Notes for :

Date:

TABLE OF CONTENTS

Programming

Programming is the process of writing instructions (code) that a computer can understand and execute to perform a specific task.

What is Python?

Python is a **high-level, interpreted programming language** that is widely used for various applications like web development, data science, automation, AI, and more.

- ***Python is a popular programming language.***
- ***It was created by Guido van Rossum and released in 1991 at* CWI (centrum wiskunde & informatica) *Netherland.***
- ***Python is a general purpose language***
- ***Python is High level language (reason it is written in general English language)***
- ***Python is dynamic (no need to writing data type)***
- ***Python code has indentation.***

History Of Python

- Guido van Rossum started working on Python at **CWI (Centrum Wiskunde & Informatica)** in the Netherlands in late 1980's
- Python **1.0** was officially released in 1991.

Why Learn Python?

- **Easy to Learn** – Python has a simple and readable syntax, making it beginner-friendly.
- **Versatile** – Used in web development, machine learning, automation, game development, and more.

- **Large Community** – Millions of developers use Python, so you can find plenty of resources and help online.

Key Features of Python:

- **Simple & Readable** – Looks almost like English.
- **Interpreted Language** – No need to compile; runs line by line.
- **Dynamically Typed** – No need to declare variable types.
- **Huge Libraries** – Supports many built-in libraries for tasks like data analysis, web development, and AI.

Why is Python Platform-Independent?

1. **Interpreted Language** – Python code is executed by the Python interpreter, which is available for multiple operating systems.

2. **Write Once, Run Anywhere** – You don't need to rewrite the code for different platforms.
3. **Cross-Platform Libraries** – Python libraries work on multiple OS without issues.

How Python was named..

python was named after the British comedy show **"Monty Python's Flying Circus."**
Guido van Rossum, Python's creator, was a fan of the show and wanted a name that was **short, unique, and a bit mysterious**. He didn't name it after the snake, but the association with a python (the reptile) became popular over time.
So, Python's name comes from **Monty Python** and not from the snake!

How Machine understands the code

Compiler converts the code (set of instructions) which will be in High-level language in to Low level language i.e., in the form of 0's and 1's .

-- A machine (computer) reads **0s and 1s** internally using **electronic circuits** that operate on the principles of **binary logic** and **digital electronics**. Here's how it works step by step

1. Electric Signals Represent 0 and 1

- Computers use **electrical voltage levels** to represent binary values:
  - **High Voltage (e.g., +5V or +3.3V)** → Represents **1**
  - **Low Voltage (0V)** → Represents **0**
- These signals travel through circuits made of **transistors** (tiny electronic switches).

2. Transistors Process Binary Data

- A **transistor** acts as a switch:
  - If **current flows** → It represents **1**
  - If **no current flows** → It represents **0**
- Billions of transistors inside processors (CPUs, RAM, storage) work together to perform calculations.

3. Logic Gates Perform Operations

- **Logic gates** (AND, OR, NOT, etc.) are built using transistors.
- They process binary inputs and produce binary outputs.
- Example:
  - **AND Gate:** `1 AND 1 = 1`, `1 AND 0 = 0`
  - **OR Gate:** `1 OR 0 = 1`, `0 OR 0 = 0`
- These gates form circuits that perform arithmetic, store data, and control operations.

4. Memory Stores 0s and 1s

- **RAM (Random Access Memory):** Uses tiny capacitors that hold electrical charges (`1`) or no charge (`0`).
- **Hard Drives (HDDs):** Use magnetic fields to store binary data.
- **Solid-State Drives (SSDs):** Use flash memory, where transistors store charge states

5. Data Transmission in Buses

- Inside a computer, data (0s and 1s) moves through **buses** (electronic pathways) connecting CPU, RAM, storage, and other components.
- The CPU reads instructions and data by interpreting **binary signals**.

6. Conversion to Human-Readable Form

- Eventually, this binary data is **converted** into human-readable text, images, sounds, and videos using software and digital-to-analog converters.

Basic “Hello world” program

```
1    print("Hello, World!")
2
```

Output:

Input & Output Functions

`print()` – Display output

```
print("Hello, World!")

```

input() – Take input from user

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

```

By Default input() stores any value entered in to string format, so use type casting to convert the datatype.

```
name = input("enter your name")  #take input from user , it is string
print(name)
print(type(name))

num = input("enter any number :")
print(num)
print(type(num)) # by default it writes str
num2= num+ 20
print(num2)

```

OUTPUT

```
samle
<class 'str'>
enter any number :15
15
<class 'str'>
Traceback (most recent call last):
  File "d:\kali\C tutorial\tempCodeRunnerFile.python", line 8, in <module>
    num2= num+ 20
          ~~~^~~~
TypeError: can only concatenate str (not "int") to str
PS D:\kali\C tutorial>
```

Variables

A **variable** is a named storage for data in memory, or the name of the container where the data is stored. In Python, you don't need to declare the type of a variable explicitly; it is dynamically assigned based on the value you store in it.

- So, python is known as the dynamic programming language.
- Variables store values and can hold different data types.
- Data types include int, float, str, list, tuple, set, dict, etc.

Program for Declaring Variables

```
1    x = 10        # Integer
2    name = "John"  # String
3    price = 99.99  # Float
4    is_active = True  # Boolean
5
```

Explaination:

- `x` is assigned an **integer** value.
- `name` is assigned a **string**.
- `price` is assigned a **floating-point** value.
- `is_active` is assigned a **boolean** value.

Rules for Naming Variables

- Must start with a letter (A-Z or a-z) or an underscore _
- Can contain letters, numbers, and underscores
- Case-sensitive (myVar and myvar are different)
- Cannot use Python keywords (like if, else, class, etc.

Invalid variable names:

```
2name = "Alice"  # ❌ Cannot start with a number
my-name = "Bob"  # ❌ Hyphens not allowed
if = 100         # ❌ Cannot use keywords

```

Strings:

```
integer_value = 100
float_value = 4.5
boolean_value = True
boolean_value2 = False

# string declarations

string_value = "python course"
string_value2 = 'python course'

# multi line string declaration
string_value3 = """python
course
"""

print(len(string_value))
print(string_value[0])
print(string_value[-1])

# how to access any part of string
# with starting and ending point
# here it will include the value in starting index
# but not include the value in end index
print(string_value[0:5])
print(string_value[0:])
print(string_value[:5])
print(string_value[:])
```

Output

```
PS D:\kali\python\Learn\Mosh> & C:/U
13
p
e
pytho
python course
pytho
python course
PS D:\kali\python\Learn\Mosh>
```

Escape Sequences in python:

```
# Escape sequences in python

# \"    - adds double quote
# \'    - adds single quote
# \\    - adds back slash
# \n   - new line

print("python  \"course")
print("python  \'course")
print("python  \\course")
print("python  \n course")
```

output:

```
PS D:\kali\python\Learn\Mosh> & C:/User
python  "course
python  'course
python  \course
python
 course
PS D:\kali\python\Learn\Mosh>
```

Using concatenation and F-string :

```
first = "Aakash"
last = "varma"
full = "first" + " " + last

format_string = f"{first} {last}"
format_string2 = f"{first}-----{last}"

print(full)
print(format_string)
print(format_string2)

print(f"""
      This type of formatting is used
      with help of f-string {first} and
      now we can add next variable {last}
""")

print(f"""
This type of formatting is used
      with help of f-string {len(first)} and
      now we can add next variable {5+6}
""")
```

Output:

```
PS D:\kali\python\Learn\Mosh> & C:/Users/kir
first varma
Aakash varma
Aakash-----varma

      This type of formatting is used
      with help of f-string Aakash and
      now we can add next variable varma


This type of formatting is used
      with help of f-string 6 and
      now we can add next variable 11

PS D:\kali\python\Learn\Mosh>
```

DataTypes

Python has various **built-in data types**.

Data Type	Example	Description
int	x = 100	Whole numbers
float	y = 10.5	Decimal numbers
str	name = "Alice"	Text (string)
bool	flag = True	Boolean values (True/False)
list	fruits = ["apple", "banana"]	Ordered, mutable collection
tuple	point = (2, 3)	Ordered, immutable collection
set	nums = {1, 2, 3}	Unordered, unique elements
dict	person = {"name": "John", "age": 30}	Key-value pairs

Category	Data Type	Example
Numeric	int	100
	float	10.5
	complex	3 + 4j
Boolean	bool	True, False
Text	str	"Hello"
Sequence	list	["a", "b"]
	tuple	("x", "y")
	range	range(5)
Set	set	{1, 2, 3}
	frozenset	frozenset({1, 2, 3})
Mapping	dict	{"key": "value"}
Binary	bytes	b'hello'
	bytearray	bytearray([65, 66])
	memoryview	memoryview(b'abc')
None	NoneType	None

Immutable & Mutablle Datatypes+

Immutable(means we cannot change the value)
Numbers
Strings
Tuples

CODE to prove that these DataTypes are immutable.

```
a= 10
print(id(a))

a=20
print(id(a))
```

---

mutable(means we can change the value)
Lists
Dictionary
Sets

1. Numeric Data Types

These include **integers, floating-point numbers, and complex numbers**.

Integer (`int`)

- Stores **whole numbers** (positive, negative, or zero).
- No decimal point.
- Unlimited precision (not fixed to 32-bit or 64-bit like in other languages).

```
a = 100       # Positive integer
b = -50       # Negative integer
c = 0         # Zero

print(type(a))  # Output: <class 'int'>

```

Floating-Point (`float`)

- Stores **decimal numbers**.
- Can represent very large or very small numbers.

```
x = 10.5
y = -3.14159
z = 1.0  # Even if it's 1.0, it's a float.

print(type(x))  # Output: <class 'float'>

```

Complex Numbers (`complex`)

- Stores numbers in the form `a + bj`, where `j` is the imaginary unit (√-1).

```
num = 3 + 4j  # 3 is the real part, 4j is the imaginary part.
print(type(num))  # Output: <class 'complex'>

# Accessing real and imaginary parts
print(num.real)  # Output: 3.0
print(num.imag)  # Output: 4.0

```

2. Boolean (bool)

- Represents `True` or `False` values.
- Used in **conditional statements** and **logical operations**.

- Internally, `True` = `1` and `False` = `0`.

```
a = True
b = False

print(type(a))  # Output: <class 'bool'>

print(5 > 2)  # Output: True
print(10 == 20)  # Output: False

```

3. Text Data Type (str)

- Represents **sequences of characters** (text).
- Defined using single (`'`), double (`"`), or triple (`''' """`) quotes.

```
s1 = "Hello, Python!"
s2 = 'Single quotes work too'
s3 = '''Multiline
string using triple quotes.'''

print(type(s1))  # Output: <class 'str'>

# String operations
print(s1[0])   # Output: H (first character)
print(s1[-1])  # Output: ! (last character)
print(s1[0:5]) # Output: Hello (substring)
print(len(s1)) # Output: 14 (length of string)

```

4. Sequence Data Types

These store **multiple values** in an **ordered manner**.

List (`list`)

- **Ordered** and **mutable** (can change elements).
- Can store different data types.

```
fruits = ["apple", "banana", "cherry", 10, 20.5]

print(type(fruits))  # Output: <class 'list'>

# Access elements
print(fruits[0])   # Output: apple
print(fruits[-1])  # Output: 20.5

# Modify elements
fruits[1] = "grape"
print(fruits)  # Output: ['apple', 'grape', 'cherry', 10, 20.5]

```

Tuple (`tuple`)

- **Ordered**, but **immutable** (cannot change elements after creation).
- Faster than lists.
- The values in tuples cant't be changed

```
colors = ("red", "green", "blue")

print(type(colors))  # Output: <class 'tuple'>

print(colors[0])  # Output: red

# colors[0] = "yellow"  # ❌ This will give an error (tuples are immutable)

```

Range (`range`)

- Represents a **sequence of numbers**.
- Commonly used in loops.

```
r = range(1, 10, 2)  # (start, stop, step)
print(list(r))  # Output: [1, 3, 5, 7, 9]

```

5. Set Data Types

Set (`set`)

- **Unordered** collection of unique elements.
- No duplicate values.
- Faster operations compared to lists.

```python
nums = {1, 2, 3, 4, 4, 5}
print(nums)  # Output: {1, 2, 3, 4, 5} (removes duplicates)

nums.add(6)
nums.remove(2)

print(nums)  # Output: {1, 3, 4, 5, 6}

```

Frozen Set (`frozenset`)

- **Immutable version of a set**.
- Cannot add or remove elements.

```python
fs = frozenset([10, 20, 30])
# fs.add(40)  # ❌ This will cause an error

```

6. Dictionary (`dict`)

- Stores **key-value pairs**.
- **Keys are unique** and **values can be of any type**

```python
student = {
    "name": "Alice",
    "age": 22,
    "marks": [85, 90, 78]
}

print(type(student))  # Output: <class 'dict'>

print(student["name"])  # Output: Alice

# Adding new key-value pair
student["city"] = "New York"
print(student)        # output {'name': 'Alice', 'age': 22, 'marks': [85, 90, 78], 'city': 'New York'}

```

7. Binary Data Types

Python provides special types for handling **binary data**.

Bytes (`bytes`)

- Immutable sequence of bytes.

- Used in handling **binary files** (images, videos, etc.).

```
b = bytes([65, 66, 67])
print(b)  # Output: b'ABC'

```

Bytearray (`bytearray`)

- Mutable version of `bytes`.

```
ba = bytearray([65, 66, 67])
ba[0] = 68  # Can be modified
print(ba)  # Output: bytearray(b'DBC')

```

Memoryview (`memoryview`)

- Provides a **memory-efficient way** to access binary data.

```
mv = memoryview(b"Hello")
print(mv[0])  # Output: 72 (ASCII of 'H')

```

8. None Type (`NoneType`)

- Represents **null or empty values**.
- **Used when a variable** has no value assigned **yet.**

```
x = None
print(type(x))  # Output: <class 'NoneType'>

```

Operators in Python

Operators are symbols that perform operations on variables and values. Python has different types of operators:

1.Arithmetic Operators

Used for mathematical operations.

```
a = 10
b = 3

print(a + b)  # Addition → 13
print(a - b)  # Subtraction → 7
print(a * b)  # Multiplication → 30
print(a / b)  # Division → 3.3333
print(a // b) # Floor Division → 3 (removes decimals)
print(a % b)  # Modulus → 1 (remainder)
print(a ** b) # Exponentiation → 1000 (10^3)

```

2.Relationalc/comparision Operators

```
x = 5
y = 10

print(x == y)  # False (Equal to)
print(x != y)  # True (Not equal to)
print(x > y)   # False (Greater than)
print(x < y)   # True (Less than)
print(x >= y)  # False (Greater than or equal to)
print(x <= y)  # True (Less than or equal to)

```

3. Logical Operators

Used to combine conditions

```
a = True
b = False

print(a and b)  # False (Both must be True)
print(a or b)   # True (At least one must be True)
print(not a)    # False (Negation)

```

4. Assignment Operators

Used to assign values

```
x = 10  # Assign
x += 5  # Equivalent to: x = x + 5 (x becomes 15)
x -= 2  # Equivalent to: x = x - 2 (x becomes 13)
x *= 3  # x = x * 3
x /= 4  # x = x / 4
x %= 2  # x = x % 2

```

5. Bitwise Operators

Used for binary-level operations.

#Bitwise AND &

```
a = 0b1010
b = 0b1100
c = a&b
print(c)     #output : 0b1000
```

#Bitwise OR " | " …|

```
a = 10
b = 20
c = a&b
print(c)
```

#Bitwise XOR ^

```
a = 0b1010
b = 0b1100
c = a^b
print(c)

# OR operators prints true is any one is true
#but XOR prints true when there are different inputs
T   T  = F
T  F   = T
F  T  =   T
F   F =  F
```

#Bitwise NOT ~

```
a = 0b1010
b = 0b1100
c = ~a   (  0 is added before as positive sign 00000 1010, Bitwise NOT means compliment 0 to 1 , 1 to 0. (we added 0 in the beginning as a positive sign)
0 1010 ➡  1 0101 (1 represents negative sign)
```

now inorder to represent this , we need to take 2's compliment

1 0101

1's compliment is 1010

Now add 1

1010

1

1011

(- sign) 1 1011 [this is the answer]

d= ~b

print(c)

print(b)

#Bitwise Left Shift << (it is always multiply by 2)

a = 0b1010

b = 0b1100

c = a<<2 (always dividing by 2)

print(c)

#Bitwise Right Shift >>

a = 0b1010

b = 0b1100

c = a>>2

print(c)

program:

```
a=10
b=20
c=0b1010
d=0x1011

e=a&b
f=a|b
g=a^b
i=~a #
j=~b #bitwise NOT

print(bin(a))
print(hex(a))
print(oct(a))
```

```python
a = 5  # 101 in binary
b = 3  # 011 in binary

print(a & b)  # 001 → 1 (AND)
print(a | b)  # 111 → 7 (OR)
print(a ^ b)  # 110 → 6 (XOR)
print(~a)     # -6 (NOT)
print(a << 1) # Left Shift → 10 (binary 1010)
print(a >> 1) # Right Shift → 2 (binary 10)

```

6. Identity Operators

Operator	Description
`is`	Returns `True` if both variables refer to the same object in memory.
`is not`	Returns `True` if both variables do **not** refer to the same object in memory.

program:

```python
a=10
b=10
print(a is b)
print(a is not b)
print(id(b))
print(id(a))
```

```python
list1=[1,3,5]
list2=[1,3,5]
print(list1 is list2) #checks object means where it is stored(location ID)
print(list1 == list2) #checks value in the location
```

7. Membership Operators

Checks wheather is in the part of the existing data, example to check userid id

```python
list1=[1,10,20]
print(10 in list1)
```

Notes by : Kiran versatile

Indentation in Python

Indentation in Python refers to the spaces or tabs at the beginning of a line of code. Unlike other programming languages that use `{}` (curly brackets) to define code blocks, Python relies on indentation to indicate block structures like loops, conditionals, and function definitions.

Why is Indentation Important?

1. **Defines Code Blocks**
   - In Python, indentation is not just for readability; it is **mandatory**.
   - Without proper indentation, Python will throw an error.
2. **Improves Readability**
   - Indentation helps in structuring code in a clear and understandable way.
3. **Prevents Errors**
   - Incorrect indentation will lead to `IndentationError` or unexpected behavior in programs

Correct indentation

```
def greet():
    print("Hello, World!")  # Indented properly under function

greet()

```

Wrong Indentation

```
def greet():
print("Hello, World!")  # Missing indentation

greet()

```

Expected Error: IndentationError: expected an indented block

Indentation in Conditional Statements:

```
age = 18
if age >= 18:
    print("You are an adult.")  # Indented inside the if block
else:
    print("You are a minor.")  # Indented inside the else block

```

Indention in LOOPS

```
for i in range(3):
    print("Iteration:", i)  # Indented inside the loop

```

Indention with Nested Structures

```
for i in range(2):
    print("Outer Loop:", i)
    for j in range(2):
        print("  Inner Loop:", j)  # Further indented for nested loop

```

How Many Spaces for Indentation?

- Python **PEP 8 (style guide)** recommends using **4 spaces per indentation level**.
- Using **tabs and spaces together** can cause errors.

Recommended 4 spaces

```
if True:
    print("Use 4 spaces per indentation level.")

```

- Python uses indentation to define code blocks.
- Always use 4 spaces per indentation level (avoid mixing spaces and tabs).
- Improper indentation leads to IndentationError.
- Proper indentation makes code readable and error-free.

Type Checking and Type Conversion in Python

Python is a dynamically typed language, meaning that variables do not have explicit types and can change type during execution. However, checking and converting types can be useful for ensuring data integrity and avoiding errors.

1. Type Checking

Type checking refers to verifying the data type of a variable or object.

Using `type()`

The `type()` function returns the type of an object.

```
x = 10
print(type(x))  # Output: <class 'int'>

y = "Hello"
print(type(y))  # Output: <class 'str'>

```

Using isinstance()

The `isinstance()` function checks whether a variable is an instance of a particular type.

```
x = 10
print(isinstance(x, int))  # Output: True
print(isinstance(x, float))  # Output: False

```

You can check for multiple types using a tuple:

```
x = 3.14
print(isinstance(x, (int, float)))  # Output: True

```

2. Type Conversion (Type Casting)

Type conversion is the process of converting one data type into another.

Implicit Type Conversion (Automatic)

Python automatically converts one type to another when needed.

```
a = 5     # int
b = 2.5   # float
c = a + b  # float (automatic conversion)
print(c)   # Output: 7.5
print(type(c))  # Output: <class 'float'>

```

Explicit Type Conversion (Manual)

Python provides built-in functions to convert types manually.

Function Converts To

int(x)	Integer
float(x)	Floating point number
str(x)	String
bool(x)	Boolean
list(x)	List
tuple(x)	Tuple
set(x)	Set
dict(x)	Dictionary

Examples for Type conversion :

```python
# Integer to float
x = 10
y = float(x)
print(y)  # Output: 10.0

# Float to integer
a = 3.9
b = int(a)
print(b)  # Output: 3 (decimal part is truncated)

# String to integer
s = "100"
num = int(s)
print(num)  # Output: 100

# Integer to string
n = 50
s = str(n)
print(s)  # Output: '50'

# List to tuple
lst = [1, 2, 3]
tpl = tuple(lst)
print(tpl)  # Output: (1, 2, 3)

# String to List
s = "hello"
l = list(s)
print(l)  # Output: ['h', 'e', 'l', 'l', 'o']

# Boolean Conversion
print(bool(0))  # Output: False
print(bool(1))  # Output: True
print(bool(""))  # Output: False
print(bool("Python"))  # Output: True
```

3. Handling Errors in Type Conversion:

When converting types, invalid conversions can cause errors.

```
s = "abc"
try:
    num = int(s)  # This will raise a ValueError
except ValueError as e:
    print(f"Error: {e}")  # Output: Error: invalid literal for int() with base 10: 'abc'

```

- **Type Checking**: Use `type()` or `isinstance()`.
- **Type Conversion**:

Implicit: Python automatically converts types where needed.

Explicit: Use built-in functions like `int()`, `float()`, `str()`, `list()`, etc.

- **Error Handling**: Use `try-except` for invalid conversions.

Functions in Python

A **function** is a block of code that performs a specific task. Functions help in **code reusability**, **organization**, and **modularity**

1. Types of Functions in Python

Python has two types of functions:

1. **Built-in Functions** – Predefined in Python (e.g., `print ()`, `len()`, `max()`, `min()`).
2. **User-Defined Functions** – Created by programmers using the `def` keyword.

2. Creating a User-Defined Function

Basic Function

```
def sample():
    print("Hello, students Good morning !")

sample()  #calling the function

#OUTPUT : Hello, students Good morning !
```

- **`def sample():` → Defines a function named `sample`.**
- `print("Hello,Students Good morning !")` → Code inside the function runs when it's called.
- `sample()` → Calls the function to execute.

3. Function with Parameters

Functions can accept values (parameters) to work with.

```
def hai(name):
    print ("hello " + name +"!")

hai("Bunty") #output : hello Bunty!
```

- `name` is a parameter that takes different values when the function is called.

4. Function with Return Value

A function can return a result using `return`

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

result = add(5,10)
print(result )   #output: 15
```

- The function `add(a, b)` takes two numbers and returns their sum.
- `return` gives back the result to the caller.

5.Default Parameter Values

If a parameter is not provided, a default value is used.

```
def hai(name="Default_Name"):
    print ("hello " + name +"!")

hai("Bunty") #output : hello Bunty!
hai()
#if we wont provide any value for decalered
# parameter in function arguments, it will take default
#value

#OUTPUT: hello Bunty!
#         hello Default_Name!
```

7. Lambda (Anonymous) Functions

A **lambda function** is a small anonymous function used for short tasks.

```
# writes odd numbers and filters means skips the numbers that are divisible by 2
"""
num = [10, 12,19, 13,25,35,36,16]

def even(x):
    return x % 2
evens = list(filter(even, num))

print (evens)
"""
#writes only even numbers that satisfies the condition inside the function
num = [10, 12,19, 13,25,35,36,16]

def even(x):
    return x % 2 ==0
evens = list(filter(even, num))

print (evens)
-------------------------------------------------------------------------------
#using Lambda function and mainly used for temporary purpose
num = [10, 12,19, 13,25,35,36,16]
evenn = list(filter( lambda x : x % 2 ==0, num)) # lambda function is mainly used for, when only
sorted, filtered, or mapped
print (evenn)
```

8. Function with Arbitrary Arguments (*args & **kwargs)

Using `*args` (Multiple Positional Arguments)

```python
def add_numbers(*numbers):
    return sum(numbers)

print(add_numbers(1, 2, 3, 4, 5))  # Output: 15

```

Using **kwargs (Multiple Keyword Arguments)

```python
def person_info(**info):
    print(info)

person_info(name="Alice", age=25, city="New York")
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

```

Functions help organize code and avoid repetition.
Use `def` to define a function.
Functions can have **parameters**, **return values**, and **default values**.
`lambda` creates small, **anonymous functions**.
`*args` allows multiple **positional** arguments.
`**kwargs` allows multiple **keyword** arguments.

Notes For Date:

Type Casting:

Program: ⬇

```
Num1= input("enter first number :")
Num2= input("enter second number :")
print(Num1+Num2)
```

Output ⬇

```
PS D:\kali\C tutorial> python -u "d:\kali\C tutorial\tempCodeRunnerFile.python"
enter first number :23
enter second number :20
2320
```

- Output is concatenated
- Input is stored in the form of “STR string” data type.
- We need to change the Datatype explicitly.
- Use explicit type casting function [int() or float() or char()….]

```
Num1= int(input("enter first number :"))
Num2= int(input("enter second number :"))
print(Num1+Num2)
```

output ⬇

```
enter first number :10
enter second number :20
30
PS D:\kali\C tutorial>
```

Implicit Type casting of Datatypes ⬇

```
Num1= 10
Num2= 20.5 # this is float datatype
add= Num1+Num2 #answer will be in float datatype int + Float = float
print(add)
print(type(add)) #lets check the type of add datatype
```

output ⬇

```
le.python"
30.5
<class 'float'>
```

Example program for practice:

Here we used
float(), to print give integer in float
oct(), to print give integer in octa decimal
hex(), to print give integer in hexa decimal
bin(), to print give integer in Binary number
ord(),

```
num = 15
f = float(num)
print(f)

H = hex(num)
print(H)

O = oct(num)
print(O)

B = bin(num)
print(B)

#for ASCII value
char = "A"
print(ord(char))
```

output ⬇

```
15.0
0xf
0o17
0b1111
65
```

Escape character ⬇

These are special characters used to represent non-printable or special characters within the string

\ : backslash
\n : newline
\t : tab
\r : carriage return

```
print(' I'll call you later')
```

output error ⬇

```
PS D:\kali\C tutorial> python -u "d:\kali\C tutorial\tempCodeRunne
le.python"
  File "d:\kali\C tutorial\tempCodeRunnerFile.python", line 1
    print(' I'll call you later')
                               ^
SyntaxError: unterminated string literal (detected at line 1)
```

To allow any string to print any special character ⬇

```
print(" I'll call you later")
```

or

```
print(' I\'ll call you later')
```

```
print(" this is important \"topic\" here ")
```

```
print(" this is \t important \n here ")
```

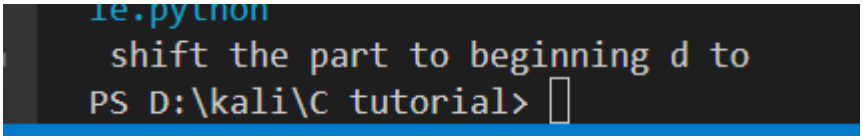
output: ⬇

```
this is         important
here
```

\r (carriage return)

```
print(" This escape character is used to \r shift the part to
beginning ")
```

output: ⬇



Notes by : Kiran versatile

Operator precedence & Associativity

Precedence	Operator(s)	Description	Associativity
1 (Highest)	()	Parentheses (Grouping)	Left to Right
2	**	Exponentiation	Right to Left
3	+x, -x, ~x	Unary Plus, Unary Minus, Bitwise NOT	Left to Right
4	*, /, //, %	Multiplication, Division, Floor Division, Modulus	Left to Right
5	+, -	Addition, Subtraction	Left to Right
6	<<, >>	Bitwise Shift Left, Bitwise Shift Right	Left to Right
7	&	Bitwise AND	Left to Right
8	^	Bitwise XOR	Left to Right
9	|	Bitwise OR	Left to Right
10	==, !=, >, <, >=, <=, is, is not, in, not in	Comparison, Identity, and Membership Operators	Left to Right
11	not	Logical NOT	Left to Right
12	and	Logical AND	Left to Right
13	or	Logical OR	Left to Right
14 (Lowest)	=, +=, -=, *=, /=, //=, %=, **=, &=, |=, ^=, >>=, <<=	Assignment Operators	Right to Left

```
print(10*2+3/3-4)
```

output: ⬇

```
17.0
```

Notes by : Kiran versatile

Types of Logic Gates in Python

AND Gate

```
salary= 2000
age= 18
if age>19 and salary>2000:
    print("True")
else:
    print("False")
```

OR Gate

```
salary= 2000
age= 18
if age>19 or salary>1000:
    print("True")
else:
    print("False")
```

NOT Gate

```
a= True
b= False
if not a:
    print("cpndition becomes false now")
else:
    print("now it will print this becase it is false")
```

NAND Gate

NAND Gate (`NOT (A AND B)`):

NOR Gate

NOR Gate (`NOT (A OR B)`)

XOR Gate

Truth Table:

A	B	XOR Output (`A != B`)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

Truth Table:

A	B	XOR Output	XNOR Output (`A == B`)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Formatting in Python

Using f-strings only in versions (Python 3.6+)

```
name = "meghana"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

output

```
My name is meghana and I am 25 years old.
```

formatting numbers(old method)

```
pi = 3.14159
print(f"Pi rounded to 2 decimal places: {pi}")
```

```
pi = 3.14159
print(f"Pi rounded to 2 decimal places: {pi:.10f}")
```

Methods ⬇

Using Method **.format()**

```
name = "raju"
age = 30
print("My name is {} \n and I am {} years old.".format(name, age))
```

output ⬇

```
My name is raju
 and I am 30 years old.
```

- With Indexed Placeholders:

```
name= "ayshu"
age= "16"
roll_no = 101
city= "piler"
print("My name is {1} and I am {0} years old.".format(age, name))
print("My name is {1} and I am {0} years old. I live in {2} adn my roll number is {3}".format(age, name,city,roll_no))
```

string Methods ⬇

```
a = "10"

# string methods

text = "  Hello, students! , project "
value = "python course"
value2 = "   python course   "

print(a.isdigit())  # to check value is in digits or not
print(text.lower())  # lowercase
print(text.upper())  # uppercase
print(text.strip())  # Remove whitespace
print(text.replace("project", "World"))  # Replace substring
print(text.split(","))  # Split into a list

print(value.title())  # capatilze the starting letter
print(value2.lstrip())  # removes white space from beginning of string
print(value2.rstrip())  # removes from end
```

LIST Methods ⬇

```
#list methods
fruits = ["apple", "banana", "cherry"]

fruits.append("orange")  # Add element at the end
fruits.insert(1,"grape")  # Insert at a specific index
fruits.remove("banana")  # Remove an element
fruits.pop()  # Remove the last item
fruits.reverse()  # Reverse the list
fruits.sort()  # Sort the list (ascending order)
```

Dictionary Methods ⬇

```
#dictionary methods
print(fruits)

student = {"name": "John", "age": 25, "city": "New York"}
```

```
print(student.keys())  # Get all keys
print(student.values())  # Get all values
print(student.items())  # Get key-value pairs as tuples

student.update({"age": 26})  # Update value
student.pop("city")  # Remove key-value pair

print(student)
```

Tuple Methods ⬇

```
#Tuple methods
numbers = (10, 20, 30, 40, 20, 50)

print(numbers.count(20))  # Count occurrences of 20
print(numbers.index(30))  # Find index of 30
```

set Methods ⬇

```
#set methods
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

print(set1.union(set2))  # Combine sets
print(set1.intersection(set2))  # Common elements
print(set1.difference(set2))  # Elements in set1 but not in set2
```

File Handling Methods ⬇

```
#file handling methods
set1.add(10)  # Add an element
set1.remove(2)  # Remove an element
```

Math Methods ⬇

```
#math methods
import math

print(math.sqrt(16))  # Square root
print(math.ceil(4.2))  # Round up
print(math.floor(4.9))  # Round down
```

```
print(math.pow(2, 3))  # Power (2^3)
print(math.factorial(5))  # Factorial of 5
```

Random module Methods ⬇

```
#random module methods
import random

print(random.randint(1, 10))  # Random integer
print(random.choice(["apple", "banana", "cherry"]))  # Random choice
print(random.shuffle([1, 2, 3, 4, 5]))  # Shuffle list
```

DateTime Methods ⬇

```
#datetime methods

from datetime import datetime

now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Format date and time
```

OS module Methods ⬇

```
#os module method
import os

print(os.getcwd())  # Get current working directory
print(os.listdir())  # List files in directory
```