CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

UNIVERSITY INSTITUTE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NUMERICAL METHODS AND OPTIMIZATION USING PYTHON
COURSE CODE- 22CSH-259/22ITH-259

DISCOVER . LEARN . EMPOWER

CHAPTER-1.1

Introduction to Python Programming for Numerical Computation:

Python is a versatile programming language widely used in various domains, including numerical computation and scientific computing. Its simplicity, readability, and a vast ecosystem of libraries make it an excellent choice for numerical tasks. In this introduction, we'll cover essential aspects of Python for numerical computation.

Getting Started:

Installation:

Visit the official Python website to download and install Python.

Consider using package managers like Anaconda that come bundled with popular numerical computing libraries.

Why Python?

Readability and ease-of-maintenance

- Python focuses on well-structured easy to read code
- Easier to understand source code.

Extensibility with libraries

• Large base of third-party libraries that greatly extend functionality. Eg., NumPy, SciPy etc.

Python Interpreter

The system component of Python is the interpreter.

- The interpreter is independent of your code and is required to execute your code.

Two major versions of interpreter are currently available:

- Python 2.7.X (broader support, legacy libraries)
- Python 3.6.X (newer features, better future support)

Variables and Objects

Variables are the basic unit of storage for a program.

• Variables can be created and destroyed.

• At a hardware level, a variable is a reference to a location in memory.

• Programs perform operations on variables and alter or fill in their values.

• An object can therefore be considered a more complex variable.

Classes vs. Objects

- Every Object belongs to a certain class.
- Classes are abstract descriptions of the structure and functions of an object.
- Objects are created when an instance of the class is created by the program.
- For example, “Fruit” is a class while an “Apple” is an object.

What is an Object?

• Almost everything is an object in Python, and it belongs to a certain class.
• Python is dynamically and strongly typed:
  - Dynamic: Objects are created dynamically when they are initiated and assigned to a class.
  - Strong: Operations on objects are limited by the type of the object.
• Every variable you create is either a built-in data type object OR a new class you created

THANK YOU

UNIVERSITY INSTITUTE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NUMERICAL METHODS AND OPTIMIZATION USING PYTHON
COURSE CODE- 22CSH-259/22ITH-259
DR. HARDEEP KAUR (E15828)

DISCOVER . LEARN . EMPOWER

Core data types: • Numbers • Strings • Lists • Dictionaries • Tuples • Files • Sets

Numbers

• Can be integers, decimals (fixed precision), floating points (variable precision), complex numbers etc.

• Simple assignment creates an object of number type such as:

• a = 3 • b = 4.56 • Supports simple to complex arithmetic operators. • Assignment via numeric operator also creates a number object:

• c = a / b • a, b and c are numeric objects.

• Try dir(a) and dir(b) . This command lists the functions available for these objects.

Strings

• A string object is a ‘sequence’, i.e., it’s a list of items where each item has a defined position. • Each character in the string can be referred, retrieved and modified by using its position.

• This order id called the ‘index’ and always starts with 0.

```
>>> S = 'Hello'
>>> len(S)
5
>>> S[0]
'H'
>>> S[4]
'o'
>>> S[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

```
>>> S[-1]
'o'
>>> S[3:]
'lo'
>>> S[2:5]
'llo'
```

Strings :

- String objects support concatenation and repetition operations.

```
>>> S + 'World!'
'HelloWorld!'
>>> S + ' World!'
'Hello World!'
>>> S * 4
'HelloHelloHelloHello'
>>> S + ' World! ' * 4
'Hello World!  World!  World!  World! '
>>> (S + ' World! ') * 4
'Hello World! Hello World! Hello World! Hello World! '
```

Lists

- List is a more general sequence object that allows the individual items to be of different types.
- Equivalent to arrays in other languages.
- Lists have no fixed size and can be expanded or contracted as needed.
- Items in list can be retrieved using the index.
- Lists can be nested just like arrays, i.e., you can have a list of lists.

Simple list:

```
>>> L = [123, 3.14, 'Hello']
>>> L
[123, 3.1400000000000001, 'Hello']
```

```
>>> L[0]
123
```

Nested list:

```
>>> DDL = [[1,2,3],
... [4,5,6],
... [7,8,9]]
>>> DDL
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> DDL[2][1]
8
```

Dictionaries:

- Dictionaries are unordered mappings of ’Name : Value’ associations.
- Comparable to hashes and associative arrays in other languages.
- Intended to approximate how humans remember associations

```
>>> D = {'name':'apple','color':'red','taste':'sweet','number':'5'}
>>> D['name']
'apple'
>>> D
{'color': 'red', 'taste': 'sweet', 'name': 'apple', 'number': '5'}
```

Files:

- File objects are built for interacting with files on the system. Same object used for any file type. User has to interpret file content and maintain integrity

```
>>> f = open('test.txt','w')
>>> f.write('Hello\t')
>>> f.write('world!\n')
>>> f.close()
>>> f = open('test.txt')
>>> text = f.read()
>>> text
'Hello\tworld!\n'
>>> print(text)
Hello	world!
```

Mutable vs. Immutable

- Numbers, strings and tuples are immutable i.,e cannot be directly changed.
- Lists, dictionaries and sets can be changed in place.

```
>>> S[0]
'H'
>>> S[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> L[1]
3.1400000000000001
>>> L[1] = 3.145
```

Tuples

• Tuples are immutable lists. • Maintain integrity of data during program execution.

Sets

- Special data type introduced since Python 2.4 onwards to support mathematical set theory operations.
- Unordered collection of unique items.
- Set itself is mutable, BUT every item in the set has to be an immutable type.
- So, sets can have numbers, strings and tuples as items but cannot have lists or dictionaries as items.

THANK YOU

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

UNIVERSITY INSTITUTE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NUMERICAL METHODS AND OPTIMIZATION USING PYTHON
COURSE CODE- 22CSH-259/22ITH-259

DISCOVER . LEARN . EMPOWER

Basic Python syntax

Python is known for its simplicity and readability, making it an excellent choice for beginners and experienced developers alike.

1. Comments: Comments start with the # symbol and are ignored by the Python interpreter.

```python
# This is a comment
```

2. Variables and Data Types:

Variables don't require explicit declaration and dynamically change types.

```python
x = 5                   # Integer
y = 3.14                # Float
name = "Python"         # String
is_true = True           # Boolean
```

3. Print Statement:

Use print() to display output.

```python
print("Hello, World!")
```

4. Indentation:

Python uses indentation to indicate blocks of code. It's crucial for readability and structure.

```python
if x > 0:
    print("Positive")
else:
    print("Non-positive")
```

5. Control Flow:

if, elif, and else statements for conditional execution.

for and while loops for iteration.

```python
# Example of a for loop
for i in range(5):
    print(i)

# Example of a while loop
counter = 0
while counter < 3:
    print("Counting:", counter)
    counter += 1
```

6. Functions:

Define functions using the def keyword.

```python
def greet(name):
    return "Hello, " + name + "!"

result = greet("Alice")
print(result)
```

7. Lists:

A versatile data structure for holding ordered elements.

```python
numbers = [1, 2, 3, 4, 5]
```

8. Dictionaries:

Store data as key-value pairs.

```python
person = {'name': 'John', 'age': 30, 'city': 'New York'}
```

9. Tuples:

Similar to lists, but immutable.

```python
coordinates = (3, 4)
```

10. Strings:

Manipulate and concatenate strings.

```python
message = "Hello"
print(message + " World")
```

11. List Comprehensions:

A concise way to create lists.

```python
squares = [x**2 for x in range(5)]
```

12. Error Handling:

Use try, except blocks for handling exceptions.

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

13. Classes:

Define classes using the class keyword.

```python
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("Woof!")
```

14. Importing Modules:

Import external libraries or modules using import.

```python
import math

result = math.sqrt(25)
```

THANK YOU

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

UNIVERSITY INSTITUTE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NUMERICAL METHODS AND OPTIMIZATION USING PYTHON

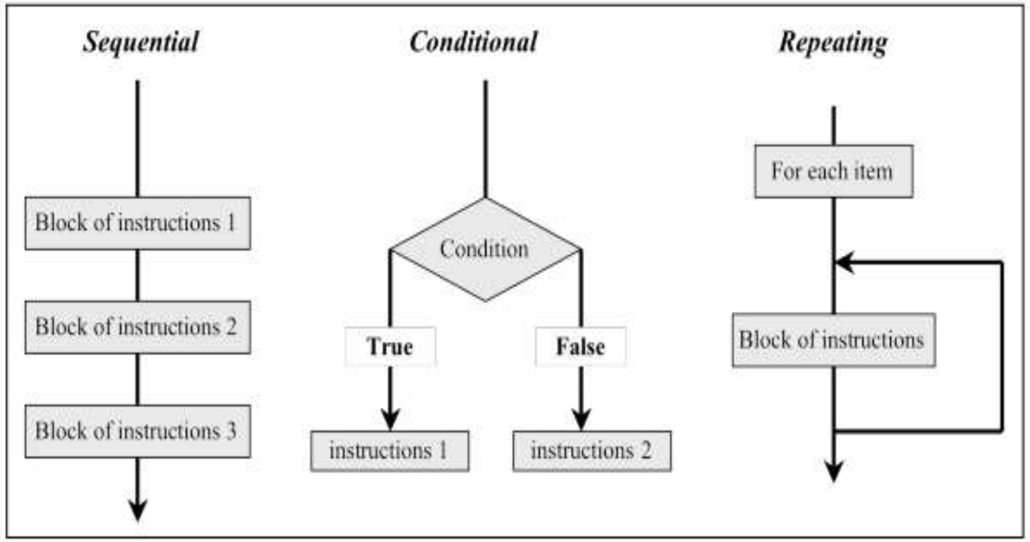
COURSE CODE- 22CSH-259/22ITH-259

DR. HARDEEP KAUR (E15828)

DISCOVER . LEARN . EMPOWER

Control Structures

- Python has thought about these issues, and offers solutions in the form of control structures:
- The if structure that allows to control if a block of instruction need to be executed, and the for structure (and equivalent), that repeats a set of instructions for a preset number of times.



Logical operators Most of the control structure we will see in this chapter test if a condition is true or false. For programmers, "truth" is easier to define in terms of what is not truth! In Python, there is a short, specific list of false values:

- An empty string, " ", is false
- The number zero and the string "0" are both false.
- An empty list, (), is false.
- The singleton None (i.e. no value) is false. Everything else is true

Comparing numbers and strings

We can test whether a number is bigger, smaller, or the same as another. All the results of these tests are TRUE or FALSE. Table lists the common comparison operators available in Python.

comparison	Corresponding question
a == b	Is a equal to b ?
a != b	Is a not equal to b ?
a > b	Is a greater than b ?
a >= b	Is a greater than or equal to b ?
a < b	Is a less than b ?
a <= b	Is a less than or equal to b ?
a in b	Is the value a in the list (or tuple) b?
a not in b	Is the value a not in the list (or tuple) b?

Combining logical operators

We can join together several tests into one, by the use of the logical operator and and or.

a and **b**	True if both **a** and **b** are true.
a or **b**	True if either **a**, or **b**, or both are true.
not **a**	True if **a** is false.

Conditional structures

- **If**
- It is used to protect a block of code that only needs to be executed if a prior condition is met (i.e. is TRUE).

```
>>> if condition:
        code block
```

Else

- When making a choice, sometimes you have two different things you want to do, depending upon the outcome of the conditional. This is done using an if …else structure that has the following format:

```
if  condition:
        block code 1
else:
        block code 2
```

Loops

loops allow you to do that. Every loop has three main parts:

• An entry condition that starts the loop • The code block that serves as the “body” of the loop

For loop

The most basic type of determinate loop is the for loop. Its basic structure is:

```
for variable in listA:
        code block
```

```
>>> names=["John","Jane","Smith"]
>>> j=0
>>> for name in names:
        j+=1
        print "The name number ",j," in the list is ",name
```

- **While loop**
- Sometimes, we face a situation where neither Python nor we know in advance how many times a loop will need to execute. This is the case for example when reading a file: we do not know in advance how many lines it has. Python has a structure for that: the while loop:

```
while TEST:
        code block;
```

- The while structure executes the code block as long as the TEST expression evaluates as TRUE. For example, here is a program that prints the number between 0 and N, where N is input:

```
>>> N=int(raw_input("Enter N --> "))
>>> print "Counting numbers from 0 to ",N,"\n"
 i=0
while  i < N+1:
        print i,"\n"
        i+=1
```

THANK YOU

UNIVERSITY INSTITUTE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NUMERICAL METHODS AND OPTIMIZATION USING PYTHON
COURSE CODE- 22CSH-259/22ITH-259

DISCOVER . LEARN . EMPOWER

Basic Python for Calculus and Algebra.

- Linear algebra is a branch of mathematics that deals with linear equations and their representations using vectors and matrices
- **Understanding Vectors, Matrices, and the Role of Linear Algebra**
- A **vector** is a mathematical entity used to represent physical quantities that have both magnitude and direction.
- **Matrices**are used to represent vector transformations, among other applications.
- In Python, NumPy is the most used library for working with matrices and vectors

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

- Python
- In [1]: import numpy as np
- In [2]: np.array([[1, 2], [3, 4], [5, 6]])
- Out[2]:
- array([[1, 2],
- [3, 4],
- [5, 6]])

A **linear system** or, more precisely, a system of linear equations, is a set of equations linearly relating to a set of variables. Here's an example of a linear system relating to the variables x_1 and x_2:

$$\begin{cases} 3x_1 + 2x_2 = 12 \\ 2x_1 - 1x_2 = 1 \end{cases}$$

- It's common to write linear systems using matrices and vectors. For example, you can write the previous system as the following **matrix product**:

$$\underbrace{\begin{bmatrix} 3 & 2 \\ 2 & -1 \end{bmatrix}}_{\text{A}} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{\text{x}} = \underbrace{\begin{bmatrix} 12 \\ 1 \end{bmatrix}}_{\text{b}}$$

- you can notice the elements of matrix **A** correspond to the coefficients that multiply x_1 and x_2. Besides that, the values in the right-hand side of the original equations now make up vector **b**.

Using Determinants to Study Linear Systems

- System with two equations given by $\boldsymbol{x_1 + x_2 = 2}$ and $\boldsymbol{x_1 + x_2 = 3}$ is inconsistent and has no solution.
- This happens because no two numbers x_1 and x_2 can add up to both 2 and 3 at the same time.
- system with two equivalent equations, such as $x_1 + x_2 = 2$ and $2x_1 + 2x_2 = 4$, then you can find an infinite number of solutions, such as (x_1=1, x_2=1), (x_1=0, x_2=2), (x_1=2, x_2=0), and so on.
- A **determinant** is a number, calculated using the matrix of coefficients, that tells you if there's a solution for the system.

- Because you'll be using scipy.linalg to calculate it, you don't need to care much about the details on how to make the calculation. However, keep the following in mind:
- If the determinant of a coefficients matrix of a linear system is **different from zero**, then you can say the system has a **unique solution**.
- If the determinant of a coefficients matrix of a linear system is **equal to zero**, then the system may have either **zero solutions** or an **infinite number of solutions**.
- Now that you have this in mind, you'll learn how to solve linear systems using matrices.
- **Using Matrix Inverses to Solve Linear Systems**

- To understand the idea behind the inverse of a matrix, start by recalling the concept of the **multiplicative inverse** of a number. When you multiply a number by its inverse, you get 1 as the result. Take 3 as an example. The inverse of 3 is 1/3, and when you multiply these numbers, you get 3 × 1/3 = 1.
- With square matrices, you can think of a similar idea. However, instead of 1, you'll get an **identity matrix** as the result.

$$\mathbf{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The identity matrix has an interesting property: when multiplied by another matrix **A** of the same dimensions, the obtained result is **A**.
- Recall that this is also true for the number 1, when you consider the multiplication of numbers.

This allows you to solve a linear system by following the same steps used to solve an equation. As an example, consider the following linear system, written as a matrix product:

$$\underbrace{\begin{bmatrix} 3 & 2 \\ 2 & -1 \end{bmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 12 \\ 1 \end{bmatrix}}_{\mathbf{b}}$$

- By calling $\mathbf{A}^{-1}$ the inverse of matrix **A**, you could multiply both sides of the equation by $\mathbf{A}^{-1}$, which would give you the following result:

$$\begin{aligned} A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

- This way, by using the inverse, $\mathbf{A}^{-1}$, you can obtain the solution **x** for the system by calculating $\mathbf{A}^{-1}\mathbf{b}$.
- It's worth noting that while non-zero numbers always have an inverse, not all matrices have an inverse. When the system has no solution or when it has multiple solutions, the determinant of **A** will be zero, and the inverse, $\mathbf{A}^{-1}$, won't exist.
- Now you'll see how to use Python with scipy.linalg to make these calculations.
- Calculating Inverses and Determinants With **scipy.linalg**

- You can calculate matrix inverses and determinants using scipy.linalg.inv() and scipy.linalg.det().
- Recall that the linear system for this problem could be written as a matrix product:

$$\underbrace{\begin{bmatrix} 1 & 9 & 2 & 1 & 1 \\ 10 & 1 & 2 & 1 & 1 \\ 1 & 0 & 5 & 1 & 1 \\ 2 & 1 & 1 & 2 & 9 \\ 2 & 1 & 2 & 13 & 2 \end{bmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 170 \\ 180 \\ 140 \\ 180 \\ 350 \end{bmatrix}}_{\mathbf{b}}$$

- Previously, you used scipy.linalg.solve() to obtain the solution 10, 10, 20, 20, 10 for the variables x_1 to x_5, respectively.
- But as you've just learned, it's also possible to use the inverse of the coefficients matrix to obtain vector **x**, which contains the solutions for the problem.
- You have to calculate $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, which you can do with the following program:

Python

```
In [1]: import numpy as np
   ...: from scipy import linalg

In [2]: A = np.array(
   ...:     [
   ...:         [1, 9, 2, 1, 1],
   ...:         [10, 1, 2, 1, 1],
   ...:         [1, 0, 5, 1, 1],
   ...:         [2, 1, 1, 2, 9],
   ...:         [2, 1, 2, 13, 2],
   ...:     ]
   ...: )

In [3]: b = np.array([170, 180, 140, 180, 350]).reshape((5, 1))

In [4]: A_inv = linalg.inv(A)

In [5]: x = A_inv @ b
   ...: x
Out[5]:
array([[10.],
       [10.],
       [20.],
       [20.],
       [10.]])
```

- Here’s a breakdown of what’s happening:
-
- **Lines 1 and 2** import NumPy as np, along with linalg from scipy. These imports allow you to use linalg.inv().
- **Lines 4 to 12** create the coefficients matrix as a NumPy array called A.
- **Line 14** creates the independent terms vector as a NumPy array called b. To make it a column vector with five elements, you use .reshape((5, 1)).
- **Line 16** uses linalg.inv() to obtain the inverse of matrix A.
- **Lines 18 and 19** use the @ operator to perform the matrix product in order to solve the linear system characterized by A and b. You store the result in x, which is printed.
- You get exactly the same solution as the one provided by scipy.linalg.solve(). Because this system has a unique solution, the determinant of matrix **A** must be different from zero. You can confirm that it is by calculating it using det() from scipy.linalg:

THANK YOU