Python notes

Toppic : Introduction to python data type and operators

- Python is a **high-level, interpreted programming language** renowned for its clear syntax and readability, which makes it easy to learn and use.
- **Easy & Readable** → Python uses simple, human-like syntax (e.g., print("Hello")), making it beginner-friendly.

- **Versatile** → Widely used for data science, web development, AI, automation, and scripting.

1,Data types in python

- There are mainly 7 data types in python They are:-
    1. Numeric Types
    2. Text Type
    3. Boolean Type
    4. Mapping type
    5. Set Type
    6. Sequence Type
    7. None Type

1.Numeric Data type

- **Integers** are whole numbers, positive or negative, without a decimal point represent by “int” (e.g., 5, -100).
- **Floating-point numbers** are numbers with a decimal point it is represented by” float” (e.g., 3.14, -0.5).
- **Complex numbers** are numbers with a real and imaginary part it is represented by “complex” (e.g., 3 + 5j).

2.Text Type

- **Strings** are sequences of characters enclosed in single, double, or triple quotes it is represented “ str” (e.g., 'hello', "world", """multi-line string""").

3.Boolean Type

- **Booleans** represent one of two values: True or False. They are primarily used in conditional statements and logical operations.

4. Sequence Types

- **Lists** are ordered, mutable(changable) collections of items. They are defined with square brackets [] and can contain different data types (e.g., [1, 'apple', 3.14]).
- **Tuples** are ordered, immutable (Unchangable)collections of items. They are defined with parentheses () (e.g., (1, 'apple', 3.14)).
- **Ranges** are sequences of numbers, often used in for loops. And it include(start , stop, step).

5. Mapping Type

- **Dictionaries** are unordered, mutable collections of **key-value pairs**. They are defined with curly braces {} .it is represented by dict (e.g., {'name': 'Alice', 'age': 30}).

6.Set Types

- set: **Sets** are unordered collections of unique items. They are defined with curly braces {} or the set() constructor (e.g., {1, 2, 3}).

7.None type

- Absence of value or null values

2. **Python Operators**

Operators are special symbols that perform operations on variables and values. The values the operator works on are called **operands**.

Types of Operators

- **Arithmetic Operators**
  - + (Addition): Adds two operands.
  - - (Subtraction): Subtracts the right operand from the left.
  - * (Multiplication): Multiplies two operands.
  - / (Division): Divides the left operand by the right, always returning a float.
  - // (Floor Division): Divides and returns the integer part of the quotient.
  - % (Modulus): Returns the remainder of the division.
  - ** (Exponentiation): Raises the left operand to the power of the right.
- **Assignment Operators**

    - =: Assigns the value on the right to the variable on the left.
    - +=, -=, *=, /=, etc.: Shorthand for operations followed by assignment (e.g., x += 5 is the same as x = x + 5).
- **Comparison Operators**
    - == (Equal to): Returns True if both operands are equal.
    - != (Not equal to): Returns True if operands are not equal.
    - > (Greater than), < (Less than)
    - >= (Greater than or equal to), <= (Less than or equal to)
- **Logical Operators**
    - and: Returns True if both statements are True.
    - or: Returns True if at least one of the statements is True.
    - not: Reverses the result; returns False if the statement is True.
- **Identity Operators**
    - is: Returns True if both variables point to the **same object in memory**.
    - is not: Returns True if both variables do **not** point to the same object.
- **Membership Operators**
    - in: Returns True if a sequence contains the specified value.
    - not in: Returns True if a sequence does not contain the specified value.

Toppic : conditional statement

Conditional statements allow you to execute different blocks of code based on whether a condition is True and False . They are a fundamental part of programming logic, enabling your programs to make decisions. The primary conditional statements in Python are if , elif, and else.

1.The IF Statement

The IF statement is the simplest form of a conditional. The code block indented under the IF statement will only run if the specified condition is” True”

Eg:

age = 18

if age >= 18:

print("You are eligible to vote.")

in this example, since the condition age >= 18 is True, the message "You are eligible to vote." is printed.

2.The else Statement

The else statement is used to provide an alternative block of code to run if the if condition is False.

Eg:

```
age = 16

if age >= 18:

    print("You are eligible to vote.")

else:

    print("You are not old enough to vote.")
```

Here, the if condition is False, so the code under the else block is executed.

3.The elif (Else-if) Statement

The elif statement is used to check for multiple conditions sequentially. It stands for "else if" and is checked only if the preceding if or elif conditions were False. You can have any number of elif statements.

Eg:

```
score = 85

if score >= 90:

    print("Grade: A")

elif score >= 80:

    print("Grade: B")

elif score >= 70:

    print("Grade: C")

else:

    print("Grade: D or F")
```

in this case, the first condition (score >= 90) is False, so Python moves to the elif statement. The condition score >= 80 is True, so the code print("Grade: B") is executed, and the program exits the conditional block.

Toppic : Iterative statement and functions

- Iterative statements are also commonly known as **loops.**
- There are mainly two types of iterative statements
    1. For loop
    2. While loop

1, For loop

- This loop is ideal for when you know the number of iterations in advance.
- Mainly uses in list of data such as ,tupple, range, strings, list etc….
- The loop continues until all elements in the sequence have been processed.

Eg: for char in "Hello":

```
    print(char)
```

```
for i in range(10):
```

```
    print(i)
```

2, While loop

- This loop is used when the number of iterations is not known beforehand.
- It continues to execute as long as a specified condition is true.
- The condition is checked at the beginning of each iteration.
- If the condition becomes false, the loop terminates.

Eg:

```
i = 1

while i <= 10:

    print(i)

    i = i + 1
```

3, Break

- If you want to out of the loop then we use the condition break.

Eg:

```
For i in range(1,6):
        If i ==4:
        Break
        print(i)
print(“outsideloop”)
```

4, continue

- After continue then skip statement after that

Eg:

```
For i in range(1,6):
        If i ==4:
        Continue
print(i)
print(“outside loop”)
```

Functions

- In python functions are usable blocks of code that perform a specific task.
- It help program modular , readable and easier to maintain.
- Built-in or user-defined blocks
- Built-in functions (provided by python)

  Eg: print(), len(),type(), sum(),max(), min() etc…

- User-defined functions(created by programmers).
- Using ‘def’ keyword, followed by function name.

Eg def function name(parameter):

#statement.

Return expression

1, simple function

Eg : defgreat():

print(“hello, welcome to kerala”)

great()

2, function with parameter

Eg : def odd-num(a,b):

Return a+b

Result =add-num(4,8)

print(“sum” , Result)

3, function with default parameter

Eg : def gret_user(name=”guest”):

print(f”hello,{guest}”)

greet_user(“Ammu”)

greet_user()

4, function with Variable Argument(*args and**keywords)*

- **args - Allows passing a variable number of positional arguments.
- **keyword - Allows passing a variable number of keyword arguments.

Eg: def display_info(*args,**keyword)

print(“positionalArgument:,” orgs)

print(“Keyword argument:,” keyword)

display_info(1,2,3,name=”Ancy” age=20)

5, Returning multiple values

Eg def calculate(a,b):

Return a+b,a-b,a*b,a/b

Add,sub, mult, div = calculate(6,8)

print(f"add:{add},sub:{sub}, mult :{mult},div{div}")

Key points

1. **Return** = return a value form of a function.
2. **Stope** =variable defined inside a function are lead to that function.

   Eg: "global knowledge" keyword used to modify global variable inside a function.

3. **Docstring =** documentation string to describe the purpose .

   eg: def greet():

   "This function prints a greeting msg"'

   print("hello")

4. **Lambda function =** Anonymous , single expresion function.

   Eg : square = lambdax:X**2

   print(square(4))

Toppic : Object oriented programming (OOPs)

- **OOPS(object oreinted programming)
- use for easy acess and creating large code use oops
- basic terms

1,class : Blue print of object or the structure that created

a, atributes: that is the features in the class

b,methods :in class is also known as functionscode security is powerfull in class

2, Object(instants) : used to acess the class , we can create n number of class in object

3, Inhertance

4, polymorphism

5, Encapulation

6, Abstraction

1. class

a, Atributes

b, Methods

here we are using the keyword such as obect to ru the class without the error

"self" is the keyword wich is using here to conflict the current object which acessing one or more object.

- **Encapsulation** allows you to bundle data (attributes) and behaviors (methods) within a class to create a cohesive unit. By defining methods to control access to attributes and its modification, encapsulation helps maintain data integrity and promotes modular, secure code.
- **Inheritance** enables the creation of hierarchical relationships between classes, allowing a subclass to inherit attributes and methods from a parent class. This promotes code reuse and reduces duplication.
- **Abstraction f**ocuses on hiding implementation details and exposing only the essential functionality of an object. By enforcing a consistent interface, abstraction simplifies interactions with objects, allowing developers to focus on what an object does rather than how it achieves its functionality.
- **Polymorphism** allows you to treat objects of different types as instances of the same base type, as long as they implement a common interface or behavior. Python's duck typing make it especially suited for polymorphism, as it allows you to access attributes and methods on objects without needing to worry about their actual class.

Toppic: Coding best pratices

General Coding Best Practices

1. Write Readable Code

- Use meaningful variable and function names (`calculate_total()` is better than `ct()`).
- Follow consistent indentation and spacing.
- Break long lines and avoid deeply nested logic.

2. Keep It DRY (Don't Repeat Yourself)

- Reuse code with functions, classes, or modules.
- Avoid copy-pasting logic—abstract it instead.

3. Comment Wisely

- Use comments to explain *why*, not *what* (the code should be self-explanatory).
- Avoid cluttering with obvious comments.

4. Use Version Control

- Git is your best friend. Commit often with clear messages.
- Branch for features, fixes, and experiments.

5. Handle Errors Gracefully

- Use try-except blocks where needed.
- Don't just silence errors—log them or raise meaningful exceptions.

6. Write Tests

- Unit tests help catch bugs early.
- Use frameworks like `unittest`, `pytest`, or `Jest` depending on your language.

7. Follow Language-Specific Conventions

- For Python: follow PEP 8 style guide.
- For JavaScript: use linters like ESLint to enforce style.

8. Optimize for Performance

- Avoid unnecessary computations or memory usage.
- Profile your code if performance matters.

9. Document Your Code

- Use docstrings or markdown files to explain how your code works.
- Make onboarding easier for others (or future you).

10. Keep Learning

- Stay updated with new tools, libraries, and patterns.
- Read others' code, contribute to open source, and ask for feedback

Toppic : control git and github

Version control is like a time machine for your code—and Git with GitHub is the dynamic duo that makes it all happen. Let's break it down so it's crystal clear:

What Is Version Control?

Version control lets you:

- Track changes to your code over time
- Collaborate with others without overwriting each other's work
- Revert to earlier versions if something breaks
- Experiment safely with new features

Git: The Engine Behind Version Control

Git is a **distributed version control system**.

Common Git Commands

Command	Purpose
`git init`	Start a new Git repository

`git clone <url>`	Copy a remote repo to your machine
`git status`	Check current changes
`git add <file>`	Stage changes for commit
`git commit -m "message"`	Save changes with a message
`git push`	Upload changes to GitHub
`git pull`	Download changes from GitHub
`git branch`	Manage branches for features or fixes
`git merge`	Combine changes from different branches

GitHub: The Cloud-Based Collaboration Hub

GitHub is a platform that hosts your Git repositories online. It adds features like:

- **Remote backups** of your code
- **Collaboration tools** (issues, pull requests, reviews)
- **Project management** (boards, milestones)
- **CI/CD integration** (automated testing and deployment)

Git + GitHub Workflow

1. Create a repositary on GitHub

2. Clone it to your local machine

3. Make changes and commit them

4. Push changes back to GitHub

5. Open a pull request if collaborating

6. Merge changes into main branch.

- Use **meaningful commit messages**: `git commit -m "Fix login bug"`
- Create **feature branches**: `git checkout -b feature/login-form`
- Pull often to avoid conflicts: `git pull origin main`
- Review code before merging via pull request.

Toppic : Uv Decomposition of vectors

Singular Value Decomposition (SVD)

For any real matrix AAA (size m×nm \times nm×n):

A =UΣVTA

= U \Sigma V^TA

=UΣVT

- UUU → Orthogonal matrix of size m×mm \times mm×m
  (columns are called **left singular vectors**)

- Σ\SigmaΣ → Diagonal matrix (size m×nm \times nm×n) with **singular values** (non-negative real numbers).

- VVV → Orthogonal matrix of size n×nn \times nn×n
  (columns are called **right singular vectors**).

🔹 Geometric Interpretation

- UUU = directions in **output space**.
- VVV = directions in **input space**.
- Σ\SigmaΣ = scales how much the input vectors (from VVV) are stretched before mapping into UUU.

So any linear transformation (matrix) is:
Rotate → Stretch → Rotate again.

Toppic: introduction,Conditional probability and Bayers Theorem

1. Introduction to Probability

- **Probability** measures the likelihood of an event occurring.
- Values range between **0 and 1**:
  - P(E)=0P(E) = 0 → impossible event
  - P(E)=1P(E) = 1 → certain event

- Formula:

$$P(E) = \frac{\text{Number of favorable outcomes}}{\text{Total number of possible outcomes}}$$

Example: Tossing a fair die → probability of getting a 4:

$$P(4) = \frac{1}{6}$$

🔹 2. Conditional Probability

- Sometimes the probability of an event depends on another event.
- **Definition:** Probability of event AA **given** that event BB has occurred:

- Definition: Probability of event A given that event B has occurred.

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Example:

In a deck of 52 cards, if we know a card drawn is red, what's the probability it's a king?

$$P(\text{King} \mid \text{Red}) = \frac{P(\text{King and Red})}{P(\text{Red})} = \frac{\frac{2}{52}}{\frac{26}{52}} = \frac{2}{26} = \frac{1}{13}$$

3. Bayes' Theorem

- **Used to reverse conditional probabilities.**
- Formula:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:

$$P(B) = \sum_i P(B|A_i) \cdot P(A_i)$$

Example:

A medical test is 95% accurate for a disease that affects 1% of population.

- $P(D) = 0.01$ (disease)
- $P(+|D) = 0.95$ (test positive if diseased)
- $P(+|\neg D) = 0.05$ (false positive)

Find probability that a person has the disease given they tested positive:

$$P(D|+) = \frac{P(+|D) \cdot P(D)}{P(+|D)P(D) + P(+|\neg D)P(\neg D)}$$

$$= \frac{0.95 \times 0.01}{0.95 \times 0.01 + 0.05 \times 0.99} = \frac{0.0095}{0.0095 + 0.0495} \approx 0.16$$

Even with a positive test, chance of having disease is **only ~16%** (because disease is rare)

- **Probability** → measures likelihood of events.
- **Conditional Probability** → probability of event under condition.
- **Bayes' Theorem** → powerful tool to update beliefs with new evidence.

Toppic : Readability and Pythonic code

Readability

Readability refers to how easily code can be read, understood, and maintained by humans. Python emphasizes readability more than many other languages.

1 .Meaningful Names:

- Variables, functions, and class names should describe their purpose.
  Avoid single letters except in loops (like i or j).

```
# Bad readability

x = 10

y = 20

z = x + y

# Good readability

width = 10

height = 20

area = width + height
```

2. Consistent Indentation and Spacing:

- Python uses indentation to define blocks; always be consistent.
- Follow **PEP 8** style guide: 4 spaces per indentation.

```
# Bad indentation
```

```
if x>10:
print("x is greater than 10")
# Good indentation
if x > 10:
    print("x is greater than 10")
```

3. Commenting and Documentation:

- Use comments to explain *why* code is written, not *what* it does.
- Use docstrings for functions/classes.

```
def calculate_area(radius):
    """
    Calculate the area of a circle given the radius.
    Formula: area = π * r^2
    """
    import math
    return math.pi * radius ** 2
```

4. Avoid Deep Nesting:

- Too many nested loops or conditionals reduce readability.
- Consider breaking code into functions.

5. Readable Structure:

- Use blank lines to separate sections.
- Keep code blocks small and organized.

Pythonic Code

Pythonic code is code that follows Python's conventions and idioms. It's not just about functionality; it's about writing code in a way that's natural for Python and takes advantage of its features.

Characteristics of Pythonic Code:

1. Use Built-in Functions:
    - Python provides many built-in functions that simplify tasks.

```
# Non-Pythonic

squares = []

for i in range(10):

    squares.append(i*i)

# Pythonic

squares = [i*i for i in range(10)]  # List comprehension
```

2. Prefer Readable Expressions:

- Use concise, clear syntax rather than verbose code.

```
# Non-Pythonic

if len(my_list) != 0:

    print("List has items")

# Pythonic

if my_list:

    print("List has items")
```

3. Follow "Easier to ask for forgiveness than permission" (EAFP):

- Instead of checking for conditions, try the operation and handle exceptions.

```
# Non-Pythonic

if 'key' in my_dict:
```

```
    value = my_dict['key']
# Pythonic
try:
    value = my_dict['key']
except KeyError:
    value = None
```

4. Use Generators and Iterators Where Appropriate:

- Saves memory and makes code elegant.

```
# Pythonic: generator expression
total = sum(x*x for x in range(10))
```

Functions and DRY Principles

1. Functions in Python

A function is a reusable block of code that performs a specific task.
Instead of writing the same logic multiple times, we can define a function once and call it whenever needed.

Key Parts of a Function:

1. Definition – created using def keyword.
2. Parameters – input values passed to the function.
3. Return Value – output produced by the function.
4. Calling – using the function when needed.

Example:

```
# Function definition
def greet(name):
    return f"Hello, {name}!"
```

```
# Function call

print(greet("Lekshmi"))

print(greet("Rahul"))
```

Output:

```
Hello, Lekshmi!

Hello, Rahul!
```

2. DRY Principle

DRY = *Don't Repeat Yourself*

It is a **software design principle** that says:
"Every piece of knowledge must have a single, unambiguous, and authoritative representation in the system."

In simple words: **Don't write the same code in multiple places. Write it once, and reuse it.**

Bad Example (Without DRY):

```
# Calculating area of rectangle in multiple places

length1, width1 = 5, 10

area1 = length1 * width1

print("Area 1:", area1)

length2, width2 = 7, 3

area2 = length2 * width2

print("Area 2:", area2)
```

Problem: The formula is repeated. If we need to change logic, we must edit it everywhere.

Good Example (With DRY using Functions):

```
def rectangle_area(length, width):
```

```
    return length * width

print("Area 1:", rectangle_area(5, 10))

print("Area 2:", rectangle_area(7, 3))
```

Advantages of the above code :

- Logic is written **once** inside the function.
- Easy to **reuse** and **modify** later.

Functions = Reusable code blocks → improve readability, modularity, and maintainability.

DRY Principle = *Don't Repeat Yourself* → avoid duplication, write code once, reuse many times.

Exception Handling and **Validation**

1. Exception Handling

Definition:

- An **exception** is an error that occurs during program execution, disrupting the normal flow.
- **Exception Handling** allows programs to deal with errors gracefully instead of crashing.

Key Concepts:

- **try** → block of code to test for errors.
- **except** → block of code to handle errors.
- **else** → block executed if no errors occur.
- **finally** → block that always executes (cleanup code).

Example in Python:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("❌ Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("❌ Cannot divide by zero.")
else:
    print("✅ Result is:", result)
finally:
    print("Program execution completed.")
```

Benefits:

- Prevents program crashes.
- Provides meaningful error messages.
- Ensures resources (files, DB connections) are safely released.

2. Validation

Definition:

- **Validation** is the process of checking input data for correctness, completeness, and security before using it.

Types of Validation:

1. **Data Type Validation** → check if input is integer, string, float, etc.
2. **Range Validation** → check if input lies within acceptable range.
3. **Format Validation** → ensure input matches pattern (e.g., email, phone number).
4. **Business Rule Validation** → check logical conditions (e.g., age > 18).

Example in Python:

```
def validate_age(age):
    if not isinstance(age, int):
        raise TypeError("Age must be an integer.")
    if age < 0 or age > 120:
        raise ValueError("Age must be between 0 and 120.")
    return True

try:
    user_age = int(input("Enter age: "))
    if validate_age(user_age):
        print("✅ Age is valid")
except Exception as e:
    print("❌ Validation failed:", e)
```

3. Relationship Between Exception Handling & Validation

- **Validation** prevents errors by **checking input upfront**.

- **Exception Handling** deals with errors that still occur during execution.
- Always **validate inputs** first.

- Use **exception handling** as a safeguard for unexpected runtime issues.
- **Exception Handling** = catching and managing errors.

- **Validation** = preventing errors by ensuring inputs are correct.

- Together, they make programs **robust, secure, and user-friendly**.