**UNIT-1**

**CHAPTER-1.1 NOTES**

**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… • ..hence easier to maintain code base

**Portability**

• Scripting language hence easily portable • Python interpreter is supported on most modern OS’s

**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.

• Objects are higher level constructs that include one or more variables and the set of operations that work on these variables.

• 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.

**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:

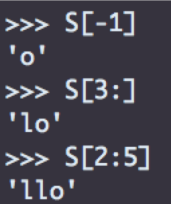
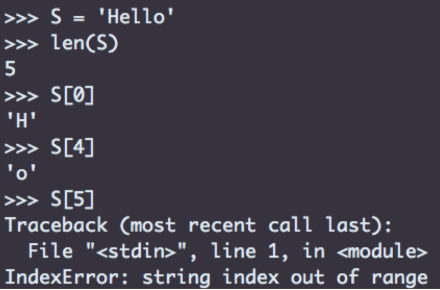
• c = a / b • a, b and c are numeric objects.

• 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.



**Strings**

• String objects support concatenation and repetition operations.



**Lists**

• List is a more general sequence object that allows the individual items to be of different types.

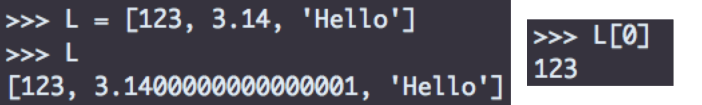
• Equivalent to arrays in other languages.

• Lists have no fixed size and can be expanded or contracted as needed.

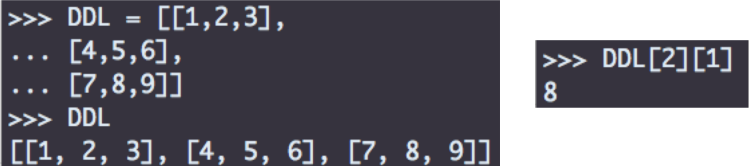
• Items in list can be retrieved using the index.

• Lists can be nested just like arrays, i.e., you can have a list of lists.

Simple list:



Nested list:

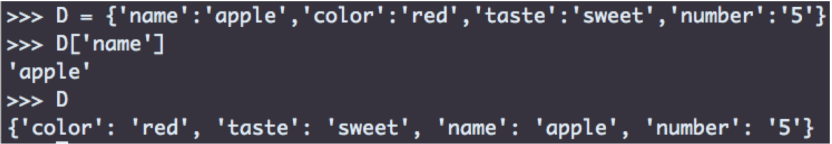


**Dictionaries:**

• Dictionaries are unordered mappings of ’Name : Value’ associations.

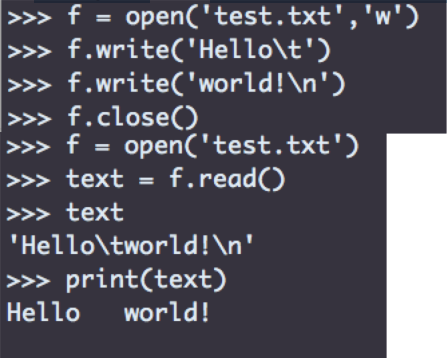
• Comparable to hashes and associative arrays in other languages.

• Intended to approximate how humans remember associations.



**Files**

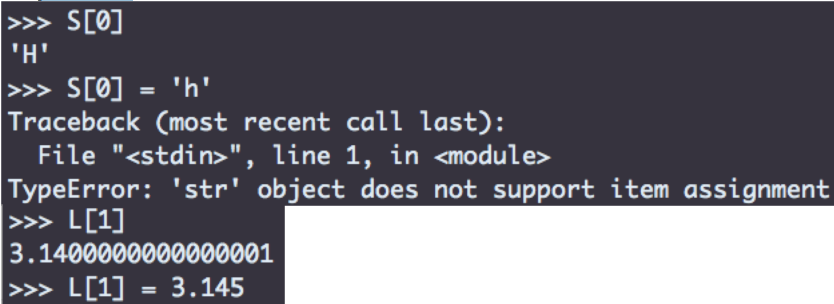
• File objects are built for interacting with files on the system. • Same object used for any file type. • User has to interpret file content and maintain integrity



**Mutable vs. Immutable**

• Numbers, strings and tuples are immutable i.,e cannot be directly changed.

• Lists, dictionaries and sets can be changed in place.



**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.

**Basic Python syntax**

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

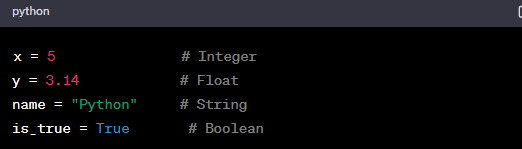
**1. Comments:**

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



**2. Variables and Data Types:**

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



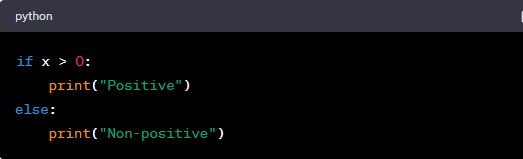
**3. Print Statement:**

Use print() to display output.



**4. Indentation:**

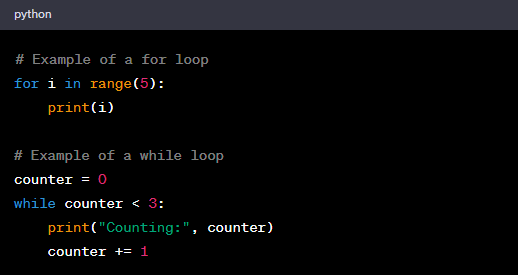
Python uses indentation to indicate blocks of code. It's crucial for readability and structure.



**5. Control Flow:**

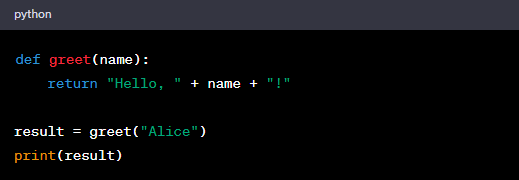
if, elif, and else statements for conditional execution.

for and while loops for iteration.



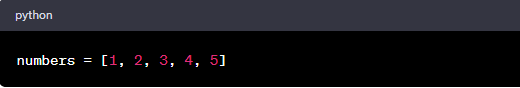
**6. Functions:**

Define functions using the def keyword.



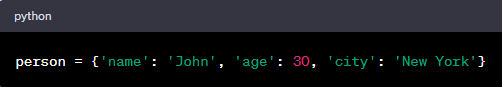
**7. Lists:**

A versatile data structure for holding ordered elements.



**8. Dictionaries:**

Store data as key-value pairs.



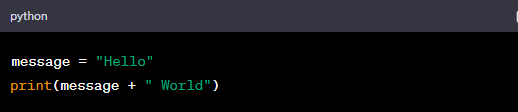
**9. Tuples:**

Similar to lists, but immutable.



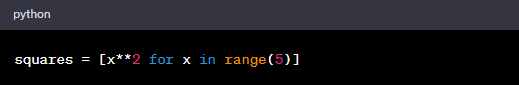
**10. Strings:**

Manipulate and concatenate strings.



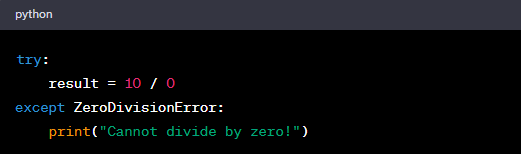
**11. List Comprehensions:**

A concise way to create lists.



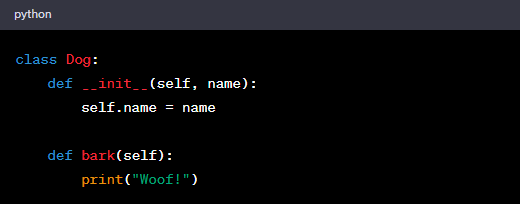
**12. Error Handling:**

Use try, except blocks for handling exceptions.



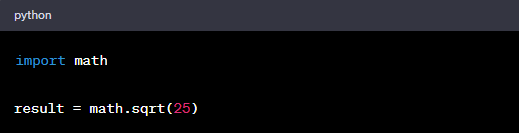
**13. Classes:**

Define classes using the class keyword.



**14. Importing Modules:**

Import external libraries or modules using import.

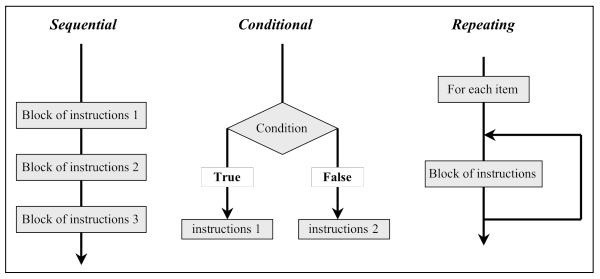


**Control Structures**

The programs we have studied so far have all been sequential, with each line corresponding to one instruction: this is definitely not optimal.

For example, we have introduced in the previous chapter the concept of lists and arrays, to avoid having to use many scalar variables to store data (remember that if we were to store the whole human genome, we would need either 30,000 scalar variables, one for each gene, or a single array, whose items are the individual genes); if we wanted to perform the same operation on each of these genes, we would still have to write one line for each gene.

In addition, the programs we have written so far would attempt to perform all their instructions, once given the input. Again, this is not always desired: we may want to perform some instructions only if a certain condition is satisfied. Again, 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. In this chapter, we will look in details on the syntax and usage of these two structures.



*Figure : The three main types of flow in a computer program: sequential, in which instructions are executed successively, conditional, in which the blocks “instructions 1” and “instructions 2” are executed if the Condition is True or False, respectively, and repeating, in which instructions are repeated over a whole list.*

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.

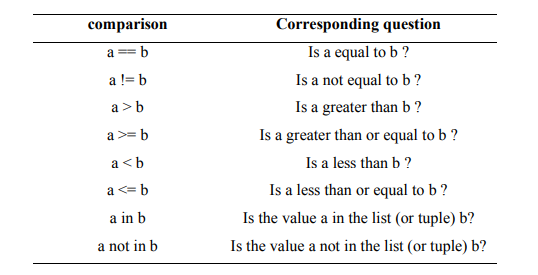
• An empty list, (), is false.

• 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. Similarly, we can test if a string comes before or after another string, based on the alphabetical order. All the results of these tests are TRUE or FALSE. Table 3.1 lists the common comparison operators available in Python.

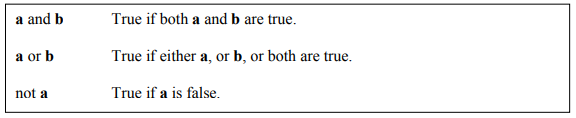
Notice that the numeric operators look a little different from what we have learned in Math: this is because Python does not use the fancy fonts available in text editors, so symbols like ≠, ≤, ≥ do not exist. Notice also that the numeric comparison for equality uses two = symbols (==): this is because the single = is reserved for assignment.



These comparisons apply both to numeric value and to strings. Note that you can compare numbers to strings, but the result can be arbitrary: I would strongly advise to make sure that the types of the variables that are compared are the same.

**Combining logical operators**

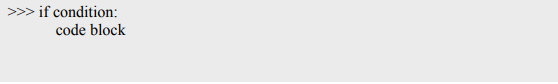
We can join together several tests into one, by the use of the logical operator and and or.



**Conditional structures**

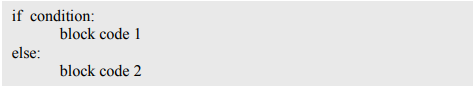
**If**

The most fundamental control structure is the if structure. 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). The generate format of an if statement is:



**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:



**Loops**

One of the most obvious things to do with an array is to apply a code block to every item in the array: 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

An exit condition if condition: block code 1 else:

block code 2 hidden=”Mypasscode” password=raw\_input(“Enter your password : “) if password == hidden: print “You entered the right password\n” else: print “Wrong password !!\n”; if CONDITION1: block code 1 elif CONDITION2: block code 2 else : block code 3 31 Obviously, all three are important. Without the entry condition, the loop won’t be executed; a loop without body won’t do any thing; and finally, without a proper exit condition, the program will never exit the loop (this leads to what is referred to an infinite loop, and often results from a bug in the exit loop).

There are two types of loops: determinate and indeterminate. Determinate loops carry their end condition with them from the beginning, and repeat its code block an exact number of times. Indeterminate loops rely upon code within the body of the loop to alter the exit condition so the loop can exit. We will see one determinate loop structure, for, and one indeterminate loop structure, while

.

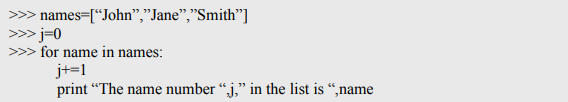
**For loop**

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



Note the syntax similar to the syntax of an if statement: the : at the end of the condition, and the indentation of the code block. A for loop is simple: it loops over all possible values of variable as found in the list listA, executing each time the code block.

For example, the Python program:

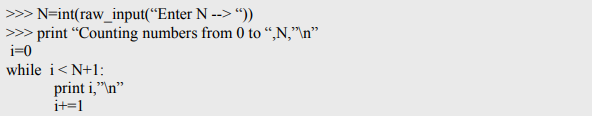


**While loop**

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:



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:



Note that it is important to make sure that the code block includes a modification of the test:

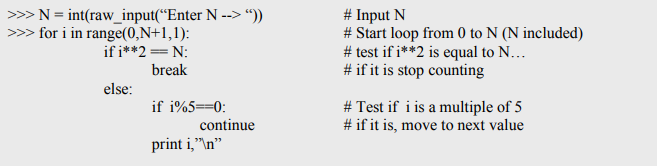
if we had forgotten the line i+=1 in the example above, the while loop would have become an infinite loop. Note that any for loop can be written as a while loop.

In practice however, it is better to use a for loop, as Python executes them faster

**Break points in loops**

Python provides two functions that can be used to control loops from inside its code block: break allows you to exit the loop, while continue skips the following step in the loop.

while TEST: code block; >>> 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 >>> N=int(raw\_input(“Enter the last integer considered --> “)) >>> Sum=0 >>> for i in range(0,N+1,1): Sum+=i\*\*2 >>> print “The sum of the squares between 0 and “,N,” is “,Sum 33 Here is an example of a program that counts and prints number from 1 to N (given as input), skipping numbers that are multiples of 5 and stopping the count if the number reached squared is equal to N:



**Basic Python for Calculus and Algebra.**

[Linear algebra](https://en.wikipedia.org/wiki/Linear_algebra) is a branch of mathematics that deals with linear equations and their representations using [vectors](https://en.wikipedia.org/wiki/Vector_(mathematics_and_physics)) and [matrices](https://en.wikipedia.org/wiki/Matrix_(mathematics)). It’s a fundamental subject in several areas of engineering, and it’s a prerequisite to a deeper understanding of [machine learning](https://realpython.com/learning-paths/machine-learning-python/).

To work with linear algebra in Python, you can count on [SciPy](https://scipy.org/), which is an open-source Python library used for scientific computing, including several modules for common tasks in science and engineering.

Of course, SciPy includes modules for [linear algebra](https://docs.scipy.org/doc/scipy/reference/linalg.html), but that’s not all. It also offers [optimization](https://realpython.com/python-scipy-cluster-optimize/), [integration](https://docs.scipy.org/doc/scipy/tutorial/integrate.html), [interpolation](https://docs.scipy.org/doc/scipy/reference/interpolate.html), and [signal processing](https://realpython.com/python-scipy-fft/) capabilities. It’s part of the [SciPy stack](https://projects.scipy.org/), which includes several other packages for scientific computing, such as [NumPy](https://realpython.com/numpy-tutorial/), [Matplotlib](https://realpython.com/python-matplotlib-guide/), [SymPy](https://www.sympy.org/), [IPython](https://realpython.com/ipython-interactive-python-shell/), and [pandas](https://realpython.com/learning-paths/pandas-data-science/).

**scipy.linalg** includes several tools for working with linear algebra problems, including functions for performing matrix calculations, such as [determinants](https://en.wikipedia.org/wiki/Determinant), [inverses](https://en.wikipedia.org/wiki/Invertible_matrix), [eigenvalues, eigenvectors](https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors), and the [singular value decomposition](https://en.wikipedia.org/wiki/Singular_value_decomposition).

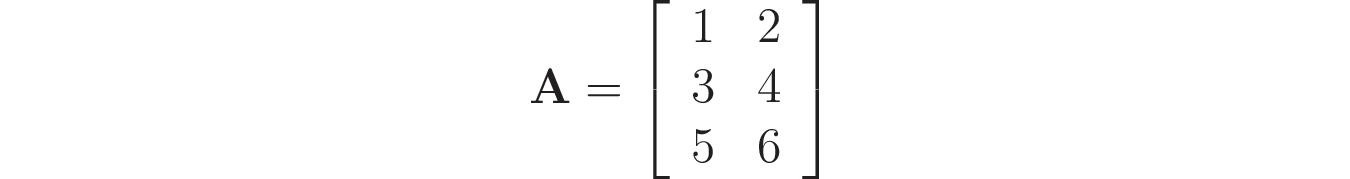
We’re going a step further, using scipy.linalg to study linear systems and build linear models for real-world problems.

In order to use scipy.linalg, you have to install and set up the SciPy library. Besides that, you’re going to use [Jupyter Notebook](https://jupyter.org/) to run the code in an interactive environment. SciPy and Jupyter Notebook are third-party packages that you need to install. For installation, you can use the [conda](https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html) or [pip](https://realpython.com/what-is-pip/) package manager. Revisit [Working With Linear Systems in Python With scipy.linalg](https://realpython.com/python-scipy-linalg/#getting-started-with-scipylinalg) for installation details.

**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. It’s a fundamental tool for solving engineering and machine learning problems. So are **matrices**, which are used to represent vector transformations, among other applications.

**Note:** In Python, [NumPy](https://numpy.org/) is the [most used library](https://www.jetbrains.com/lp/python-developers-survey-2020/#FrameworksLibraries) for working with matrices and vectors. It uses a special type called [ndarray](https://realpython.com/numpy-array-programming/) to represent them. As an example, imagine that you need to create the following matrix:



With NumPy, you can use np.array() to create it, providing a nested list containing the elements of each row of the matrix:

**Python**

**In [1]: import numpy as np**

**In [2]: np.array([[1, 2], [3, 4], [5, 6]])**

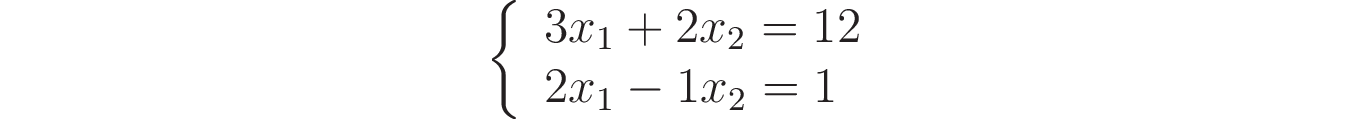
**Out[2]:**

**array([[1, 2],**

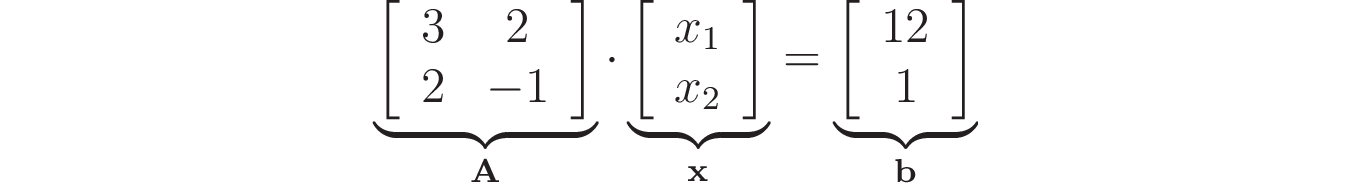
**[3, 4],**

**[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:



Here you have two equations involving two variables. In order to have a **linear** system, the values that multiply the variables *x*₁ and *x*₂ must be constants, like the ones in this example. It’s common to write linear systems using matrices and vectors. For example, you can write the previous system as the following **matrix product**:



Comparing the matrix product form with the original system, 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**.

In [Working With Linear Systems in Python With scipy.linalg](https://realpython.com/python-scipy-linalg/#using-scipylinalgsolve), you’ve seen how to solve linear systems using scipy.linalg.solve(). Now you’re going to learn how to use determinants to study the possible solutions and how to solve problems using the concept of matrix inverses.

Solving Problems Using Matrix Inverses and Determinants

Matrix inverses and determinants are tools that allow you to get some information about the linear system and also to solve it. Before going through the details on how to calculate matrix inverses and determinants using scipy.linalg, take some time to remember how to use these structures.

**Using Determinants to Study Linear Systems**

As you may recall from your math classes, not every linear system can be solved. You may have a combination of equations that’s inconsistent and has no solution. For example, a system with two equations given by *x*₁ + *x*₂ = 2 and *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.

Besides that, some systems can be solved but have more than one solution. For example, if you have a system with two equivalent equations, such as *x*₁ + *x*₂ = 2 and 2*x*₁ + 2*x*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](https://en.wikipedia.org/wiki/Coefficient_matrix), 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**.

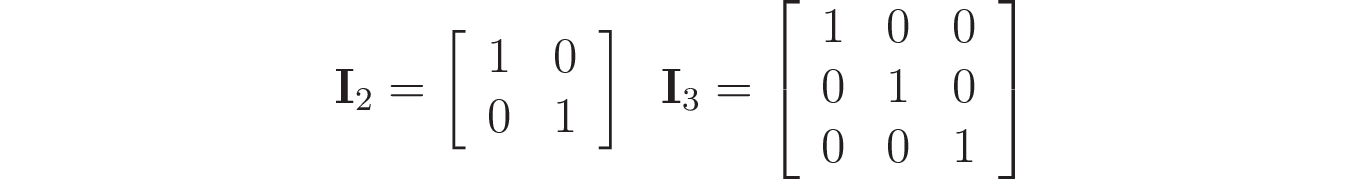
If the determinant of a coefficients matrix of a linear system is **equal to zero**, then the system may have either **zero solutions** or an **infinite number of solutions**.

Now that you have this in mind, you’ll learn how to solve linear systems using matrices.

**Using Matrix Inverses to Solve Linear Systems**

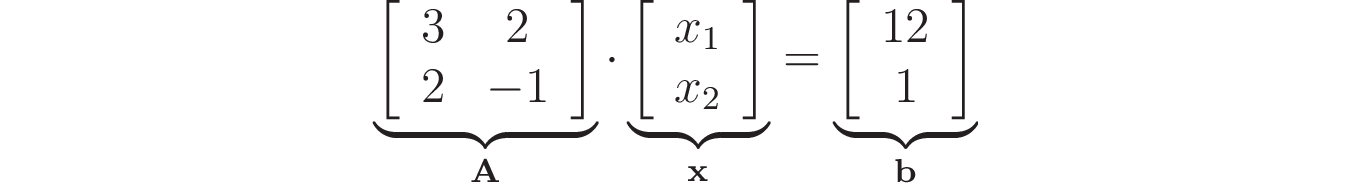
To understand the idea behind the inverse of a matrix, start by recalling the concept of the **multiplicative inverse** of a number. When you multiply a number by its inverse, you get 1 as the result. Take 3 as an example. The inverse of 3 is 1/3, and when you multiply these numbers, you get 3 × 1/3 = 1.

With square matrices, you can think of a similar idea. However, instead of 1, you’ll get an **identity matrix** as the result. An identity matrix has ones in its diagonal and zeros in the elements outside of the diagonal, like the following examples:

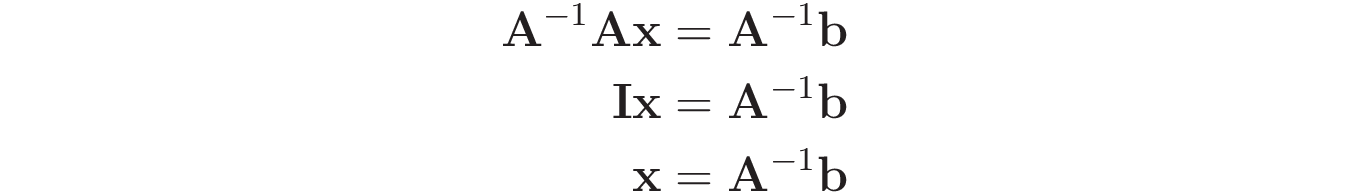


The identity matrix has an interesting property: when multiplied by another matrix **A** of the same dimensions, the obtained result is **A**. Recall that this is also true for the number 1, when you consider the multiplication of numbers.

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:



By calling **A**⁻¹ the inverse of matrix **A**, you could multiply both sides of the equation by **A**⁻¹, which would give you the following result:



This way, by using the inverse, **A**⁻¹, you can obtain the solution **x** for the system by calculating **A**⁻¹**b**.

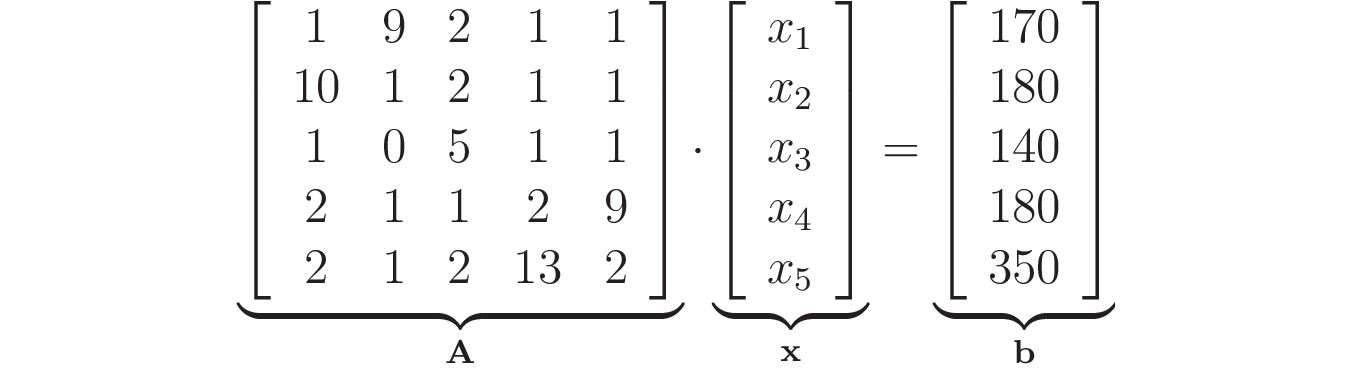
It’s worth noting that while non-zero numbers always have an inverse, not all matrices have an inverse. When the system has no solution or when it has multiple solutions, the determinant of **A** will be zero, and the inverse, **A**⁻¹, won’t exist.

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:



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 **x** = **A**⁻¹**b**, which you can do with the following program:

**Python**

1In [1]: import numpy as np

2 ...: from scipy import linalg

3

4In [2]: A = np.array(

5 ...: [

6 ...: [1, 9, 2, 1, 1],

7 ...: [10, 1, 2, 1, 1],

8 ...: [1, 0, 5, 1, 1],

9 ...: [2, 1, 1, 2, 9],

10 ...: [2, 1, 2, 13, 2],

11 ...: ]

12 ...: )

13

14In [3]: b = np.array([170, 180, 140, 180, 350]).reshape((5, 1))

15

16In [4]: A\_inv = linalg.inv(A)

17

18In [5]: x = A\_inv @ b

19 ...: x

20Out[5]:

21array([[10.],

22 [10.],

23 [20.],

24 [20.],

25 [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:

Python

In [6]: linalg.det(A)

Out[6]:

45102.0

As expected, the determinant isn’t zero. This indicates that the inverse of **A**, denoted as **A**⁻¹ and calculated with inv(A), exists, so the system has a unique solution. **A**⁻¹ is a square matrix with the same dimensions as **A**, so the product of **A**⁻¹ and **A** results in an identity matrix. In this example, it’s given by the following:

Python

In [7]: A\_inv

Out[7]:

array([[-0.01077558, 0.10655847, -0.03565252, -0.0058534 , -0.00372489],

[ 0.11287748, -0.00512172, -0.04010909, -0.00658507, -0.0041905 ],

[ 0.0052991 , -0.01536517, 0.21300608, -0.01975522, -0.0125715 ],

[-0.0064077 , -0.01070906, -0.02325839, -0.01376879, 0.08214713],

[-0.00931223, -0.01902355, -0.00611946, 0.1183983 , -0.01556472]])

Now that you know the basics of using matrix inverses and determinants, you’ll see how to use these tools to find the coefficients of polynomials.