My Python Course Notes

Structured Revision for Every Lesson

Siddharth Patel
HTW Berlin

July 12, 2025

Contents

1 Print Function – Full Usage Guide

```python
# PRINT FUNCTION – FULL USAGE GUIDE

# Basic Syntax:
# print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

# Parameters:
# *objects → One or more objects to be printed (comma-separated).
# sep      → String inserted between objects. Default is ' ' (space).
# end      → String appended after the last object. Default is '\n' (new line).
# file     → A file-like object (stream); default is sys.stdout.
# flush    → If True, forcibly flush the stream. Default is False.

# ------------------------------------------

# 1. Basic print
print("Hello, World!")  # Hello, World!

# 2. Printing multiple objects
print("Hello", "Python", 3)  # Hello Python 3

# 3. Using 'sep' to change separator
print("2025", "05", "27", sep="-")  # 2025-05-27

# 4. Using 'end' to avoid new line
print("Loading", end="...")  # Loading...

# 5. Using custom separator and end together
print("Name", "Age", sep=": ", end=" years\n")  # Name: Age years

# 6. Printing to a file
with open("output.txt", "w") as f:
    print("Saving this line to a file.", file=f)

# 7. Forcing flush (useful in loops/real-time output)
import time
for i in range(3):
    print(i, end=" ", flush=True)
    time.sleep(0.5)  # Output appears immediately

# 8. Printing escape characters
print("Line1\nLine2")         # New line
print("Tabbed\tSpace")        # Tab space
print("He said \"hello\"")    # Quotes inside string

# 9. Printing with formatted strings (f-strings)
name = "Siddhart"
age = 21
print(f"Hello, my name is {name} and I am {age} years old.")

# 10. Using print with unpacking
nums = [1, 2, 3, 4]
print(*nums)               # 1 2 3 4
print(*nums, sep=", ")     # 1, 2, 3, 4

# 11. Printing Unicode/emojis (note: removed for LaTeX safety)
print("Python is fun")
```

Additional Functions Used in This Lesson

Referenced Functions – Syntax and Output Type

Function	Syntax	Return / Output Type
`with open()`	`with open("file.txt", "w") as f:`	File object
`print(..., file=f)`	`print("text", file=f)`	Writes to file, returns `None`
`range()`	`range(3)` or `range(start, stop, step)`	Range object (iterable)
`time.sleep()`	`time.sleep(seconds)`	None (pauses execution)

2 Input Function – Full Usage Guide

```python
# INPUT FUNCTION – FULL USAGE GUIDE

# Basic Syntax:
# input(prompt='')

# Parameters:
# prompt → A string, written to standard output without a trailing newline,
#           to ask the user for input. Default is an empty string ''.
# Returns → A string entered by the user (always str type).
# Notes   → Always returns a string. You need to convert it using int(), float(), etc. if needed.

# ------------------------------------------

# 1. Basic usage with no prompt
user_input = input()
print("You entered:", user_input)

# 2. Input with a prompt
name = input("Enter your name: ")
print("Hello,", name)

# 3. Converting input to integer
age = int(input("Enter your age: "))
print("You will be", age + 1, "next year.")

# 4. Converting input to float
height = float(input("Enter your height in meters: "))
print("Your height in cm is", height * 100)

# 5. Reading multiple values (as strings)
x, y = input("Enter two words separated by space: ").split()
print("Word 1:", x)
print("Word 2:", y)

# 6. Reading and converting multiple values to int
a, b = map(int, input("Enter two integers: ").split())
print("Sum =", a + b)

# 7. Reading many values into a list of ints
numbers = list(map(int, input("Enter multiple numbers: ").split()))
print("You entered:", numbers)

# 8. Handling invalid input using try/except
try:
    salary = float(input("Enter your monthly salary: "))
    print("Yearly salary:", salary * 12)
except ValueError:
    print("Invalid input! Please enter a number.")


```

Referenced Functions – Syntax and Output Type

Function / Statement	Syntax	Return / Output Type
.split()	`string.split()` or `string.split("delimiter")`	List of strings
map()	`map(function, iterable)`	Map object (can be converted to list)
list()	`list(iterable)`	List object

try / except	`try:`
	`code`
	`except ErrorType:`
	`fallback`
		Flow control – no return value; handles runtime errors

3 Math Operators – Full Usage Guide

```python
# MATH OPERATORS – FULL USAGE GUIDE

# Basic Syntax:
# <operand1> <operator> <operand2>

# Operators:
# +   Addition            →  a + b
# -   Subtraction         →  a - b
# *   Multiplication      →  a * b
# /   Division            →  a / b
# //  Floor Division      →  a // b
# %   Modulus (Remainder) →  a % b
# **  Exponentiation      →  a ** b

# ------------------------------------------

# 1. Addition
print("1 + 1 =", 1 + 1)

# 2. Subtraction
print("2 - 3 =", 2 - 3)

# 3. Multiplication
print("4 * 5 =", 4 * 5)

# 4. Division (always returns float)
print("6 / 3 =", 6 / 3)

# 5. Floor Division (truncates decimals)
print("7 // 2 =", 7 // 2)

# 6. Rounded division result using round()
number1 = 1.85
number2 = 1.35
number3 = 1.5
print(f"{number1} rounded is:", round(number1))  # 2
print(f"{number2} rounded is:", round(number2))  # 1
print(f"{number3} rounded is:", round(number3))  # 2

# 7. Exponentiation
print("3 ** 3 =", 3 ** 3)  # 27

# 8. Modulus (Remainder)
print("20 / 6 =", 20 / 6)      # Division
print("20 % 6 =", 20 % 6)      # Remainder (2)

# 9. Operator Precedence in Python:
# 1. ()
# 2. **
# 3. * and /
# 4. + and -
# Evaluated left to right within same level
```

4 Strings – Full Usage Guide

```
# STRINGS – FULL USAGE GUIDE

# Basic Explanation:
# A string is a sequence of characters enclosed in single (' ') or double (" ") quotes.
# Strings are immutable in Python.

# ------------------------------------------
# 1. Creating Strings
name = 'math'     # single-quoted string
subject = "math"  # double-quoted string

# 2. String Addition and Printing
print("math" + "works")     # mathworks
print("math", "works")      # math works

# 3. String Multiplication
string1 = "hello"
string2 = "world"
number = 5

print(string1, string2)     # hello world
print(string1 + string2)    # helloworld
print(string1 * number)     # hellohellohellohellohello

# 4. Invalid Concatenation Example
# print(string1 + number)   # TypeError: can only concatenate str (not "int")

# STRING METHODS – TOP 10 DEFINITIONS

text = "hello WORLD"

# 5. capitalize()
# Returns string with first character uppercased, rest lowercased.
print(text.capitalize())  # Hello world

# 6. lower()
# Converts all characters to lowercase.
print(text.lower())       # hello world

# 7. title()
# Capitalizes first letter of each word.
print(text.title())       # Hello World

# 8. casefold()
# Aggressive lowercase, suitable for comparisons.
text2 = "Straße"
print(text2.casefold())   # strasse

# 9. upper()
# Converts all characters to uppercase.
print(text.upper())       # HELLO WORLD

# 10. count()
# Counts how many times a substring appears.
print(text.count("l"))            # 3
print(text.count("l", 3, 6))      # 1

# 11. find()
# Finds index of substring, or -1 if not found.
print(text.find("WORLD"))        # 6
```

```
print(text.find("not_here"))     # -1

# 12. replace()
# Replaces substring with another.
print(text.replace("WORLD", "Python"))     # hello Python
print(text.replace("l", "X", 2))           # heXXo WORLD

# 13. swapcase()
# Swaps uppercase to lowercase and vice versa.
print("Hello World".swapcase())  # hELLO wORLD

# 14. join()
# Joins elements of iterable with separator.
words = ["hello", "world"]
print("-".join(words))           # hello-world
```

Referenced Methods – Syntax and Output Type

Method / Function	Syntax	Return / Output Type
.capitalize()	str.capitalize()	str
.lower()	str.lower()	str
.title()	str.title()	str
.casefold()	str.casefold()	str
.upper()	str.upper()	str
.count()	str.count(substring, start, end)	int
.find()	str.find(substring, start, end)	int
.replace()	str.replace(old, new, count)	str
.swapcase()	str.swapcase()	str
.join()	"separator".join(iterable)	str

5 If, Else, and Conditional Operators

```
# IF / ELSE / ELIF – FULL USAGE GUIDE

# Basic Syntax:
# if condition:
#     block of code
# elif another_condition:
#     another block
# else:
#     fallback block

# Conditional Operators:
# ==   → Equal to                  → (x == y)
# !=   → Not equal to              → (x != y)
# <    → Less than                 → (x < y)
# <=   → Less than or equal to     → (x <= y)
# >    → Greater than              → (x > y)
# >=   → Greater than or equal to  → (x >= y)

# Logical Operators:
# and  → True if both are True     → (x > 5 and x < 10)
# or   → True if at least one is True → (x > 5 or x < 3)
# not  → Inverts the truth value   → not (x > 5)

# ------------------------------------------

# 1. Simple if statement
x = 10
if x > 5:
    print("x is greater than 5")

# 2. if-else statement
if x % 2 == 0:
    print("x is even")
else:
    print("x is odd")

# 3. if-elif-else ladder
grade = 85
if grade >= 90:
    print("Grade: A")
elif grade >= 80:
    print("Grade: B")
elif grade >= 70:
    print("Grade: C")
else:
    print("Grade: F")

# 4. Nested if statements
number = 42
if number > 0:
    if number % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    print("Negative number or zero")

# 5. Using logical 'and'
age = 25
if age > 18 and age < 65:
```

```
    print("Adult and working age")

# 6. Using logical 'or'
language = "Python"
if language == "Python" or language == "Java":
    print("Popular programming language")

# 7. Using logical 'not'
is_logged_in = False
if not is_logged_in:
    print("User not logged in")

# 8. Short form if-else (Ternary Expression)
# → Python provides a one-line shorthand for simple if-else statements.
# → Syntax: value_if_true if condition else value_if_false
# → Returns: One of two values based on the boolean result of the condition.

value = 8

# Traditional if-else version:
if value % 2 == 0:
    result = "Even"
else:
    result = "Odd"

print("Traditional form:", result)  # Even

# Shortened using ternary expression:
result = "Even" if value % 2 == 0 else "Odd"
print("Ternary form:", result)      # Even

```

Referenced Operators – Syntax and Output Type

Operator	Syntax	Return / Output Type
== (Equal)	`x == y`	`bool`
!= (Not Equal)	`x != y`	`bool`
< (Less Than)	`x < y`	`bool`
<= (Less Than or Equal)	`x <= y`	`bool`
> (Greater Than)	`x > y`	`bool`
>= (Greater Than or Equal)	`x >= y`	`bool`
and (Logical AND)	`x > 5 and x < 10`	`bool`
or (Logical OR)	`x < 5 or x > 10`	`bool`
not (Logical NOT)	`not (x > 5)`	`bool`
Ternary Expression	`value1 if condition else value2`	Result of `value1` or `value2`

6 While Loop – Full Usage Guide

```
# WHILE LOOP – FULL USAGE GUIDE

# Basic Syntax:
# while condition:
#     block of code

# The code inside the loop runs repeatedly as long as the condition is True.

# ------------------------------------------

# 1. Basic while loop
counter = 0
while counter < 5:
    print("Counter is:", counter)
    counter += 1  # same as: counter = counter + 1

# 2. Using break to exit loop early
i = 0
while True:
    if i == 3:
        print("Breaking at", i)
        break
    print(i)
    i += 1

# 3. Using continue to skip to next iteration
x = 0
while x < 5:
    x += 1
    if x == 3:
        continue  # skips printing 3
    print("x =", x)

# 4. while loop with else block
z = 0
while z < 3:
    print("z =", z)
    z += 1
else:
    print("Loop ended normally (no break)")

# 5. Infinite loop (be careful!)
# while True:
#     print("This runs forever")

# 6. Compound condition
n = 0
while n < 10 and n != 7:
    print(n)
    n += 2

# ------------------------------------------
# Counter update operators

# → counter += 1  → same as counter = counter + 1
# → counter -= 1  → same as counter = counter - 1
# → counter *= 2  → same as counter = counter * 2
# → counter /= 2  → same as counter = counter / 2

# Note:
```

```
# Python does NOT support the ++ or -- operators like other languages.
# Using x++ or x-- will cause a SyntaxError.
```

Referenced Keywords and Operators – Syntax and Output Type

Keyword / Operator	Syntax	Return / Effect
while	`while condition:`	Repeats block while condition is `True`
break	`break`	Immediately exits the nearest enclosing loop
continue	`continue`	Skips current iteration and continues with the next
+=	`x += y`	Updates: `x = x + y`
-=	`x -= y`	Updates: `x = x - y`
***=**	`x *= y`	Updates: `x = x * y`
/=	`x /= y`	Updates: `x = x / y`
++ / --	Not supported in Python	Causes `SyntaxError`

7 For Loop – Full Usage Guide

```python
# FOR LOOP – FULL USAGE GUIDE

# Basic Syntax:
# for variable in iterable:
#     block of code

# ------------------------------------------
# 1. Using range()
# range(stop) → from 0 to stop-1
# range(start, stop) → from start to stop-1
# range(start, stop, step)

for i in range(5):
    print("i =", i)

for i in range(2, 6):
    print("From 2 to 5:", i)

for i in range(10, 0, -2):
    print("Countdown by 2:", i)

# ------------------------------------------
# 2. Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("Fruit:", fruit)

# 3. Iterating over a string
text = "hello"
for char in text:
    print("Char:", char)

# 4. Iterating over a tuple
# A tuple is an ordered, immutable collection.
# You can iterate through it just like a list.
coordinates = (10, 20, 30)
for value in coordinates:
    print("Value:", value)


# 5. Iterating over a set
# A set is an unordered collection of unique elements.
# Iteration works, but order is not guaranteed.
unique_numbers = {1, 2, 3}
for num in unique_numbers:
    print("Unique:", num)


# 6. Iterating over a dictionary
# A dictionary stores key-value pairs.
# Iterating over it by default gives you the keys.
person = {"name": "Alice", "age": 25}
for key in person:
    print("Key:", key, "| Value:", person[key])

# 7. Iterating with .items()
# .items() returns a list of (key, value) pairs from a dictionary.
# Useful when you need both key and value at once.
for key, value in person.items():
    print(f"{key} => {value}")
```

```python

# 8. Using enumerate()
# enumerate() gives both the index and the item during iteration.
colors = ["red", "green", "blue"]
for index, color in enumerate(colors):
    print(f"{index}: {color}")


# 9. Using break
for n in range(5):
    if n == 3:
        break
    print("Breaking loop at:", n)

# 10. Using continue
for n in range(5):
    if n == 2:
        continue
    print("Continuing:", n)

# 11. Using else with for
for n in range(3):
    print(n)
else:
    print("Loop completed without break.")

# ---------------------------------------------
# range() - Recap:
# range(stop)
# range(start, stop)
# range(start, stop, step)
# returns a range object which is an iterable of numbers
```

Referenced Iteration Helpers – Syntax and Output Type

Function / Method	Syntax	Return / Output Type
tuple	`(a, b, c)`	Iterable (ordered, immutable)
set	`{a, b, c}`	Iterable (unordered, unique)
dict.items()	`dict.items()`	Iterable of `(key, value)` pairs
enumerate()	`enumerate(iterable)`	Iterable of `(index, item)` tuples

8 Functions – Full Usage Guide

```python
# FUNCTIONS - FULL USAGE GUIDE

# Basic Syntax:
# def function_name(parameters):
#     block of code

# ------------------------------------------
# 1. Defining a basic function
def greet():
    print("Hello from the function!")

# Call the function AFTER defining it
greet()

# ------------------------------------------
# 2. DOs and DON'Ts
# Don't call a function before it's defined:
# greet()  # This would raise NameError if called before definition

# Always define before calling:
def welcome():
    print("Welcome to Python!")

welcome()

# ------------------------------------------
# 3. Function that performs a task (prints something)
def print_sum(a, b):
    print("The sum is:", a + b)

print_sum(4, 5)  # Output: The sum is: 9

# ------------------------------------------
# 4. Function that calculates and returns a value
def get_sum(a, b):
    return a + b

result = get_sum(10, 20)
print("Returned sum:", result)

# ------------------------------------------
# 5. Local vs Global Variables

# Global variable
counter = 100

def increase_counter():
    # Local variable (does not affect global counter)
    counter = 0
    counter += 1
    print("Local counter:", counter)

increase_counter()
print("Global counter remains:", counter)

# ------------------------------------------
# 6. Using 'global' keyword to modify global variable
count = 0

def modify_global():
```

```
    global count
    count += 1
    print("Modified global count:", count)

modify_global()
print("Global count after function:", count)
```

9 Lists – Introduction

```python
# LISTS - INTRODUCTION

# What is a list?
# A list is a built-in data structure in Python that stores an ordered collection of items.
# Lists are mutable, meaning you can change their contents after creation.
# Defined using square brackets: []

# ------------------------------------------
# 1. Defining a simple list (homogeneous)
fruits = ["apple", "banana", "cherry"]
print("Fruits:", fruits)

# 2. List of numbers
numbers = [1, 2, 3, 4, 5]
print("Numbers:", numbers)

# 3. List of mixed data types
mixed = ["hello", 42, 3.14, True]
print("Mixed list:", mixed)

# 4. Empty list
empty = []
print("Empty list:", empty)

# 5. Nested list (list inside a list)
matrix = [[1, 2], [3, 4]]
print("Nested list:", matrix)

# 6. List using list() constructor
from_string = list("hello")
print("List from string:", from_string)  # ['h', 'e', 'l', 'l', 'o']

# 7. List from range()
range_list = list(range(5))
print("List from range():", range_list)  # [0, 1, 2, 3, 4]
```

List Types and Creation Methods

List Type	Syntax / Example	Notes
Homogeneous List	`[1, 2, 3, 4]`	All elements of the same type
String List	`["apple", "banana"]`	List of strings
Mixed-Type List	`["hi", 42, 3.14, True]`	Supports multiple data types
Empty List	`[]`	No elements yet
Nested List	`[[1, 2], [3, 4]]`	List inside a list
From String	`list("abc")`	Converts string into list of characters
From Range	`list(range(5))`	Converts range object into list

10 List Methods – General Usage

```
# LIST METHODS – GENERAL PURPOSE

# Lists are mutable and support many built-in methods for adding, removing, searching, and modifying.

# 1. append()
# → Adds a single element to the end
items = ["pen", "book"]
items.append("pencil")
print("append():", items)

# 2. insert()
# → Inserts an element at a specific index
items.insert(1, "eraser")
print("insert():", items)

# 3. remove()
# → Removes the first matching value
items.remove("book")
print("remove():", items)

# 4. pop()
# → Removes and returns element at index (default = last)
removed = items.pop()
print("pop():", removed)
print("After pop:", items)

# 5. clear()
# → Empties the list
temp = ["a", "b"]
temp.clear()
print("clear():", temp)

# 6. copy()
# → Returns a shallow copy
original = ["x", "y", "z"]
cloned = original.copy()
print("copy():", cloned)

# 7. extend()
# → Adds multiple elements from another iterable
tools = ["pen", "pencil"]
tools.extend(["marker", "sharpener"])
print("extend():", tools)

# 8. index()  [Use with care]
# → Finds index of first match; raises error if not found
names = ["Alice", "Bob", "Charlie"]
try:
    idx = names.index("Bob")
    print("index():", idx)
except ValueError:
    print("Name not found")

# Alternative using 'in'
print("Eve" in names)  # False

# 9. count()
# → Counts number of times an item appears
letters = ["a", "b", "a", "c", "a"]
print("count():", letters.count("a"))  # 3
```

```python

# 10. reverse()
# → Reverses the list in-place
words = ["start", "middle", "end"]
words.reverse()
print("reverse():", words)

# 11. sort() [Only works if items are comparable]
# → Sorts the list (only if all items can be compared)
languages = ["python", "c", "java"]
languages.sort()
print("sort():", languages)
```

Common List Methods – Overview Table

Method	Description	Return Value
`append(x)`	Add element `x` to the end	`None`
`insert(i,x)`	Insert `x` at index `i`	`None`
`remove(x)`	Remove first occurrence of `x`	`None`
`pop(i)`	Remove and return item at index `i` (last by default)	Element
`clear()`	Remove all items from the list	`None`
`copy()`	Return a shallow copy of the list	List copy
`extend(iter)`	Append elements from iterable	`None`
`index(x)`	Return first index of `x` (error if not found)	Integer
`count(x)`	Count occurrences of `x`	Integer
`reverse()`	Reverse items in-place	`None`
`sort()`	Sort the list in-place	`None`

11 List Methods – Numeric Lists Only

```python
# LIST METHODS – NUMERIC LISTS ONLY

# These methods are especially useful and commonly used with lists that contain only numbers.

# Numeric list for demonstration
numbers = [5, 2, 8, 3, 5, 1, 8]

# 1. sort()
# → Sorts the list in ascending order (in-place)
numbers.sort()
print("sort():", numbers)  # [1, 2, 3, 5, 5, 8, 8]

# 2. reverse()
# → Reverses the list order in-place
numbers.reverse()
print("reverse():", numbers)  # [8, 8, 5, 5, 3, 2, 1]

# 3. count()
# → Counts occurrences of a specific number
print("count(5):", numbers.count(5))  # 2

# 4. max()
# → Returns the largest number in the list
print("max():", max(numbers))  # 8

# 5. min()
# → Returns the smallest number in the list
print("min():", min(numbers))  # 1

# 6. sum()
# → Returns the sum of all elements
print("sum():", sum(numbers))  # 32

# 7. average (manual)
# → Average = sum / count
average = sum(numbers) / len(numbers)
print("Average:", average)  # 4.571...

# 8. sorted()
# → Returns a new sorted list without modifying the original
original = [10, 3, 7]
sorted_list = sorted(original)
print("original:", original)       # [10, 3, 7]
print("sorted():", sorted_list)    # [3, 7, 10]
```

List Methods for Numbers – Reference Table

Method	Description	Return Value
`sort()`	Sort list in ascending order (in-place)	`None`
`reverse()`	Reverse the list (in-place)	`None`
`count(x)`	Count how many times `x` appears	Integer
`max(lst)`	Return maximum element	Element
`min(lst)`	Return minimum element	Element
`sum(lst)`	Sum of all list items	Numeric
`sorted(lst)`	Return a new sorted list	New list
`avg = sum()/len()`	Compute average (manual)	Float

12 2D Lists and Nested Loops

```python
# 2D LISTS AND NESTED LOOPS - FULL GUIDE

# A 2D list is a list of lists (matrix/grid style)

# Defining a 3x3 grid
num_grid = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# ------------------------------------------
# 1. Accessing a specific element (row 0, column 0)
print("Top-left element:", num_grid[0][0])  # Output: 1

# 2. Accessing other positions
print("Middle element:", num_grid[1][1])    # Output: 5
print("Bottom-right:", num_grid[2][2])      # Output: 9

# ------------------------------------------
# 3. Iterating through 2D list using nested for-loops

# Outer loop goes over each row
for row in num_grid:
    # Inner loop goes over each column (element in that row)
    for column in row:
        print(column, end=" ")
    print()  # For new line after each row
```

13 Tuples – Full Usage Guide

```
# TUPLES – FULL USAGE GUIDE

# 1. What is a tuple?
# → A tuple is an immutable, ordered collection of items.
# → Defined using parentheses: ()

# 2. Tuple vs List – Key Differences:
# → Tuples use ()     → Lists use []
# → Tuples are immutable → Lists are mutable (can be changed)
# → Tuples have fewer built-in methods
# → Tuples are faster and used for fixed data

# ------------------------------------------
# 3. Defining tuples

empty_tuple = ()
single_item = ("apple",)   # Note the comma!
fruits = ("apple", "banana", "cherry")

print("Fruits tuple:", fruits)
print("First item:", fruits[0])

# ------------------------------------------
# 4. What can we do with tuples?

# Tuple concatenation
more_fruits = ("mango", "orange")
combined = fruits + more_fruits
print("Combined tuple:", combined)

# Tuple unpacking
x, y, z = fruits
print("Unpacked:", x, y, z)

# Cannot change a tuple element
# fruits[0] = "kiwi"  # TypeError

# Cannot append or remove items
# fruits.append("kiwi")  # AttributeError
# fruits.remove("banana")  # AttributeError

# You can delete the whole tuple
temp = (1, 2, 3)
del temp
# print(temp)  # NameError if uncommented

# ------------------------------------------
# 5. Built-in functions usable with tuples

values = (10, 20, 5, 30)

# 1. len()
print("Length:", len(values))  # 4

# 2. max()
print("Maximum:", max(values))  # 30

# 3. min()
print("Minimum:", min(values))  # 5

```

```python
# 4. tuple() constructor
sample_list = ["x", "y", "z"]
converted = tuple(sample_list)
print("Converted to tuple:", converted)
```

Tuple Functions – Syntax and Return Type

Function	Syntax	Return Type
len()	`len(tuple)`	`int` (length of tuple)
max()	`max(tuple)`	Largest item from tuple
min()	`min(tuple)`	Smallest item from tuple
tuple()	`tuple(iterable)`	A new tuple object

14 Sets – General Usage Guide

```python
# SETS - GENERAL USAGE GUIDE

# 1. What is a set?
# → A set is an unordered, unindexed collection with no duplicate elements.
# → Defined using curly braces {} or the set() constructor.

# ------------------------------------------
# 2. Creating a set

# From a list with duplicates
values = [2, 3, 3, 4, 5]
unique_values = set(values)
print("Set from list:", unique_values)  # {2, 3, 4, 5}

# Direct set definition
colors = {"red", "green", "blue"}
print("Set of colors:", colors)

# Empty set must be created using set()
empty_set = set()
print("Empty set:", empty_set)

# ------------------------------------------
# 3. General set methods

# 1. add()
colors.add("yellow")
print("After add():", colors)

# 2. remove()
colors.remove("green")  # Raises error if not found
print("After remove():", colors)

# 3. discard()
colors.discard("blue")  # Safe: no error if not found
print("After discard():", colors)

# 4. pop()
item = colors.pop()  # Removes random item (sets are unordered)
print("Popped:", item)

# 5. clear()
numbers = {1, 2, 3}
numbers.clear()
print("After clear():", numbers)

# 6. copy()
original = {"a", "b"}
cloned = original.copy()
print("Copy of set:", cloned)

# 7. update()
colors.update({"white", "black"})
print("After update():", colors)

# ------------------------------------------
# Sets do NOT support indexing or duplicate elements.
```

General Set Methods – Syntax and Return Type

Method	Description	Return Type
`add(x)`	Adds element `x` to the set	`None`
`remove(x)`	Removes `x`; error if not found	`None`
`discard(x)`	Removes `x`; no error if not found	`None`
`pop()`	Removes and returns an arbitrary element	Element
`clear()`	Removes all items from the set	`None`
`copy()`	Returns a shallow copy of the set	Set copy
`update(iter)`	Adds elements from another iterable	`None`

15 Sets – Mathematical Operations

```python
# SETS - MATHEMATICAL OPERATIONS

# We use sets to perform classic mathematical operations like:
# union, intersection, difference, symmetric difference, etc.

a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

# ------------------------------------------
# 1. union() - combines both sets
print("Union:", a.union(b))       # {1, 2, 3, 4, 5, 6}
print("Using | :", a | b)

# 2. update() - adds elements from b to a
a1 = a.copy()
a1.update(b)
print("Update:", a1)

# ------------------------------------------
# 3. intersection() - common elements
print("Intersection:", a.intersection(b))  # {3, 4}
print("Using & :", a & b)

# 4. intersection_update() - keeps only common elements in a
a2 = a.copy()
a2.intersection_update(b)
print("Intersection Update:", a2)

# ------------------------------------------
# 5. difference() - items in a but not in b
print("Difference (a - b):", a.difference(b))  # {1, 2}
print("Using - :", a - b)

# 6. difference_update() - removes items in b from a
a3 = a.copy()
a3.difference_update(b)
print("Difference Update:", a3)

# ------------------------------------------
# 7. symmetric_difference() - elements in a or b but not both
print("Symmetric Difference:", a.symmetric_difference(b))  # {1, 2, 5, 6}
print("Using ^ :", a ^ b)

# 8. symmetric_difference_update() - modifies a to symmetric diff
a4 = a.copy()
a4.symmetric_difference_update(b)
print("Symmetric Difference Update:", a4)

# ------------------------------------------
# 9. issubset()
print("Is a subset of b:", a.issubset(b))  # False

# 10. issuperset()
print("Is a superset of b:", a.issuperset(b))  # False

# 11. isdisjoint()
x = {10, 20}
print("Is disjoint:", a.isdisjoint(x))  # True (no common elements)
```

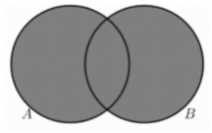
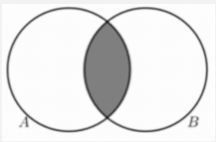
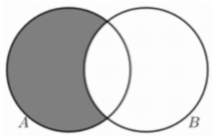
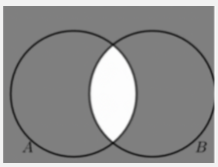
Set Operations – Syntax and Result

Method / Operator	Description	Return Type
`a.union(b)` / `a | b`	All unique elements from both sets	New set
`a.update(b)`	Adds all elements from b to a	`None` (in-place)
`a.intersection(b)` / `a & b`	Elements common to both sets	New set
`a.intersection_update(`	Keeps only common elements in a	`None` (in-place)
`a.difference(b)` / `a - b`	Elements in a but not in b	New set
`a.difference_update(b)`	Removes items in b from a	`None` (in-place)
`a.symmetric_difference` / `a ^ b`	Elements in a or b, not both	New set
`a.symmetric_difference`	Updates a to symmetric diff	`None` (in-place)

Set Comparison – Membership and Relationship

Method	Description	Returns
`a.issubset(b)`	True if all elements of a are in b	Boolean
`a.issuperset(b)`	True if a contains all elements of b	Boolean
`a.isdisjoint(b)`	True if a and b have no elements in common	Boolean

Set Operations – Visual Explanation

Set Operation	Diagram	Explanation	`In Python`
Union		$A \cup B$ is the set of all elements that are a member of A, or B, or both.	union()
Intersection		$A \cap B$ is the set of all elements that are a member of both A and B.	intersection()
Difference		$A \setminus B$ is the set of all elements of A that are not in B.	difference()
Symmetric Difference		$A \Delta B$ is the set of elements in either A or B, but not both.	symmetric_difference()

16 Dictionary

```python
# DICTIONARIES – BASIC USAGE GUIDE

# 1. What is a dictionary?
# → A dictionary is a collection of key-value pairs.
# → Each key must be unique, and values can be of any type.

# Unlike lists or tuples (which store values), dictionaries store data like a "map":
#   Key   →   Value

# ------------------------------------------
# 2. Creating dictionaries

# Method 1: Using curly braces
student = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}

# Method 2: Using dict() constructor
profile = dict(name="Bob", age=25)
print("Created with dict():", profile)

# ------------------------------------------
# 3. Accessing values

print("Name:", student["name"])  # Alice
print("Age:", student["age"])    # 21

# If key doesn't exist, it raises a KeyError
# print(student["grade"])  # KeyError

# ------------------------------------------
# 4. Adding new key-value pairs

student["grade"] = "A"
print("Updated student:", student)

# ------------------------------------------
# 5. Iterating over a dictionary (keys by default)

for key in student:
    print("Key:", key, "| Value:", student[key])
```

17 Dictionary – Most Important Methods

```python
# DICTIONARY METHODS - MOST IMPORTANT & COMMON

person = {
    "name": "Alice",
    "age": 22,
    "country": "Germany"
}

# ------------------------------------------
# 1. get(key[, default])
# → Returns the value for the key if it exists; otherwise returns default (or None)
print("get('name'):", person.get("name"))         # Alice
print("get('gender'):", person.get("gender"))     # None
print("get('gender', 'Not specified'):", person.get("gender", "Not specified"))

# ------------------------------------------
# 2. keys()
# → Returns a view of all keys
print("Keys:", person.keys())  # dict_keys(['name', 'age', 'country'])

# ------------------------------------------
# 3. values()
# → Returns a view of all values
print("Values:", person.values())  # dict_values(['Alice', 22, 'Germany'])

# ------------------------------------------
# 4. items()
# → Returns a view of all key-value pairs as tuples
print("Items:", person.items())  # dict_items([('name', 'Alice'), ('age', 22), ...])

# ------------------------------------------
# 5. pop(key)
# → Removes a key and returns its value
age = person.pop("age")
print("Popped 'age':", age)
print("After pop():", person)

# ------------------------------------------
# 6. popitem()
# → Removes and returns the last inserted (key, value) pair
last_item = person.popitem()
print("Popped last item:", last_item)
print("After popitem():", person)

# ------------------------------------------
# 7. update(other_dict)
# → Merges another dictionary into current one
person.update({"name": "Bob", "gender": "Male"})
print("After update():", person)

# ------------------------------------------
# 8. clear()
# → Removes all key-value pairs from dictionary
temp = {"x": 1, "y": 2}
temp.clear()
print("After clear():", temp)

# ------------------------------------------
# 9. copy()
# → Returns a shallow copy of the dictionary
```

```
original = {"a": 1, "b": 2}
duplicate = original.copy()
print("Copy:", duplicate)
```

```
# FAMOUS DICTIONARY EXAMPLES USING: get(), keys(), values(), items()

student = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science",
    "grade": "A"
}

# ------------------------------------------
# 1. .get() - Safe value access
# → Famous for avoiding KeyError if the key doesn't exist

# GOOD: get() returns default instead of crashing
print("Country:", student.get("country", "Not specified"))  # Not specified

# BAD: This would raise KeyError
# print(student["country"])

# ------------------------------------------
# 2. .keys() - Useful for looping or checking presence
print("Keys in student dict:")
for key in student.keys():
    print("-", key)

# Check if "age" is a key
if "age" in student.keys():
    print("Yes, 'age' is a key.")

# ------------------------------------------
# 3. .values() - Check or search values
print("Values in student dict:")
for value in student.values():
    print("-", value)

# Check if a specific value exists
if "Computer Science" in student.values():
    print("Found the major!")

# ------------------------------------------
# 4. .items() - Iterate over both key and value (most common in loops)
print("Student Info:")
for key, value in student.items():
    print(f"{key} → {value}")

# ------------------------------------------
# BONUS: Use in condition
if "grade" in student:
    if student["grade"] == "A":
        print("Excellent student!")

# ------------------------------------------
# 5. Finding Key from Value (reverse lookup)

# Let's say we want the key for value "A"
target_value = "A"
```

```
57
58 # Using a loop to search for matching value
59 for key, value in student.items():
60     if value == target_value:
61         print(f"Key for value '{target_value}' is: {key}")
62
63 # ------------------------------------------
64 # 6. Finding Value from Key (already known way)
65 # Just standard access
66 print("Grade is:", student["grade"])  # A
```

Common Dictionary Methods – Reference Table

Method	Description	Return Type
`get(key, default)`	Returns value if key exists, else default or None	Value or None
`keys()`	Returns all keys in the dictionary	dict_keys view
`values()`	Returns all values in the dictionary	dict_values view
`items()`	Returns key-value pairs as tuples	dict_items view
`pop(key)`	Removes and returns value for given key	Value
`popitem()`	Removes and returns the last key-value pair	(key, value) tuple
`update(dict)`	Updates dict with another dict or key-value pairs	None
`clear()`	Removes all items from the dictionary	None
`copy()`	Returns a shallow copy of the dictionary	New dict

18 Comprehensions – List Comprehensions

```python
# LIST COMPREHENSIONS – PYTHON GUIDE

# → List comprehensions provide a concise way to create or transform lists.
# → Syntax: [ expression for item in iterable if condition ]
#
# Parts:
# expression → The output expression (what should be added to the new list)
# item       → The looping variable
# iterable   → Any iterable like range, list, tuple, etc.
# condition  → (Optional) Only items satisfying this condition are included


# ------------------------------------------
# 1. Squares of numbers from 1 to 10
squares = [x**2 for x in range(1, 11)]
print("Squares:", squares)
# Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# ------------------------------------------
# 2. Filter odd squares from the above list
odd_squares = [x for x in squares if x % 2 != 0]
print("Odd Squares:", odd_squares)
# Output: [1, 9, 25, 49, 81]

# ------------------------------------------
# 3. Filter words starting with 'b'
words = ['apple', 'banana', 'cherry', 'durian', 'elderberry']
b_words = [word for word in words if word.startswith('b')]
print("Words starting with 'b':", b_words)
# Output: ['banana']

# ------------------------------------------
# 4. Example without condition (copying elements)
copied = [w for w in words]
print("Copied:", copied)

# ------------------------------------------
# 5. Example with inline modification
uppercased = [w.upper() for w in words]
print("Uppercased:", uppercased)
```

List Comprehension Syntax Breakdown

Component	Description
`expression`	The result expression evaluated and added to the list. Can be a variable or formula.
`item`	The looping variable assigned in each iteration.
`iterable`	An iterable like `range()`, list, tuple, or string.
`condition` (optional)	A filter. Only items satisfying this condition are included.

19 Comprehensions – Tuple Comprehensions

```python
# TUPLE COMPREHENSIONS – PYTHON GUIDE

# ➜ Tuple comprehensions work just like list comprehensions,
#   but use parentheses ( ) and the tuple() constructor.
#
# ➜ Syntax:
#   tuple_result = tuple(expression for item in iterable if condition)

# Note:
# Although the syntax uses ( ), it does not create a tuple by default –
# you must explicitly wrap the comprehension in the tuple() constructor.


# ------------------------------------------
# 1. Tuple of squares (1 to 10)
squaretuples = tuple(x**2 for x in range(1, 11))
print("Tuple of Squares:", squaretuples)
# Output: (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

# ------------------------------------------
# 2. Filter odd numbers from existing list of squares
squares = [x**2 for x in range(1, 11)]
odd_squaretuples = tuple(x for x in squares if x % 2 != 0)
print("Odd Squares as Tuple:", odd_squaretuples)
# Output: (1, 9, 25, 49, 81)

# ------------------------------------------
# 3. Filter words starting with 'b' from a tuple
words = ('apple', 'banana', 'cherry', 'durian', 'elderberry')
b_words = tuple(word for word in words if word.startswith('b'))
print("Words starting with 'b':", b_words)
# Output: ('banana',)
```

Tuple Comprehensions – Syntax Overview

Component	Description
`tuple()` wrapper	Required to convert the generator into a tuple.
`(expression for item in iterable)`	Generator-style expression inside `tuple()`.
`expression`	Defines what value to include.
`condition` (optional)	Filters the items added to the final tuple.

20 Comprehensions – Dictionary and Set

```python
# DICTIONARY & SET COMPREHENSIONS – PYTHON GUIDE

# ➜ Comprehensions allow building new dictionaries and sets in a single line.
# ➜ They use similar syntax to list comprehensions with minor differences.

# ------------------------------------------
# DICTIONARY COMPREHENSION
# Syntax:
# {key_expression: value_expression for item in iterable if condition}

# Example: Square values in a dictionary only if value is even
numbers = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

squares = {key: value**2 for key, value in numbers.items() if value % 2 == 0}
print("Squared Even Values:", squares)
# Output: {'b': 4, 'd': 16}

# ------------------------------------------
# SET COMPREHENSION
# Syntax:
# {expression for item in iterable if condition}

# Example: Squaring all values in a set
nums = {-2, -1, 0, 1, 2}
squared_set = {num**2 for num in nums}
print("Squared Set:", squared_set)
# Output: {0, 1, 4}
```

Comprehension Syntax Comparison Table

Comprehension Type	Syntax Example
List	`[x for x in iterable if condition]`
Tuple	`tuple(x for x in iterable if condition)`
Dictionary	`{k: v for (k, v) in iterable if condition}`
Set	`{x for x in iterable if condition}`

21 Classes and Objects

```python
# KLASSEN UND OBJEKTE – PYTHON GUIDE

# → A class is a blueprint or template for creating objects.
# → An object is a concrete instance of a class with its own data and behavior.

# ------------------------------------------
# Example: A simple bank account class

class Account:
    # The constructor method (__init__) is called when an object is created
    def __init__(self, name, number):
        self.name = name          # Account holder
        self.number = number      # Account number
        self.balance = 0          # Initial balance is 0

    # Method to deposit money into the account
    def deposit(self, amount):
        self.balance += amount    # Increase balance by the deposit amount

    # Method to withdraw money from the account
    def withdraw(self, amount):
        if self.balance < amount:
            raise ValueError("Insufficient funds")
        self.balance -= amount    # Decrease balance by the withdrawal amount

# ------------------------------------------
# Creating instances (objects) of the Account class

account1 = Account("Otto Schmidt", "123456789")
account2 = Account("Luisa Meier", "987654321")

# ------------------------------------------
# Using object methods
account1.deposit(100)
account1.withdraw(40)

account2.deposit(200)

# ------------------------------------------
# Accessing attributes
print("Account 1 - Name:", account1.name)
print("Account 1 - Balance:", account1.balance)

print("Account 2 - Name:", account2.name)
print("Account 2 - Balance:", account2.balance)
```

Overview – Class Terminology

Concept	Explanation
`class`	A blueprint or template for creating objects
`object`	An instance of a class with its own data (attributes)
`self`	Refers to the current object instance (used inside methods)
`__init__()`	The constructor method, automatically called when creating an object
Attribute	A variable attached to an object (e.g., `self.name`)
Method	A function defined inside a class (e.g., `deposit()`)
Instance	A specific object created from a class (e.g., `account1`)

22 Objects – Using Class Instances

```python
# OBJEKTVERWENDUNG – KONTOBEISPIEL

# → This example shows how to create and interact with objects
# → Using methods defined inside the Account class

# Assuming the class Account is already defined

# ------------------------------------------
# Creating objects from the class

account1 = Account("Otto Schmidt", "123456")
account2 = Account("Luisa Meier", "789012")

# ------------------------------------------
# Depositing money
account1.deposit(1500)
account2.deposit(500)
account2.deposit(800)

# ------------------------------------------
# Withdrawing money (if enough balance)
account1.withdraw(500)
print("Account 1 Balance:", account1.balance)  # Output: 1000

account2.withdraw(200)
print("Account 2 Balance:", account2.balance)  # Output: 1100

# ------------------------------------------
# Note:
# account1.balance refers to the attribute of a specific object
# Inside the class, we use self.balance since self refers to the current instance
```

Object Usage vs. `self` in Classes

Context	Syntax / Explanation
Outside the Class	`account1.balance`, `account2.deposit(100)`
Inside the Class	`self.balance`, `self.deposit(...)`
Why?	`self` refers to the object on which the method was called
Example	`account1.withdraw(500)` → `self = account1` inside the method

23 Private Attributes, Getter and __str__ Method

```python
# PRIVATE ATTRIBUTES – GETTERS & __str__ EXAMPLE

# → A class with private attributes
# → Demonstrates data protection using __ (double underscore)
# → Includes a getter method and __str__() for safe access

class Account:
    def __init__(self, name, number):
        self.__name = name           # private attribute
        self.__number = number       # private attribute
        self.__balance = 0           # private attribute initialized to 0

    def deposit(self, amount):
        self.__balance += amount     # internally modifying balance is allowed

    def withdraw(self, amount):
        if self.__balance < amount:
            raise ValueError("Insufficient funds")
        self.__balance -= amount

    def getBalance(self):
        return self.__balance        # getter for balance

    def __str__(self):
        return f"Name: {self.__name} Kontonummer: {self.__number} Guthaben: {self.__balance}"

# ------------------------------------------
# Object creation and access
account3 = Account("Simon Frank", 6543211)
account3.deposit(1000)

# Attempting direct access (NOT RECOMMENDED)
# account3.__balance -= 1000000  Illegal access (will not modify balance)

# Safe balance access
print(account3.getBalance())  #1000

# Print uses __str__ internally
print(account3)  # Name: Simon Frank Kontonummer: 6543211 Guthaben: 1000
```

Private Attributes and Access Methods

Access Type	Description
self.__attribute	Private attributes within the class. Only accessible inside the class.
obj.getAttribute()	Getter method for safe external access.
obj.__attribute	Direct access from outside is not allowed (causes error or unexpected behavior).
print(obj)	Automatically calls the `__str__()` method, which returns formatted info.

24 Inheritance – Extending Classes with super()

```python
# INHERITANCE IN PYTHON (Vererbung)
# → Extending a base class (Account) with a derived class (SavingsAccount)

# Assuming Account class is already defined as in earlier chapters
class Account:
    def __init__(self, name, number):
        self.__name = name
        self.__number = number
        self.__balance = 0

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

    def withdraw(self, amount):
        if self.__balance < amount:
            raise ValueError("Insufficient funds")
        self.__balance -= amount

    def getBalance(self):
        return self.__balance

    def __str__(self):
        return f"Name: {self.__name}, Kontonummer: {self.__number}, Guthaben: {self.__balance}"

# Subclass – Sparkonto mit Zinsen
class SavingsAccount(Account):
    def __init__(self, name, number, balance, interest):
        super().__init__(name, number)      # Call parent constructor
        self.deposit(balance)               # Set starting balance using inherited method
        self.__interest = interest          # Private interest rate for this account

    def add_interest(self):
        # Compute interest and deposit
        interest_amount = self.getBalance() * self.__interest
        self.deposit(interest_amount)

# ----------------------------------------
# Creating and using subclass objects

savings_account = SavingsAccount("Max Mustermann", "123456", 1000, 0.05)
print(savings_account.getBalance())    # Before interest: 1000

savings_account.add_interest()         # Add interest
print(savings_account.getBalance())    # After interest: 1050.0
```

Inheritance – Overview

Term	Explanation
`class Subclass(BaseClass):`	Creates a derived class that inherits from the base class.
`super().__init__()`	Calls the constructor of the base class.
`self.deposit(...)`	Calls a method from the base class.
`self.__attribute`	Private attribute of the base class – accessible only via methods.
`add_interest()`	New method of the subclass to calculate and add interest.

25 File Access in Python

```python
# Lesson 25: File Access in Python

# ------------------------------------------
# 1. Opening and Closing Files
# ------------------------------------------

# Open a text file for reading
txtfile = open('file.txt', 'r')

# Open a binary file for writing
binfile = open('file.bin', 'wb')

# Open with encoding (e.g., for German umlauts)
file_with_umlauts = open('umlauts.txt', 'r', encoding='utf-8')

# Always close files after use
txtfile.close()
binfile.close()

# ------------------------------------------
# 2. Recommended: Using `with` for safer file handling
# ------------------------------------------

with open('file.txt', 'r') as file:
    content = file.read()
print(content)  # file is automatically closed after this block

# ------------------------------------------
# 3. Reading Entire File vs. Line-by-Line
# ------------------------------------------

# Read all lines into a list
with open('file.txt', 'r') as file:
    lines = file.readlines()

for i, line in enumerate(lines, start=1):
    print(f"{i}: {line}", end='')

# Read file line by line using readline()
with open('file.txt', 'r') as file:
    line = file.readline()
    while line != '':
        print(line, end='')
        line = file.readline()

# Even shorter with iterator
with open('file.txt', 'r') as file:
    for line in file:
        print(line, end='')

# ------------------------------------------
# 4. Writing to a Text File
# ------------------------------------------

with open('file.txt', 'w') as file:
    file.write('This is the first line.\n')
    file.write('This is the second line.\n')
    file.write('This is the third line.\n')

# ------------------------------------------
```

```python
# 5. Reading from a Binary File
# ------------------------------------------
with open('file.bin', 'rb') as file:
    bytes_data = file.read(10)  # read 10 bytes

for byte in bytes_data:
    print(byte)

# ------------------------------------------
# 6. Writing to a Binary File
# ------------------------------------------
with open('file.bin', 'wb') as file:
    file.write(bytes([0, 1, 2, 3, 4, 5]))
```

File Access – Modes and Methods Overview

Concept / Function	Description
open('file.txt', 'r')	Open text file for reading
open('file.txt', 'w')	Open text file for writing (overwrites content)
open(..., encoding='utf-8')	Open file with specific character encoding
file.read()	Read entire file content as a string
file.readlines()	Read all lines into a list
file.readline()	Read one line at a time
for line in file	Loop through file line by line
file.write(...)	Write a string to file (no auto line break)
with open(...) as f	Automatically closes file after block
file.close()	Manually closes file (not needed with `with`)
open(..., 'rb')	Open binary file for reading
open(..., 'wb')	Open binary file for writing
bytes([0,1,2])	Convert list to byte array for binary write

```python
# File Access Modes - What They Mean

# ------------------------------------------
# Common Text File Modes
# ------------------------------------------

# 'r'  = Read mode (default); file must exist
# 'w'  = Write mode; creates file if it doesn't exist, overwrites if it does
# 'a'  = Append mode; writes to end of file, creates it if it doesn't exist
# 'x'  = Create mode; creates file, fails if it already exists

# Example:
file = open('example.txt', 'r')  # Read
file.close()

file = open('example.txt', 'w')  # Write (overwrites)
file.close()

file = open('example.txt', 'a')  # Append
file.close()

file = open('new_file.txt', 'x')  # Create (raises error if exists)
file.close()

# ------------------------------------------
# Binary File Modes (for non-text data like images or audio)
# ------------------------------------------

# 'rb' = Read binary
# 'wb' = Write binary (overwrite)
# 'ab' = Append binary
# 'xb' = Create binary

# Example:
with open('image.jpg', 'rb') as file:
    data = file.read()

with open('audio.mp3', 'wb') as file:
    file.write(bytes([1, 2, 3]))
```

File Access Modes – Overview

Mode	Description	Use Case	`Example Code`
'r'	Read (text) – file must exist	Reading config or data files	`open('file.txt', 'r')`
'w'	Write (text) – overwrite or create	Writing logs or reports	`open('log.txt', 'w')`
'a'	Append (text) – add to end, create if needed	Appending logs or results	`open('log.txt', 'a')`
'x'	Exclusive write – fail if file exists	Safe one-time file creation	`open('unique.txt', 'x')`
'rb'	Read binary – file must exist	Reading image or binary data	`open('image.png', 'rb')`
'wb'	Write binary – overwrite or create	Saving structured bytes or images	`open('output.bin', 'wb')`
'ab'	Append binary – add to end, create if needed	Streaming sensor or audio bytes	`open('audio.raw', 'ab')`
'xb'	Exclusive binary write – fail if file exists	Writing protected firmware or assets	`open('firmware.bin', 'xb')`

26 Exceptions in Python

```python
# Lesson: Exceptions in Python

# 1. Handling runtime errors with try-except blocks

try:
    result = 5 / 0
except ZeroDivisionError:
    print("An error occurred: Division by zero")

# 2. Handling file errors specifically and generally

try:
    file = open("my_file.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("The file was not found")
except:
    print("An unspecified error has occurred")

# 3. Raising exceptions manually

x = -1
if x < 0:
    raise ValueError("The value must not be negative.")

# 4. Handling raised exceptions with custom messages

try:
    x = int(input("x = "))
    if x < 0:
        raise ValueError("The value must not be negative.")
except ValueError as exception:
    print(exception)

```

Common Python Exception Types

Exception Type	Description
`Exception`	Base class for all built-in exceptions.
`ArithmeticError`	Base class for numeric calculation errors.
`ZeroDivisionError`	Raised when division or modulo by zero takes place.
`OverflowError`	Raised when a calculation exceeds the limits for a numeric type.
`FloatingPointError`	Raised when a floating-point calculation fails.
`AssertionError`	Raised when an assert statement fails.
`AttributeError`	Raised when an invalid attribute reference occurs.
`EOFError`	Raised when `input()` hits an end-of-file condition.
`ImportError`	Raised when an import fails.
`ModuleNotFoundError`	Raised when a module cannot be found.
`IndexError`	Raised when a sequence subscript is out of range.
`KeyError`	Raised when a dictionary key isn't found.
`NameError`	Raised when a variable is not found.
`UnboundLocalError`	Raised when a local variable is referenced before assignment.
`MemoryError`	Raised when an operation runs out of memory.
`OSError`	Raised for system-related errors.
`FileNotFoundError`	Raised when a file or directory is requested but doesn't exist.
`PermissionError`	Raised when trying to perform an operation without proper access rights.
`TimeoutError`	Raised when a system function times out.
`TypeError`	Raised when an operation or function is applied to the wrong type.
`ValueError`	Raised when an operation receives an argument of correct type but improper value.
`UnicodeError`	Raised when encoding or decoding Unicode fails.
`IndentationError`	Raised when there is incorrect indentation.
`SyntaxError`	Raised when Python code is syntactically incorrect.
`RuntimeError`	Raised when an error is detected that doesn't fall in other categories.