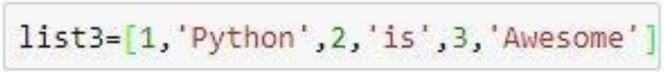


```
list3[1:4]
```

end index

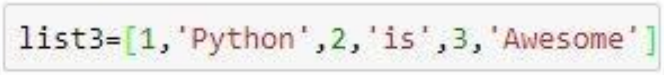
```
['Python', 2, 'is']
```

Adding elements to an existing List



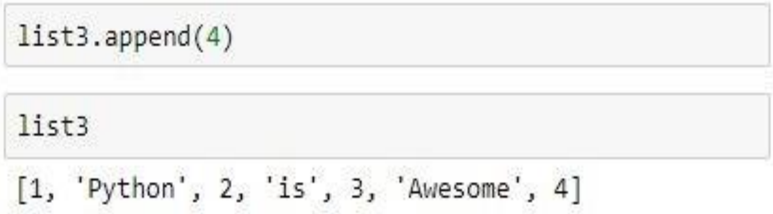
```
list3=[1,'Python',2,'is',3,'Awesome']
```

Adding elements to an existing List



```
list3=[1,'Python',2,'is',3,'Awesome']
```

Adding a single element

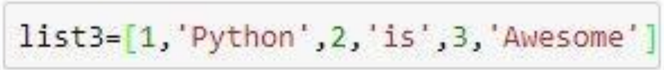


```
list3.append(4)
```

```
list3
```

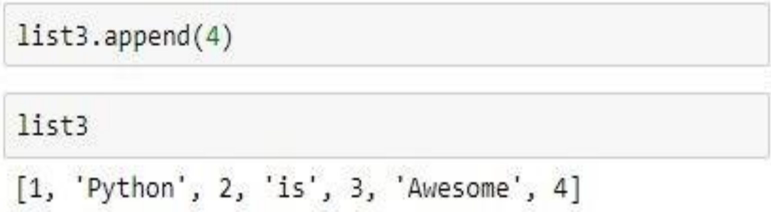
```
[1, 'Python', 2, 'is', 3, 'Awesome', 4]
```

Adding elements to an existing List



```
list3=[1,'Python',2,'is',3,'Awesome']
```

Adding a single element

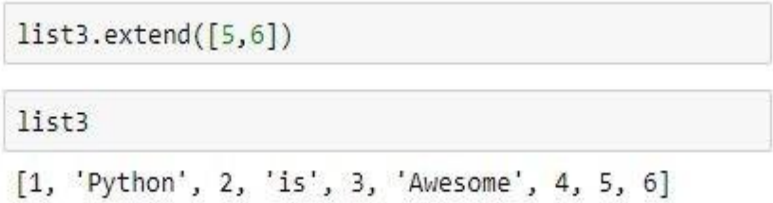


```
list3.append(4)
```

```
list3
```

```
[1, 'Python', 2, 'is', 3, 'Awesome', 4]
```

Adding multiple elements

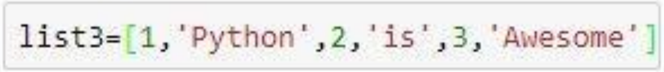


```
list3.extend([5,6])
```

```
list3
```

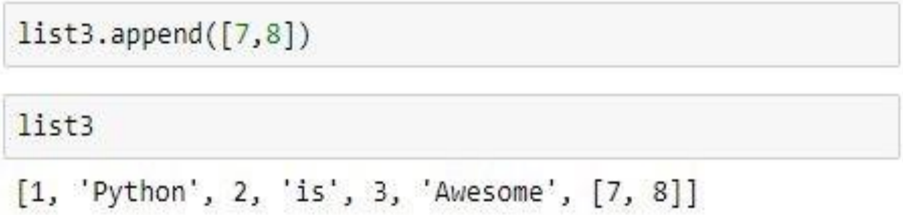
```
[1, 'Python', 2, 'is', 3, 'Awesome', 4, 5, 6]
```

Adding elements to an existing List



```
list3=[1,'Python',2,'is',3,'Awesome']
```

Adding list to a list

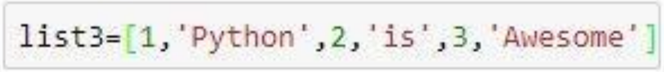


```
list3.append([7,8])
```

```
list3
```

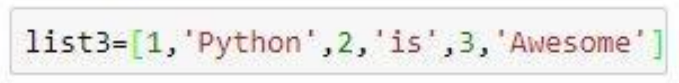
```
[1, 'Python', 2, 'is', 3, 'Awesome', [7, 8]]
```

Deleting elements of a List



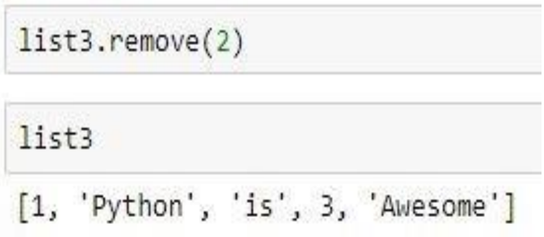
```
list3=[1,'Python',2,'is',3,'Awesome']
```

Deleting elements of a List



```
list3=[1,'Python',2,'is',3,'Awesome']
```

Deleting an element by value

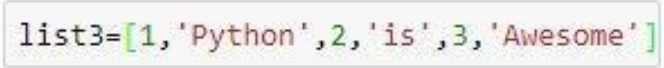


```
list3.remove(2)
```

```
list3
```

```
[1, 'Python', 'is', 3, 'Awesome']
```

Deleting elements of a List



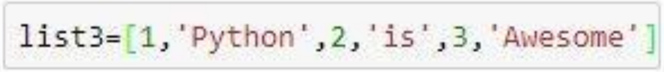
```
list3=[1,'Python',2,'is',3,'Awesome']
```

Indexing

This returns the whole list.

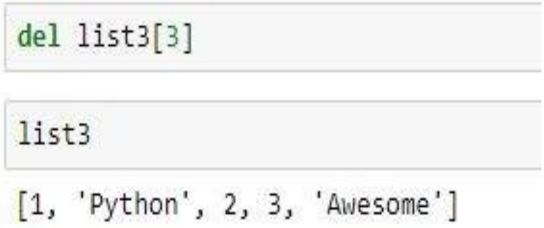
Negative indices- The indices we mention can be negative as well. A negative index means traversal from the end of the list.

Deleting elements of a List



```
list3=[1,'Python',2,'is',3,'Awesome']
```

Deleting an element by index →



```
del list3[3]
```

```
list3
```

```
[1, 'Python', 2, 3, 'Awesome']
```

List Data Structure: Summary

Store

Represent

Manipulate

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Manipulate

Represent

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Represent

Manipulate

- Extract values
- Add values (append, extend, insert)
- Remove values (del, remove, pop)
- Looping over values

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Represent

- Ordered/Sequential

Manipulate

- Extract values
- Add values (append, extend, insert)
- Remove values (del, remove, pop)
- Looping over values

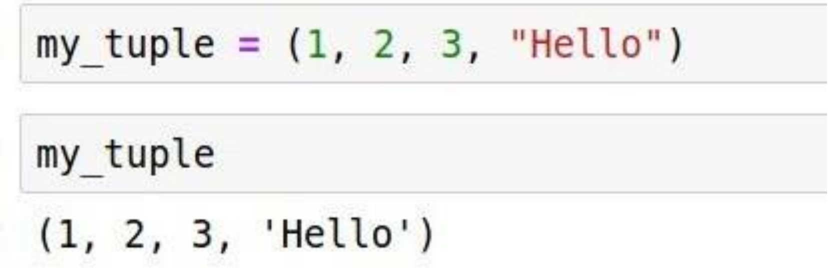
Tuple

Tuples are very similar to lists. However they have one key difference - **immutability.**

Once an element is inside a tuple, it can not be reassigned.

Tuples use parenthesis: **(1,2,3)**

Tuple in Python (Data Structure)



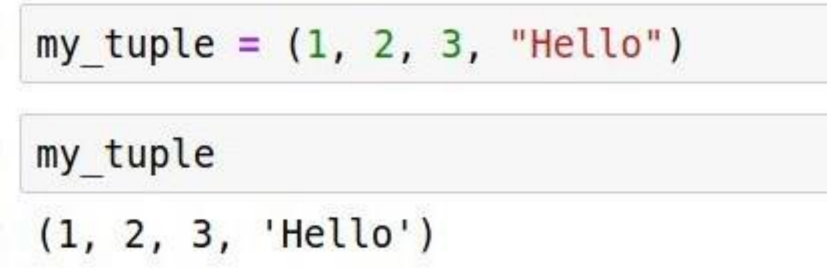
```
my_tuple = (1, 2, 3, "Hello")
```

```
my_tuple
```

```
(1, 2, 3, 'Hello')
```

Tuple in Python (Data Structure)

- Ordered collection of elements



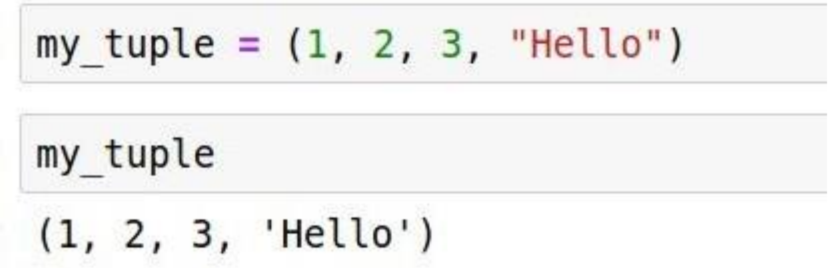
```
my_tuple = (1, 2, 3, "Hello")
```

```
my_tuple
```

```
(1, 2, 3, 'Hello')
```

Tuple in Python (Data Structure)

- Ordered collection of elements
- Immutable



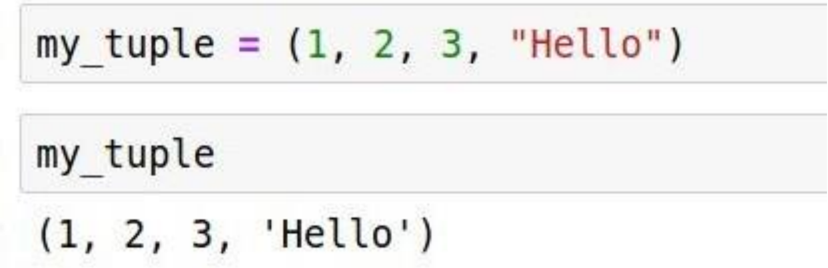
```
my_tuple = (1, 2, 3, "Hello")
```

```
my_tuple
```

```
(1, 2, 3, 'Hello')
```

Tuple in Python (Data Structure)

- Ordered collection of elements
- Immutable
- Uses circular brackets in syntax



```
my_tuple = (1, 2, 3, "Hello")
```

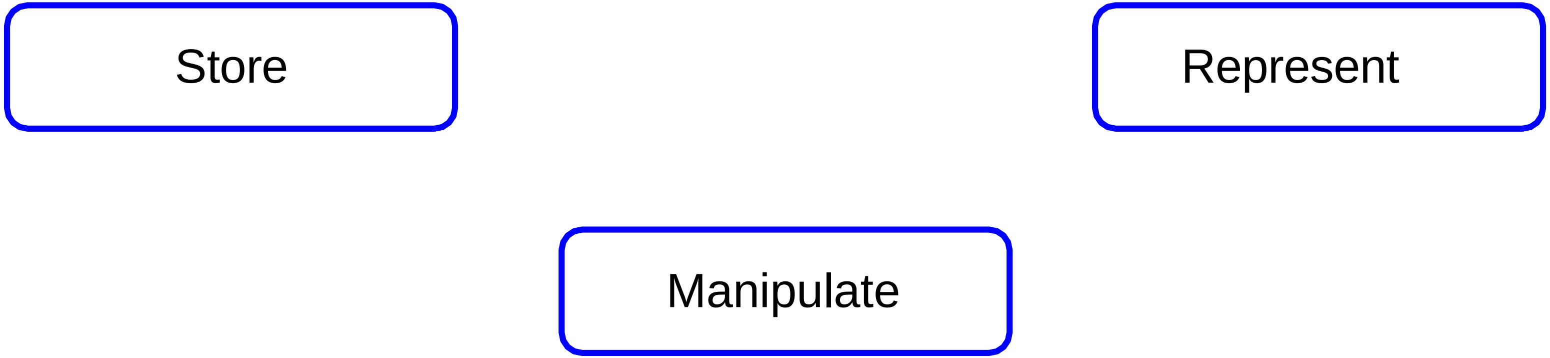
```
my_tuple
```

```
(1, 2, 3, 'Hello')
```

Benefits of using Tuple

- Faster than lists
- Provide security over updation
- Unlike lists, can be used as key for dictionaries

List Data Structure: Summary



Store

Represent

Manipulate

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Manipulate

Represent

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Represent

Manipulate

- Extract values
- Add values (append, extend, insert)
- Remove values (del, remove, pop)
- Looping over values

List Data Structure: Summary

Store

- Multiple values
- Multiple data types

Represent

- Ordered/Sequential

Manipulate

- Extract values
- Add values (append, extend, insert)
- Remove values (del, remove, pop)
- Looping over values

Tuple Data Structure: Summary

Store

- Multiple values
- Multiple data types

Manipulate

- Extract values
- Looping over values

Represent

- Ordered/Sequential

Dictionary

- Dictionaries are unordered mappings for storing objects.

- Previously we saw how lists store objects in an ordered sequence, dictionaries use a key-value pairing instead.

- This key-value pair allows users to quickly grab objects without needing to know an index location.

- Dictionaries use curly braces and colons to signify the keys and their associated values.

{'key1':'value1','key2':'value2'}

- So when to choose a list and when to choose a dictionary?

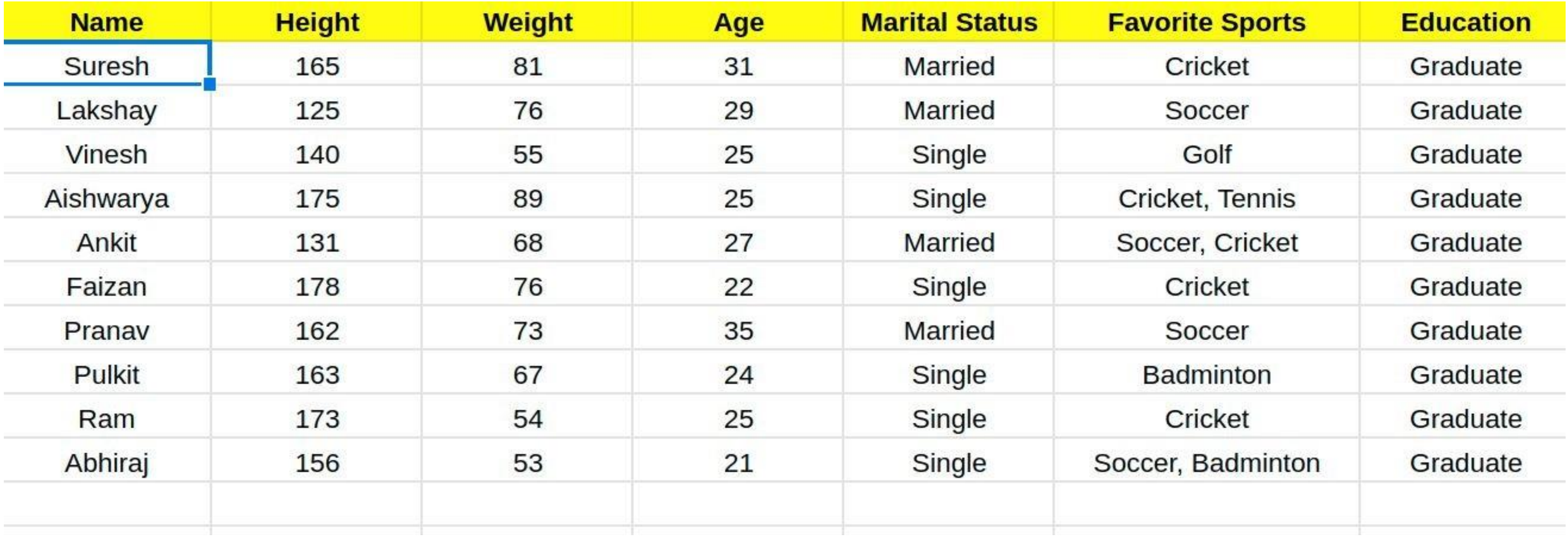
- **Dictionaries:** Objects retrieved by key name. Cannot be indexed or sliced

- **Lists:** Objects retrieved by location.

Ordered Sequences can be indexed or sliced.

Multiple columns

Employee General Info:



Name	Height	Weight	Age	Marital Status	Favorite Sports	Education
Suresh	165	81	31	Married	Cricket	Graduate
Lakshay	125	76	29	Married	Soccer	Graduate
Vinesh	140	55	25	Single	Golf	Graduate
Aishwarya	175	89	25	Single	Cricket, Tennis	Graduate
Ankit	131	68	27	Married	Soccer, Cricket	Graduate
Faizan	178	76	22	Single	Cricket	Graduate
Pranav	162	73	35	Married	Soccer	Graduate
Pulkit	163	67	24	Single	Badminton	Graduate
Ram	173	54	25	Single	Cricket	Graduate
Abhiraj	156	53	21	Single	Soccer, Badminton	Graduate

- Height in Cm
- Weight in Kgs
- Age in Years

```
Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]
```

Lists

- Height in Cm
- Weight in Kgs
- Age in Years

.

.

- Some info

Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]

Lists

How Many Lists?

Lists

```
Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]
```

Lists

```
Names = [“Ramesh”, “Suresh”, “Sudesh”]

Height = [150, 145, 165]

Weight = [56, 60, 65]

Age = [23, 45, 58]
```

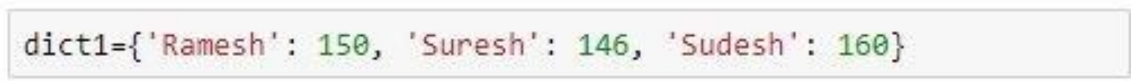
Dictionary

```
employee_info = {
          “names” : [“Ramesh”, “Suresh”, “Sudesh”],

          “height” :   [150, 145, 165],

          “weight” :  [56, 60, 65],

          “age” :       [23, 45, 58]
}
```

What is a Dictionary?

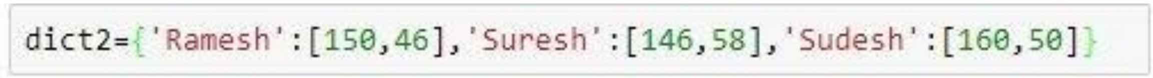
- A dictionary is an **unordered** data structure.
- Elements are separated by a comma and stored as key : value pair.
- A dictionary is enclosed within curly brackets.

Some examples of Dictionary -



```
dict1={'Ramesh': 150, 'Suresh': 146, 'Sudesh': 160}
```

key : value, where value is a number

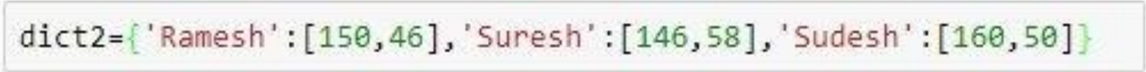


```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

key : value, where value is a List

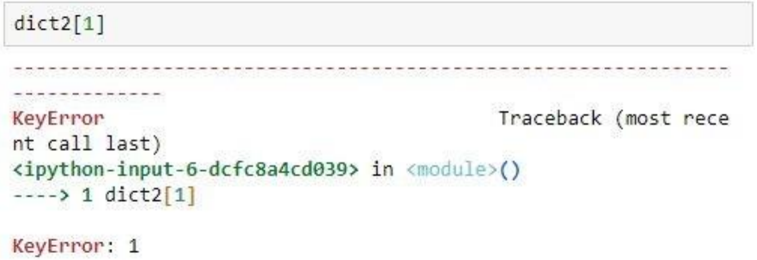
Accessing elements of a Dictionary

Elements are accessed by **keys** rather than index.



```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

Dictionary accessed by index →

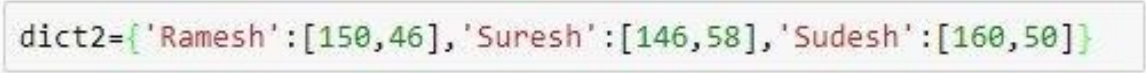


```
dict2[1]
```

```
---------------------------------------------------------------
-------------
KeyError                                  Traceback (most rece
nt call last)
<ipython-input-6-dcfc8a4cd039> in <module>()
----> 1 dict2[1]

KeyError: 1
```

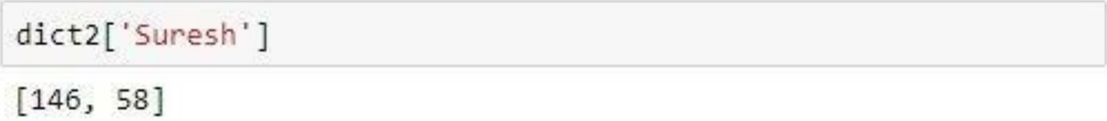
Accessing elements of a Dictionary

Elements are accessed by keys rather than index.



```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

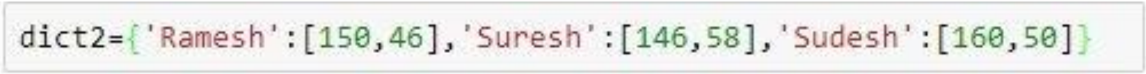
Dictionary accessed by key →



```
dict2['Suresh']
```

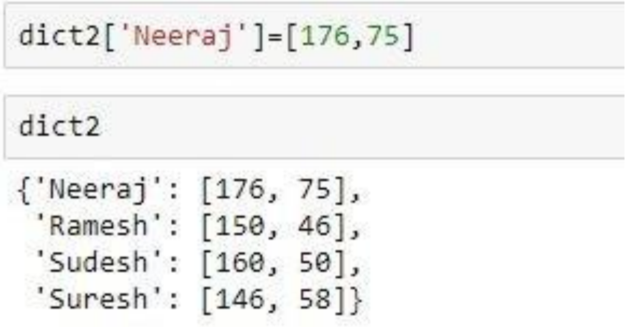
```
[146, 58]
```

Adding elements to a Dictionary



```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

Adding a single element →

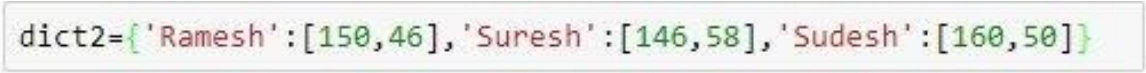


```
dict2['Neeraj']=[176,75]
```

```
dict2
```

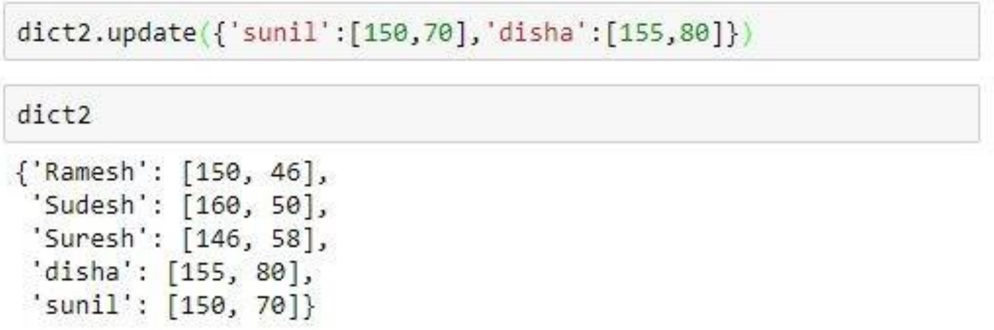
```
{'Neeraj': [176, 75],
 'Ramesh': [150, 46],
 'Sudesh': [160, 50],
 'Suresh': [146, 58]}
```

Adding elements to a Dictionary



```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

Adding multiple elements at once →

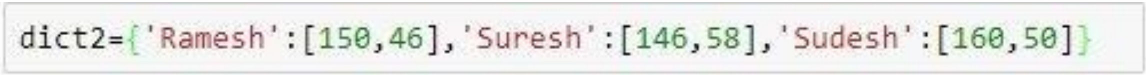


```
dict2.update({'sunil':[150,70],'disha':[155,80]})
```

```
dict2
```

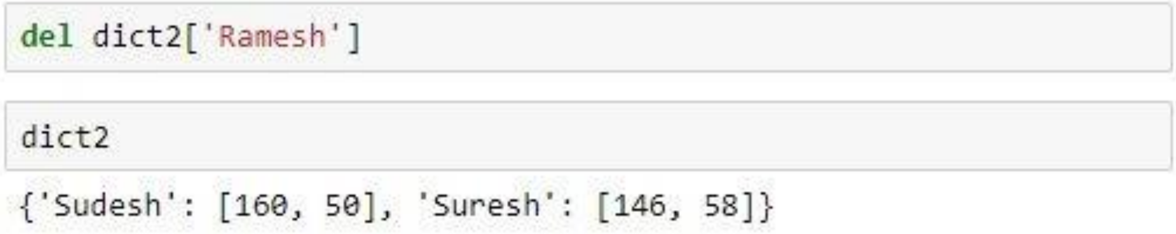
```
{'Ramesh': [150, 46],
 'Sudesh': [160, 50],
 'Suresh': [146, 58],
 'disha': [155, 80],
 'sunil': [150, 70]}
```

Deleting element of a Dictionary



```
dict2={'Ramesh':[150,46],'Suresh':[146,58],'Sudesh':[160,50]}
```

Deleting an element →

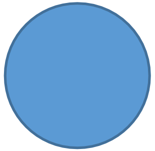


```
del dict2['Ramesh']
```

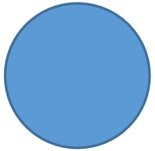
```
dict2
```

```
{'Sudesh': [160, 50], 'Suresh': [146, 58]}
```

Functions

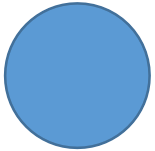


Radius = 1
cm



Radius = 1 cm

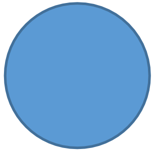
Area?



Radius = 1 cm

Pseudo-Code:

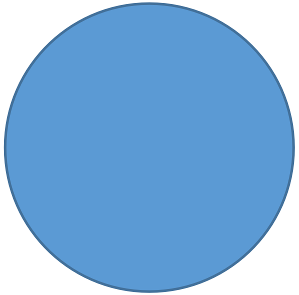
```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```



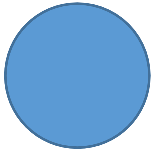
Radius = 1 cm

Pseudo-Code:

```
Area of Circle:
    Task 1. Take radius 1
    Task 2. Calculate 1*1
    Task 3. Multiply 3.14 by 1*1
```



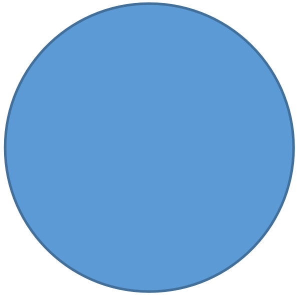
Radius = 3 cm



Radius = 1 cm

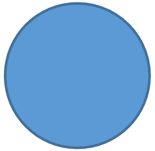
Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```



Radius = 3 cm

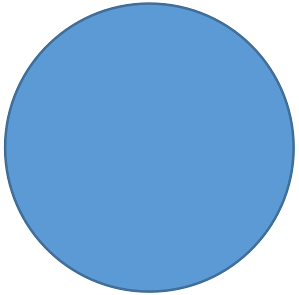
Area?



Radius = 1 cm

Pseudo-Code:

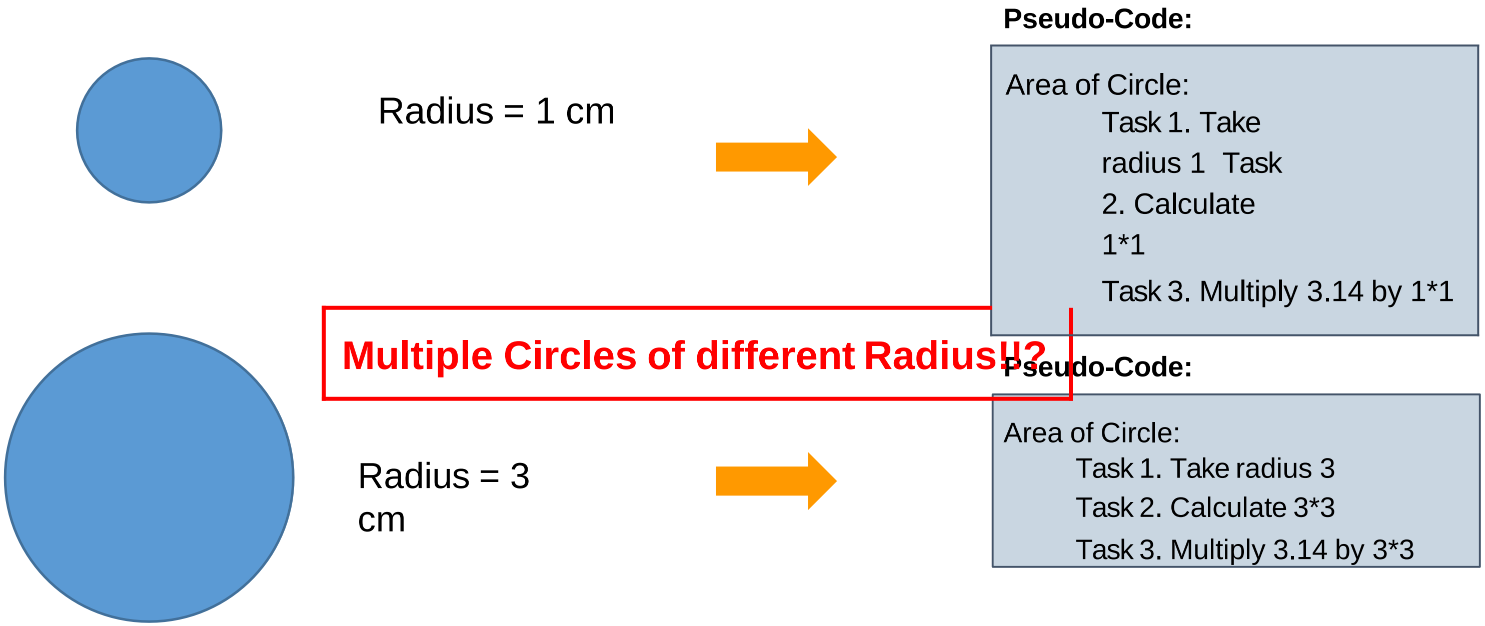
```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```



Radius = 3 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 3
        Task 2. Calculate 3*3
        Task 3. Multiply 3.14 by 3*3
```



Radius = 1 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```

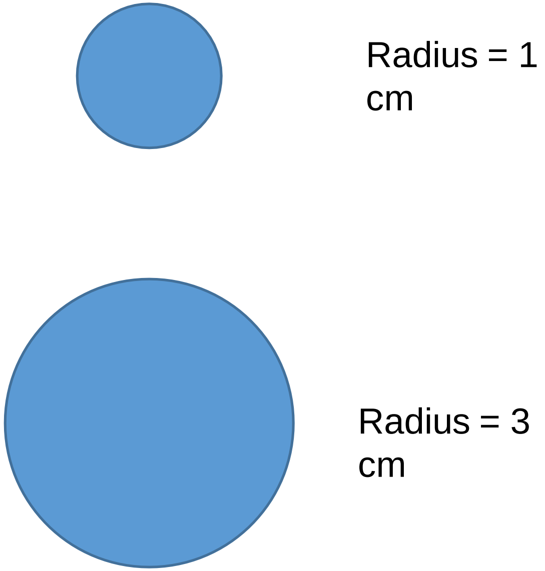
Multiple Circles of different Radius!!?

Radius = 3 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 3
        Task 2. Calculate 3*3
        Task 3. Multiply 3.14 by 3*3
```

Solution #1: Looping

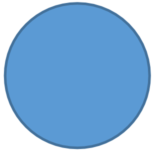


Pseudo-Code:

list of radius

```
for r in radius_list:

        Area of Circle:
                Task 1. Take radius r
                Task 2. Calculate r*r
                Task 3. Multiply 3.14 by r*r
                Task 4. You get the area
```

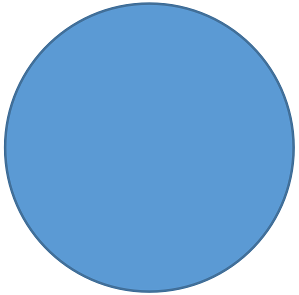
Solution #2: Function



Radius = 1 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```

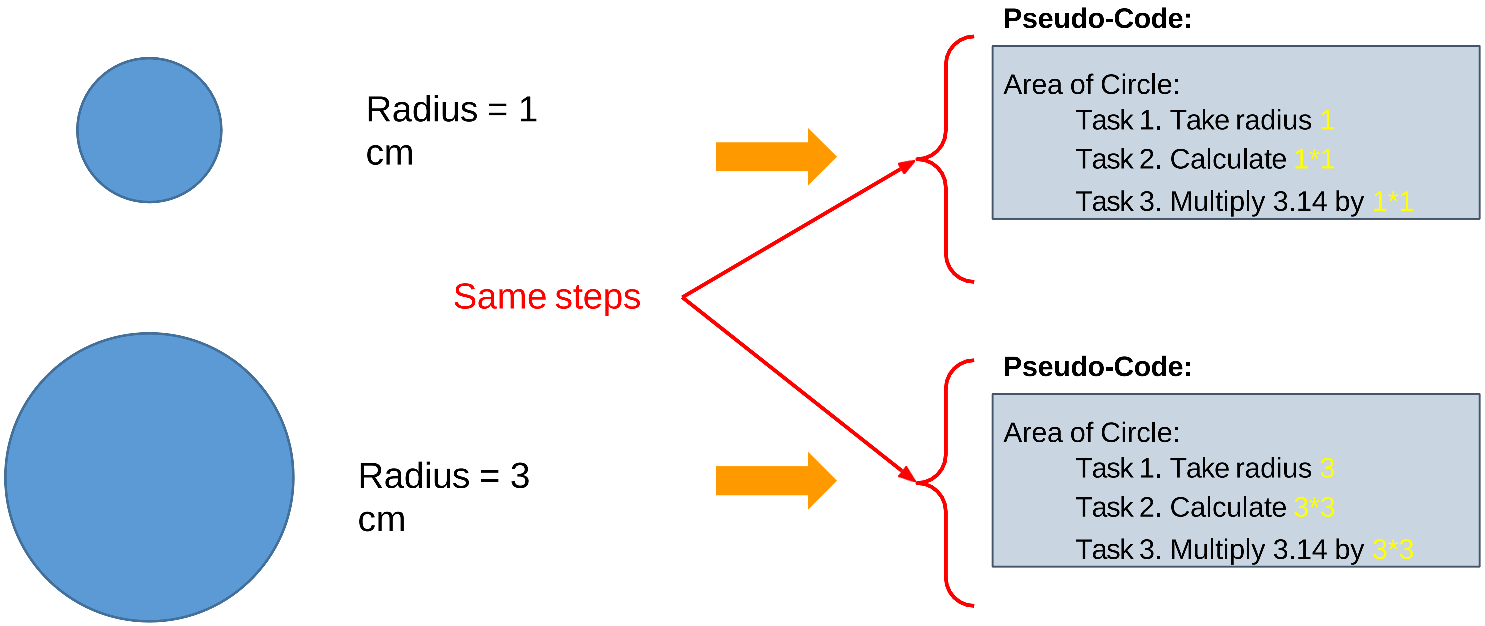


Radius = 3 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 3
        Task 2. Calculate 3*3
        Task 3. Multiply 3.14 by 3*3
```

Solution #2: Function



Radius = 1 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 1
        Task 2. Calculate 1*1
        Task 3. Multiply 3.14 by 1*1
```

Same steps

Radius = 3 cm

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius 3
        Task 2. Calculate 3*3
        Task 3. Multiply 3.14 by 3*3
```

Function in Python

Code
:

```
def area_circle(r):

        area = 3.14 * r *

        r  return area
```

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius r
        Task 2. Calculate r*r
        Task 3. Multiply 3.14 by r*r
        Task 4. You get the area
```

What are Functions?

What are functions?

Code:

```
def area_circle(r):

    area = 3.14 * r *

    r  return area
```

What are Functions?

What is functions?

- Reusable piece of code

Code :

```
def area_circle(r):
        area = 3.14 * r *
        r  return area
```

What are Functions?

What is functions?

- Reusable piece of code
- Created for solving specific problem

Code
:

```
def area_circle(r):
        area = 3.14 * r *
        r  return area
```

Function: Syntax

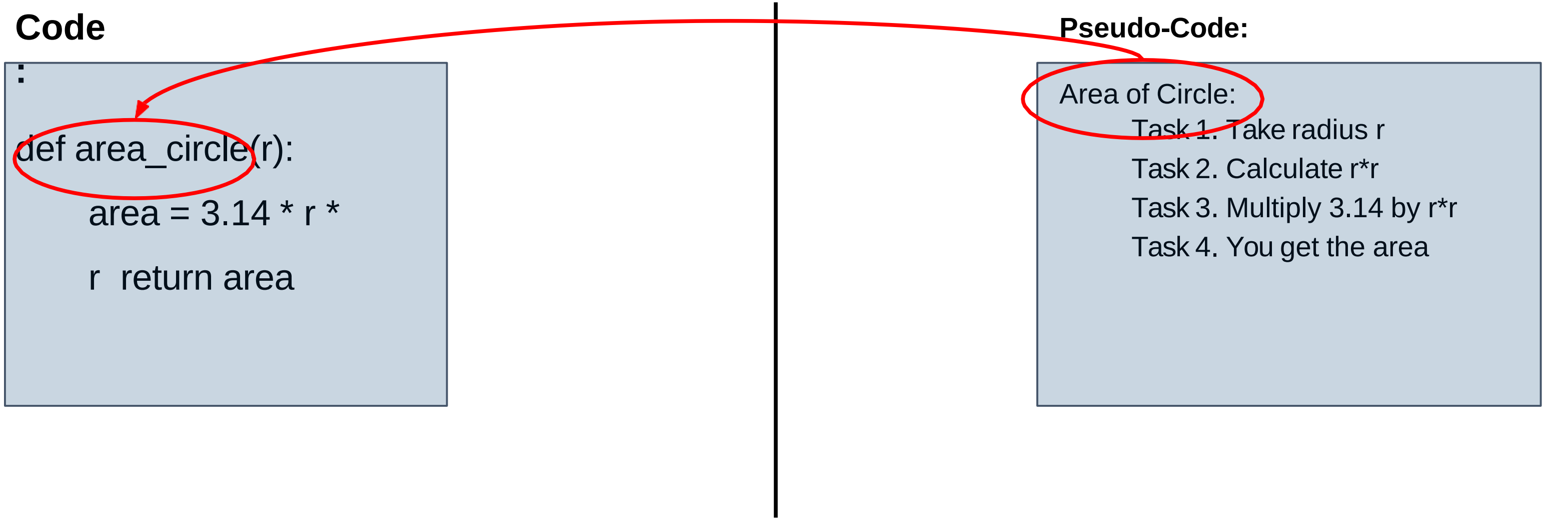
Code:

```
def area_circle(r):
        area = 3.14 * r *
        r  return area
```

Pseudo-Code:

```
Area of Circle:
        Task 1. Take radius r
        Task 2. Calculate r*r
        Task 3. Multiply 3.14 by r*r
        Task 4. You get the area
```

Function: Syntax



Code
:

```
def area_circle(r):
      area = 3.14 * r *
      r  return area
```

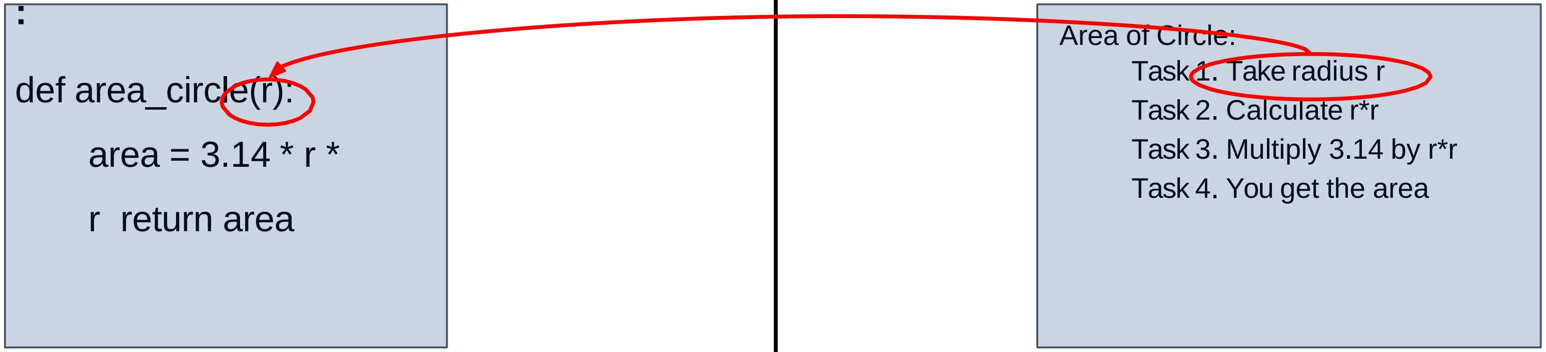
Pseudo-Code:

```
Area of Circle:
      Task 1. Take radius r
      Task 2. Calculate r*r
      Task 3. Multiply 3.14 by r*r
      Task 4. You get the area
```

Function: Syntax

Code :

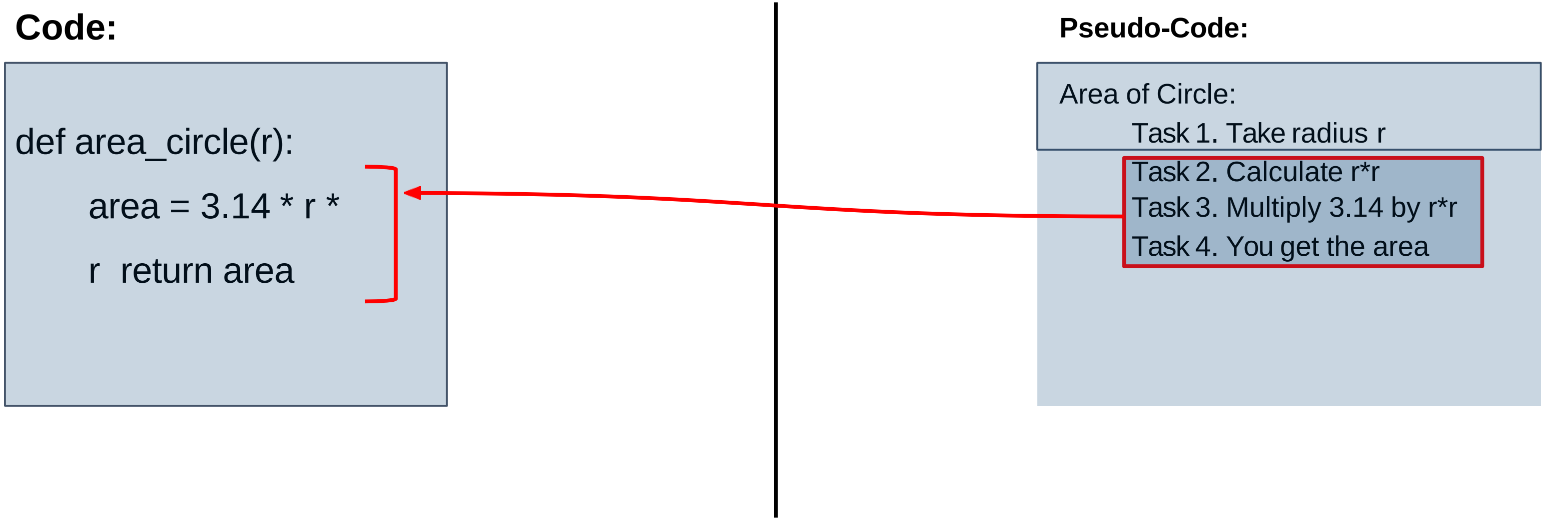
Pseudo-Code:



```
def area_circle(r):
        area = 3.14 * r *
        r  return area
```

```
Area of Circle:
        Task 1. Take radius r
        Task 2. Calculate r*r
        Task 3. Multiply 3.14 by r*r
        Task 4. You get the area
```

Function:
Syntax



Code:

```
def area_circle(r):
    area = 3.14 * r *
    r  return area
```

Pseudo-Code:

```
Area of Circle:
    Task 1. Take radius r
    Task 2. Calculate r*r
    Task 3. Multiply 3.14 by r*r
    Task 4. You get the area
```

Function: Syntax

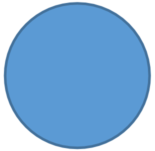
Code :

```
def area_circle(r):
    area = 3.14 * r *
    r  return area
```

Pseudo-Code:

```
Area of Circle:
    Task 1. Take radius r
    Task 2. Calculate r*r
    Task 3. Multiply 3.14 by r*r
    Task 4. You get the area
    Task 5. Return the area
```

Functions

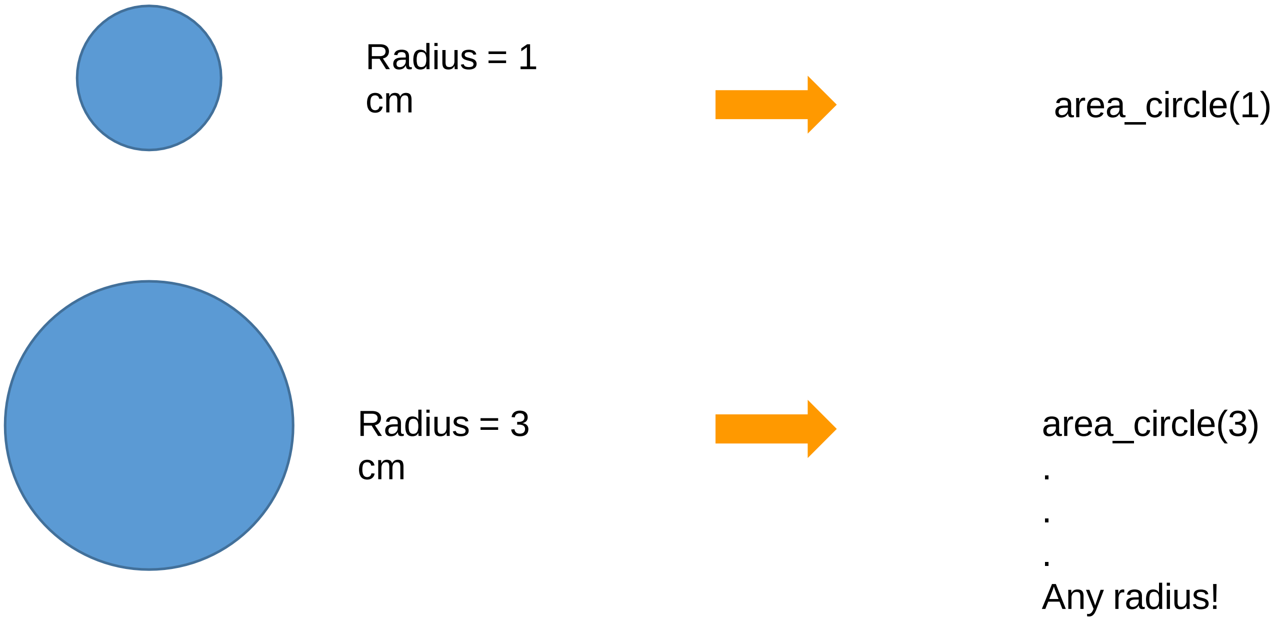


Radius = 1
cm

→

area_circle(1)

Functions



Radius = 1 cm → area_circle(1)

Radius = 3 cm → area_circle(3)

.

.

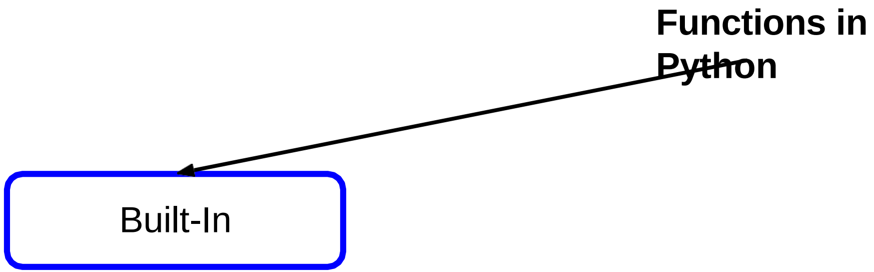
.

Any radius!

Types of Functions

Functions in Python

Types of Functions

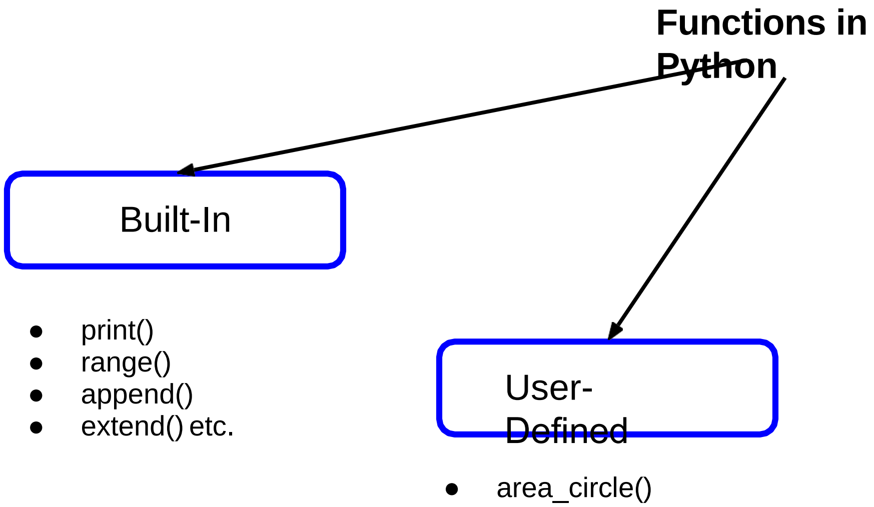


Functions in Python

Built-In

- print()
- range()
- append()
- extend() etc.

Types of Functions



Functions in Python

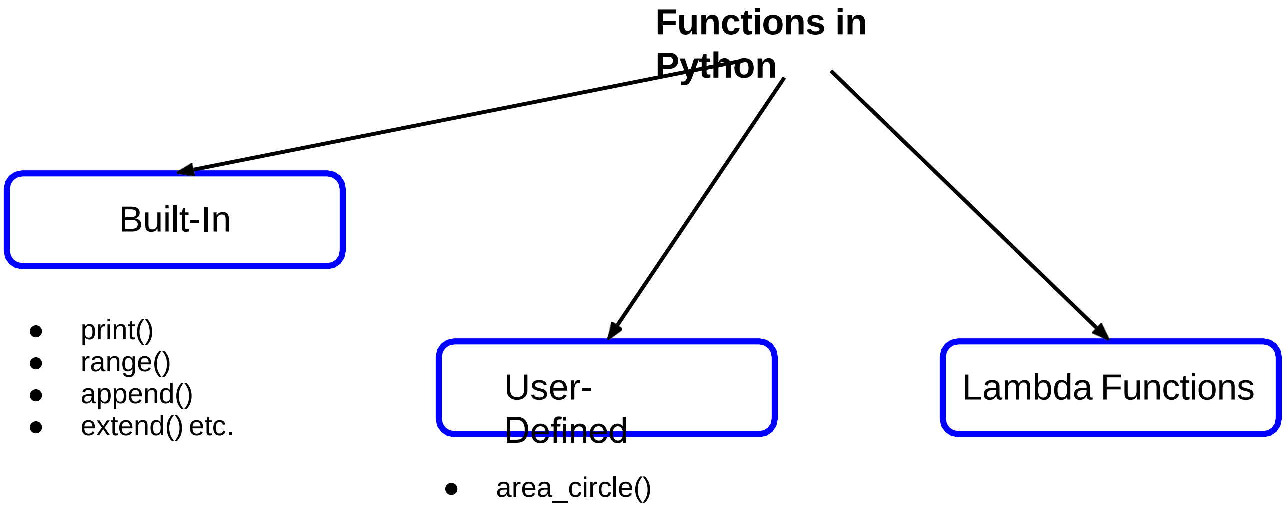
Built-In

- print()
- range()
- append()
- extend() etc.

User-Defined

- area_circle()

Types of Functions



Functions in Python

Built-In

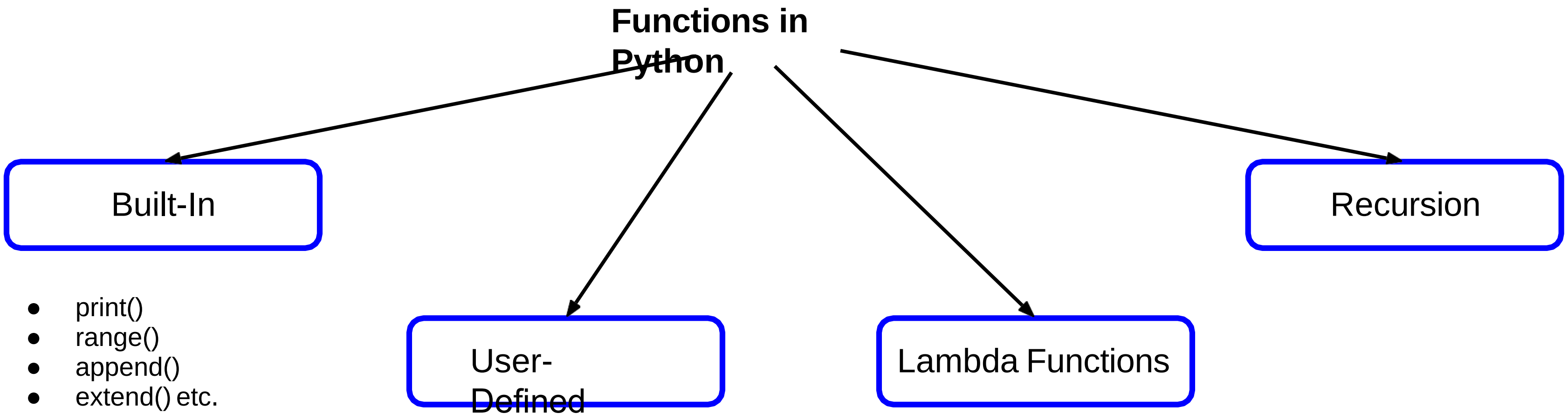
- print()
- range()
- append()
- extend() etc.

User-Defined

- area_circle()

Lambda Functions

Types of Functions



Functions in Python

Built-In

User-Defined

Lambda Functions

Recursion

- print()
- range()
- append()
- extend() etc.

- area_circle()

File Handling

Python Files I/O

Printing to the Screen:

- The simplest way to produce output is using the *print* statement where you can pass zero or more expressions, separated by commas. This function converts the expressions you pass it to a string and writes the result to standard output as follows:

```
print "Python is really a great language,", "isn't it?";
```

- This would produce following result on your standard screen:

```
Python is really a great language, isn't it?
```

Reading Keyboard Input:

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are:

```
raw_input
input
```

The *raw_input* Function:

- The *raw_input([prompt])* function reads one line from standard input and returns it as a string (removing the trailing newline):

```
str = raw_input("Enter your input: ");
print "Received input is : ", str
```

- This would prompt you to enter any string and it would display same string on the screen. When I typed "Hello Python!", it output is like this:

```
Enter your input: Hello Python
Received input is : Hello Python
```

- **The *input* Function:**
- The *input([prompt])* function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you:

```
str = input("Enter your input: ");
print "Received input is : ", str
```

- This would produce following result against the entered input:

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is : [10, 20, 30, 40]
```

Opening and Closing Files:

- Until now, you have been reading and writing to the standard input and output. Now we will see how to play with actual data files.
- Python provides basic functions and methods necessary to manipulate files by default. You can do your most of the file manipulation using a **file** object.
- **The *open* Function:**

  Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object which would be utilized to call other support methods associated with it.
- **Syntax:**

```
file object = open(file_name [, access_mode][,
    buffering])
```

Paramters detail:

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode:** The access_mode determines the mode in which the file has to be opened ie. read, write append etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r)
- **buffering:** If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

A list of the different modes of opening a file:

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

A list of the different modes of opening a file:

wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The *file* object atrributes:

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object:

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

- **Example:**

```
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

- This would produce following result:

```
Name of the file: foo.txt
Closed or not : False
Opening mode : wb
Softspace flag : 0
```

The *close()* Method:

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

- **Syntax:**

  fileObject.close();

  Example:

  ```
  fo = open("foo.txt", "wb")
  print "Name of the file: ", fo.name
  fo.close()
  ```

- This would produce following result:

  ```
  Name of the file: foo.txt
  ```

Reading and Writing Files:

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

The *write()* Method:

- The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string:

Syntax:

```
fileObject.write(string);
```

Example:

```
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\r\nYeah its
   great!!\r\n");
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content

```
Python is a great language.
Yeah its great!!
```

The *read()* Method:

The *read()* method read a string from an open file. It is important to note that Python strings can have binary data and not just text.

Syntax:

```
fileObject.read([count]);
```

Here passed parameter is the number of bytes to be read from the opend file. This method starts reading from the beginning of the file and if *count* is missing then it tries to read as much as possible, may be until the end of file.

Example:

```
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
fo.close()
```

This would produce following result:

```
Read String is : Python is
```

File Positions:

- The *tell()* method tells you the current position within the file in other words, the next read or write will occur at that many bytes from the beginning of the file:
- The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.
- If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example:

```
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
position = fo.tell();
print "Current file position : ", position
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
fo.close()
```

- This would produce following result:

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```

Renaming and Deleting Files:

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module you need to import it first and then you can all any related functions.

The rename() Method:

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax:

```
os.rename(current_file_name, new_file_name)
```

Example:

```
import os
os.rename( "test1.txt", "test2.txt" )
```

The *delete()* Method:

You can use the *delete()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax:

```
os.remove(file_name)
```

Example:

```
import os
os.remove("test2.txt")
```

Directories in Python:

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The *mkdir()* Method:

You can use the *mkdir()* method of the os module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax:

```
os.mkdir("newdir")
```

Example:

```
import os # Create a directory "test"
os.mkdir("test")
```

The *chdir()* Method:

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax:

```
os.chdir("newdir")
```

Example:

```
import os
os.chdir("/home/newdir")
```

The *getcwd()* Method:

The *getcwd()* method displays the current working directory.

Syntax:

```
os.getcwd()
```

Example:

```
import os
os.getcwd()
```

The *rmdir()* Method:

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax:

```
os.rmdir('dirname')
```

Example:

```
import os
os.rmdir( "/tmp/test" )
```

File & Directory Related Methods:

There are three important sources which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows:

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods.: This provides methods to process files as well as directories.

For More Details

- ✓ https://data-flair.training/blogs/python-list-comprehension/
- ✓ https://data-flair.training/blogs/python-tuple/
- ✓ https://data-flair.training/blogs/python-dictionary/
- ✓ https://data-flair.training/blogs/python-function/
- ✓ https://data-flair.training/blogs/file-handling-in-python/

Thank You