

TOPIC 1

What is Problem?

- A Problem is a state of difficulty that need to be resolved

What is Algorithm?

- A set of precise steps that describe exactly the tasks to be performed and the order in which they are to be carried out.
- Pseudocode , Flowcharts

Steps to developing a Program

- Define
 - Define the problem.
- Outline
 - Outline the solution.
- Develop
 - Develop the outline into an algorithm.
- Test
 - Test the algorithm for correctness.
- Code
 - Code the algorithm into a specific programming language.
- Run
 - Run the program on the computer.
- Document and maintain
 - Document and maintain the program.

What is Pseudo Code?

- It is English that has been formalized and abbreviated to look like the high-level computer languages.
- START, END,IF,ELSEIF , PRINT , DISPLAY , SHOW , OUTPUT, PUT
- Sample keyword for repetition:
- **FOR**
- **WHILE / ENDWHILE**
- **REPEAT / UNTIL**
- **DO/WHILE/ENDWHILE**

Example:

“IF student_attendance_status is part_time THEN

add 1 to part_time_count

ELSE

add 1 to full_time_count

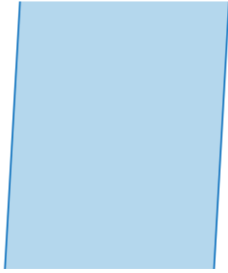
ENDIF “”

FLOWCHART

- **Flowchart is generally drawn from top to bottom**
- **All boxes of flowchart must be connected with arrow.**
- **All boxes of flowchart must be connected with arrow.**
- **All boxes of flowchart must be connected with arrow.**



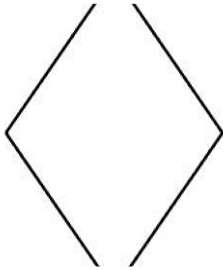
- **TERMINAL (START OR END)**



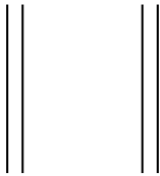
- INPUT/OUTPUT



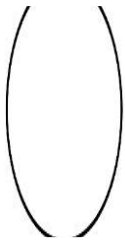
-Process (calculation)



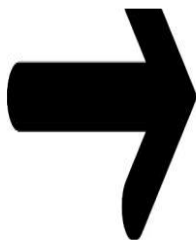
-Decision (IF Statements)



-Predefined process



-On-page Connector



-FLOW (DIRECTION)

TOPIC – 2

Types of Operators

Operators	Type
+, -, /, *, %	Arithmetic Operators
==, !=, >, <, <=, >=	Comparison Operators
&&, (), !	Logical Operators
=, +=, -=, *=, /=, %=	Assignment Operators
++, --	Increment and Decrement Operators
&, , ^, ~, >>, <<	Bitwise Operators

Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Logical Operators

Operator	Action	Definition
&&	AND	New relational expression is true if both expressions are true
	OR	New relational expression is true if either expression is true
!	NOT	Reverses the value of an expression – true expression becomes false, and false becomes true

Types of Selection

1. Simple Selection (simple IF)
2. Simple Selection with NULL FALSE
3. Combine Selection (combined IF)
4. Nested Selection (Linear and Non-Linear)

Simple Selection

- Simple selection uses a straightforward `IF` statement to make a decision based on a condition.

```
IF temperature > 30 THEN
    PRINT "It's hot outside."
END IF
```

Simple Selection with NULL FALSE

- In some programming languages or contexts, you might check for null values explicitly to handle cases where data might be missing or undefined.

```
IF userName IS NOT NULL AND userName = "John" THEN
    PRINT "Hello, John!"
END IF
```

Combine Selection (combined IF)

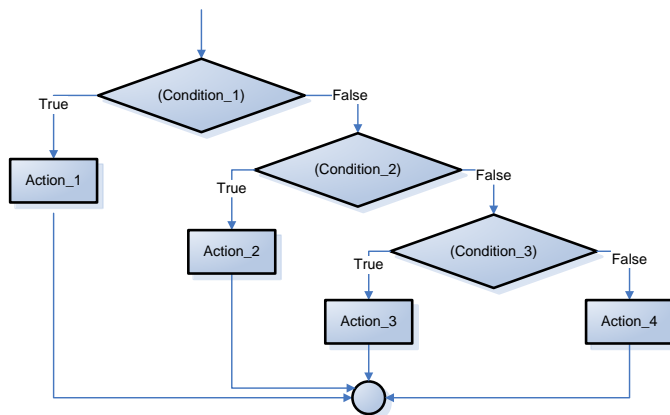
- Combine selection uses logical operators to check multiple conditions in a single `IF` statement.

```
IF age >= 18 AND hasID = TRUE THEN
    PRINT "You are eligible to vote."
END IF
```

Nested Selection (Linear and Non-Linear)

Nested selection involves placing one IF statement inside another. This can be linear (one inside another) or non-linear (multiple branches).

Linear Nested

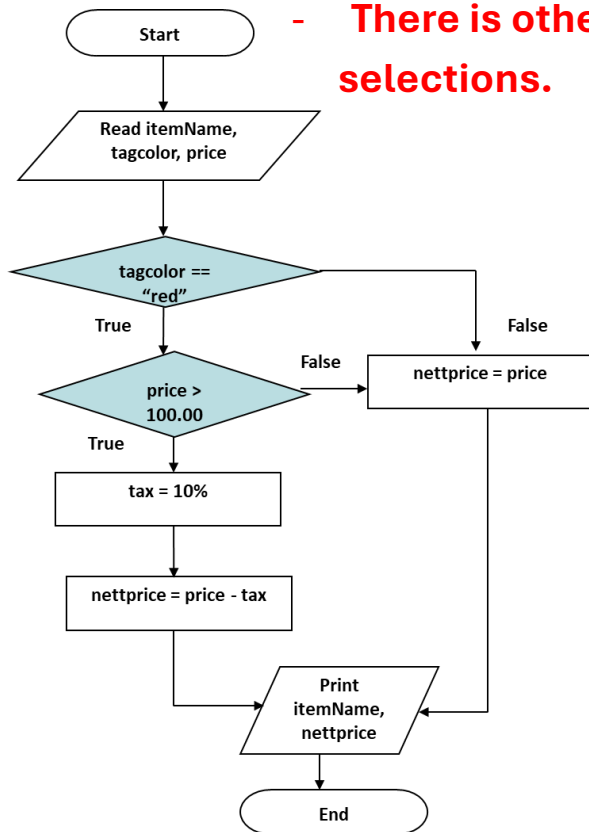


```
IF weather = "rainy" THEN
  IF temperature < 15 THEN
    PRINT "It's a cold rainy day."
  END IF
END IF
```

- **If one condition is not correct , will directly go to another condition.**

Non-linear Nested

- There is other processes between the conditions selections.



```
IF score >= 90 THEN
    PRINT "Grade: A"
    IF extraCredit > 0 THEN
        PRINT "With extra credit."
    ELSE
        PRINT "Without extra credit."
    END IF
ELSE
    PRINT "Grade: B"
END IF
```

Repetition

- **For Loop**
- **While Loop**
- **Do While Loop (Known as Post test loop)**

2 Types of LOOP

- **Pre-test lopp**
- **Post test loop (Use in C, C#,Java)**

Repetition Structure

- **1. Can be used to control execution of the loop (loop control variable)**
- **2.It will increment or decrement each time a loop repeats**
- **3.Must be initialized before entering loop**

For Loop

- A for loop in Python is typically used when the number of iterations is known beforehand.
- The for loop in Python is often used to iterate over a sequence (like a **list, tuple, string, or range**).
- The loop control variable is automatically managed by the loop structure.

```
for i in range(5): # Pre-defined range 0, 1, 2, 3, 4
    print("Iteration number:", i)
```

While Loop

- A while loop is used when the number of iterations is not known beforehand and depends on a condition.
- **The while loop in Python continues to execute as long as the condition is True.**
- The loop control variable must be explicitly managed (**initialized, condition-checked, and updated**) by the programmer.

```
user_input = ""
while user_input != "exit":
    user_input = input("Enter something (type 'exit' to quit): ")
    print("You entered:", user_input)
```

TOPIC – 3

What is Programming Language?

- Is a set of rules that provides a way of telling a computer what operations to perform.
 - It provides a linguistic framework for describing computations
 - Is a notational system for describing computation in a machine-readable and human-readable form.
 - Is a tool for developing executable models for a class of problem domains.
-
- **Source Code:** The entire block of code from variable initialization to the `if-else` statement and the `print` functions.
 - **Syntax:** The `if-else` statement syntax is correct because it includes a colon after the condition and proper indentation.
 - **Output:** Depending on the values of `a` and `b`, the output will be either "The result is: 30" or "a is not less than b".
 - **Console:** The text box (within the IDE or terminal) where the output "The result is: 30" will be printed.

Python is

- easy to learn,
- relatively fast,
- object-oriented,
- strongly typed,
- widely used, and
- portable.

Compared to:

- C is much faster but much harder to use.
- Java is about as fast and slightly harder to use.
- Perl is slower, is as easy to use, but is not strongly typed.

- Free
- Portable
- Indentation
- Object-Oriented
- Powerful

- PYTHON is Interpreting Language.

Difference Between Compiling and Interpreting

Aspect	Compiling	Interpreting
Translation	Translates entire source code into machine code before execution.	Translates and executes source code line by line.
Output	Generates an executable file (e.g., .exe).	No intermediate file, directly executes source code.
Execution Speed	Generally faster, as machine code is executed directly by the processor.	Generally slower, as translation happens during execution.
Platform Dependency	Platform-specific executables; may require recompilation for different platforms.	Platform-independent; requires an interpreter on the target machine.
Development Cycle	Requires compilation step before execution; slower iteration during development.	Direct execution; faster iteration and easier testing during development.
Distribution	Distribute executable files; source code can remain hidden.	Source code is distributed and executed by the interpreter.
Error Detection	Errors are detected at compile time.	Errors are detected at runtime.
Examples	C, C++, Rust	Python, JavaScript, Ruby

Comment in Python

- Use “#”
- “”

TOPIC 4 (Implementing PYTHON)

*** REMEMBER the DATA TYPES

Type	Declaration	Example	Usage
Integer	int	x = 124	Numbers without decimal point
Float	float	x = 124.56	Numbers with decimal point
String	str	x = "Hello world"	Used for text
Boolean	bool	x = True or x = False	Used for conditional statements
NoneType	None	x = None	Whenever you want an empty variable

Variable Declaration Rules

1. Variable Names Must Start with a Letter or an Underscore

```
# Valid variable names
name = "Alice"
_age = 30
```

2. Variable Names Can Only Contain Alphanumeric Characters and Underscores

```
# Valid variable names
user_name = "Alice"
user123 = "Bob"
```

3. Variable Names Cannot Be a Reserved Keyword

```
# Invalid variable name (using a reserved keyword)
# class = "Class" # This will cause a syntax error
```

False	class	finally	is	return	None	continue	for
lambda	try	True	def	from	nonlocal	while	and
del	global	notwith	as	elif	if	or	yield
assert	else	import	pass	break	except	in	raise


4. Variable Names Should Be Descriptive

```
# Descriptive variable name
student_age = 20

# Less descriptive variable name
sa = 20
```

Order Precedence Rules

Parenthesis
Power
Multiplication / division/
Modulus
Addition
Left to Right



Boolean Value

- YES OR NO
- True or False

Decision Structures in Python with Examples

1. One Way Decision (if)

Executes a block of code if the condition is true

```
# One way decision
age = 18
if age >= 18:
    print("You are an adult.")
```

2. Two-Way Decision (if else)

Executes one block of code if the condition is true and another block if the condition is false.

```
# Two-way decision
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

3. Multiway Decision (if elif)

Executes one of several blocks of code depending on multiple conditions.

```
# Multiway decision
score = 75
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

Pass Statement

A placeholder statement that does nothing; used when a statement is required syntactically but no action is needed.

```
# Pass statement
age = 20
if age >= 18:
    pass # Later code will be added here
else:
    print("You are a minor.")
```

```
# No output because pass does nothing
```


Return Statement

Exits a function and optionally passes an expression back to the caller.

```
# Return statement in a function
def check_age(age):
    if age >= 18:
        return "You are an adult."
    else:
        return "You are a minor."

result = check_age(21)
print(result)
```

break Statement

The `break` statement is used to exit a loop prematurely. When `break` is encountered inside a loop, the loop is terminated, and control is transferred to the statement immediately following the loop.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

continue Statement

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

The `continue` statement is used to skip the current iteration of a loop and proceed to the next iteration. When `continue` is encountered, the remaining code inside the loop is skipped, and the next iteration begins.

```
# Output: 1 3 5 7 9
```

Explanation of range(start, stop, step):

start: The starting value of the sequence (inclusive).

stop: The stopping value of the sequence (exclusive).

step: The step value determines the increment (or decrement if negative).

```
print("Counting backwards:")  
for i in range(10, 0, -1):  
    print(i)
```

```
Counting backwards:  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1
```

Using else Statement with While Loop – Example

```
count = 0
```

```
while count < 5:
```

```
    print(count, " is less than 5")
```

```
    count = count + 1
```

```
else:
```

```
    print(count, " is not less than 5")
```

Chapter – 7

Functions

- A function in Python is a block of reusable code that performs a specific task.

Using “def” to create a function

```
def greet(name):  
    """This function greets the user."""  
    print(f"Hello, {name}!")  
  
# Calling the function  
greet("Alice")
```

Arguments and Parameters:

Parameter: A variable in a function definition. It is a placeholder for the value that will be passed to the function when it is called.

Argument: The actual value or expression passed to a function when calling it.

Pass by Reference and Pass by Value:

Pass by reference: When you pass a mutable object (like a list or dictionary) to a function, the function can modify the object.

```
def modify_list(lst):  
    lst.append(4)  
  
numbers = [1, 2, 3]  
modify_list(numbers)  
print(numbers) # Output: [1, 2, 3, 4]
```

Pass by value: When you pass immutable objects (like integers, strings, tuples) to a function, a copy of the object is passed. Changes inside the function do not affect the original object.

```
def increment(num):  
    num += 1  
  
x = 10  
increment(x)  
print(x) # Output: 10 (x remains unchanged)
```

Fruitful (Return Values) and Void (Non-fruitful) Functions:

Fruitful Function: A function that returns a value using the `return` statement.

```
def add_numbers(a, b):  
    return a + b  
  
result = add_numbers(3, 5)  
print(result) # Output: 8
```

Void Function (Non-fruitful): A function that performs an operation but does not return any value explicitly (implicitly returns `None`).

```
def greet(name):  
    print(f"Hello, {name}!")  
  
greet("Bob")  
# Output: Hello, Bob!
```

***Python Built-in Functions to Remember

- `print()`: Outputs messages or variables to the console.
- `len()`: Returns the length (number of items) of an object like a string, list, or tuple.
- `input()`: Reads input from the user via the console.
- `type()`: Returns the type of an object (e.g., `int`, `str`, `list`).
- `int()`: Converts a string or number to an integer.
- `float()`: Converts a string or number to a floating-point number.
- `str()`: Converts an object into a string representation.
- `list()`: Creates a list from iterable objects like tuples or converts a string to a list.
- `tuple()`: Creates a tuple from iterable objects or converts a list to a tuple.
- `dict()`: Creates a dictionary or converts a sequence of key-value pairs into a dictionary.
- `range()`: Generates a sequence of numbers.
- `sorted()`: Returns a new sorted list from the elements of any iterable.
- `sum()`: Returns the sum of all elements in an iterable.
- `max()`: Returns the maximum element from an iterable or a series of arguments.
- `min()`: Returns the minimum element from an iterable or a series of arguments.
- `abs()`: Returns the absolute value of a number.
- `all()`: Returns `True` if all elements of an iterable are true (or if the iterable is empty).
- `any()`: Returns `True` if any element of an iterable is true. If the iterable is empty, it returns `False`.
- `callable()`: Checks if the object is callable (e.g., functions, methods).

- **enumerate()**: Returns an enumerate object that yields tuples containing a count (index) and the values obtained from iterating over a sequence.
 - **filter()**: Constructs an iterator from elements of an iterable for which a function returns true.
 - **map()**: Applies a function to all items in an input iterable.
 - **zip()**: Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the input iterables.
 - **chr()**: Returns the string representing a character whose Unicode code point is the integer.
 - **ord()**: Returns the Unicode code point for a given character.
 - **round()**: Rounds a floating-point number to a specified number of decimals or to the nearest integer.
 - **dir()**: Returns a list of attributes and methods of any object (without the `__` methods).
 - **eval()**: Evaluates a string containing a Python expression.
 - **globals()**: Returns the dictionary representing the current global symbol table.
 - **locals()**: Returns the dictionary representing the current local symbol table.
- append()**: Adds an element to the end of a list.
- pop()**: Removes and returns the last element from a list, or removes and returns the element at a specified index.
- insert()**: Inserts an element at a specified position in a list.
- index()**: Returns the index of the first occurrence of a value in a list.

1	capitalize()	Capitalizes first letter of string
2	center(width, fillchar)	Returns a space-padded string with the original string centered to a total of width columns
3	count(str, beg= 0,end=len(string))	Counts how many times str occurs in string, or in a substring of string if starting index beg and ending index end are given
3	decode(encoding='UTF-8',errors='strict')	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
4	encode(encoding='UTF-8',errors='strict')	Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
5	endswith(suffix, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; Returns true if so, and false otherwise
6	expandtabs(tabsize=8)	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if <u>tabsize</u> not provided

7	find(str, beg=0 end=len(string))	Determine if str occurs in string, or in a substring of string if starting index beg and ending index end are given; returns index if found and -1 otherwise
8	index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found
9	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise
10	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic false otherwise
11	isdigit()	Returns true if string contains only digits and false otherwise
12	islower()	Returns true if string has at least 1 cased character and all cased characters are lowercase and false otherwise
13	isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise
14	isspace()	Returns true if string contains only whitespace characters and false otherwise

15	istitle()	Returns true if string is properly "titlecased" and false otherwise
16	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise
17	join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string
18	len(string)	Returns the length of the string
19	ljust(width[, fillchar])	Returns a space-padded string with the original string left-justified to a total of width columns
20	lower()	Converts all uppercase letters in string to lowercase
21	lstrip()	Removes all leading whitespace in string
22	maketrans()	Returns a translation table to be used in translate function.
23	max(str)	Returns the max alphabetical character from the string str

32	startswith(str, beg=0, end=len(string))
	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; Returns true if so, and false otherwise
33	strip([chars])
	Performs both lstrip() and rstrip() on string
34	swapcase()
	Inverts case for all letters in string
35	title()
	Returns "titlecased" version of string, that is, all words begin with uppercase, and the rest are lowercase
36	translate(table, deletechars="")
	Translates string according to translation table str(256 chars), removing those in the del string
37	upper()
	Converts lowercase letters in string to uppercase
38	zfill (width)
	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero)
39	isdecimal()
	Returns true if a unicode string contains only decimal characters and false otherwise

24	min(str)	Returns the min alphabetical character from the string str
25	replace(old, new [, max])	Replaces all occurrences of old in string with new, or at most max occurrences if given
26	rfind(str, beg=0, end=len(string))	Same as find(), but search backwards in string
27	rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string
28	rjust(width,[, fillchar])	Returns a space-padded string with the original string right-justified to a total of width columns.
29	rstrip()	Removes all trailing whitespace of string
30	split(str="", num=string.count(str))	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given
31	splitlines(num=string.count('\n'))	Splits string at all (or num) NEWLINES and returns a list of each line with NEWL removed

Chapter (Stings Slicing)

Indexing

Strings in Python are indexed, meaning each character in the string has a position, starting from 0 for the first character. Negative indices count from the end of the string.





```
my_string = "Hello"
print(my_string[0]) # Output: 'H'
print(my_string[-1]) # Output: 'o'
```

Slicing

Slicing allows you to extract a substring (a portion of the string) from a string by specifying a range of indices. The syntax for slicing is `string[start:end:step]`.

```
my_string = "Hello, World!"
print(my_string[7:]) # Output: 'World!'
print(my_string[:5]) # Output: 'Hello'
print(my_string[7:12]) # Output: 'World'
```

0	1	2	3	4	5	6	7	8
M	i	n	e	c	r	a	f	t
-9	-8	-7	-6	-5	-4	-3	-2	-1

- **First Character**
`firstChar = myString[0]` . . . 
- **Last Character**
`lastChar = myString[-1]` . . . 
- **All characters but the first**
`slice = myString[1:]` . . . 
- **All characters but the last**
`slice = myString[:-1]` . . . 

The `in` Operator:

The `in` operator checks for membership, whether a value exists within an iterable (like strings, lists, tuples, etc.).

```
my_string = "Hello, World!"
if 'Hello' in my_string:
    print("Found 'Hello' in the string")
else:
    print("Not found")
```

*** Remember this functions()

`"AbC aBc".lower()` → `abc abc`

`"abc abc".replace("c ", "xx")` → `abxxabc`

`"abc abc".startswith("ab")` → `True`

`"AbC aBc".swapcase()` → `aBc AbC`

`"Abc abc".upper()` → `ABC ABC`

`"abc abc".capitalize()` → `Abc abc`

`"abc abc".count("b")` → `2`

`"abc abc".islower()` → `True`

```
>>> greet = ' Hello Bob '
>>> greet.lstrip()
'Hello Bob '
>>> greet.rstrip()
' Hello Bob'
>>> greet.strip()
'Hello Bob'
>>>
```

Strings cannot be modified; instead, create a new one.

```
>>> s = "GATTACA"
>>> s[3] = "C"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> s = s[:3] + "C" + s[4:]
>>> s
'GATCACA'
>>> s = s.replace("G","U")
>>> s
'UATCACA'
```

String Formatting Methods:

String formatting in Python allows you to insert values into strings in a controlled manner. There are several ways to format strings, but here we focus on the `format()` method and f-strings (formatted string literals).

```
name = "Alice"
age = 30
message = "My name is {} and I am {} years old.".format(name, age)
print(message)
# Output: My name is Alice and I am 30 years old.
```

Positional Arguments:

```
print("Hello, {0}. Your balance is {1}".format("Alice", 230.2346))
# Output: Hello, Alice. Your balance is 230.2346
```

Keyword Arguments

```
print("Hello, {name}. Your balance is {balance}".format(name="Alice", balance=230.2346))
# Output: Hello, Alice. Your balance is 230.2346
```

F-strings Format

```
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."
print(message)
# Output: My name is Alice and I am 30 years old.
```


Chapter – 9 (lists and tuples)

LIST	TUPLE
Syntax for list is slightly different comparing with tuple	Syntax for tuple is slightly different comparing with lists
<code>weekdays=['Sun','Mon', 'wed',46,67]</code> <code>type(Weekdays)</code> <code>class<'lists'></code>	<code>twdays = ('Sun', 'mon', 'tue', 634)</code> <code>type(twdays)</code> <code>class<'tuple'></code>
List uses [and] (square brackets) to bind the elements.	Tuple uses rounded brackets(and) to bind the elements.
List can be edited once it is created in python. Lists are mutable data structure.	A tuple is a list which one cannot edit once it is created in Python code. The tuple is an immutable data structure
More methods or functions are associated with lists.	Compare to lists tuples have Less methods or functions.

TUPLE	DICTIONARY
Order is maintained.	Ordering is not guaranteed.
They are immutable	Values in a dictionary can be changed.
They can hold any type, and types can be mixed.	Every entry has a key and a value
Elements are accessed via numeric (zero based) indices	Elements are accessed using key's values
There is a difference in syntax and looks easy to define tuple	Differ in syntax, looks bit complicated when compare with Tuple or lists

Lists:

Lists are mutable sequences, typically used to store collections of homogeneous items. In Python, lists are defined by enclosing comma-separated values within square brackets [].

```
# Creating a list
my_list = [1, 2, 3, 4, 5]
print(my_list) # Output: [1, 2, 3, 4, 5]

# Lists can contain different data types
mixed_list = [1, "Hello", 3.5, True]
print(mixed_list) # Output: [1, 'Hello', 3.5, True]
```

Append vs. Concatenate

- The concatenate operator + uses two lists and creates a bigger one
- Append is a method which adds an element to the right end of a list – any type of data

List Built-in Functions

append(): Adds an element to the end of the list.

```
my_list.append(6)
print(my_list) # Output: [1, 2, 3, 4, 5, 6]
```

`extend()`: Extends a list by appending elements from an iterable.

```
another_list = [7, 8, 9]
my_list.extend(another_list)
print(my_list) # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`insert()`: Inserts an element at a specified position.

```
my_list.insert(0, 0)
print(my_list) # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`remove()`: Removes the first occurrence of a value from the list.

```
my_list.remove(3)
print(my_list) # Output: [0, 1, 2, 4, 5, 6, 7, 8, 9]
```

pop(): Removes and returns the last element from the list, or removes an element at a specified index.

```
popped_element = my_list.pop()
print(popped_element) # Output: 9
print(my_list) # Output: [0, 1, 2, 4, 5, 6, 7, 8]
```

index(): Returns the index of the first occurrence of a value in the list

```
index = my_list.index(5)
print(index) # Output: 4
```

Split – returns a list

“lets try some splitting here”.split(“ “) => ['lets', 'try', 'some', 'splitting', 'here']

Tuples

1. Cannot be changed
2. Immutable
3. Ordered
4. Have different data types
5. Hashable
6. Comparable
7. Count() and index() works
8. Sort(), reverse(), append() don't work as it cannot change.

```
my_tuple = (1, 2, 'Hello', 3.5)
print(my_tuple)      # Output: (1, 2, 'Hello', 3.5)
print(my_tuple[2])   # Output: Hello
```

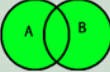



Sets

1. Unordered
2. Mutable
3. Unique elements
4. Cannot contain “lists” in set
5. Using “set()” can convert list to set

```
my_set = {1, 2, 3, 'Hello'}
print(my_set)      # Output: {1, 2, 3, 'Hello'}

# Adding elements to a set
my_set.add(4)
print(my_set)      # Output: {1, 2, 3, 4, 'Hello'}

# Removing elements from a set
my_set.remove(3)
print(my_set)      # Output: {1, 2, 4, 'Hello'}
```

Method	Operator		Description
union			Contains all elements that are in set A or in set B
intersection	&		Contains all elements that are in both sets A and B
difference	-		Contains all elements that are in A but not in B
<u>symmetric_difference</u>	^		Contains all elements that are either <ul style="list-style-type: none"> in set A but not in set B or in set B but not in set A

```

setA = {1, 2, 3, 4, 5}
setB = {3, 4, 5, 6, 7}

print("Set A:", setA)
print("Set B:", setB)

print("Union:", setA | setB)           # Un
print("Intersection:", setA & setB)   # In
print("Difference:", setA - setB)     # Di
print("Symmetric Difference:", setA ^ setB)

```

```

Set A: {1, 2, 3, 4, 5}
Set B: {3, 4, 5, 6, 7}
Union: {1, 2, 3, 4, 5, 6, 7}
Intersection: {3, 4, 5}
Difference: {1, 2}
Symmetric Difference: {1, 2, 6, 7}

```

Dictionary

1. Key-Value Pairs
2. Mutable
3. Unordered
4. Keys are immutable

1. clear()

The clear() method removes all items from the dictionary.

```
# Creating a dictionary
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
print("Original Dictionary:", my_dict)

# Clearing the dictionary
my_dict.clear()
print("Dictionary after clear():", my_dict) # Output: {}
```


`copy()`

The `copy()` method returns a shallow copy of the dictionary.

```
# Making a copy of the dictionary
copy_dict = my_dict.copy()
print("Copy of dictionary:", copy_dict) # Output: {'name': 'Alice', 'age': 30, 'city': 'N
```

`get(key[, d])`

The `get()` method returns the value for `key`. If `key` does not exist, it returns `d` (defaulting to `None` if not provided).

```
# Accessing values using get()
age = my_dict.get('age')
print("Age:", age) # Output: 30

# Specifying a default value
address = my_dict.get('address', 'Unknown')
print("Address:", address) # Output: Unknown
```

`items()`

The `items()` method returns a new view of the dictionary's items as (key, value) pairs.

```
# Getting items as key-value pairs
items = my_dict.items()
print("Items:", items) # Output: dict_items([('name', 'Alice'), ('age', 30), ('city', 'N
```

keys ()

The `keys ()` method returns a new view of the dictionary's keys.

```
# Getting keys
keys = my_dict.keys()
print("Keys:", keys) # Output: dict_keys(['name', 'age', 'city'])
```

update ()

The `update ()` method updates the dictionary with the key/value pairs from another dictionary, overwriting existing keys.

```
# Updating the dictionary
my_dict.update({'age': 31, 'address': '123 Main St'})
print("Updated dictionary:", my_dict) # Output: {'name': 'Alice', 'age': 31, 'city': 'New York', 'address': '123 Main St'}
```

values ()

The `values ()` method returns a new view of the dictionary's values.

```
# Getting values
values = my_dict.values()
print("Values:", values) # Output: dict_values(['Alice', 31, 'New York', '123 Main St'])
```

`fromkeys(seq[, v])`

The `fromkeys()` method creates a new dictionary with keys from `seq` and values set to `v` (defaulting to `None` if not provided).

```
# Creating a dictionary from keys
keys = ['name', 'age', 'city']
default_value = 'Unknown'
new_dict = dict.fromkeys(keys, default_value)
print("New Dictionary from keys:", new_dict) # Output: {'name': 'Unknown', 'age': 'Un
```

`pop(key[, d])`

The `pop()` method removes and returns the value associated with `key`. If `key` is not found and `d` is provided, it returns `d`. If `d` is not provided and `key` is not found, it raises `KeyError`.

```
# Removing and returning a value using pop()
age = my_dict.pop('age')
print("Popped age:", age) # Output: 31
print("Dictionary after pop:", my_dict) # Output: {'name': 'Alice', 'city': 'New York'}

# Using pop() with default value
address = my_dict.pop('address', 'No address')
print("Popped address:", address) # Output: 123 Main St (previously updated)
```

`popitem()`

The `popitem()` method removes and returns an arbitrary (key, value) pair from the dictionary. It raises `KeyError` if the dictionary is empty.

```
# Removing and returning an arbitrary item using popitem()
item = my_dict.popitem()
print("Popped item:", item) # Output: ('city', 'New York')
print("Dictionary after popitem:", my_dict) # Output: {'name': 'Alice'}
```

Chapter 11 (file Handling)

- OPENING FILE
 - Associate an external file with a program object
- READING/WRITE FILE
 - Manipulate the file object
 - Reading from or writing to the file object
- CLOSING FILE
 - Once done, close the file.

File Access Mode

Text File Mode	Binary File Mode	Description	Notes
w	<u>wb</u>	Write only	<ul style="list-style-type: none">If file not exist, file is created.If file exists, python will delete existing data and overwrite the file.
r	<u>rb</u>	Read only	<ul style="list-style-type: none">File must exist, otherwise getting I/O error
a	ab	Append	<ul style="list-style-type: none">File in write mode only, new data will be added to the end of existing data.If file not exists, file is created
w+	<u>w+b</u> or <u>wb+</u>	Write and read	<ul style="list-style-type: none">Opens the file for both reading and writing.The text is overwritten and deleted from an existing file.
r+	<u>r+b</u> or <u>rb+</u>	Read and write	<ul style="list-style-type: none">Opens the file for both reading and writing.If the file does not exist, an I/O error gets raised.
a+	<u>a+b</u> or <u>ab+</u>	Append and read	<ul style="list-style-type: none">Can read and write in the file.If the file doesn't already exist, file is createdNew written text will be added at the end, following the previously written data.
x		Exclusive creation mode	<ul style="list-style-type: none">Open the file for writing, but only if the file does not already exist.If the file exists, an error is raised.

```
# Reading from a file
with open('example.txt', 'r') as f:
    content = f.read() # Read entire file content into a string
    print(content)
```

`name = open("filename")`

opens the given file for reading, and returns a file object

`name.read()` - file's entire contents as a string

`name.readline()` - next line from file as a string

`name.readlines()` - file's contents as a list of lines

the lines from a file object can also be read using a for loop

Error Handling

1. Syntax Errors (Compile-time Errors):

Syntax errors occur when the syntax (grammar) of the code is incorrect.

```
# Missing colon after if statement
if True
    print("Hello, World!")
```

2. Runtime Errors (Exceptions):

Runtime errors, also known as exceptions, occur while a program is running if something unexpected happens.

```
# Division by zero error
numerator = 10
denominator = 0
result = numerator / denominator
print("Result:", result)
```

```
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    result = numerator / denominator
ZeroDivisionError: division by zero
```

3. Logical Errors:

Logical errors occur when the code executes without throwing any syntax or runtime errors, but produces incorrect results due to a mistake in the algorithm or logic of the program. These errors are often the most difficult to debug because they do not cause Python to report an error.

```
# Incorrect calculation of average
numbers = [10, 20, 30, 40, 50]
total = sum(numbers)
average = total / len(numbers) # Incorrect calculation
print("Average:", average)
```

```
maketfile
```

```
Average: 30
```

Incorrect Result

try, except Blocks:

```
# Example of try-except block
try:
    num1 = 10
    num2 = 0
    result = num1 / num2 # Division by zero will raise ZeroDivisionError
    print("Result:", result)
except ZeroDivisionError as e:
    print("Error:", e) # Output: Error: division by zero
```

Try-finally Example

```
# Example of try-except-finally block
try:
    f = open('example.txt', 'r')
    content = f.read()
    print(content)
except FileNotFoundError as e:
    print("Error:", e) # Handle file not found error
finally:
    if 'f' in locals():
        f.close() # Always close the file, whether an exception occurred or not
```

ZeroDivisionError

NameError

IndentationError

IOError

EOFError

- **Logging is a means of tracking events that happen when some software runs.**
- **Logging module provides a set of functions for simple logging and for following purposes**