



Python for Beginners

Srinivasa.R
Bangalore, India

Table of contents

Basics: Variables, data types, operators, control flow, and functions.

Data Structures: Lists, tuples, dictionaries, and comprehensions.

OOP: Classes, inheritance, polymorphism, encapsulation, and abstraction.

File Handling & Modules: Reading/writing files, importing modules, and creating packages.

Error Handling & Debugging: Try-except blocks, custom exceptions, and logging.

Intermediate Python: Lambda functions, decorators, generators, and concurrency.

Advanced Topics: Python standard library, metaprogramming, performance optimization.

Specialized Areas: Databases, data analysis, visualization, machine learning.

Basic overview

Strings in Python are surrounded by either single quotation marks, or double quotation marks.

'Hello' is the same as "Hello".

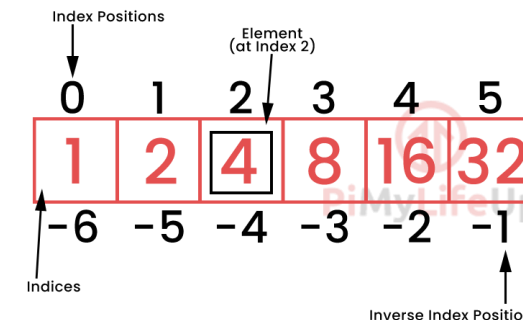
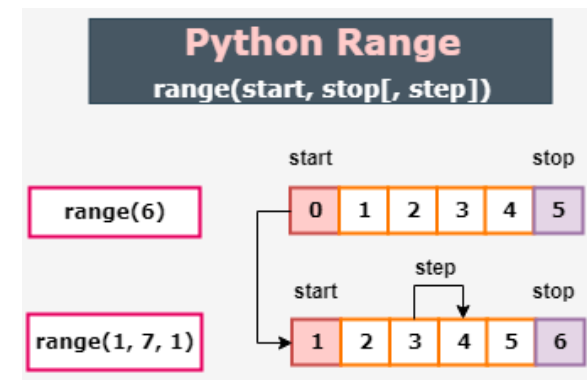
You can assign a multiline string to a variable by using three quotes or three single quotes
you write Python (.py) les in a text editor

Comments starts with a #, and Python will ignore them: for multiple Ctrl+/
Ctrl+Z

[ASCII table - Table of ASCII codes, characters and symbols](#)

[Python - String Methods](#)

 H e l l o
+0 1 2 3 4 5
 →
-6 5 4 3 2 1
←



Escape Characters

\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Python Editors:

Python IDLE
Visual Studio Code
Spyder
Visual Studio
PyCharm
Wing Python IDE
Jupyter Notebook

15 decimal places
ASCII value
Case sensitive

Types

1 x = 1 # i n t
2 y = 2. 8 # f l o a t
3 z = 3 + 2 j # c o m p l e x
4 x = 'Hello' # s t r i n g

Data Variables

Python Variables are memory spaces that hold information, such as numbers or text, that the Python Code can later use to perform tasks.

1. The variable name should start with an underscore or letter.

Example: `_educba`, `xyz`, `ABC`

2. Using a number at the start of a variable name is not allowed.

Examples of incorrect variable names: `1variable`, `23alpha`, `390say`

3. The variable name can only include alphanumeric characters and underscore.

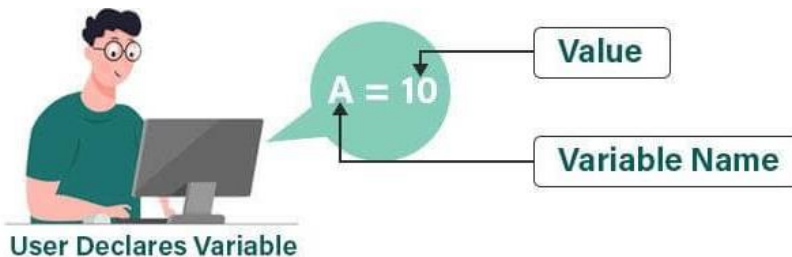
Example: `learn_educba`, `c5`, `A777_21`

4. Variable names in Python are case-sensitive.

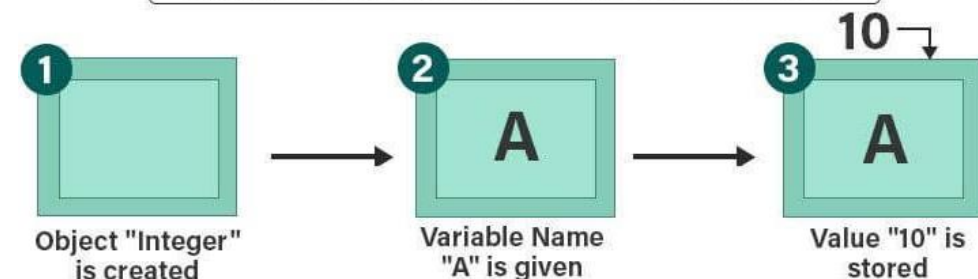
Examples of different variable names: `educba`, `Educba`, `EDUCBA`

5. Reserved words in Python cannot be a variable name.

Example: `while`, `if`, `print`, `while`



WHAT HAPPENS IN PYTHON MEMORY?



```
a = [1,2,3,4,5]
a1 = (1,2,3,4,5)
a3 = {1,2,3,4,5}
a4 = {'a':1,'b':2,'c':3,'d':4,'e':5}
a5 = {
    'Name': Vishnu,
    'Class': First std,
    'DOB': May 7th
}
b = int(input('Please enter value = '))
c = float(input('please enter value = '))
d = input('please enter value = ')
arr = [1,2,3]
arr1 = [[1,2,3],[3,4,5],[6,7,8]]
arr2 = [1,2,3,4]
x, y, z = 1, 2, 3
x1[], y1[], z1[] = 0
x2, y2, z2 = []
x3, y3, z3 = "Orange", "Banana", "Cherry"
fruits = ["apple", "banana", "cherry"]
```


Data Types

•Numeric Types:

- int: Represents whole numbers (integers), e.g., 5, -10.
- float: Represents numbers with a decimal point (floating-point numbers), e.g., 3.14, -0.5.
- complex: Represents complex numbers with a real and imaginary part, e.g., 1 + 2j.

•Sequence Types:

- str: Represents sequences of characters (strings), enclosed in single or double quotes, e.g., "hello", 'Python'.
- list: Represents ordered, mutable sequences of items, enclosed in square brackets, e.g., [1, 2, "three"].
- tuple: Represents ordered, immutable sequences of items, enclosed in parentheses, e.g., (1, "two", 3.0).
- range: Represents an immutable sequence of numbers, often used in loops.

•Mapping Type:

- dict: Represents unordered collections of key-value pairs (dictionaries), enclosed in curly braces, e.g., {"name": "Alice", "age": 30}.

•Set Types:

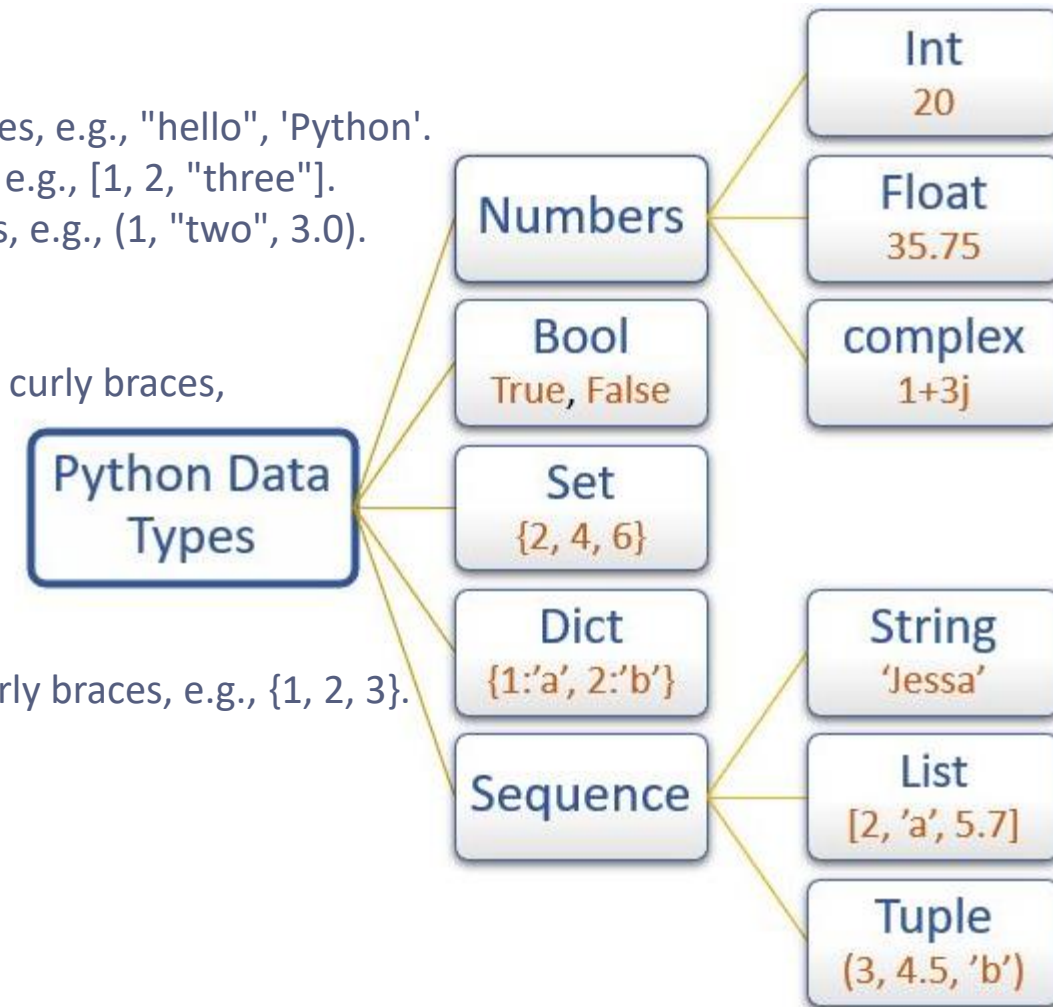
- set: Represents unordered collections of unique, mutable elements, enclosed in curly braces, e.g., {1, 2, 3}.
- frozenset: Represents unordered collections of unique, immutable elements.

•Boolean Type:

- bool: Represents truth values, either True or False.

•Binary Types:

- bytes: Represents immutable sequences of bytes.
- bytearray: Represents mutable sequences of bytes.
- memoryview: Provides a memory-efficient way to access the internal data of an object without copying it.



Data Operators

Comparison

==	Equal	<code>x == y</code>
!=	Not equal	<code>x != y</code>
>	Greater than	<code>x > y</code>
<	Less than	<code>x < y</code>
>=	Greater than or equal to	<code>x >= y</code>
<=	Less than or equal to	<code>x <= y</code>

+	Addition	<code>x + y</code>
-	Subtraction	<code>x - y</code>
*	Multiplication	<code>x * y</code>
/	Division	<code>x / y</code>
%	Modulus	<code>x % y</code>
**	Exponentiation	<code>x ** y</code>
//	Floor division	<code>x // y</code>

Arithmetic

in	Returns True if a sequence with the specified value is present in the object	<code>x in y</code>
not in	Returns True if a sequence with the specified value is not present in the object	<code>x not in y</code>

Membership

&	AND	Sets each bit to 1 if both bits are 1	<code>x & y</code>
	OR	Sets each bit to 1 if one of two bits is 1	<code>x y</code>
^	XOR	Sets each bit to 1 if only one of two bits is 1	<code>x ^ y</code>
~	NOT	Inverts all the bits	<code>~x</code>
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	<code>x << 2</code>
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	<code>x >> 2</code>

Bitwise

is	Returns True if both variables are the same object	<code>x is y</code>
is not	Returns True if both variables are not the same object	<code>x is not y</code>

Identity

and	Returns True if both statements are true	<code>x < 5 and x < 10</code>
or	Returns True if one of the statements is true	<code>x < 5 or x < 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

Logical

Assignment Operators

=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 3</code>	<code>x = x + 3</code>
-=	<code>x -= 3</code>	<code>x = x - 3</code>
*=	<code>x *= 3</code>	<code>x = x * 3</code>
/=	<code>x /= 3</code>	<code>x = x / 3</code>
%=	<code>x %= 3</code>	<code>x = x % 3</code>
//=	<code>x //= 3</code>	<code>x = x // 3</code>
**=	<code>x **= 3</code>	<code>x = x ** 3</code>
&=	<code>x &= 3</code>	<code>x = x & 3</code>
=	<code>x = 3</code>	<code>x = x 3</code>
^=	<code>x ^= 3</code>	<code>x = x ^ 3</code>
>>=	<code>x >>= 3</code>	<code>x = x >> 3</code>
<<=	<code>x <<= 3</code>	<code>x = x << 3</code>
:=	<code>print(x := 3)</code>	<code>x = 3</code>

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Control flow

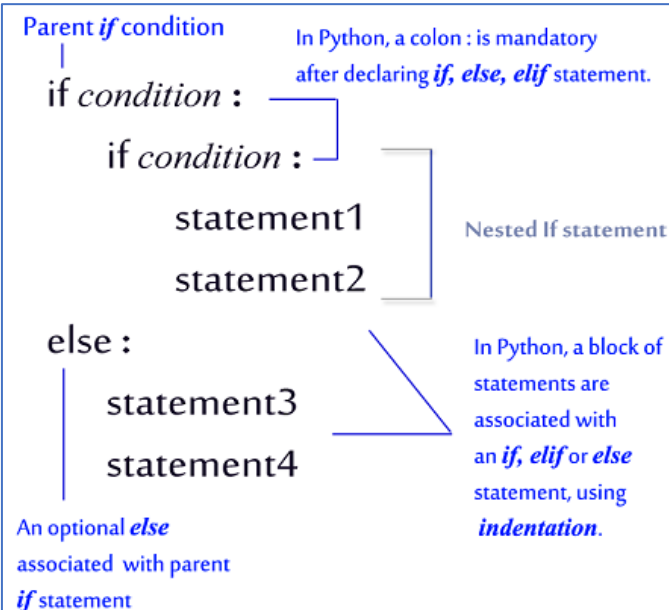
If statement

The primary keywords used for conditional statements in Python are `if`, `elif` (short for "else if"), and `else`. Each `if`, `elif`, and `else` statement must end with a colon (:).

if statement:

if-else statement:

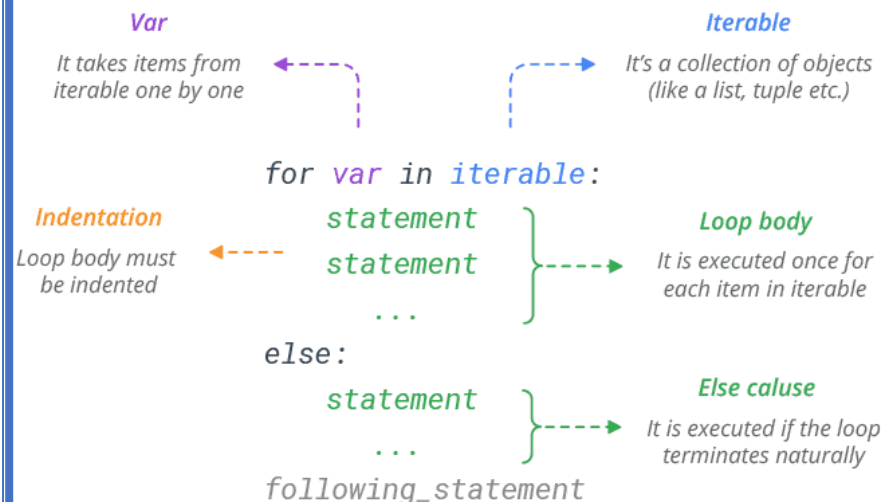
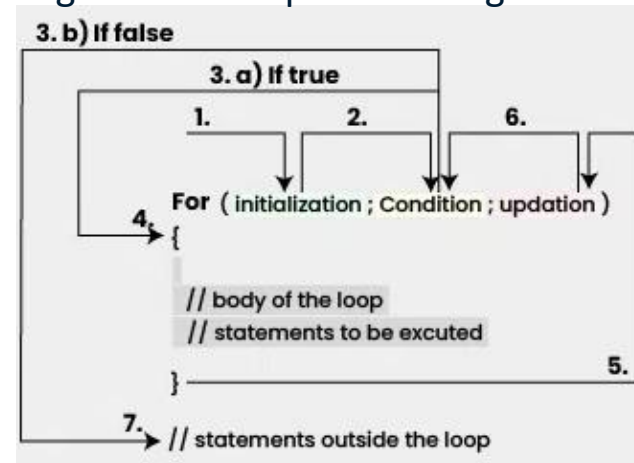
if-elif-else statement (Elif Ladder):



For loop

Known Iterations:

The for loop is primarily used for iterating over a sequence (like a list, tuple, string, or range) or other iterable objects. It is suitable when the number of iterations is known or determined by the length of the sequence being traversed.



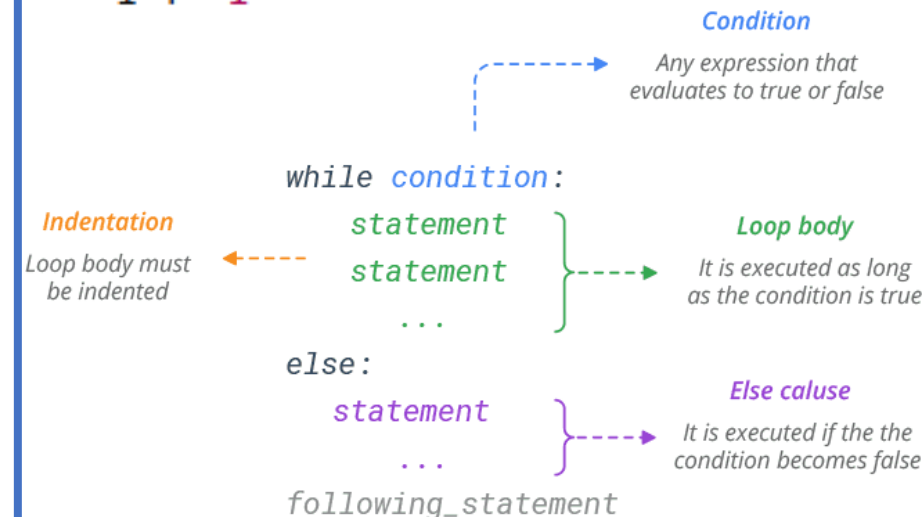
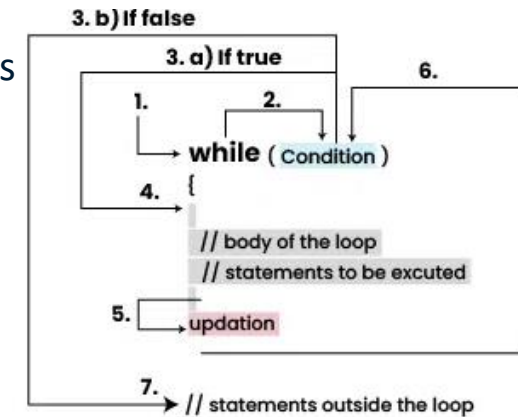
While loop

Unknown Iterations:

The while loop is used to repeatedly execute a block of code as long as a specified condition remains true. It is suitable when the number of iterations is unknown beforehand and depends on a condition being met or becoming false.

number of iterations is dependent on a dynamic condition.

```
i = 1
while i < 6:
    print(i)
    i += 1
```



Data Built in functions

abs():	Returns the absolute value of a number.
bool():	Converts a value to a Boolean (True or False).
chr():	Returns the character represented by a Unicode code point.
dir():	Returns a list of names in the current scope or the attributes of an object.
format():	Formats a specified value.
frozenset():	Returns a new frozenset object (an immutable set).
getattr():	Returns the value of a named attribute of an object.
hasattr():	Returns True if an object has a given named attribute.
hex():	Converts an integer to a hexadecimal string.
isinstance():	Returns True if an object is an instance of a specified class or of a subclass thereof.
issubclass():	Returns True if a specified class is a subclass of another specified class.
max():	Returns the largest item in an iterable or the largest of two or more arguments.
min():	Returns the smallest item in an iterable or the smallest of two or more arguments.
object():	Returns a new featureless object.
ord():	Returns the Unicode code point for a one-character string.
pow():	Returns the result of raising a number to a power.
repr():	Returns a string containing a printable representation of an object.
reversed():	Returns a reverse iterator.
round():	Rounds a number to a specified number of decimal places.
sum():	Sums the items of an iterable.
vars():	Returns the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute.
zip():	Creates an iterator that aggregates elements from multiple iterables.
map():	Applies a given function to each item of an iterable and returns an iterator.
filter():	
reduce():	
sorted():	Returns a new sorted list from the items in an iterable
next():	Retrieves the next item from an iterator.
all():	Returns True if all elements of an iterable are true (or if the iterable is empty).
any():	Returns True if any element of an iterable is true.
enumerate():	Returns an enumerate object, which yields pairs of (index, item).
map(), filter():	For applying functions to iterables.
Uppercase()	
Lowercase()	

dict():	Creates a new dictionary.
float():	Converts a value to a floating-point number.
help():	
id():	Returns the identity of an object.
input():	Reads a line from input, converts it to a string, and returns it.
int():	Converts a value to an integer.
len():	Returns the length (the number of items) of an object.
list():	Creates a new list.
open():	Opens a file and returns a corresponding file object.
print():	Prints objects to the text stream file, separated by sep and followed by end.
range():	Returns a sequence of numbers.
str():	Converts a value to a string.
tuple():	Creates a new tuple.
type():	Returns the type of an object.
set():	Creates a new set.
append():	Adds an element to the end of the list.
copy():	Returns a shallow copy of the list.
clear():	Removes all elements from the list.
count():	Returns the number of times a specified element appears in the list.
index():	Returns the index of the first occurrence of a specified element.
insert():	Inserts an element at a specified position.
pop():	Removes and returns the element at the specified position (or the last element if not specified).
remove():	Removes the first occurrence of a specified element.
reverse():	Reverses the order of the elements in the list.
sort():	Sorts the list in ascending order (by default).

Data structure - List, Tuple, set and dictionary

[] is a list, and is *mutable*, in that its size can vary

List is a collection which is ordered and changeable. Allows duplicate members.

() is a tuple. Tuples are *immutable*, in that their sizes cannot vary.

Immutable objects can be hashed, which is an important property.

You can concatenate tuples, but it returns a *new* tuple,

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

{ } is a set, in that it contains *unique, immutable* objects. { } is also used to create a dictionary, where the dictionary has a *set* of keys

Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members..

{ } is a dictionary. You can store key value pairs in it e.g. Person = {'name': one_loop, 'age':69}

Dictionary is a collection which is ordered** and changeable. No duplicate members. (e.g., strings, numbers, tuples),

	Lists	Tuples	Sets	Dictionaries
Ordering	Ordered	Ordered	Unordered	Ordered <small>Unordered before Python 3.7</small>
Indexing	Indexed	Indexed	Not Indexed	Keyed
Mutability	Mutable	Immutable	Mutable <small>Only Adding and Removing</small>	Mutable
Duplicates Allowed	Yes	Yes	No	Yes <small>Only in values and not in keys</small>
Types Allowed	Mutable and Immutable	Mutable and Immutable	Only Immutable	Only Immutable <small>in keys</small>

List

General purpose
Most widely used data structure
Grow and shrink size as needed
Sequence type
Sortable

Set

Store non-duplicate items
Very fast access vs Lists
Math Set ops (union, intersect)
Unordered

Tuple

Immutable (can't add/change)
Useful for fixed data
Faster than Lists
Sequence type

Dict

Key/Value pairs
Associative array, like Java HashMap
Unordered

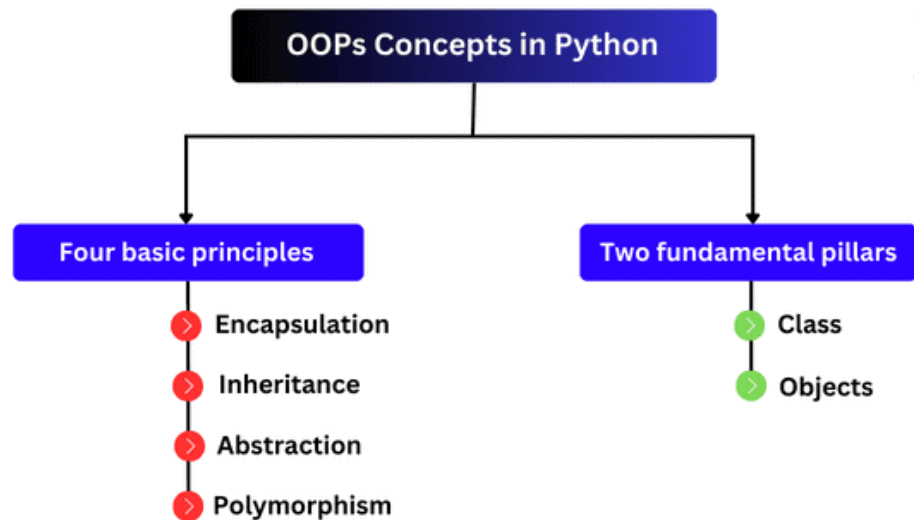
Python Dictionary

```
py_dict = { 1: 'Apple', 2: 'OnePlus' }
```

key value key value
 ↓ ↓ ↓ ↓
 └── Item 1 ──┘ └── Item 2 ──┘

```
my_list = [1, 2, 3]
my_tuple = (4, 5, 6)
my_set = {7, 8, 9}
my_dict = {'a': 10, 'b': 11, 'c': 12}
```

OOP – Object Oriented Programming



Object-Oriented Programming (OOP) in Python is a programming paradigm that organizes code around objects rather than functions and logic. It aims to model real-world entities and their interactions, leading to more modular, reusable, and maintainable code.

•Classes and Objects:

•**Class:** A blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that objects of that class will possess.

•**Object:** An instance of a class. When an object is created from a class, it inherits the attributes and methods defined in that class.

•Inheritance:

- A mechanism that allows a new class (subclass or child class) to inherit attributes and methods from an existing class (superclass or parent class).
- It promotes code reuse and establishes a hierarchical relationship between classes.

•Polymorphism:

- The ability of objects of different classes to be treated as objects of a common type.
- It allows the same method name to be used for different implementations in different classes, leading to more flexible and extensible code.

•Abstraction:

- The process of hiding complex implementation details and exposing only the necessary functionalities to the user.
- It focuses on what an object does rather than how it does it, simplifying interaction with the object.

•Encapsulation:

- The bundling of data (attributes) and the methods that operate on that data within a single unit (the class).
- It restricts direct access to some of an object's components, promoting data integrity and preventing unintended modifications.

Contd...

1. Defining a Class:

A class is defined using the class keyword, followed by the class name (conventionally capitalized) and a colon.

```
class MyClass:  
    x = 5
```

```
p1 = MyClass()  
print(p1.x)
```

2. The __init__ Method (Constructor):

The __init__ method is a special method, often referred to as the constructor, which is automatically called when a new object (instance) of the class is created. It's used to initialize the object's attributes. The first parameter, self, refers to the instance itself.

Create a class named Person, use the __init__() method to assign values for name and age:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

3. Instance Methods:

Methods are functions defined within a class that operate on the object's data. They also take self as their first parameter.

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def myfunc(self):  
        print("Hello my name is " + self.name)
```

4. Creating Objects (Instantiating the Class):

To create an object from a class, you call the class name as if it were a function, passing any required arguments for the __init__ method

```
p1 = Person("John", 36)
```

5. Accessing Attributes and Calling Methods:

You can access an object's attributes and call its methods using the dot (.) operator. `print(p1.name)` # prints the

```
.           nameprint(p1.age)      # prints the  
           agep1.myfunc()          # calls the method
```


Contd...

Class: A Colleton of functions and attributes, attached to a specific name, which represents an abstract concept.

Attribute: A named piece of data (i.e. variable associated with a class.

Object: A single concrete instance generated from a class

Instances of Classes: Classes can be viewed as factories or templates for generating new object instances. Each object instance takes on the properties of the class from which it was created.

```
class Person:
def __init__(self, name, age):
self.name = name # Initialize the 'name' attribute
self.age = age # Initialize the 'age' attribute
```

```
def display_info(self):
print(f"Name: {self.name}, Age: {self.age}")
```

```
# Creating an instance of the Person class
person1 = Person("Alice", 30)
```

```
# Accessing and using the object's attributes and methods
person1.display_info()
```

Create a class (blueprint)
Create a instance (Object)
Class vs instance
Instance attributes: defined in __init__(self)
Class attribute

```
# position, name, age, level, salary
se1 = ["Software Engineer", "Max", 20, "Junior", 5000]
se2 = ["Software Engineer", "Lisa", 25, "Senior", 7000]
```

```
class SoftwareEngineer:

    # class attribute
    alias = "Keyboard Magician"

    def __init__(self, name, age, level, salary):
        # instance attributes
        self.name = name
        self.age = age
        self.level = level
        self.salary = salary

    # instance method
    def code(self):
        print(f"{self.name} is writing code...")

    def code_in_language(self, language):
        print(f"{self.name} is writing code in {language}...")

# instance
se1 = SoftwareEngineer("Max", 20, "Junior", 5000)
se2 = SoftwareEngineer("Lisa", 25, "Senior", 7000)

se1.code()
se2.code()
```


Contd...

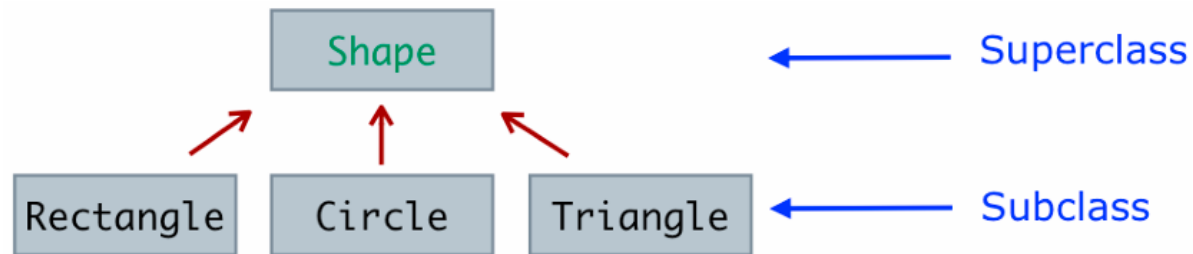
Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

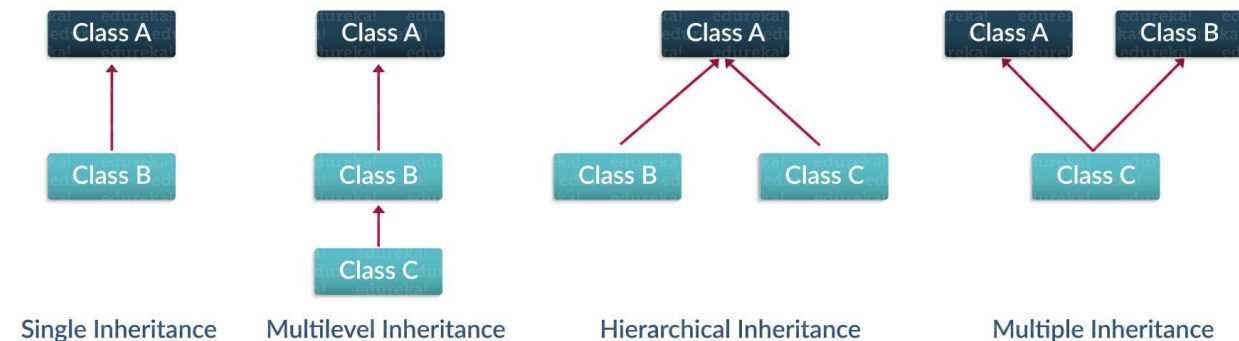
Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Class inheritance is designed to model relationships of the type "x is a y" (e.g. "a triangle is a shape")



Types Of Inheritance



Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Two objects of different classes but supporting the same set of functions or attributes can be treated identically. The implementations may differ internally, but the outward "appearance" is the same.

Two different classes that contain the function `area()`

```
class Rectangle(Shape):
    def __init__(self, w, h):
        Shape.__init__(self)
        self.width = w
        self.height = h

    def area(self):
        return self.width*self.height
```

```
class Circle(Shape):
    def __init__(self, rad):
        Shape.__init__(self)
        self.radius = rad

    def area(self):
        return math.pi*(self.radius**2)
```

Instances of the two classes can be treated identically...

Result of `area()` in Rectangle

Result of `area()` in Circle

```
>>> l = []
>>> l.append( Rectangle(4,5) )
>>> l.append( Circle(3) )
>>> for someshape in l:
...     print someshape.area()
...
20
28.2743338823
```

Cond...

Abstraction

Abstraction is that feature in **OOP** concept wherein the user is kept unaware of the basic implementation of a function property. The user is only able to view basic functionalities whereas the internal details are **hidden**.

_x is called a “protected” attribute (one underscore)
__x is called a “private” attribute (double underscore)

Data abstraction in Python	
Available details of a person	
	Name
	Address
	Tax information
	Contact number
	Favourite food
	Hobbies
	Blood group

Encapsulation

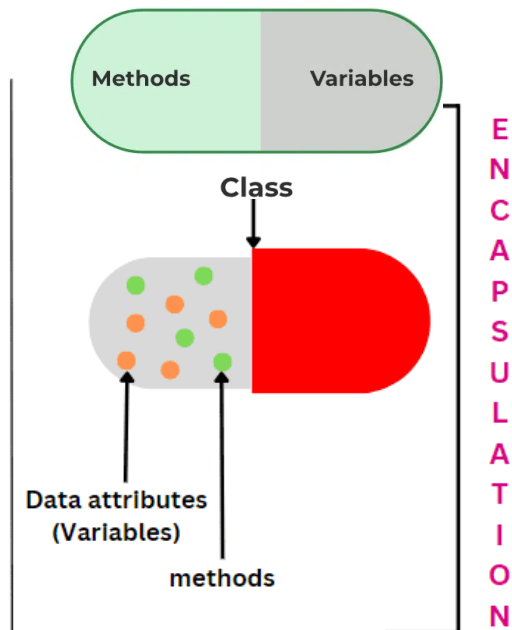
class Person:

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age  
  
def display(self):  
    print(self.name)  
    print(self.age)
```

Data attributes

Method

Wrapping up data and methods associated with that data into a single unit is called encapsulation in Python



File Handling and Modules

Sl no	Text file	Binary file
1	Consist of data in ASCII	consists of data in binary form
2	Suitable to store Unicode characters	Suitable to store binary such as images, videos, audio files, etc
3	Each line in a text file is terminated with a special character(EOL)	There is no EOL - End of line) character
4	Operations on text files are slower than binary files as data needs to be translated to binary	Operations on binary files are faster as no translation is required
5	Web Standards: HTML, XML, CSS, JSON, etc. Source Code: C, APP, JS, PY, Java, etc. Documents: TXT, TEX, RTF, etc. Tabular Data: CSV, TSV, etc. Configuration Files: INI, CFG, REG, etc	Document Files: .pdf, .doc, .xls, etc. Image Files: .png, .jpg, .gif, .bmp, etc. Video Files: .mp4, .3gp, .mkv, .avi, etc. Audio Files: .mp3, .wav, .mka, .aac, etc. Database Files: .mdb, .accde, .frm, .sqlite, etc. Archive Files: .zip, .rar, .iso, .7z, etc. Executable Files: .exe, .dll, .class, etc

open()	obj = open("File_name", mode)
close()	F_obj.close()
read()	F_obj.read() or F_obj.read(n)
readline()	F_obj.readline() or F_obj.read(n)
readlines()	F_obj.readlines()
write()	F_obj.write("data")
writelines()	F_obj.writelines(LST)

Understanding File Objects

file.name: Name of the file. file.mode: Mode in which the file was opened ('r', 'w', etc.). file.closed: Boolean indicating whether the file is closed.

Sequence Of Operations

- 1.Open/Create File
- 2.Read from/Write to file
- 3.Close File

Understanding File Objects

file.name: Name of the file.

file.mode: Mode in which the file was opened ('r', 'w', etc.).

file.closed: Boolean indicating whether the file is closed.

```
print('<----- Reading all Characters ----->')
my_file = open('data.txt', 'r')
contents = my_file.read()
print(contents)
my_file.close()
```


Contd...

File modes and operation

Sr. No	File Mode Symbol	Description
1	r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	r+	Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
4	rb+	Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
5	w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	wb+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	a	Opens a file for appending. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
9	a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.
10	a	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.

Error handling and debugging

Types of Errors in Python

1. Syntax Errors
2. Division by Zero:
3. Accessing Undefined Variables:
4. Out-of-Bounds Indexing:
5. File Not Found:

Logical Errors

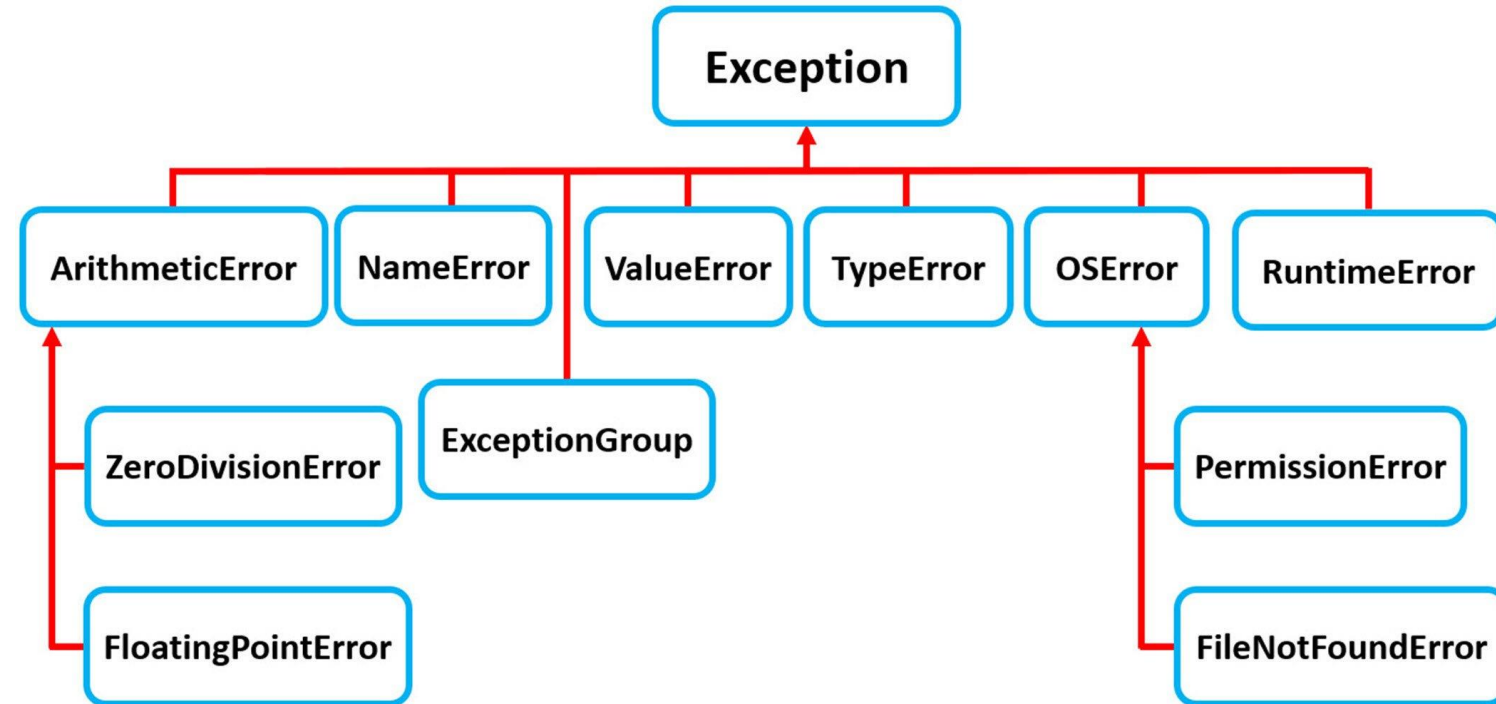
1. Incorrect Algorithm:
2. Off-by-One Error:
3. Wrong Data Types:
4. Misplaced Conditions:

Exceptions

1. ValueError:

2. TypeError: Common Exceptions

3. KeyError: Here are some common exceptions in Python:
- ZeroDivisionError: Division by zero.
 - IndexError: List index out of range.
 - KeyError: Dictionary key not found.
 - TypeError: Invalid operation for a data type.
 - ValueError: Invalid value for a function or operation.
 - Syntax Errors:



```
except ValueError:  
    ("You must enter a number")  
except ZeroDivisionError:  
    ("You can't divide by zero")
```

Error Handling & Debugging: Try-except blocks, custom exceptions, and logging.

Intermediate python - Iterator and Generator

An iterator is an object that represents a stream of data and allows you to traverse through its elements one by one. Iterators implement two special methods: `__iter__()` and `__next__()`.

- `__iter__()` returns the iterator object itself.
- `__next__()` returns the next item in the sequence.
- If there are no more items, it raises a `StopIteration` exception.

```
nums = [1, 2, 3, 4]
obj = iter(nums)
print(next(obj))
print(next(obj))
print(next(obj))
print(next(obj))
```

1
2
3
4

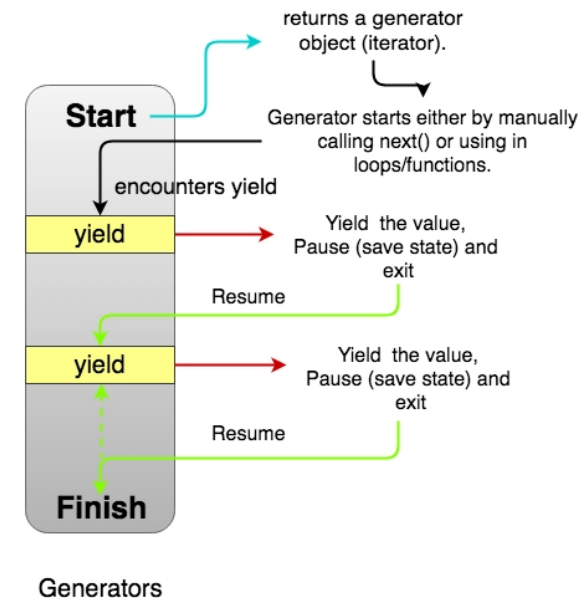
A generator uses the 'yield' keyword in Python. A Python iterator, on the other hand, does not. Every time 'yield' pauses the loop in Python, the Python generator stores the states of the local variables. An iterator does not need local variables

```
def nums():
    for i in range(1, 5):
        yield i

obj = nums()
print(next(obj))
print(next(obj))
print(next(obj))
print(next(obj))
print(next(obj))
```

1
2
3
4

Feature	Iterators	Generators
Definition	Objects implementing <code>__iter__()</code> and <code>__next__()</code> methods.	Functions with <code>yield</code> for value generation.
Memory Use	May require storing the entire dataset in memory.	Generate values lazily; uses less memory.
Ease of Use	Requires implementing <code>__iter__()</code> and <code>__next__()</code> manually.	Defined with a function and <code>yield</code>



Contd...

```
my_list = [1, 2, 3, 4]
iterator = iter(my_list)
```

```
print(next(iterator)) # Output: 1
print(next(iterator)) # Output: 2
print(next(iterator)) # Output: 3
print(next(iterator)) # Output: 4
print(next(iterator)) # Raises StopIteration
```

```
def my_generator():
    yield 1
    yield 2
    yield 3
    yield 4
```

```
gen = my_generator()

print(next(gen)) # Output: 1
print(next(gen)) # Output: 2
print(next(gen)) # Output: 3
print(next(gen)) # Output: 4
print(next(gen)) # Raises StopIteration
```

Feature	Generators	Iterators
Definition	Functions that yield values using yield.	Objects with <code>__iter__</code> and <code>__next__</code> .
Creation	Defined using a function and yield.	Created by implementing iterator methods.
Memory Usage	Lazy evaluation, saves memory.	Depends on data; can use more memory.
State	Maintains state automatically.	Needs manual state management.
Reusability	Can't be reused after exhaustion.	Can be reused if reinitialized.
Ease of Use	Easier with concise syntax.	Requires more code to implement.

Feature	Iterator	Generator
Definition	An object with <code>__iter__()</code> and <code>__next__()</code> methods.	A function with one or more <code>yield</code> statements.
Memory Usage	Can be memory-intensive as all data is stored in memory.	More memory efficient as it generates values on the fly.
Syntax	Requires implementing a class with <code>__iter__()</code> and <code>__next__()</code> methods.	Easier to write using functions and the <code>yield</code> keyword.
State	Maintains state between iterations manually.	Automatically maintains state and local variables between <code>yield</code> calls.
Complexity	More complex to implement.	Simpler and more readable.

Lambda

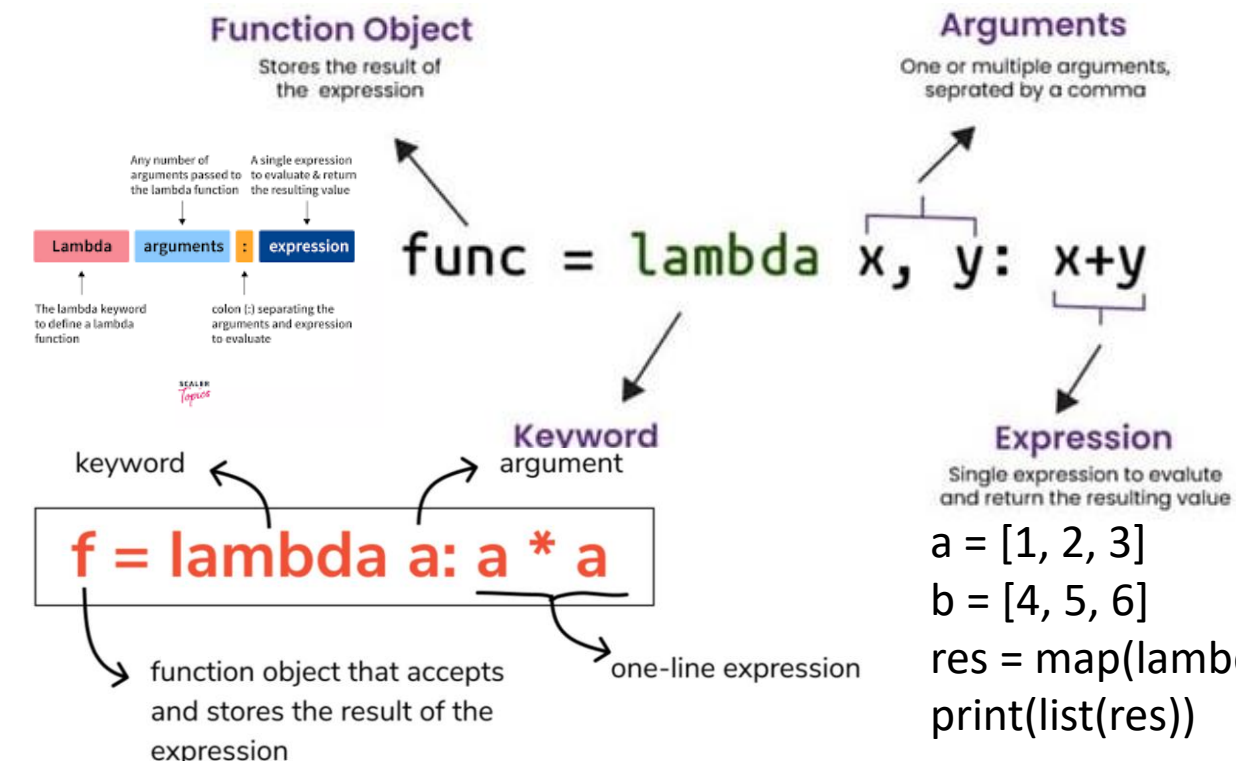
A Python lambda function is a small, anonymous function defined using the lambda keyword. It can take any number of arguments but can only have one expression.

Simple Lambda Function:

```
add = lambda a, b: a + b
print(add(5, 3)) # Output: 8
```

Lambda with map():

```
numbers = [1, 2, 3, 4]
doubled_numbers = list(map(lambda x: x * 2, numbers))
print(doubled_numbers) # Output: [2, 4, 6, 8]
```



Lambda with filter():

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # Output: [2, 4, 6]
```

Lambda as a Return Value:

```
def multiplier(n):
    return lambda a: a * n
doubler = multiplier(2)
tripler = multiplier(3)
print(doubler(10)) # Output: 20
print(tripler(10)) # Output: 30
```

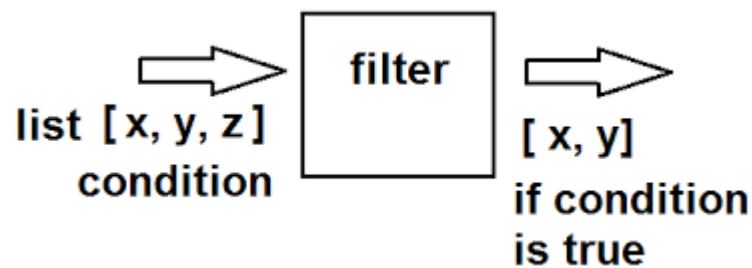
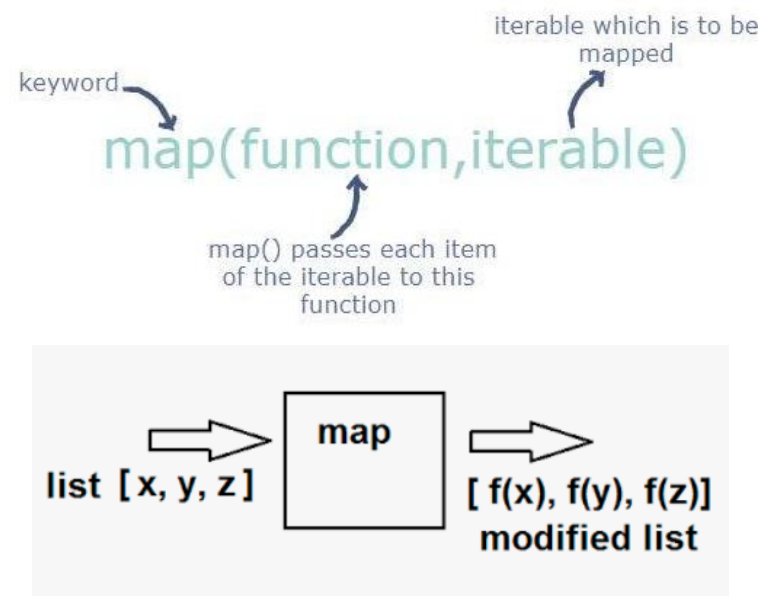
Lambda with Conditional Logic:

```
check_parity = lambda num: "Even" if num % 2 == 0 else "Odd"
print(check_parity(7)) # Output: Odd
print(check_parity(10)) # Output: Even
```

```
a = [1, 2, 3]
b = [4, 5, 6]
res = map(lambda x, y: x + y, a, b)
print(list(res))
```

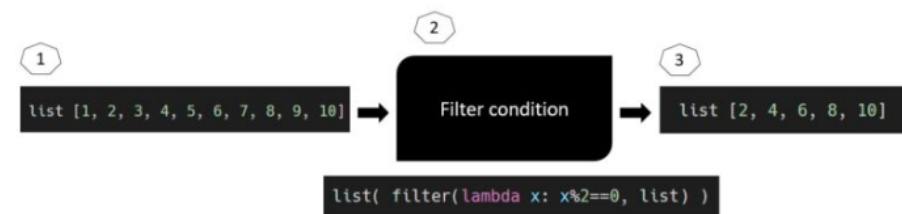
[5, 7, 9]

Map and Filter



```
1 | filter(function or None, iterable)
```

Result: Filter function returns an iterator object having only the filtered data as per your conditions (as written in the function/lambda).



```
def myfunc(a, b):  
    return a + b
```

```
x = map(myfunc, ('apple', 'banana',  
                'cherry'), ('orange', 'lemon',  
                'pineapple'))
```

```
print(x)
```

```
#convert the map into a list, for  
readability:  
print(list(x))
```

```
<map object at 0x034244F0>  
['appleorange', 'bananalemon', 'cherrypineapple']
```

```
def myfunc1(a, b):  
    return a + b
```

```
y = map(myfunc1, ('hi hello'), ('how'))
```

```
print(y)
```

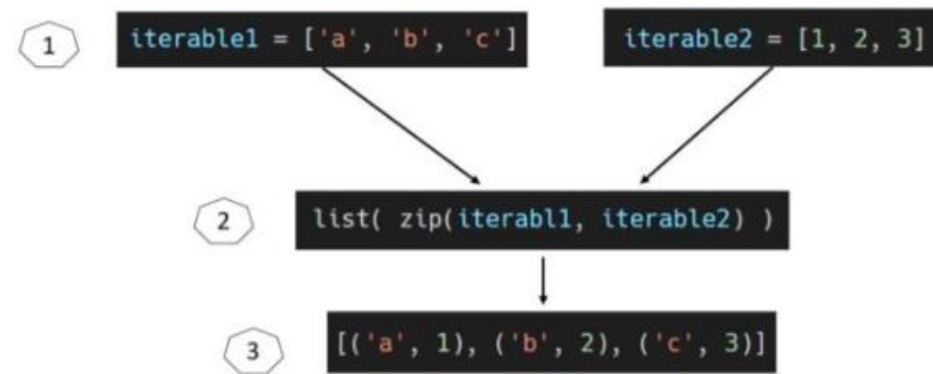
```
#convert the map into a list, for  
readability:  
print(list(y))
```

```
<map object at 0x145e33b0f5b0>  
['hh', 'io', ' w']
```

```
1 | zip(*iterables)
```

where,

iterable—an iterable like sets, lists, tuples, dictionaries, etc.



Comprehensions

Comprehensions in Python provide a concise and expressive way to create new sequences (lists, dictionaries, and sets) from existing iterables. They offer a more readable and efficient alternative to traditional for loops for many common data transformation and filtering tasks.

- List Comprehensions:** Used to create new lists.

```
squares = [x**2 for x in range(10)]
even_numbers = [x for x in range(20) if x % 2 == 0]
```

- Dictionary Comprehensions:** Used to create new dictionaries.

```
names = ["Alice", "Bob", "Charlie"]
ages = [30, 24, 35]
name_age_dict = {name: age for name, age in zip(names, ages)}
```

- Set Comprehensions:** Used to create new sets. Sets are unordered collections of unique elements.

```
unique_letters = {char for char in "hello world" if char.isalpha() }
```

Decorators

decorators are a powerful and flexible way to modify or extend the behavior of functions or methods, without changing their actual code.

- A decorator is essentially a [function](#) that takes another function as an argument and returns a new function with enhanced functionality.
- Decorators are often used in scenarios such as logging, authentication and memorization, allowing us to add additional functionality to existing functions or methods in a clean, reusable way.

```
def pal_dec(func):
    def wrapper():
        print('*****')
        func()
        print('-----')
        return
    return wrapper
```

```
@pal_dec
def pal():
    a = input('please enter word = ')
    if a[::-1] == a[:-1]:
        print('true')
    else:
        print('false')

pal()
```

The @ syntax is syntactic sugar for applying a decorator.

A decorator is a design pattern that allows you to modify or enhance the behavior of a function or class without permanently altering its source code

Libraries

Python libraries are collections of pre-written code, modules, and packages that provide a wide range of functionalities, allowing developers to perform various tasks without writing code from scratch. They contain functions, classes, and routines that can be readily integrated into your programs.

Key aspects of Python libraries:

- Code Reusability:
- Modularity:
- Simplified Development:
- Installation

Extensive Ecosystem:

Data Science and Machine Learning:

Web Development:

Image Processing:

Scientific Computing:

Web Scraping:



NumPy:

Fundamental for numerical computing, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions.



Pandas:

Essential for data manipulation and analysis, offering powerful data structures like DataFrames for working with tabular data.



Matplotlib:

A widely used library for creating static, interactive, and animated visualizations in Python.

Requests:

Simplifies making HTTP requests, commonly used for interacting with web APIs and web scraping.



Scikit-learn:

A comprehensive library for machine learning, offering tools for classification, regression, clustering, and more.

Numpy

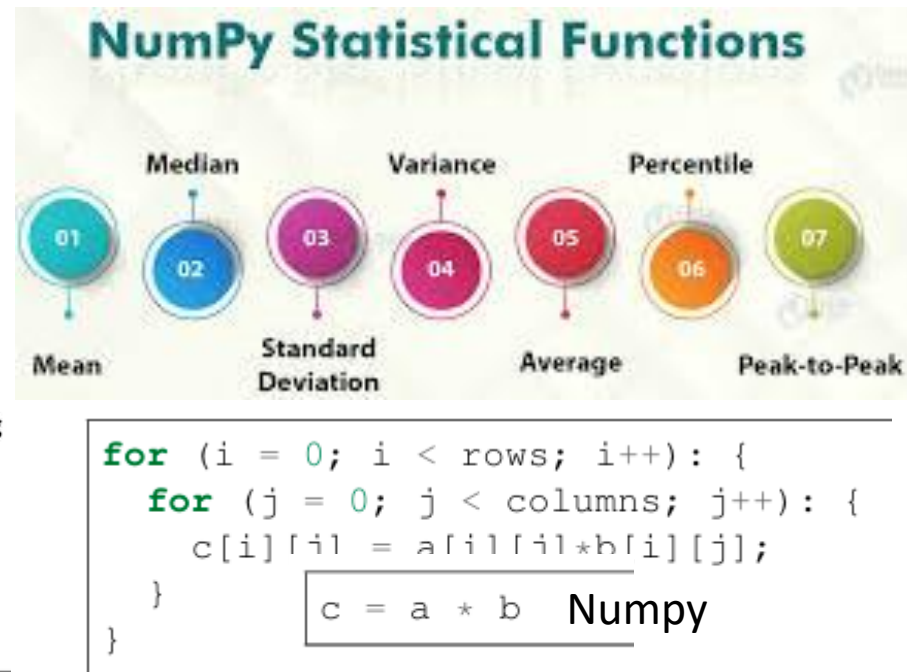
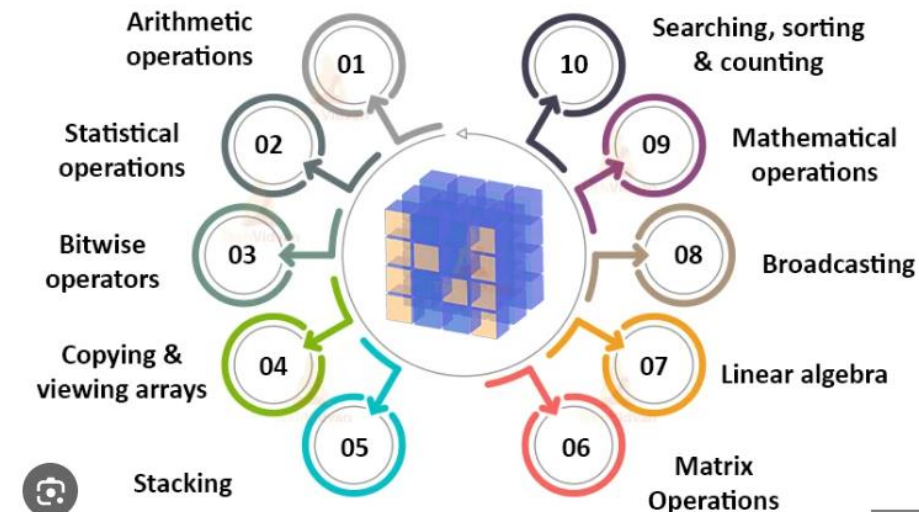
NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays.

NumPy is a library that helps us handle large and multidimensional arrays and matrices. It provides a large collection of powerful methods to do multiple operations.

NumPy (Numerical Python) is a fundamental Python library for scientific computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays efficiently.

Uses of NumPy



#1	$a + b$	<code>np.add(a,b)</code>
#2	$a - b$	<code>np.subtract(a,b)</code>
#3	$a * b$	<code>np.multiply(a,b)</code>
#4	a / b	<code>np.divide(a,b)</code>
#5	$a \% b$	<code>np.mod(a,b)</code>
#6	$a ** b$	<code>np.power(a,b)</code>
#7	$1/a$	<code>np.reciprocal(a)</code>

Pandas

Pandas is a software library written for the Python programming language specifically designed for data manipulation and analysis. It provides powerful and flexible data structures, most notably the DataFrame and Series, which are optimized for working with tabular and labeled data.

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

Pandas is open-source Python library which is used for data manipulation and analysis. It consist of data structures and functions to perform efficient

Why Pandas is widely used:

- Ease of Use:**

Its intuitive API makes it accessible for both beginners and experienced data scientists.

- Flexibility and Power:**

It can handle various data types and structures, from simple tables to complex time series.

- Performance:**

While not inherently multi-threaded, computationally intensive operations are implemented in C or Cython for efficiency.

- Open Source:**

It is free and open-source, benefiting from a large and active community.

Creating a DataFrame from a Dictionary:

Reading Data from a CSV File:

Selecting Columns:

Filtering Rows:

What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.



Pandas

Matplotlib:

Matplotlib is a Python library for data visualization, primarily used to create static, animated, and interactive plots. It provides a wide range of plotting functions to visualize data effectively.

Matplotlib is a powerful and versatile open-source plotting library for Python, designed to help users visualize data in a variety of formats.

Different Types of Plots

- 1. Line Graph
- 2. Bar Chart
- 3. Histogram
- 4. Scatter Plot
- 5. Pie Chart
- 6. 3D Plot

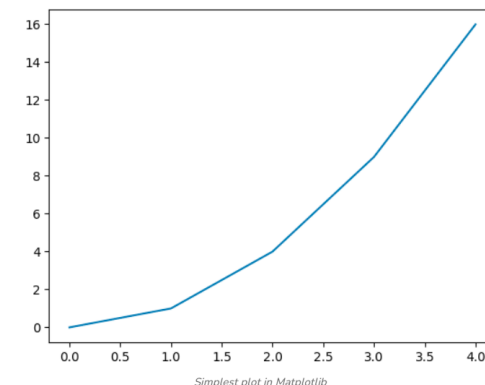
Steps

- Import Matplotlib:** Start by importing matplotlib.pyplot as plt.
- Create Data:** Prepare your data in the form of lists or arrays.
- Plot Data:** Use plt.plot() to create the plot.
- Customize Plot:** Add titles, labels, and other elements using methods like plt.title(), plt.xlabel(), and plt.ylabel().
- Display Plot:** Use plt.show() to display the plot.

Key Features of Matplotlib

- Versatile Plotting:** Create a wide variety of visualizations, including line plots, scatter plots, bar charts, and histograms.
- Extensive Customization:** Control every aspect of your plots, from colors and markers to labels and annotations.
- Seamless Integration with NumPy:** Effortlessly plot data arrays directly, enhancing data manipulation capabilities.
- High-Quality Graphics:** Generate publication-ready plots with precise control over aesthetics.
- Cross-Platform Compatibility:** Use Matplotlib on Windows, macOS, and Linux without issues.
- Interactive Visualizations:** Engage with your data dynamically through interactive plotting features.

```
import matplotlib.pyplot as plt
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]
plt.plot(x, y)
plt.show()
```



Requests:

Request module allow us to easily handle HTTP in Python by sending HTTP requests and receiving responses with simple code.



Python Requests Library is a simple and powerful tool to send **HTTP** requests and interact with web resources. It allows you to easily send **GET, POST, PUT, DELETE, PATCH, HEAD** requests to web servers, handle responses, and work with REST APIs and web scraping tasks:

```
import requests
response = requests.get("https://www.geeksforgeeks.org/")
print(response.status_code)
```

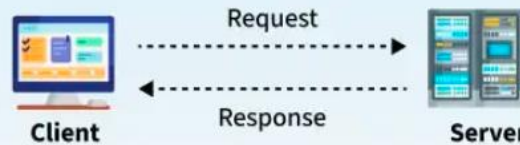
Key Features of Request Module

GET: Retrieve data from a server.

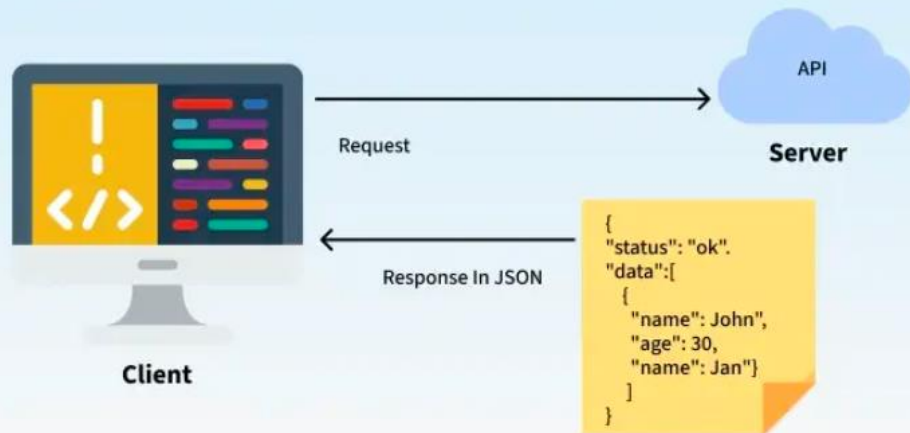
POST: Send data to a server.

PUT: Update existing data on a server.

DELETE: Remove data from a server.



Ease of making API Calls using Request Module



pip install requests

*requests.get(url, params={key: value}, **kwargs)*

Sending GET Requests with Parameters

HTTP Request Methods

Response object

Response Methods

Authentication using Python Requests

SSL Certificate Verification

Accessing a site with invalid SSL:


Providing a custom certificate:

Session Objects






Error Handling with Requests

Short cut keys for pycharm

Code Editing:

- `Ctrl + Space`: Code completion.
- `Ctrl + Shift + Space`: Smart code completion.
- `Ctrl + /`: Comment/uncomment line.
- `Ctrl + Shift + /`: Comment/uncomment block.
- `Ctrl + D`: Duplicate line.
- `Ctrl + Y`: Delete line.
- `Ctrl + Alt + L`: Reformat code.
- `Ctrl + Shift + J`: Join lines.
- `Ctrl + Shift + Up/Down`: Move line up/down.
- `Ctrl + Alt + Enter`: Add line before current.
- `Shift + Enter`: Add line after current.
- `Ctrl + Shift + F8`: View breakpoints.
- `Alt + 0`: Show structure view.
- `Ctrl + Q`: Quick documentation lookup.
- `Ctrl + F1`: Show error description. 


Search:

- `Ctrl + F`: Find in file. 
- `Ctrl + R`: Replace in file. 
- `Ctrl + Shift + F`: Find in files (across the project) according to a Reddit thread. 
- `Shift + Shift` (or `Ctrl + Shift + N`): Search everywhere (files, classes, actions, etc.) according to a Reddit thread. 
- `Alt + ``: VCS operations popup. 











Pycharm shortcut
keys

Running and Debugging:

- `Shift + F10`: Run.
- `Shift + F9`: Debug.
- `F7`: Step into.
- `Alt + 5`: Activate debug window.
- `Ctrl + Alt + F5`: Attach to process.
- `Ctrl + Alt + Shift + T`: Refactor this. 

Navigation:

- `Ctrl + N`: Go to class. 
- `Ctrl + Shift + N`: Go to file. 
- `Ctrl + G`: Go to line. 
- `Alt + F7`: Find usages. 
- `Ctrl + Click` (or `Cmd + Click` on macOS) on a symbol: Go to declaration. 
- `Ctrl + Shift + F7`: Highlight usages in file. 
- `Ctrl + Left Mouse Click` on a symbol: Go to declaration (macOS: `Cmd + Click`) [according to a Reddit thread](#). 
- `Shift + F2` and `F2`: Navigate between code issues (next/previous error). 

Installation

[Download Python | Python.org](#)

[Download PyCharm: The Python IDE for data science and web development by JetBrains](#)

[pip · PyPI](#)

For libraries – install in pycharm



The screenshot shows two terminal windows in PyCharm. The top terminal window has a tab labeled 'Terminal' and 'Local'. It displays the output of a previous command: six 1.17.0, threadpoolctl 3.6.0, and tzdata 2025.2. Below this, the command 'pip install pandas' is entered in the prompt. A yellow highlight is placed over the text 'Pip install pandas'. The bottom terminal window also has a tab labeled 'Terminal' and 'Local'. It displays the output of the 'pip list' command, showing a table of installed packages: contourpy 1.3.2, cyclor 0.12.1, and fonttools 4.59.0. A yellow highlight is placed over the text 'Pip list'.

```
Terminal Local x + v
six 1.17.0
threadpoolctl 3.6.0
tzdata 2025.2
(.venv) PS C:\Users\Admin\Desktop\Python\Srini\230725> pip install pandas

Terminal Local x + v
Try the new cross-platform PowerShell https://aka.ms/pscore6
(.venv) PS C:\Users\Admin\Desktop\Python\Srini\230725> pip list
Package Version
-----
contourpy 1.3.2
cyclor 0.12.1
fonttools 4.59.0
```

The background is a deep blue gradient with a complex, futuristic circuit pattern. The pattern consists of numerous thin, white, angular lines that resemble a microchip or a data network. Several circular nodes are scattered throughout, some of which are glowing with a bright blue and white light, creating a sense of depth and technological sophistication. The overall aesthetic is clean, modern, and high-tech.

Thank you