Introduction to Python

Lecture Note

Mathematics 2023 – 2024

Imad A. Aziz

Introduction to Python Programming

Python is a modern, general-purpose, object-oriented, high-level programming language with a clean and expressive syntax. The following features make for easy code development and debugging in python:

- Python code is interpreted: There is no need to compile the code. Your code is read by a python interpreter and made into executable instructions for your computer in real time.
- Python is dynamically typed: There is no need to declare the type of a variable or the type of an input to a function.
- Python has automatic garbage collection or memory management: There is no need to explicitly allocate memory for variables before you use them or deallocate them after use.

However, keep in mind that these features also make pure python code slower (than, say C) in repetitious loops because of repeated checking for the type of objects. Therefore, many python modules (such as numpy, which we shall see in detail soon), have C or other compiled code, which is then wrapped in python to take advantage of python's usability without losing speed.

Installing Python

Go to www.python.org and download the latest version of Python. It should be painless to install. Then Go to https://www.jetbrains.com and download "PyCharm Community Edition" the IDE for Pure Python Development.

Input and Output

Output: - The print command

The fastest way to print the value of an object in python is with the print command. Python **print()** function prints the message to the screen or any other standard output device.

Example: -

```
name = "John"
age = 30
print("Name:", name)
print("Age:", age)
```

How print() works in Python?

You can pass variables, strings, numbers, or other data types as one or more parameters when using the print() function. Then, these parameters are represented as strings by their respective str() functions. To create a single output string, the transformed strings are concatenated with spaces between them.
In this code, we are passing two parameters name and age to the print function.

```
name = "Alice"
age = 25
print("Hello, my name is", name, "and I am", age, "years
old.")
```

or

```
name = "Alice"
age = 25
print("Hello, my name is{name} and I am {age} years old.")
```

Input: - The Input command
Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

input (prompt)

input (): This function first takes the input from the user and converts it into a string. The type of the returned object always will be <class 'str'>. It does not evaluate the

expression it just returns the complete statement as String. For example, Python provides a built-in function called input which takes the input from the user. When the input function is called it stops the program and waits for the user's input. When the user presses enter, the program resumes and returns what the user typed.

Example: -

```
val = input("Enter your value: ")
print(val)
```

Variable

In Python, variables are names that can be assigned a value and then used to refer to that value throughout your code.
Variables are fundamental to programming for two reasons:

1. Variables keep values accessible: For example, you can assign the result of some time-consuming operation to a variable so that your program doesn't have to perform the operation each time you need to use the result.
2. Variables give values context: The number 28 could mean lots of different things, such as the number of students in a class, the number of times a user has accessed a website, and so on. Giving the value 28 a name like *num_students* makes the meaning of the value clear.

Variable names

There are just a couple of rules to follow when naming your variables.
• Variable names can contain letters, numbers, and the underscore.
• Variable names cannot contain spaces.
• Variable names cannot start with a number.
• Case matters—for instance, temp and Temp are different.

Example: -

```
x=2
student_name=”arkan”
err1=0.002
```

The Assignment Operator

An operator is a symbol, such as +, that performs an operation on one or more values. For example, the + operator takes two numbers, one to the left of the operator and one to the right, and adds them together.

Values are assigned to variable names using a special symbol called the assignment operator (=) . The = operator takes the value to the right of the operator and assigns it to the name on the left.

Example: -

```
x=2
x = y = z = 6
x, y, z = 1, 2.39, 'cat'
(x, y, z )= (1, 2.39, 'cat')
```

Python Data Types

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instances (object) of these classes. The following are the standard or built-in data types in Python:

Numeric Data Type in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python.

Integers – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.

Float – This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.

Complex Numbers – Complex number is represented by a complex class. It is specified as (real part) + (imaginary part)j. For example – 2+3j

Example: -

```
a = 5
print("Type of a: ", type(a))
b = 5.0
print("\nType of b: ", type(b))
c = 2 + 4j
print("\nType of c: ", type(c))
```

Arithmetic Operators in Python

There are 7 arithmetic operators in Python. The lists are given below:

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power (Exponent): Returns first raised to power second	x ** y

Example: -

```
val1 = 2
val2 = 3
# using the addition operator
res = val1 + val2
print(res)
```

Precedence of Arithmetic Operators in Python

Let us see the precedence and associativity of Python Arithmetic operators.

Operator	Description	Associativity	
**	Exponentiation Operator	right-to-left	
%, *, /, //	Modulos, Multiplication, Division, and Floor Division	left-to-right	
+, –	Addition and Subtraction operators	left-to-right	

Variable modifying operators

Some additional arithmetic operators that modify variable values:

Operator	Effect	Equivalent to...
x += y	Add the value of y to x	x = x + y
x -= y	Subtract the value of y from x	x = x - y
x *= y	x *= y Multiply the value of x by y	x = x * y
x = x / y	x /= y Divide the value of x by y	x = x / y

Comparison Operators in Python

In Python Comparison of Relational operators compares the values. It either returns **True** or **False** according to the condition.

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

= is an assignment operator and == comparison operator.

Logical Operators in Python

Python Logical operators perform **Logical AND**, **Logical OR**, and **Logical NOT** operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	not x

Precedence of Logical Operators in Python

The precedence of Logical Operators in python is as follows:

1. Logical not
2. logical and
3. logical or

String Data Type

Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

Creating String

Strings in Python can be created using single quotes or double quotes or even triple quotes.

Example: -

```
String1 = 'Welcome to the Geeks World'
String2 = "I'm a Geek"
String3 = '''I'm a Geek and I live in a world of
"Geeks"'''
String4 = '''Geeks
            For
            Life'''
String5= String1+ String2
print(String1)
print(type(String1))
print(String2)
print(String3)
print(String4)
print(String5)
```

Accessing elements of String

In Python, individual characters of a String can be accessed by using the method of Indexing. Negative Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

Example: -

```
String1 = "GeeksForGeeks"
print("Initial String: ")
print(String1)
print("\nFirst character of String is: ")
print(String1[0])
# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
# Printing range of characters
print("\n characters of String is: ")
print(String1[2:5])
```

Python Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed some of them here.

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
bool()	Returns the Boolean value of the specified object
chr()	Returns a character from the specified Unicode code.
complex()	Returns a complex number

float()	Returns a floating point number
input()	Allowing user input
int()	Returns an integer number
len()	Returns the length of an object
list()	Returns a list
max()	Returns the largest item in an iterable
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
reversed()	Returns a reversed iterator
round()	Rounds a numbers
sorted()	Returns a sorted list
str()	Returns a string object
sum()	Sums the items of an iterator
tuple()	Returns a tuple

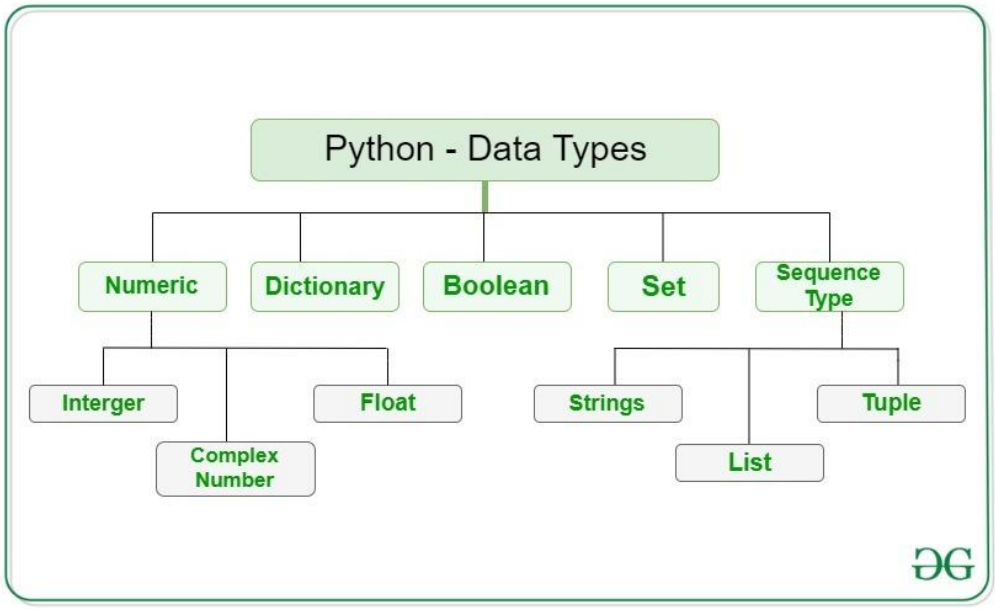
Example: -

```
x = input('x=')
y = input('y=')
z=int(x)+int(y)
print(z)
```

Python Data Types

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are classes and variables are instances (objects) of these classes. The following are the standard or built-in data types in Python:

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary
- Binary Types (memory view, byte array, bytes)



Sequence Data Type in Python

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence types in Python:

- Python String
- Python List
- Python Tuple

String Data Type

Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

Creating String

Strings in Python can be created using single quotes or double quotes or even triple quotes.

Example: -

```
String1 = 'Welcome to the Geeks World'
String2 = "I'm a Geek"
String3 = '''I'm a Geek and I live in a world of "Geeks"'''
String4 = '''Geeks
            For
            Life'''
String5= String1+ String2
print(String1)
print(type(String1))
print(String2)
print(String3)
print(String4)
print(String5)
```

Accessing elements of String

In Python, individual characters of a String can be accessed by using the method of Indexing. Negative Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

Example: -

```
String1 = "GeeksForGeeks"
print("Initial String: ")
print(String1)
print("\nFirst character of String is: ")
print(String1[0])
# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
# Printing range of characters
print("\n characters of String is: ")
print(String1[2:5])
```

List Data Type

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating List

Lists in Python can be created by just placing the sequence inside the square brackets[].

Example: -

```
List1 = []
print(List)
 List2 = ['GeeksForGeeks']
print(List2)
List3 = ["Geeks", "For", "Geeks"]
print(List3[0])
List4 = [2,5,8,3,-2,0,12]
print(List4[3])
```

Multi-Dimensional List:

Example: -

```
List1 = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12,
16, 20]]
print(List1)
print(List1[1])
print(List1[1][3])
```

Python Access List Items

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

Tuple Data Type

Just like a list, a tuple is also an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by a tuple class.

Creating a Tuple

In Python, tuples are created by placing a sequence of values separated by a ‘comma’ with or without the use of parentheses for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.). Note: Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing ‘comma’ to make it a tuple.

Example: -

```
# Creating an empty tuple
Tuple1 = ()
print("Initial empty Tuple: ")
```

```
print(Tuple1)
# Creating a Tuple with
# the use of Strings
Tuple1 = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple1)
# Creating a Tuple with
# the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))
# Creating a Tuple with the
# use of built-in function
Tuple1 = tuple('Geeks')
print("\nTuple with the use of function: ")
print(Tuple1)
# Creating a Tuple
# with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'geek')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

Access Tuple Items

In order to access the tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple. The index must be an integer. Nested tuples are accessed using nested indexing.

Example: -

```
# Python program to
# demonstrate accessing tuple
tuple1 = tuple([1, 2, 3, 4, 5])
# Accessing element using indexing
print("First element of tuple")
print(tuple1[0])
# Accessing element from last
# negative indexing
print("\nLast element of tuple")
print(tuple1[-1])
print("\nThird last element of tuple")
print(tuple1[-3])
```

Membership Operators in Python

In Python, in and not in are the membership operators that are used to test whether a value or variable is in a sequence.

Operators	Description
in	True if value is found in the sequence
not in	True if value is not found in the sequence

Example: -

```
a=[2,5,-2,5,0,3]
k=2 in a
print(k)
```

Methods in Python

In Python, methods are functions that are associated with an object and can manipulate its data or perform actions on it. They are called using dot notation, with the object name

followed by a period and the method name. Methods are an important part of object-oriented programming in Python.

String Methods

Python has a set of built-in methods that you can use on strings. They are listed some of them here.

Method	Description
count()	Returns the number of times a specified value occurs in a string
endswith()	Returns true if the string ends with the specified value
find()	Searches the string for a specified value and returns the position of where it was found
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isspace()	Returns True if all characters in the string are whitespaces
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
lower()	Converts a string into lower case
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
upper()	Converts a string into upper case

Example: -

```
txt = "I love applers, apple are my favorite fruit"
x = txt.count("apple")
print(x)
```

Example: -

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")
print(x)
```

Example: -

```
txt = "welcome to the jungle"
x = txt.split()
print(x)
```

List Methods

Python has a set of built-in methods that you can use on lists. They are listed some of them here.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position

pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Example: -

```
fruits = ['apple', 'banana', 'cherry']
fruits.append("orange")
```

Example: -

```
fruits = ['apple', 'banana', 'cherry']
fruits.insert(1, "orange")
```

Example: -

```
fruits = ['apple', 'banana', 'cherry']
cars = ['Ford', 'BMW', 'Volvo']
fruits.extend(cars)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Example: -

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)
```

Control Statements

if..else Statements

Usually statements in a program are executed one after another. However, there are situations when we have more than one option to choose from, based on the outcome of certain conditions. This can be done using if..else conditional statements. Conditional statements let us write program to do different tasks or take different paths based on the outcome of the conditions.

There are three ways to write if..else statements:

- **if Statement**

If the simple code of block is to be performed if the condition holds true then the if statement is used. Here the condition mentioned holds true then the code of the block runs otherwise not.

Syntax:

```
if condition:
   # Statements to execute if
   # condition is true
```

Example: To find y where $y = \sqrt{x}$

```
import math
x=float(input('x='))
if x>=0:
   y=math.sqrt(x)
print(y)
```

- **if..else Statement**

In conditional if Statement the additional block of code is merged as else statement which is performed when if condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
```

```
    # Executes this block if
    # condition is false
```

Example: To find y where $y = \begin{cases} x^2 & x < 0 \\ \sqrt{x} & x \leq 0 \end{cases}$

```
import math
x=float(input('x='))
if x>=0:
    y = math.sqrt(x)
else:
    y = x ** 2
print(y)
```

- **Nested if Statement**

if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and by this, we can see multiple conditions to be satisfied.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Example: To find y where $y = \sqrt{x}$ only for even numbers

```
import math
x=int(input('x='))
y=0
if x%2==0:
   if x>=0:
      y=math.sqrt(x)
print(y)
```

- **if-elif Statement**

The if-elif statement is shortcut of if..else chain. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

Syntax:

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

Example: To find y where $y = \begin{cases} x^2 & x < 0 \\ 2 & x = 0 \\ \sqrt{x} & x < 0 \end{cases}$

```
import math
x=float(input('x='))
if x>0:
    y = math.sqrt(x)
elif x<0:
    y = x ** 2
else:
    y=2
print(y)
```

for loops

for loops are used when you have a block of code which you want to repeat a fixed number of times. The for-loop is always used in combination with an iterable object, like a list or a range. The Python for statement iterates over the members of a sequence in order, executing the block each time. Contrast the for statement with the "while" loop, used when a condition needs to be checked each iteration or to repeat a block of code forever.

Example: Python For Loop using List

```
li = [2,-4,0,7]
for i in li:
   print(i)
print('-----')
```

Example: Python For Loop using Tuple

```
seq = (2,1,5)
for i in seq:
   print(i)
print('-----')
```

Example: Python For Loop using String

```
str = "Mathematics Department"
for i in str:
   print(i)
```

Example: Python For Loop using Range

```
for i in range(10):
   print(i)
print('----------')
for i in range(3,8):
   print(i)
print('----------')
for i in range(1,10,2):
   print(i)
print('----------')
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Example: To check n is prime or not

```
n=int(input('n='))
f=True
for i in range(2,n):
    if n%i==0:
        f=False
```

```
        break
if f:
    print('is prime')
else:
    print('is not prime')
```

Else with For loop

In most of the programming languages (C/C++, Java, etc), the use of else statement has been restricted with the if conditional statements. But Python also allows us to use the else condition with for loops.

Example: To check n is prime or not

```
n=int(input('n='))
for i in range(2,n):
    if n%i==0:
        print('is not prime')
        break
else:
    print('is prime')
```

Nested Loops

In Python programming language there are two types of loops which are for loop and while loop. Using these loops we can create nested loops in Python. Nested loops mean loops inside a loop. For example, while loop inside the for loop, for loop inside the for loop, etc.

Example:

```
for i in range(5):
  for j in range(3):
    print(f'({i},{j})')
```

Example: Find The sum of all elements in the List L=[[2,7,-1],[3,5,1],[2,0,-1]]

```
L=[[2,7,-1],[3,5,1],[2,0,-1]]
s=0
```

```
for row in L:
    for elm in row:
        s=s+elm
print(f'The sum of L is {s}')
    print(f'({i},{j})')
```

Python While Loop

Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

```
while expression:
    statement(s)
```

Example: To convert integer number n to binary

```
x=int(input('x='))
b=''
while x>0:
  b=str(x%2)+b
  x=x//2
print(b)
```

While loop with else

As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

Note: The else block just after for/while is executed only when the loop is NOT terminated by a break statement.

Example: To find the root of $x\text{^}2 - 1 = 0$ in range $[a, b]$ using bisection method

```
def f(x):
```

```
    y=x**2-1
    return y


a=float(input('a='))
b=float(input('b='))
e=0.0001
while abs(a-b)>e:
    c=(a+b)/2
    if f(a)*f(c)>0:
        a=c
    elif f(a)*f(c)<0:
        b=c
    else:
        print(f'{c} is exact root')
        break
else:
    print(f'{c} is approximate root')
```

Try Except

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Some of the common Exception Errors are :

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when the built-in function receives a wrong argument
- **EOFError:** if End-Of-File is hit without reading any data
- **ImportError:** if it is unable to find the module

Try and Except statement is used to handle these errors within our code in Python. The try block is used to check some code for errors i.e the code inside the try block will execute when there is no error in the program. Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax:

```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
```

Example: -

```
try:
    x=int(input('x='))
    y=1/x
    print(y)
except:
    print('invalid type')
```

if the error expected can write

```
try:
    x=int(input('x='))
    y=1/x
    print(y)
except ValueError:
    print('invalid type')
except ZeroDivisionError:
    print('divided by zero')
```

List Comprehension

A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.
Python List comprehension provides a much shorter syntax for creating a new list based on the values of an existing list. Here is an example of using list comprehension to find the square of the number in Python.

Example:

```
numbers = [1, 2, 3, 4, 5]
squared = [x ** 2 for x in numbers]
print(squared)
```

or

```
squared = [x ** 2 for x in [1, 2, 3, 4, 5]]
print(squared)
```

Python List Comprehension Syntax
Syntax: newList = [expression(element) for element in oldList if condition]

Parameter:
expression: *Represents the operation you want to execute on every item within the iterable.*
element: *The term "variable" refers to each value taken from the iterable.*
iterable: *specify the sequence of elements you want to iterate through. (e.g., a list, tuple, or string).*
condition: *(Optional) A filter helps decide whether or not an element should be added to the new list.*
Return: *The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.*

Iteration with List Comprehension
In this example, we are assigning 1, 2, and 3 to the list and we are printing the list using List Comprehension.

Example:

```
List = [c for c in [1, 2, 3]]
print(List)
```

Even list using List Comprehension
In this example, we are printing the even numbers from 0 to 10 using List Comprehension.

Example:

```
list = [i for i in range(11) if i % 2 == 0]
print(list)
```

Matrix using List Comprehension
In this example, we are assigning integers 0 to 2 to 3 rows of the matrix and printing it using List Comprehension.

Example:

```
matrix = [[j+i for j in range(3)] for i in range(3)]
print(matrix)
```

List Comprehensions vs For Loop
There are various ways to iterate through a list. However, the most common approach is to use the for loop. Let us look at the below example:

Example:

```
List = []
for character in 'Geeks 4 Geeks!':
    List.append(character)
print(List)
```

Above is the implementation of the traditional approach to iterate through a list, string, tuple, etc. Now, list comprehension in Python does the same task and also makes the program simpler.
List Comprehensions translate the traditional iteration approach using for loop into a simple formula hence making them easy to use. Below is the approach to iterate through a list, string, tuple, etc. using list comprehension in Python.

Example:

```
List = [character for character in 'Geeks 4 Geeks!']
print(List)
```

Conditionals in List Comprehension
We can also add conditional statements to the list comprehension. We can create a list using range(), operators, etc. and cal also apply some conditions to the list using the if statement.

Example:

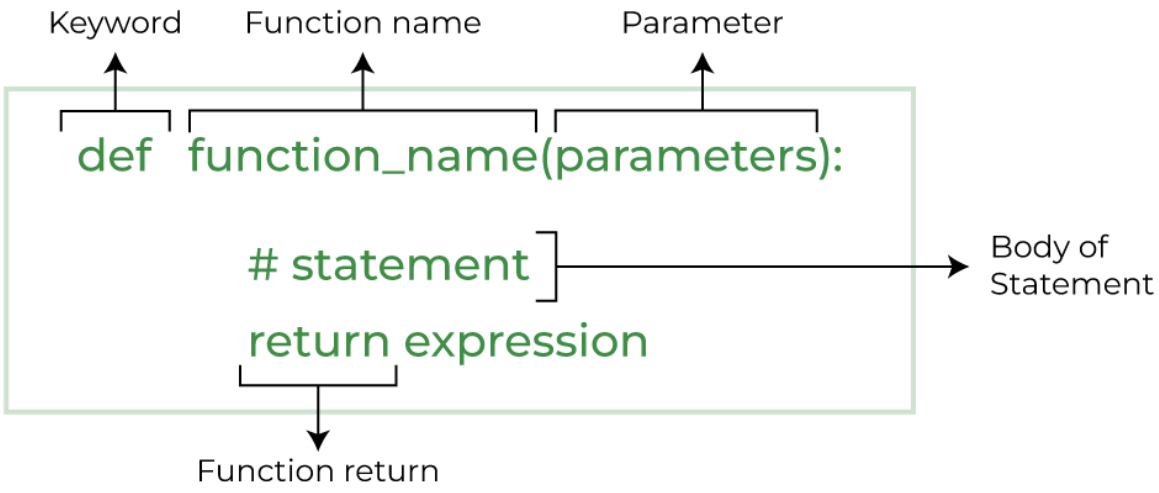
```
lis = ["Even number" if i % 2 == 0 else "Odd number" for i
in range(8)]
print(lis)
```

Function

Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again

Function Declaration

The syntax to declare a function is:



Example: - To create the function to check the number is prime or not.

```
def prim(x):
    p=True
    for k in range(2,x):
        if x%k==0:
            p=False
            break
    return p


x=int(input('x='))
L=prim(x)
print(L)
```

Defining and calling a function with parameters

Syntax

```
def fun_name(parameter: data_type) -> return_type:
    # body of the function
    return expression1
```

The following example uses arguments and parameters that you will learn later in this article so you can come back to it again if not understood.

Example: -

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2
    return num3

num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Example: - To find the volume of cylinder.

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v
```

```
r=int(input('r='))
h=int(input('h='))
v=cyl_vol(r,h)
print(v)
```

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- Default argument
- Keyword arguments (named arguments)
- Positional arguments

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

Example: -

```
def cyl_vol(r,h=5):
    from math import pi
    v=r**2*pi*h
    return v

v=cyl_vol(2)
print(v)
```

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Example: -

```
def cyl_vol(r,h):
    from math import pi
```

```
    v=r**2*pi*h
    return v


v1=cyl_vol(r=2,h=4)
print(v1)
v2=cyl_vol(h=4,r=2)
print(v2)
```

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to radius and the second argument (or value) is assigned to height. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places.

Example: -

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


v1=cyl_vol(2,4)
print(v1)
v2=cyl_vol(4,2)
print(v2)
```

Modules

In this section, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, from statements in Python, how we can use the alias to rename the module, etc.

What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

Let's create a simple cylin.py in which we define two functions, one volume and another surface area of cylinder

Example: -

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


def cyl_area(r,h):
    from math import pi
    a=2*r*pi*h+2*r**2*pi
    return a
```

Import module in Python

We can import the functions, and classes defined in a module to another module using the import statement in some other Python source file.

```
import cylin
a=cylin.cyl_area(2,4)
```

Python Import from Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

```
from cylin import cyl_area
a=cyl_area(2,4)
```

Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

```
from cylin import *
```

```
a=cyl_area(2,4)
v=cyl_vol(2,4)
```

Renaming the Python module

We can rename the module while importing it using the keyword.

```
import cylin as c
a=c.cyl_area(2,4)
```

Packages

We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently. For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the packages in Python.

What is a Python Package?

Python modules may contain several classes, functions, variables, etc. whereas Python packages contain several modules. In simpler terms, Package in Python is a folder that contains various modules as files.

Creating Package

Let's create a package in Python named mypckg that will contain two modules mod1 and mod2. To create this module, follow the below steps:

- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. __init__.py
- Then create two modules mod1 and mod2 in this folder.

Import Modules from a Package

We can import these Python modules using the from…import statement and the dot(.) operator.

Syntax:

```
import package_name.module_name
```

Object-Oriented Programming

Try Except

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Some of the common Exception Errors are :

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when the built-in function receives a wrong argument
- **EOFError:** if End-Of-File is hit without reading any data
- **ImportError:** if it is unable to find the module

Try and Except statement is used to handle these errors within our code in Python. The try block is used to check some code for errors i.e the code inside the try block will execute when there is no error in the program. Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax:

```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
```

Example: -

```
try:
    x=int(input('x='))
    y=1/x
    print(y)
except:
    print('invalid type')
```

if the error expected can write

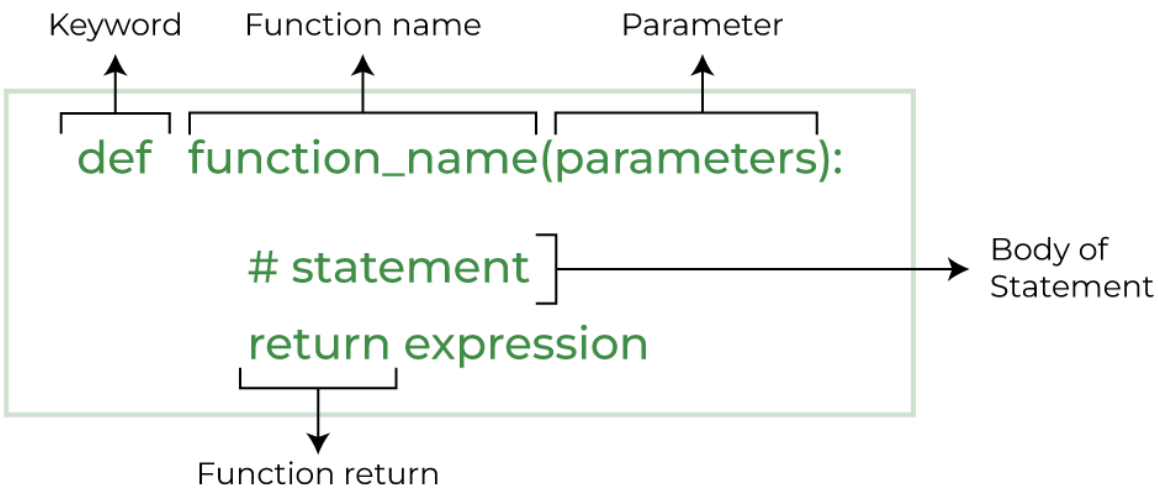
```
try:
    x=int(input('x='))
    y=1/x
    print(y)
except ValueError:
    print('invalid type')
except ZeroDivisionError:
    print('divided by zero')
```

Functions

Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again

Function Declaration

The syntax to declare a function is:



Example: - To create the function to check the number is prime or not.

```
def prim(x):
    p=True
```

```
    for k in range(2,x):
        if x%k==0:
            p=False
            break
    return p


x=int(input('x='))
L=prim(x)
print(L)
```

Defining and calling a function with parameters

Syntax

```
def fun_name(parameter: data_type) -> return_type:
    # body of the function
    return expression1
```

The following example uses arguments and parameters that you will learn later in this article so you can come back to it again if not understood.

Example: -

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2
    return num3


num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Example: - To find the volume of cylinder.

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


r=int(input('r='))
h=int(input('h='))
v=cyl_vol(r,h)
print(v)
```

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- Default argument
- Keyword arguments (named arguments)
- Positional arguments

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

Example: -

```
def cyl_vol(r,h=5):
    from math import pi
    v=r**2*pi*h
    return v
```

```
v=cyl_vol(2)
print(v)
```

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Example: -

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


v1=cyl_vol(r=2,h=4)
print(v1)
v2=cyl_vol(h=4,r=2)
print(v2)
```

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to radius and the second argument (or value) is assigned to height. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places.

Example: -

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


v1=cyl_vol(2,4)
print(v1)
v2=cyl_vol(4,2)
print(v2)
```

Modules

In this section, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, from statements in Python, how we can use the alias to rename the module, etc.

What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

Let's create a simple cylin.py in which we define two functions, one volume and another surface area of cylinder

Example: -

```
def cyl_vol(r,h):
    from math import pi
    v=r**2*pi*h
    return v


def cyl_area(r,h):
    from math import pi
    a=2*r*pi*h+2*r**2*pi
    return a
```

Import module in Python

We can import the functions, and classes defined in a module to another module using the import statement in some other Python source file.

```
import cylin
a=cylin.cyl_area(2,4)
```

Python Import from Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

```
from cylin import cyl_area
a=cyl_area(2,4)
```

Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

```
from cylin import *
a=cyl_area(2,4)
v=cyl_vol(2,4)
```

Renaming the Python module

We can rename the module while importing it using the keyword.

```
import cylin as c
a=c.cyl_area(2,4)
```

Packages

We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently. For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the packages in Python.

What is a Python Package?

Python modules may contain several classes, functions, variables, etc. whereas Python packages contain several modules. In simpler terms, Package in Python is a folder that contains various modules as files.

Creating Package

Let's create a package in Python named mypckg that will contain two modules mod1 and mod2. To create this module, follow the below steps:

- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. __init__.py
- Then create two modules mod1 and mod2 in this folder.

Import Modules from a Package

We can import these Python modules using the from…import statement and the dot(.) operator.

Syntax:

```
import package_name.module_name
```

Object-Oriented Programming

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Syntax: Class Definition

```
class ClassName:
    # Statement
```

Syntax: Object Definition

```
obj = ClassName()
print(obj.atrr)
```

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.

Creating a Python Class
Here, the class keyword indicates that you are creating a class followed by the name of the class

Example: -

```
class Point:
    def move(self):
        print('move method of point')
    def draw(self):
        print('draw method of point')


p=Point()
p.move()
```

Object of Python Class

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Self Parameter

Self represents the instance of the class. By using the "self" we can access the attributes and methods of the class in Python. It binds the attributes with the given arguments.

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.

Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Example: -

```
 class Point:
    xorigin=0
    yorigin=0
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def move(self):
        print('move method of point')

    def draw(self):
        print('draw method of point')


p=Point(3,2)
print(p.xorigin)
print(p.x)
p.draw()
```

Pass Statement

The program's execution is unaffected by the pass statement's inaction. It merely permits the program to skip past that section of the code without doing anything. It is

frequently employed when the syntactic constraints of Python demand a valid statement but no useful code must be executed.

Syntax

```
class MyClass:
    pass
```

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Introduction to NumPy

NumPy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays. It is the fundamental package for scientific computing with Python. It is open-source software.

Features of NumPy

NumPy has various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy in Python can also be used as an efficient multi-dimensional container of generic data. Arbitrary data types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

install NumPy in PyCharm

To install the Python NumPy package in PyCharm. NumPy is a general-purpose array-processing Python package. It provides a high-performance multidimensional array object, and tools for working with these arrays. As one of the most important tool while working with data manipulation, we often need to install NumPy package. It's a third party module, i.e. we need to install it separately. There can be multiple ways of installing NumPy package. While using PyCharm the easiest way to install is using PyCharm User interface, by following the steps as discussed below:

NumPy Array Creation

There are various ways of Numpy array creation in Python. They are as follows:

1. You can create an array from a regular Python list or tuple using the array() function. The type of the resulting array is deduced from the type of the elements in the sequences. Let's see this implementation:

Example: -

```
import numpy as np
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)
```

2. Often, the element is of an array is originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation. For example: np.zeros, np.ones, np.full, np.empty, etc.

To create sequences of numbers, NumPy provides a function analogous to the range that returns arrays instead of lists.

Example: -

```
c = np.zeros((3, 4))
print ("An array initialized with all zeros:\n", c)
d = np.full((3, 3), 6, dtype = 'complex')
print ("An array initialized with all 6s.","Array type is
complex:\n", d)
e = np.random.random((2, 2))
print ("A random array:\n", e)
```

3. arange: This function returns evenly spaced values within a given interval. Step size is specified.

Example: -

```
f = np.arange(0, 30, 5)
print ("A sequential array with steps of 5:\n", f)
```

4. linspace: It returns evenly spaced values within a given interval.

Example: -

```
g = np.linspace(0, 5, 10)
print ("A sequential array with 10 values between","0 and
5:\n", g)
```

5. Reshaping array: We can use reshape method to reshape an array.

Example: -

```
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])
newarr = arr.reshape(2, 2, 3)
print ("Original array:\n", arr)
print("---------------")
print ("Reshaped array:\n", newarr)
```

6. Flatten array: We can use flatten method to get a copy of the array collapsed into one dimension.

Example: -

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
flat_arr = arr.flatten()
print ("Original array:\n", arr)
print ("Fattened array:\n", flat_arr)
```

Numpy Array Indexing

Knowing the basics of NumPy array indexing is important for analyzing and manipulating the array object. NumPy in Python offers many ways to do array indexing.

- Slicing: Just like lists in Python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.

- Integer array indexing: In this method, lists are passed for indexing for each dimension. One-to-one mapping of corresponding elements is done to construct a new arbitrary array.
- Boolean array indexing: This method is used when we want to pick elements from the array which satisfy some condition.

Example: -

```
import numpy as np
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])
temp = arr[:2, ::2]
print ("Array with first 2 rows and alternate","columns(0
and 2):\n", temp)
temp = arr[[0, 1, 2, 3], [3, 2, 1, 0]]
print ("\nElements at indices (0, 3), (1, 2), (2,
1),","(3, 0):\n", temp)
cond = arr > 0 # cond is a boolean array
temp = arr[cond]
print ("\nElements greater than 0:\n", temp)
```

Indexing using index arrays

Indexing can be done in numpy by using an array as an index. In case of slice, a view or shallow copy of the array is returned but in index array a copy of the original array is returned. Numpy arrays can be indexed with other arrays or any other sequence with the exception of tuples. The last element is indexed by -1 second last by -2 and so on.

Example: -

```
import numpy as np
a = np.arange(10, 1, -2)
print("\n A sequential array with a negative step: \n",a)
```

```
newarr = a[np.array([3, 1, 2 ])]
print("\n Elements at these indices are:\n",newarr)
```

Example: -

```
import numpy as np
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
arr = x[np.array([1, 3, -3])]
print("\n Elements are : \n",arr)
```

Types of Indexing

There are two types of indexing:

Basic Slicing and indexing: Consider the syntax x[obj] where x is the array and obj is the index. Slice object is the index in case of basic slicing. Basic slicing occurs when obj is:

- a slice object that is of the form start : stop : step
- an integer
- or a tuple of slice objects and integers

All arrays generated by basic slicing are always view of the original array.

Example: -

```
import numpy as np
a = np.arange(20)
print("\n Array is:\n ",a)
print("\n a[-8:17:1]  = ",a[-8:17:1])
print("\n a[10:]  = ",a[10:])
```

Example: -

```
import numpy as np
a = np.array([[0, 1, 2, 3, 4, 5]
              [6, 7, 8, 9, 10, 11]
              [12, 13, 14, 15, 16, 17]
              [18, 19, 20, 21, 22, 23]
```

```
                [24, 25, 26, 27, 28, 29]
                [30, 31, 32, 33, 34, 35]]
print("\n Array is:\n ",a)
print("\n a[0, 3:5]  = ",a[0, 3:5])
print("\n a[4:, 4:]  = ",a[4:, 4:])
print("\n a[:, 2]  = ",a[:, 2])
print("\n a[2:;2, ::2]  = ",a[2:;2, ::2])
```

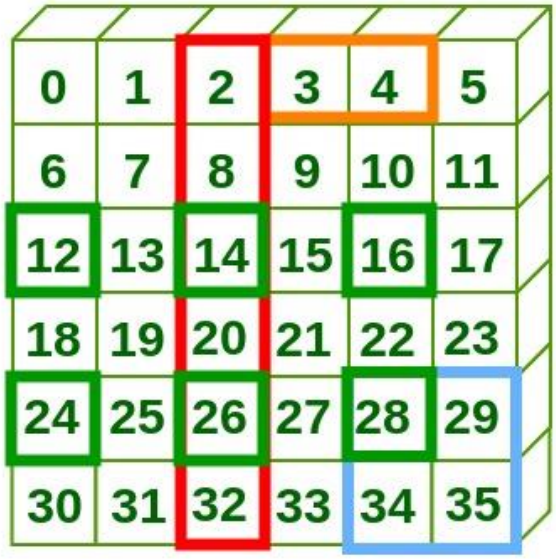
The figure below makes the concept more clear:

```
>>> a[0,3:5]
array( [3,4] )

>>> a[4:, 4:]
array( [ 28, 29],
       [ 34, 35] ] )

>>> a[ :, 2]
array( [2, 8, 14, 20, 26, 32] )

>>> a[2 : : 2, : : 2]
array( [ 12, 14, 16],
       [ 24, 26, 28] ] )
```



Advanced indexing : Advanced indexing is triggered when obj is –

- an ndarray of type integer or Boolean
- or a tuple with at least one sequence object
- is a non tuple sequence object

Advanced indexing returns a copy of data rather than a view of it. Advanced indexing is of two types integer and Boolean.

Purely integer indexing : When integers are used for indexing. Each element of first dimension is paired with the element of the second dimension. So the index of the elements in this case are (0,0),(1,0),(2,1) and the corresponding elements are selected.

Example: -

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])
print(a[[0,1,2],[0,0,1]])
```

Boolean Array Indexing:

This indexing has some boolean expression as the index. Those elements are returned which satisfy that Boolean expression. It is used for filtering the desired element values.

Example: -

```
import numpy as np
a = np.array([10, 40, 80, 50, 100])
print(a[a>50])
```

Example: -

```
import numpy as np
a = np.array([10, 40, 80, 50, 100])
print(a[a%40==0]**2)
```

Example: -

```
import numpy as np
b = np.array([[5, 5],[4, 5],[16, 4]])
sumrow = b.sum(-1)
print(b[sumrow%10==0])
```

Basic Array Operations and Binary Operators

In numpy, arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays. These operations include some basic Mathematical operation as well as Unary and Binary operations.

Example: -

```
import numpy as np
```

```
 a = np.array([[1, 2],
               [3, 4]])
b = np.array([[4, 3],
               [2, 1]])
print ("Adding 1 to every element:", a + 1)
print ("\nSubtracting 2 from each element:", b - 2)
print ("\nArray sum:\n", a + b)
print ("Array multiplication:\n", a*b)
print ("Matrix multiplication:\n", a.dot(b))
```

Numpy Mathematical Function

NumPy contains a large number of various mathematical operations. NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

Example: -

```
import numpy as np
a = np.array([0, np.pi/2, np.pi])
print ("Sine values of array elements:", np.sin(a))
a = np.array([0, 1, 2, 3])
print ("Exponent of array elements:", np.exp(a))
print ("Square root of array elements:", np.sqrt(a))
```

Unary operators

Many unary operations are provided as a method of ndarray class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter.

Example: -

```
import numpy as np
arr = np.array([[1, 5, 6],
                [4, 7, 2],
```

```
                [3, 1, 9]])
print ("Largest element is:", arr.max())
print ("Row-wise maximum elements:",arr.max(axis = 1))
print ("Column-wise minimum elements:",arr.min(axis = 0))
print ("Sum of all array elements:",arr.sum())
```

Numpy Linear Algebra

The Linear Algebra module of NumPy offers various methods to apply linear algebra on any numpy array. One can find: rank, determinant, trace, etc. of an array. eigen values of matrices matrix and vector products (dot, inner, outer,etc. product), matrix exponentiation, solve linear or tensor equations and much more.

Example: -

```
import numpy as np
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])
print("Rank of A:", np.linalg.matrix_rank(A))
print("\nTrace of A:", np.trace(A))
print("\nDeterminant of A:", np.linalg.det(A))
print("\nInverse of A:\n", np.linalg.inv(A))
print("\nMatrix A raised to power 3:\n",
np.linalg.matrix_power(A, 3))
print("transpose of A:",A. transpose())
```

Matrix eigenvalues Functions

numpy.linalg.eigh(a, UPLO='L') : This function is used to return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.Returns two objects, a 1-D array containing the eigenvalues of a, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

Example: -

```
from numpy import linalg as geek
a = np.array([[1, -2j], [2j, 5]])
print("Array is :",a)
c, d = geek.eigh(a)
print("Eigen value is :", c)
print("Eigen value is :", d)
```

numpy.linalg.eig(a) : This function is used to compute the eigenvalues and right eigenvectors of a square array.

Example: -

```
from numpy import linalg as geek
a = np.diag((1, 2, 3))
print("Array is :",a)
c, d = geek.eig(a)
print("Eigen value is :",c)
print("Eigen value is :",d)
```

Solving equations and inverting matrices
numpy.linalg.solve() : Solve a linear matrix equation, or system of linear scalar equations.Computes the “exact” solution, x, of the well-determined, i.e., full rank, linear matrix equation ax = b.

Example: -

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([8, 18])
print(("Solution of linear equations:",np.linalg.solve(a,
b)))
```

Introduction to pyplot

matplotlib.pyplot is a collection of functions that make ***matplotlib*** work like MATLAB. Each ***pyplot*** function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In ***matplotlib.pyplot*** various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the documentation refers to the axes part of a figure and not the strict mathematical term for more than one axis).

Example: -Generating visualizations with pyplot is very quick:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to ***plot***, ***matplotlib*** assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0; therefore, the x data are [0, 1, 2, 3].

plot is a versatile function, and will take an arbitrary number of arguments. For example, to plot x versus y, you can write:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

Formatting the style of your plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style

string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

Example: -

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis((0, 6, 0, 20))
plt.show()
```

See the plot documentation for a complete list of line styles and format strings. The axis function in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one function call using arrays.

Example: -

```
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(0., 5., 0.2)
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

The basic functions

Let's have a look at some of the basic functions that are often used in matplotlib.

Method	Description
plot()	it creates the plot at the background of computer, it doesn't displays it. We can also add a label as it's argument that by what name we will call this plot – utilized in legend()
show()	it displays the created plots
xlabel()	it labels the x-axis
ylabel()	it labels the y-axis
title()	it gives the title to the graph
xticks()	it decides how the markings are to be made on the x-axis

yticks()	it decides how the markings are to be made on the y-axis
legend()	pass a list as it's arguments of all the plots made, if labels are not explicitly specified then add the values in the list in the same order as the plots are made

Example: -

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 2 * (np.pi), 0.1)
y1 = np.sin(x)
y2=np.cos(x)
plt.plot(x, y1,'r:')
plt.plot(x, y2,'b--')
plt.xlabel('Time')
plt.ylabel('Speed')
plt.legend(['car','plane'])
plt.axis((-1,6,-2,2))
plt.title('The Graph')
plt.show()
```

Customization of Plots

Here, we discuss some elementary customizations applicable to almost any plot.

Example: -

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 2 * (np.pi), 0.5)
y = np.sin(x)
plt.plot(x, y,color='green', linestyle='dashed', linewidth
= 3,
         marker='o', markerfacecolor='blue',
markersize=12)
plt.show()
```

Subplots

Subplots are required when we want to show two or more plots in same figure. We can do it in two ways using two slightly different methods.

Example: -

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-2, 2, 0.1)
y1=x
y2=x*2
y3=x**2
y4=x**3
fig, ax = plt.subplots(2, 2)
ax[0, 0].plot(x, y1)
ax[0, 1].plot(x, y2)
ax[1, 0].plot(x, y3)
ax[1, 1].plot(x, y4)
ax[0, 0].set_title("Linear")
ax[0, 1].set_title("Double")
ax[1, 0].set_title("Square")
ax[1, 1].set_title("Cube")
fig.tight_layout()
plt.show()
```