Table of Contents

Introduction

Purpose

The purpose of this document is to provide a structured learning and reference resource for Python programming, covering both fundamental and advanced concepts. It is intended to help learners and developers strengthen their understanding of Python and apply it effectively in real-world tasks.

Scope

This document spans core Python features including data types, operators, conditions, loops, functions, object-oriented programming, file handling, exception handling, and more. It provides explanations, examples, and tables to make the concepts clear and practical.

Procedure

The content was compiled and organized through two years of study and practice, with notes, examples, and explanations digitized into this document. Each section has been broken down systematically to ensure gradual progression and quick referencing.

Problem

Many learners struggle with scattered resources and inconsistent explanations when starting with Python. This document addresses that issue by consolidating knowledge into a single, well-structured reference, making it easier to study, revise, and apply Python concepts.

Printing

The print() function in Python is used to display information on the screen. It can output text, numbers, variables, and even results of expressions. Printing is often the first step in learning Python because it allows you to see what your code is doing.

```
print (1,2,3, sep="<")
```
```
1<2<3
```

- Sep by default is ‘ ’

```
for i in range(1,11):
   print(i, end= " ")
```
```
1 2 3 4 5 6 7 8 9 10
```

- End by default is ‘\n’, which prints each item on a new line

Type Function

The **type() function** is used to determine the data type of an object. It is very helpful when you want to know what kind of value (string, integer, float, list, etc.) a variable holds.

It can also be used to **create new types (classes)** when given three arguments, but in everyday coding, it's mainly used to check variable types.

```
print(type(10))
print(type(3.14))
 print(type("Hello"))
print(type(True))

print(type([1, 2, 3]))
print(type((1, 2, 3)))
print(type({"a": 1}))
print(type({1, 2, 3}))

x = 42
y = "Python"
print(type(x))
print(type(y))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
<class 'int'>
<class 'str'>
```

Operators

Operators in Python are special symbols that perform operations on values and variables. Arithmetic operators deal with numbers (addition, subtraction, etc.), while bitwise operators work on the binary representation of integers.

Arithmetic Operators

Operator	Name	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus (Remainder)	5 % 2	1
**	Exponentiation	2 ** 3	8

Bitwise Operators

Operator	Name	Example	Result (Decimal)	Result (Binary)
&	AND	5 & 3	1	0101 & 0011 = 0001
|	OR	3 | 5	7	0101 | 0011 = 0111
^	XOR (Exclusive)	5 ^ 3	6	0101 ^ 0011 = 0110
~	NOT (Inversion)	~5	-6	~0101 = ...1010 (2's complement)
<<	Left Shift	5 << 1	10	0101 << 1 = 1010
>>	Right Shift	5 >> 1	2	0101 >> 1 = 0010

Notes

- N << K = N * 2^K
- N >> K = floor(N / (2^K))
- 12 % 4.5 = 3.0
- 2 % -4 = -2

Type Conversion Functions

Type conversion means **changing the data type** of a value or variable into another type. Python provides built-in functions for converting between common types like integers, floats, strings, and more.

There are **two types** of conversions:

- **Implicit Conversion (Type Casting by Python):** Python automatically converts smaller types into larger types (e.g., int → float).
- **Explicit Conversion (Type Casting by Programmer):** You manually convert using built-in functions like int(), float(), str(), etc.

Function	Description	Example	Output
int()	Converts to integer (drops decimals)	int(3.9) int("42")	3 42
float()	Converts to floating-point number	float(7) float("3.14")	7.0 3.14
str()	Converts to string	str(100) str(3.14)	"100" "3.14"
bool()	Converts to Boolean (True / False)	bool(1) bool(0) bool("")	True False False
list()	Converts to list	list("abc")	['a','b','c']
tuple()	Converts to tuple	tuple([1,2,3])	(1,2,3)
set()	Converts to set (removes duplicates)	set([1,2,2,3])	{1,2,3}
dict()	Converts to dictionary (pairs needed)	dict([(1,"a"),(2,"b")])	{1:"a",2:"b"}

Input Function

The **input() function** is used to take input from the user.

- It always returns the input as a **string** (str), even if the user types a number.
- You can optionally display a **prompt message** inside the function.
- If you need the input as an integer or float, you must convert it using int() or float().

```
name = input("Enter your name: ")
print("Hello,", name)

age = int(input("Enter your age: "))
print("You are", age, "years old.")

num = float(input("Enter a decimal number: "))
print("You entered:", num)

data = input()
print("You typed:", data)
```

```
Enter your name: Youssef
Hello, Youssef
Enter your age: 20
You are 20 years old.
Enter a decimal number: 71.17
You entered: 71.17
You typed: 17
```

Strings

A **string** in Python is a sequence of characters enclosed in **single quotes (')**, **double quotes (")**, or **triple quotes (''' or """)**.

- Strings are **immutable** (cannot be changed after creation).
- They support **indexing**, **slicing**, and many built-in methods.

Creating Strings

```
s1 = 'Hello'
s2 = "World"
s3 = '''Multiline
String'''
```

Indexing and Slicing

```
text = "Python"
print(text[0])
print(text[-1])
print(text[0:4])  # 4 not included
print(text[::2])
print(text[:])
print(text[0:5:2]) # start, end, step

print('Py' in text) # check

print(text[::-1]) # Reverse
```

```
P
n
Pyth
Pto
Python
Pto
True
nohtyp
```

- String Indices start from 0 to length(string)-1 , or from the end from -1 till -length(string)

String Functions and Methods

Function/Method	Description	Example	Output
len(s)	Length of string	len("Hello")	5
s.lower()	Convert to lowercase	"Hello".lower()	"hello"
s.upper()	Convert to uppercase	"Hello".upper()	"HELLO"
s.title()	Capitalize each word	"python basics".title()	"Python Basics"
s.capitalize()	Capitalize first letter	"hello".capitalize()	"Hello"
s.strip()	Remove spaces (both sides)	" hi ".strip()	"hi"
s.lstrip()	Remove left spaces	" hi".lstrip()	"hi"
s.rstrip()	Remove right spaces	"hi ".rstrip()	"hi"
s.replace(a, b)	Replace substring	"hi hi".replace("hi","hey")	"hey hey"
s.split(delim)	Split into list	"a,b,c".split(",")	['a','b','c']
delim.join(list)	Join list into string	"-".join(["a","b"])	"a-b"
s.find(sub)	First index of substring	"hello".find("l")	2
s.rfind(sub)	Last index of substring	"hello".rfind("l")	3
s.index(sub)	Like find(), error if not found	"hello".index("e")	1
s.count(sub)	Count occurrences	"banana".count("a")	3
s.startswith(x)	Check if starts with substring	"hello".startswith("he")	True
s.endswith(x)	Check if ends with substring	"hello".endswith("lo")	True
s.isalpha()	Only letters?	"abc".isalpha()	True
s.isdigit()	Only digits?	"123".isdigit()	True
s.isalnum()	Letters & digits?	"abc123".isalnum()	True
s.isspace()	Only whitespace?	" ".isspace()	True

Function/Method	Description	Example	Output
s.islower()	All lowercase?	"abc".islower()	True
s.isupper()	All uppercase?	"ABC".isupper()	True
s.swapcase()	Swap case	"Hello".swapcase()	"hELLO"
s.center(width)	Center align	"hi".center(6)	" hi "
s.ljust(width)	Left align	"hi".ljust(6)	"hi "
s.rjust(width)	Right align	"hi".rjust(6)	" hi"
s.zfill(width)	Pad with zeros	"7".zfill(3)	"007"

String Concatenation and Repetition

```
a = "Hello"
b = "World"
print(a + " " + b)
print(a * 3)
```

```
Hello World
HelloHelloHello
```

String Formatting

```
name = "Youssef"
age = 21

print(f"My name is {name}, I am {age} years old.")

print("My name is {}, I am {} years old.".format(name, age)) # Puts name in first {} then age in second {}

print("My name is {0}, I am {1} years old.".format(name, age)) # name is index 0, age is index 1 based on order of (name,age)
```

```
My name is Youssef, I am 21 years old.
My name is Youssef, I am 21 years old.
My name is Youssef, I am 21 years old.
```

Escape Characters

Escape Code	Meaning	Example	Output
\n	New line	"Hello\nWorld"	Hello World
\t	Tab space	"Hello\tWorld"	Hello World
\\	Backslash	"C:\\Python"	C:\Python
\'	Single quote	'It\'s ok'	It's ok
\"	Double quote	"He said \"Hi\""	He said "Hi"
\r	Carriage return (cursor moves to start of line)	"Hello\rWorld"	World
\b	Backspace (removes previous char)	"Hello\bWorld"	HellWorld
\f	Form feed (page break, rarely used)	"Hello\fWorld"	Hello (new page) World
\ooo	Octal value	"\110\145\154\154\157"	Hello
\xhh	Hex value	"\x48\x65\x6C\x6C\x6F"	Hello

Conditions

Conditional statements allow a program to **make decisions** based on certain conditions.

- They use **Boolean expressions** (True / False).
- The main keywords are: if, elif, and else.

Syntax

```
age = 18

if age < 18:
   print("You are a minor.")

elif age == 18:
   print("You are exactly 18.")

else:
   print("You are an adult.")
```

```
You are exactly 18.
```

Comparison Operators

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	7 > 3	True
<	Less than	2 < 9	True
>=	Greater than or equal to	7 >= 7	True
<=	Less than or equal to	4 <= 5	True

Logical Operators

Operator	Meaning	Example	Result
and	True if both are True	(5 > 2 and 10 > 5)	True
or	True if at least one is True	(5 > 10 or 10 > 5)	True
not	Reverses the result	not(5 > 2)	False

Nested Conditions

```
x = 10
if x > 0:
   if x % 2 == 0:
      print("Positive even number")
   else:
      print("Positive odd number")
```

```
Positive even number
```

Shorthand Conditions

```
x = 5
if x > 0: print("Positive")

print("Even") if x % 2 == 0 else print("Odd")
```

```
Positive
Odd
```

Loops

Loops allow you to **repeat a block of code** multiple times.
Python mainly has **two types of loops**:

1. for loop → iterates over a sequence (list, string, tuple, etc.)
2. while loop → repeats as long as a condition is True

For Loop

```
for i in range(5):
    print("Iteration:", i)
```

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

```
for i in range(1,10,3): # start, end, step
    print("Iteration:", i)
```

```
Iteration: 1
Iteration: 4
Iteration: 7
```

```
text="Mercury"
for index in range(len(text)):
    print(index,text[index])
for char in text:
    print(char)
```

```
0 M
1 e
2 r
3 c
4 u
5 r
6 y
M
e
r
c
u
r
y
```

While Loop

```
x = 1
while x <= 5:
   print("Number:", x)
   x += 1
```

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Loop Control Statements

Keyword	Description	Example
break	Exits the loop completely	Stops loop when condition met
continue	Skips the current iteration, moves to next	Skips certain values
pass	Does nothing (placeholder)	Useful when writing empty loops

---

Nested Loops

```
for i in range(3):
   for j in range(2):
      print(f"i={i}, j={j}")
```

```
i=0, j=0
i=0, j=1
i=1, j=0
i=1, j=1
i=2, j=0
i=2, j=1
```

Else with Loops

```
for i in range(3):
   print(i)
else:
   print("Loop finished!")
```

```
0
1
2
Loop finished!
```

Functions

A **function** is a reusable block of code that performs a specific task.

- Functions help make programs **modular, reusable, and organized**.
- They are defined using the def keyword.
- Python also supports **built-in functions** (len(), print(), type(), etc.) and **user-defined functions**.

Syntax

```
def greet(name):
    """This function greets the user by name"""
    return f"Hello, {name}!"

print(greet("Youssef"))
```

```
Hello, Youssef!
```

Default Parameters

```
def greet(name="Guest"):
    print("Hello,", name)

greet()
greet("Youssef")
```

```
Hello, Guest
Hello, Youssef
```

Multiple Returns

```
def stats(x, y):
    return x+y, x-y, x*y

s, d, p = stats(10, 5)
print(s, d, p)
```

```
15 5 50
```

Function Arguments

Type	Description	Example
Positional Arguments	Passed in order	add(3, 4)
Keyword Arguments	Passed with key=value	add(a=3, b=4)
Default Arguments	Take default value if not given	greet("Ali"), greet()
Variable Length (*args)	Accepts many values as tuple	def f(*args): print(args)
Keyword Variable Length (**kwargs)	Accepts many key-value pairs as dict	def f(**kwargs): print(kwargs)

Lambda Function

```
square = lambda x: x*x
print(square(5))
```

```
25
```

- **Often used with map(), filter(), sorted().**

Recursion

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)

print(factorial(5))
```

```
120
```

- Recursion is when a function keeps calling itself with different arguments until a condition is met

Commonly Used Built-in Functions

Function	Description	Example	Output
print()	Displays output	print("Hello")	Hello
len()	Returns length of object	len("Python")	6
type()	Returns type of object	type(3.14)	<class 'float'>
int()	Converts to integer	int("10")	10
float()	Converts to float	float("3.5")	3.5
str()	Converts to string	str(99)	"99"
bool()	Converts to Boolean	bool("")	False
list()	Converts to list	list("abc")	['a','b','c']
tuple()	Converts to tuple	tuple([1,2,3])	(1,2,3)
set()	Converts to set (removes duplicates)	set([1,2,2,3])	{1,2,3}
dict()	Creates dictionary	dict([(1,"a"), (2,"b")])	{1:'a',2:'b'}
sum()	Returns sum of iterable	sum([1,2,3])	6
max()	Returns largest element	max([4,7,2])	7
min()	Returns smallest element	min([4,7,2])	2
abs()	Returns absolute value	abs(-7)	7
round()	Rounds number	round(3.14159, 2)	3.14
pow()	Returns x to the power of y	pow(2, 3)	8
sorted()	Returns sorted list	sorted([3,1,2])	[1,2,3]
reversed()	Returns reversed iterator	list(reversed([1,2,3]))	[3,2,1]
range()	Returns sequence of numbers	list(range(5))	[0,1,2,3,4]
input()	Takes input from user	input("Enter: ")	User's input
help()	Displays documentation	help(len)	Docstring of len()

Function	Description	Example	Output
dir()	Lists attributes/methods	dir([])	List methods of list
id()	Returns object’s memory address	id("abc")	(some int)
any()	True if any element is True	any([0,1,0])	True
all()	True if all elements are True	all([1,2,3])	True
zip()	Combines iterables element-wise	list(zip([1,2],[‘a’,’b’]))	[(1,'a'),(2,'b')]
enumerate()	Returns index + value	list(enumerate(['a','b']))	[(0,'a'),(1,'b')]
map()	Applies function to iterable	list(map(str, [1,2,3]))	['1','2','3']
filter()	Filters elements	list(filter(lambda x: x>2, [1,2,3,4]))	[3,4]
sorted()	Sorts elements	sorted(["c","a","b"])	['a','b','c']

Math Functions

Function	Description	Example	Output
math.sqrt(x)	Square root	math.sqrt(16)	4.0
math.factorial(x)	Factorial	math.factorial(5)	120
math.gcd(a, b)	Greatest common divisor	math.gcd(12, 18)	6
math.lcm(a, b) (3.9+)	Least common multiple	math.lcm(4, 6)	12
math.ceil(x)	Smallest integer ≥ x	math.ceil(4.3)	5
math.floor(x)	Largest integer ≤ x	math.floor(4.9)	4
math.trunc(x)	Truncate decimal (remove fraction)	math.trunc(3.99)	3
math.pow(x, y)	x raised to power y (float)	math.pow(2, 3)	8.0
math.exp(x)	e^x (exponential)	math.exp(2)	7.389…
math.log(x)	Natural log (base e)	math.log(10)	2.302…
math.log10(x)	Logarithm base 10	math.log10(100)	2
math.log2(x)	Logarithm base 2	math.log2(8)	3
math.sin(x)	Sine (x in radians)	math.sin(math.pi/2)	1.0
math.cos(x)	Cosine	math.cos(0)	1.0
math.tan(x)	Tangent	math.tan(math.pi/4)	1.0
math.degrees(x)	Convert radians → degrees	math.degrees(math.pi)	180.0
math.radians(x)	Convert degrees → radians	math.radians(180)	3.1415…
math.pi	Constant π	math.pi	3.1415…
math.e	Constant e	math.e	2.718…
math.inf	Infinity constant	math.inf > 10**1000	True
math.nan	Not a number (NaN)	math.isnan(math.nan)	True

Scope

Scope defines the **region of a program** where a variable can be accessed.
Python uses the **LEGB Rule** to resolve variables:

1. **L → Local**: Inside the current function.
2. **E → Enclosing**: Variables in enclosing (outer) functions.
3. **G → Global**: Variables declared at the top level of a script or module.
4. **B → Built-in**: Predefined names in Python (e.g., len, print).

Python searches in this order when you use a variable.

LEGB Rule

```
x = "global"   # Global variable

def outer():
   x = "enclosing"  # Enclosing variable

   def inner():
      x = "local"  # Local variable
      print(x)    # Will print "local"

   inner()
   print(x)       # Will print "enclosing"

outer()
print(x)          # Will print "global"
```

Global Variables

```
x = 10

def modify():
   global x
   x = 20   # modifies global variable

modify()
print(x)  # 20
```

20

- Without the keyword global, this gives an error, as without the global keyword, you can only read global variables without modifying them

Nonlocal Variables

```
def outer():
    x = "outer"
    def inner():
        nonlocal x
        x = "inner modified"
    inner()
    print(x)

outer()  # inner modified
```

```
inner modified
```

Mutability

Mutable objects: Their contents **can be changed** after creation.

Immutable objects: Their contents **cannot be changed**; any modification creates a **new object** in memory.

This affects **performance, memory, and bug prevention**.

Mutability Table

Type	Mutable?	Example of Change
int, float, complex	❌ No	x = x + 1 → creates new object
str	❌ No	"hi" + "!" → new string
tuple	❌ No	Cannot add/remove items
list	✅ Yes	append(5) modifies in place
dict	✅ Yes	d["a"]=1 adds key
set	✅ Yes	add(10) modifies in place

Effect Of Functions

- Python always passes **object references** to functions.
- If the object is **immutable**, you can't modify it → looks like pass-by-value.
- If the object is **mutable**, you can modify it → looks like pass-by-reference.

Object Type	Inside Function Change	Effect Outside Function	Passing
Immutable (int, float, complex, str, tuple)	Creates new object	❌ No effect	By Value
Mutable (list, dict, set)	Modifies same object	✅ Affects original	By Reference

Exception Handling

- An **exception** is an error that occurs during program execution.
- If not handled, Python stops the program and shows an error message (traceback).
- Exception handling allows you to **catch** errors and decide what to do instead.

Syntax

```
try:
   x = 10 / 0
except:
   print("An error occurred!")
```

```
An error occurred!
```

- **Without handling, this would have crashed with ZeroDivisionError.**

Catching Specific Exceptions

```
try:
   num = int("abc")

except ValueError:
   print("Invalid number format!")

except ZeroDivisionError:
   print("Cannot divide by zero!")
```

```
Invalid number format!
```

Else

- The else block runs if no exceptions occur

```
try:
   result = 5 / 1
except ZeroDivisionError:
   print("Division error")
else:
   print("No error, result is:", result)
```

```
No error, result is: 5.0
```

Finally

- The finally block **always runs**, whether an error happens or not. Useful for cleanup (closing files, connections, etc.):

```
try:
    f = open("data.txt", "r")
    content = f.read()

except FileNotFoundError:
    print("File not found!")

finally:
    print("Closing file (if open)")
```

Raising Exceptions

- You can **raise** your own exceptions with raise:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("You can't divide by zero!")
    return a / b

try:
    print(divide(10, 0))

except ZeroDivisionError as e:
    print("Error:", e)
```

Common Built-in Exceptions

Exception	When it occurs
ValueError	Wrong value type (int("abc"))
TypeError	Wrong data type ("a" + 5)
ZeroDivisionError	Division by zero
FileNotFoundError	File doesn’t exist
IndexError	Index out of range ([1,2][5])
KeyError	Dictionary key not found
NameError	Using an undefined variable

Files and File Handling

- Files let you **store data permanently** (unlike variables in memory).
- You can **read** (input) and **write** (output) files.
- Python provides the built-in open() function for file operations.

Syntax

```
file = open("filename.txt", "mode")
```

Modes

Mode	Description
"r"	Read (default). Error if file doesn't exist.
"w"	Write (creates new file or overwrites existing).
"a"	Append (adds to file if exists).
"x"	Create new file (error if file exists).
"b"	Binary mode (e.g., "rb", "wb" for images).
"t"	Text mode (default).

Reading from Files

```
f = open("data.txt", "r")

print(f.read())       # read entire file
print(f.read(5))      # read first 5 characters
print(f.readline())   # read one line
print(f.readlines())  # read all lines as list

f.close()
```

Writing to Files

```
f = open("data.txt", "w")
f.write("Hello, world!\n")
f.write("This is Python.\n")
f.close()
```

- **Using "w" overwrites the file. Use "a" to append.**

With Statement

- **Using with automatically closes the file, even if an error occurs:**

```
with open("data.txt", "r") as f:
   for line in f:
      print(line.strip())
```

```
with open("log.txt", "a") as f:
   f.write("New log entry\n")
```

File Methods

Method	Description
read(size)	Reads up to size characters (or whole file if omitted).
readline()	Reads one line.
readlines()	Reads all lines into a list.
write(string)	Writes text to file.
writelines(list)	Writes multiple lines.
seek(offset)	Moves the file pointer.
tell()	Returns current file pointer position.
close()	Closes the file (important for saving).

Tuple

- A **tuple** is an **ordered, immutable collection** in Python.
- Similar to a list, but **cannot be modified** (no adding, removing, or changing items after creation).
- Tuples are useful for storing **fixed sets of data**.

Syntax

```
# Empty tuple
t1 = ()

# Single element tuple (comma is required!)
t2 = (5,)

# Multiple different elements
t3 = (1, 2, 3, "hello")

# Without parentheses (tuple packing)
t4 = 1, 2, 3

# Nested tuple
t5 = (1, (2, 3), [4, 5])
```

Accessing Elements

- **Just like a String**

```
nums = (10, 20, 30, 40)

print(nums[0])
print(nums[-1])
print(nums[1:3])
```

```
10
40
(20,30)
```

Immutability

```
nums = (1, 2, 3)
nums[0] = 100
```

```
Error: 'tuple' object does not support item assignment
```

Tuple Operations

Operation	Example	Result
Concatenation	(1,2) + (3,4)	(1,2,3,4)
Repetition	(1,2) * 3	(1,2,1,2,1,2)
Membership	3 in (1,2,3)	True
Length	len((1,2,3))	3

Tuple Methods

Function / Method	Description	Example
len(tuple)	Returns number of items	len((1,2,3)) → 3
max(tuple)	Returns largest item	max((5,7,2)) → 7
min(tuple)	Returns smallest item	min((5,7,2)) → 2
sum(tuple)	Returns sum of elements (numbers only)	sum((1,2,3)) → 6
tuple(iterable)	Converts iterable into tuple	tuple([1,2,3]) → (1,2,3)
.count(x)	Counts occurrences of x	(1,2,2,3).count(2) → 2
.index(x)	Returns first index of x	(10,20,30).index(20) → 1

Packing and Unpacking

```
# Packing
person = ("Alice", 25, "Engineer")

# Unpacking
name, age, job = person # (name, * info) = person, info will contain age, job as a tuple
print(name)
print(age)
print(job)
```

```
Alice
25
Engineer
```

Nested Tuples

```
nested = ((1, 2), (3, 4), (5, 6))
print(nested[1][0])
```

```
3
```

As_Integer_Ratio Function

```
# Transforms floats into (numerator, denominator) tuple

print((2.5).as_integer_ratio())
print((0.125).as_integer_ratio())
print((3.0).as_integer_ratio())
```

```
(5,2)
(1,8)
(3,1)
```

Swapping

```
# Swapping Through Tuples

x = 5
y= 10
(x,y) = (y,x)
print(x,y)
```

```
10 5
```

List

- A **list** is an **ordered, mutable collection** in Python.
- Lists can store **mixed data types** (numbers, strings, objects, even other lists).
- They are **dynamic** → you can add, remove, or modify items after creation.

Syntax

```
# Empty list
l1 = []

# List with values
l2 = [1, 2, 3, "hello"]

# Nested list
l3 = [1, [2, 3], [4, 5]]

# Using list() constructor
l4 = list((10, 20, 30))

# List comprehension
l5 = [x**2 for x in range(5)]  # [0, 1, 4, 9, 16]
```

Accessing Elements

- **Just like String**

```
nums = [10, 20, 30, 40, 50]

print(nums[0])
print(nums[-1])
print(nums[1:4])  # (slicing)
```

```
10
50
[20, 30, 40]
```

Modifying Lists

```
nums = [1, 2, 3]
nums[0] = 100     # Change element
nums.append(4)    # Add at end
nums.insert(1, 200) # Insert at index 1
nums.remove(3)    # Remove first occurrence of 3
popped = nums.pop() # Remove last item
```

List Methods

Method	Description	Example
append(x)	Adds x to end	[1,2].append(3) → [1,2,3] [1,2].append([3,4]) → [1,2,[3,4]]
extend(iterable)	Adds all elements	[1,2].extend([3,4]) → [1,2,3,4]
insert(i, x)	Insert at index	[1,2].insert(1,99) → [1,99,2] [1,2].insert(1,[3,4]) → [1,3,4,2]
remove(x)	Removes first occurrence of x	[1,2,3].remove(2) → [1,3]
pop(i)	Removes & returns element i is end element by default	[1,2,3].pop() → 3
clear()	Removes all items	[1,2].clear() → []
index(x)	Returns index of x	[1,2,3].index(2) → 1
count(x)	Counts occurrences	[1,2,2,3].count(2) → 2
sort()	Sorts list in place	[3,1,2].sort() → [1,2,3] [3,1,2].sort(reverse=True) → [1,2,3]
sorted()	Returns new sorted list	sorted([3,1,2]) → [1,2,3]
reverse()	Reverses in place	[1,2,3].reverse() → [3,2,1]
reversed()	Returns new reversed list	reversed([3,1,2]) → [2,1,3]
copy()	Shallow copy of list	new = old.copy()

Built-in Functions with List

Function	Description	Example
len(list)	Number of items	len([1,2,3]) → 3
sum(list)	Sum of numeric values	sum([1,2,3]) → 6
max(list)	Largest value	max([1,9,5]) → 9
min(list)	Smallest value	min([1,9,5]) → 1
list(iterable)	Converts to list	list("abc") → ['a','b','c']

List Comprehensions

```
squares = [x**2 for x in range(5)]     # [0,1,4,9,16]
evens = [x for x in range(10) if x%2==0] # [0,2,4,6,8]
l = [range(1,10,3)] # [1,4,7]
```

Nested Lists

```
matrix = [[1,2,3], [4,5,6], [7,8,9]]

print(matrix[1][2])
```

```
6
```

List Assignment

```
l1 = [1, 2, 3]
l2 = l1     # Both point to the same list

l2.append(4)
print(l1)   # [1, 2, 3, 4] → l1 changed too!
```

```
[1,2,3,4]
```

Shallow Copy

```
l1 = [1, 2, 3]
l2 = l1.copy()      # method
l3 = list(l1)       # constructor
l4 = l1[:]          # slicing

l2.append(4)
l3.append(17)
l4.append(20)
print(l1)  # [1, 2, 3]
print(l2)  # [1, 2, 3, 4]
```

```
[1,2,3]
[1,2,3,4]
```

Deleting

```
l = [1,2,3]
del l
# print(l) → NameError: l is not defined

l = [10, 20, 30, 40]
del l[1]
print(l)  # [10, 30, 40]

del l[:] # removes all elements from list
```

Sorting

```
words = ["apple", "banana", "kiwi", "pear"]

words.sort(reverse=True)       # Descending
print(words)  # ['pear', 'kiwi', 'banana', 'apple']

words.sort(key=len)            # By length
print(words)  # ['pear', 'kiwi', 'apple', 'banana']

words.sort(key=len, reverse=True)          # By length and Descending
print(words)  # [banana, apple, kiwi, pear]
```

```
l1 = [3, 1, 2]
l2 = l1.sort()

print(l1)  # [1, 2, 3]
print(l2)  # None
```

Joining and Splitting

```
words = ["Python", "is", "fun"]
sentence = " ".join(words)
print(sentence)
```

```
Python is fun
```

```
csv = "apple,banana,cherry"
fruits = csv.split(",")
print(fruits)
```

```
['apple', 'banana', 'cherry']
```

```
s = "hello"
chars = list(s)
print(chars)
```

```
['h', 'e', 'l', 'l', 'o']
```

```
s = "Python is great"
words = s.split()
print(words)
```

```
['Python', 'is', 'great']
```

Dictionary

- A **dictionary** is a collection of **key-value pairs**.
- Keys must be **unique** and **immutable** (str, int, tuple, etc.).
- Values can be of **any type** and can be duplicated.
- Dictionaries are **unordered** (insertion order is preserved in Python 3.7+).

Syntax

```
# Empty dictionary
d = {}

# With values
student = {"name": "Alice", "age": 20, "grade": "A"}

# Using dict() constructor
info = dict(name="Bob", age=25)
```

Accessing Elements

```
student = {"name": "Alice", "age": 20}

print(student["name"])      # Alice
print(student.get("age"))   # 20
print(student.get("grade", "N/A"))  # Safe lookup, returns "N/A" if key missing (None by default)
```

Removing Elements

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student.pop("age")       # removes by key
del student["grade"]     # removes by key
student.popitem()        # removes last key
student.clear()          # empties dict
```

Updating and Adding Elements

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student["name"]="Youssef"
student["Country"]="Egypt"
print(student)
```

```
{'name': 'Youssef', 'age': 20, 'grade': 'A', 'Country': 'Egypt'}
```

Dictionary Methods

Method	Description	Example
d.keys()	Returns all keys	student.keys() → dict_keys(['name', 'age'])
d.values()	Returns all values	student.values() → dict_values(['Alice', 20])
d.items()	Returns key-value pairs	student.items() → [('name','Alice'),('age',20)]
d.get(k, default)	Safe access with default	student.get("grade","N/A")
d.pop(k)	Remove item by key	student.pop("age")
d.update(other)	Merge/update dicts	student.update({"grade":"B"})
d.copy()	Shallow copy	new_dict = student.copy()

Iterating Through a Dictionary

```
student = {"name": "Alice", "age": 20, "grade": "A"}

for key in student:          # iterate keys
   print(key, student[key])

for key, value in student.items():   # iterate pairs
   print(key, value)
```

```
name Alice
age 20
grade A
name Alice
age 20
grade A
```

Nested Dictionaries

```
students = {
   "s1": {"name": "Alice", "age": 20},
   "s2": {"name": "Bob", "age": 22}
}
print(students["s1"]["name"])
```

```
Alice
```

Dictionary Comprehension

```
squares = {x: x**2 for x in range(5)}
print(squares)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Membership Testing

```
student = {"name" : "Youssef" , "age" : 18 }
print("age" in student)
```

```
True
```

Set

- A **set** is an **unordered collection** of **unique** elements.
- Mutable (you can add/remove elements), but elements themselves must be **immutable** (int, str, tuple, etc.).
- Useful for removing duplicates, set operations (union, intersection, etc.).

Syntax

```
# Empty set
s = set()
s = {}          # {} creates a dict, not a set!

# With elements
nums = {1, 2, 3, 4}
letters = set("hello")   # {'o', 'l', 'e', 'h'}
```

Adding and Removing Elements

```
s = {1, 2, 3}

s.add(4)          # Add one element
s.update([5, 6])    # Add multiple

s.remove(2)       # Remove element (error if missing)
s.discard(10)     # Remove safely (no error)
s.pop()           # Remove random element
s.clear()         # Empty set
```

Set Operations

Operation	Symbol	Example	Result
Union	| or .union()	{1,2,3} | {2,3,4}	{1,2,3,4}
Intersection	& or .intersection()	{1,2,3} & {2,3,4}	{2,3}
Difference	- or .difference()	{1,2,3} - {2,3,4}	{1}
Symmetric Difference	^ or .symmetric_difference()	{1,2,3} ^ {2,3,4}	{1,4}

Set Methods

Method	Description
add(x)	Add element
update(iterable)	Add multiple elements
remove(x)	Remove element (error if missing)
discard(x)	Remove element safely
pop()	Remove & return random element
clear()	Remove all elements
union(other)	Returns union
intersection(other)	Returns intersection
difference(other)	Returns difference
symmetric_difference(other)	Returns elements not in both
issubset(other)	Check if set is a subset
issuperset(other)	Check if set is a superset
isdisjoint(other)	Check if no elements in common

- **A is a subset of B when everything in A is also in B**
- **A is a superset of B when A has everything B has**

- **Proper subset (⊂) → A is a subset of B, but A ≠ B. (A< B)**
- **Proper superset (⊃) → A is a superset of B, but A ≠ B. (A>B)**

Frozen Set

- **Immutable version of a set.**
- **Elements can't be added/removed.**

```
fs = frozenset([1, 2, 3])
# fs.add(4) → ERROR
```

Membership Testing

```
s = {1, 2, 3}
print(2 in s)
print(5 in s)
```

```
True
False
```

Iterating Through Set

```
for item in {1, 2, 3}:
    print(item)
```

```
1
2
3
```

Miscellaneous

Commenting

```
# This is a one-line comment
"""
This is a
multi-line comment
"""
```

Constants

- **Python does not have true constants.**
- **By convention, use UPPERCASE names for variables meant to stay constant.**

False Values

- **False**
- **None**
- **0 (int, float, complex, decimal, fraction)**
- **"" (empty string)**
- **[] (empty list)**
- **{} (empty dict)**
- **set() (empty set)**
- **tuple() (empty tuple)**

Everything else is truthy.

Identity vs Equality

- **== → checks value equality**
- **is → checks object identity (memory address)**

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True (same values)
print(a is b)  # False (different objects)
```

```
True
False
```

None

- **Represents no value / empty value.**
- **Commonly used for optional/default function arguments.**

```
x = None
if x is None:
   print("x is empty")
```

```
x is empty
```

Docstrings

- **Used to document functions, classes, modules.**
- **Written inside triple quotes right after definition.**

```
def add(a, b):
   """Return the sum of a and b"""
   return a + b

print(add.__doc__)
```

```
Return the sum of a and b
```

Pass Statement

- **Placeholder for empty blocks (to avoid syntax error).**

```
def todo_function():
   pass
```

OOP

- **OOP is a programming paradigm that organizes code into objects.**
- **Objects combine data (attributes/variables) and behavior (methods/functions).**
- **Helps with code reusability, modularity, and abstraction.**
- **Class → A blueprint/template for creating objects.**
- **Object (Instance) → A concrete occurrence of a class.**
- **Attributes → Variables inside a class.**
- **Methods → Functions inside a class.**

Syntax

```
class Car:
    def __init__(self, brand, model,color):
        self.__brand = brand        # Private attribute
        self._model = model         # Protected attribute
        self.color = color                  # Public attribute

    def drive(self):            # method
        print(f"{self.__brand} {self._model} is driving")

# Create object
my_car = Car("Tesla", "Model S", "Red")

# Access attributes & methods
print(my_car.color)
my_car.drive()
```

```
Red
Tesla Model S is driving
```

The __init__ Constructor

- **Special method called automatically when creating an object.**
- **Used to initialize attributes.**

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

Self Keyword

- **Refers to the current instance of the class.**
- **Must be the first parameter in methods.**

Class vs Instance Attributes

```
class Dog:
   species = "Mammal"   # Class attribute (shared)

   def __init__(self, name):
      self.name = name # Instance attribute (unique)

dog1 = Dog("Buddy")
dog2 = Dog("Max")

print(dog1.species)
print(dog1.name)
```

```
Mammal
Buddy
```

Encapsulation

- **Bundling data & methods together, restricting direct access.**

```
class Account:
   def __init__(self, balance):
      self.__balance = balance  # private attribute

   def deposit(self, amount):
      self.__balance += amount

   def get_balance(self):
      return self.__balance
```

Single Inheritance

- **A class can inherit attributes/methods from another.**

```
class Animal:
   def speak(self):
      print("Animal speaks")

class Dog(Animal):
   def speak(self):
      print("Dog barks")

dog = Dog()
dog.speak()
```

```
Dog barks
```

Multiple Inheritance

```
class A:
   def method_a(self):
      print("Method from A")

class B:
   def method_b(self):
      print("Method from B")

class C(A, B):  # Multiple inheritance
   def method_c(self):
      print("Method from C")

obj = C()
obj.method_a()
obj.method_b()
obj.method_c()
```

```
Method from A
Method from B
Method from C
```

```
class A:
   def method_a(self):
      print("Method from A")


class B:
   def method_a(self):
      print("Method from B")


class C(A, B):  # Multiple inheritance
   pass


obj = C()
obj.method_a()
```

```
Method from A
```

- **Python uses Method Resolution Order (MRO) to decide which parent to search first when multiple parents have the same method.**

Multilevel Inheritance

- **A class inherits from another class, which itself inherits from another class.**

```
class A:
   def method_a(self):
      print("Method from A")

class B(A):
   def method_b(self):
      print("Method from B")

class C(B):  # Inherits from B, which inherits from A
   def method_c(self):
      print("Method from C")

obj = C()
obj.method_a()
obj.method_b()
obj.method_c()
```

```
Method from A
Method from B
Method from C
```

Super Function

```
class A:
   def __init__(self):
      print("Init from A")

class B(A):
   def __init__(self):
      super().__init__()  # Calls A's constructor , we can use A instead of super as well
      print("Init from B")

obj = B()
```

```
Init from A
Init from B
```

Polymorphism

- **Same method name behaves differently depending on the object.**

```
class Animal:
    def speak(self):
            print("Animal speaks")

class Cat(Animal):
   def speak(self): print("Meow")

class Dog(Animal):
   def speak(self): print("Bark")

for animal in [Cat(), Dog()]:
   animal.speak()
```

```
Meow
Bark
```

Abstraction

- **Hiding implementation details, showing only essential features.**
- **Achieved with abstract classes (via abc module).**

```
from abc import ABC, abstractmethod

class Shape(ABC):
   @abstractmethod
   def area(self):
      pass # or raise NotImplementedError if we want a must-implementation in the child(pure virtual function)
```

Special Methods

Method	Purpose
__init__	Constructor
__str__	String representation
__len__	Length support
__add__	Overload + operator
__eq__	Equality check

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"{self.title} ({self.pages} pages)"

    def __len__(self):
        return self.pages

b = Book("Python Guide", 300)
print(b)
print(len(b))
```

```
Python Guide (300 pages)
300
```

Access Modifiers

- **Python does not enforce access restrictions like Java or C++, but it follows naming conventions:**
- **Public → default (variable)**
- **Protected → _variable (convention, still accessible)**
- **Private → __variable (name mangling)**

```
class Example:
    def __init__(self):
        self.public = "Public"
        self._protected = "Protected"
        self.__private = "Private"

obj = Example()
print(obj.public)      # ☑ Accessible
print(obj._protected)  # ⚠ Accessible but conventionally "protected"
# print(obj.__private)  # ❌ Error
print(obj._Example__private)  # ☑ Name mangling (still accessible)
```

Static Methods and Attributes

- **Static attributes are shared across all instances.**
- **Static methods don't take self, but belong to the class.**

```
class Example:
    count = 0  # Static attribute

    def __init__(self):
        Example.count += 1

    @staticmethod
    def greet():
        print("Hello from static method")

obj1 = Example()
obj2 = Example()

print(Example.count)
Example.greet()
```

```
2
Hello from static method
```

Operator Overloading

```
class Point:
   def __init__(self, x, y):
      self.x, self.y = x, y

   def __add__(self, other):
      return Point(self.x + other.x, self.y + other.y)

   def __eq__(self, other):
      return self.x == other.x and self.y == other.y

   def __repr__(self):
      return f"Point({self.x}, {self.y})"

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = Point(2, 3)

print(p1 + p2)
print(p1 == p3)
```

```
Point(6, 8)
True
```

Mro and Dir Function

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>]
```

```
class Example:
   def hello(self): pass

obj = Example()
print(dir(obj))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'hello']
```

Conclusion

This document has walked through Python fundamentals, advanced concepts, and object-oriented programming, while highlighting practical examples and important use cases. It was created with the intention of helping learners, developers, and enthusiasts strengthen their understanding of Python and apply it confidently in real-world scenarios.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution