Variables and Data types / Literals

* A variable is used to store values.

```
x = 10      # integer
y = 3.14    # float
name = 'Neena'  # string
is_active = True  # boolean
```

type(x) – to check type

x = variable / identifier

= → assignment operator

10 → datatyp / value.

Operators

* Operators are symbols or keywords that perform operation on variables. They help in manipulating data, perform calculations and control logic.

1. Arithmetic

addition (+), Subtraction (-), multiplication (*), division (/),

floor division (//), modulus (%), exponentiation (**)

↓ removes decimal point

↓ gives reminder of division

↓ power.

a = 10, b = 3

a ** 3 = 10^3 = 1000.

* Only 2 operations possible with String (+, *)

=> str + str = 'Neena' + 'Saram'

=> str + int = 'Neena' * 3.

2. Assignment

=, +=, -=, *=, /=, ...

* Operators are used to apply mathematical and logical operations on given data.

a = 10

a += 1 → a = a + 1

= 10 + 1 = 11

3. Comparision

==, >=, <=, >, <, !=

a = 10
b = 10

a == b → True.

a > b → False.

4. Logical / Relational.

a	b	or	and
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F

* we use this in conditional programming.

a = 10, b = 5

(a or b) > 15 → False.

(a and b) < 15 → True.

5. Membership

in, not in.

name = 'Neena'

'N' in name → True

'Y' in name → False.

'A' not in name → False.

6. Identity operator

is, is not

a = 10, b = 10

a is b → True.

a is not b → False.

Conditional Statements

1. if Statement

=> if condition.

```
age = 18
if age >= 18:
    print(" ")
```

* Conditional statement is a command that directs the programs flow of exec based on whether a specific condition is met (T or F)

* Conditional statement controls structure to make decision on program, and execute whether the condition is true / false

2. if-else statement

```
if condition:
    pri---
else:
```

```
temp = 15
if temp > 15:
    print(" ")
else:
    print(" ")
```

3. if-elif-else. – multiple condition but only 1 result

```
if condition1:
  - -
elif condition2:
  - -
else:
  - -
```

```
marks = 85
if marks >= 90:
    print(" ")
elif marks >= 80:
    print(" ")
else:
    print(" ")
```

4. Nested if

```
if condition1:
  - -
  if condition2:
```

```
num = 10
if num > 0:
    print(" ")
if num % 2 == 0:
    print(" ")
```

LOOPS

* a loop in python is a control flow statement that allows a block of code to be executed repeatedly as long as a certain condition is met or for every item in a sequence.

1. for loop

for variable in collection

```
for i in range(1,6)
    print(i)
```

2. while loop

Step1 – initialization
Step2 - condition
Step3 - operation

```
Count = 1
while count <= 5:
    print (count)
    count += 1.
```

3. Conditional with while.

```
a = 10
b = 10
while a<b:
    if (a%2 = 0):
        print(" ")
    else:
        print(' ')
```

* loop is a control structure where a block of code repeats until the specified condition met.

4. Nested while.

```
while():
{
    while();
    {
    }
}
```

* pass → pass logic / body to make prog correct.
* Break → It will terminate a block / loop / execution.
* Continue → To skip the iteration.

Break

```
for num in range(1,10):
    if num==5:
        print("found, stopping loop")
        break
    print(num).
```

Continue

```
for num in range(1,10):
    if num==5:
        print('ignore')
        continue
    print(num).
```

pass

```
for num in range(1,6):
    if num==3:
        pass
    print(num).
```

Functions

* defining functions

```
def greet():
    print("Hello, How are you")
greet()
```

def → Keyword to define a function

greet → function name.

() → nothing here.

* a function is a self-contained, named block of organized reusable code that is designed to perform specific task.

1. Definition

```
def func():
    pass

def func(para):
    pass
```

2. Invoking

```
func()

func(arg)
```

3. Module

1. 1. write file. filename.py

4. functions with loop

```
def oddeven(num):
    if num%2==0
        return 'even'
    else:
        return 'odd'
```

```
li=[1,2,3,4]
for i in li:
    oddeven(i)
```

5. Scope of variables

i) local — within body

ii) global — anywhere

```
name = 'Neena' → global.
def info():
    Age=27 → local.
    print("your info is", name, Age)
print(Age) → can't access local variable.
```

6. def func(*para)

↳ accept n values (tuple)

def func(**para).

↳ accept n value in dict format.

7) Recursion:

A function calling itself to solve a smaller part of same problem.

```
def countdown(n):
    if n == 0:
        print("Blast")
    else
        print(n)
        countdown(n-1) # recursive call.

countdown(5)
```

5
4
3
2
1
Blast

8) lambda.

* function without name.
* use when we need small function for short time
* one-line function using lambda keyword.

Syntax: lambda arguments: expression
↳ actual value.

Normal.

```
def add(x,y):
    return x+y
```

lambda

```
add = lambda x,y : x+y
print(add(2,3))
```

9) map()

* map() applies a function to every item in an iterable (a list)
* It maps (transforms) each item to something else

Syntax: map(func, iterable)
↳ list, tuple.

```
num = [1, 2, 3, 4]
doubled = map(lambda x : x * 2, num)
print (list(doubled))
```

10) Zip() – unzip()

```
li = [1, 2, 3, 4]
li2 = [5, 6, 7, 8]
res = zip (li1, li2)
list (res)
[(1,5), (2,6), (3,7), (4,8)]
```

11) Filter:

* Similar to map but keeps only True.

```
def oddeven (num):
    if num % 2 == 0:
        return True.
    else:
        return False.
```

```
li = [1, 2, 3, 4]
res = filter (oddeven, li)
[2, 4]
```

Data Structures

1. list, tuples
2. Sets
3. Dictionaries
4. list comprehension, dict comprehension

1. Lists

```
li = [1,2,3,4,5]
li
```

ordered → have specific position

Mutable → can change, add, remove.

Allow duplicates → Same value can appear. [1,1,2,3,4,2]

Any datatype → mix of int, str, float [1, 'Neena', 72.1]

Nested list → list in list [1, 2, [3,4],7,5]

Operations

Access item → li[0] → accest 1st item

-ve index → li[-1] → access last item

Slice list → li[1:3] → get part of list
(n-1)

Change list → li[0] = 10 → modify item at index.

add item (end) → li.append(6) → item add at end.

add item (position) → li.insert(2,99) → insert at specific index.

extend list → li.extend([2,3,4]) → add multiple items

remove → li.remove(3) → remove item.

remove by index → li.pop(2) → remove by position

delete item → del li[1] → delete by index.

clear list → li.clear() → empty the entire list

length of list → len(li)

Sort (ascending) → li.sort()

reverse list → li.reverse()

count occurance of x → li.count(1)

index of first x → li.index(x)

Sum of list → sum(li)

Small element → min(li)

large element → max(li)

2. Tuple.

tup = (1,2,3,4,5)

tup = (1,)

(1,) → must include (,) for one-element tuple.

If not (1) is just a element in parenthesis

*Ordered

* Immutable.

* allow duplicates

* any datatype.

* Nested Tuple

access item → tup[1]

-ve index → tup[-2]

Slice tuple → tup[1:3]

check if exists → 2 in my tup.

Count value → tup.count(2) → count occurance of 2.

→ len(tup)

→ sum(tup)

→ min(tup)

→ max(tup)

3. Sets

Set = {1, 2, 3, 4, 5}

* unordered
* unique (no duplicates)
* Mutable (partially) → add / remove
* Any datatype
* Cannot have lists → {1, 2, (3, 4), 5} is okay.

Add item → set.add(6) → single element

update multiple items → set.update([7, 8])

remove item → Set.remove(3) (error if not present)

discard item → set.discard(3) (no error if not present)

popitem → set.pop() → removes random element.

Clear set → set.clear() → empty the set.

Delete set → del sd → delete entire set.

union → A.union(B)

Intersection → A.intersection(B) or A & B.

Difference → A.difference(B) or A - B.

Symmetric difference → A.Symmetricdifference(B) or A ^ B

→ len(set)

→ 2 in set

→ Set.copy() → make a copy.

→ A.isdisjoint(B) → True if set have no common items

→ A.isSubset(B) → True if A is Subset of B.

→ A.issuperset(B) → True if A is Superset of B

Dictionary

Dict = { key : value }

Dict = {'Name' : 'Neena', 'Age' : 22, 'place' : 'Adilabad'}

- different data types
- Ordered
- Keys are unique.

Accessing:

Dict["name"] → Neena

Dict.get ("Age") → 22.

Dict. get("Age", 'Not found") → Not found.

Add / update → Dict["city"] = " Hyderabad"

remove Specific → Dict.pop("Age")

remove last item → Dict. popitem()

delete Specific → del Dict ["Age"]

clear all → Dict. clear().

get all keys → Dict. keys()

get all values → Dict. values()

items get all key value pair → Dict.items()

copy dictionary → new = Dict. copy()

update one dictionary with another → dict.update (other_dict)

list	→ mutable	ordered	allow dupli
tuple	→ immutable	ordered	allow dupli
Set	→ mutable	unordered	no dupli
dict	→ mutable.	ordered	Key-value mapping

5. List Comprehension.

[expression for item in iterable if condition]

num = [i for in range(5)] → [0,1,2,3,4]

even-num = [i for in rang(5) if i%2==0]

6. Dictionary Comprehension.

{key:value for item in iterable if condition}

Square = {i: i*i for i in range(5)} → {0:0, 1:1, 2:4, 3:9, 4:16}

even-sq = {i: i*i for i in range(10) if i%2==0} → {0:0, 2:4, 4:16, 6:36, 8:64

OOP's

object oriented programming is a programming based on concept of object rather than func & logic

class : blueprint

object : copy of blueprint

attribute : properties (identify diff properties)

method : functionality (it includes process)

Ex:

instagram → class.

user → object

name, dob, mobile → attributes

post, comment → methods.

Benifits of oop

- code reusability
- easy debugging & maintain
- support realword modeling

* Creating class ⇒ class class_name:

* Creating object ⇒ obj-name = class_name()

* Accessing attributes & methods ⇒ Obj-name.attribute or methods

```
Class human: # class creation
    def talk(self) # method 1
        print("I can talk")
    def walk (self) # method 2
        print ("I can walk")

neena = human() # object creation
neena.talk()
neena.walk()
```

* Self is default parameter.

Creating Attribute

init → constructor method used to define properties

```
Class human:
    def __init__(name,self, name, dob):
        self.name = name.
        self.dob = dob.
    def info(self):
        print ("My name is", self.name)
        print ("My dob is", self.dob)

obj = human ('Neena', "13-06-2003")
obj.name → neena
obj.info.
```

properties of oops:

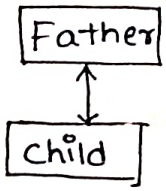
1. Inheritance - derive previous properties/methods
2. polymorphism - multiple form
3. Encapsulation - enable to hide data
4. Abstraction - enable to hide method

Types of Inheritance

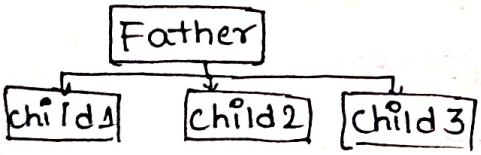
1. Single Inheritance - Single parent and Single child

```
Class parent:
    def property(self):
        print("This is parent property")
father = parent()
father.property()
```

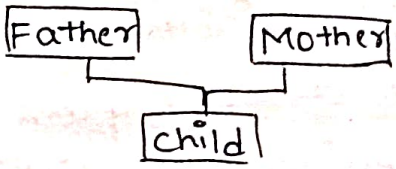
```
class child(parent):
    def property2(self):
        print('child property').
child1 = child()
child1.property()
child1.property2()
```



2. Multiple inheritance.

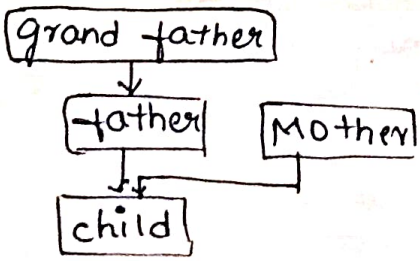


3. Multilevel inheritance.

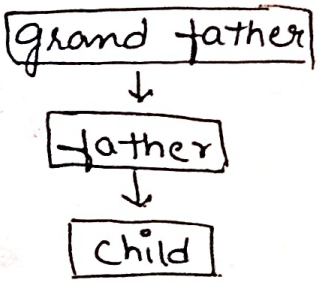


4. hybrid inheritance.

* Combination of single, multiple & multilevel.



5. Hyerarical inheritance.



Why?

* to maintain relation b/w classes.
* to remove redundance/duplicate code
* code reusability
* Organised code

Encapsulation

* encapsulation is binding/combining different variables and method into single class.
* provides better control over data
* prevents accidental modification of data
* promotes modular programming.
* achieves encapsulation through
  - → public
  - → private
  - → protected

Working

1. Data hiding: variables are kept private & protected, they are not accessible directly from outside of the class. They can accessible/modified through methods
2. Access through method: method acts as interface through which external code interacts with data stored in variable
3. control and Security: only allow manupulation via methods.

```
class public:
    def __init__(self):
        self.name = "John" # public attribute → accessible from inside & outside of class
    def display_name(self):
        print(self.name) # public method → accessible from anywhere & print value directly
obj = public()
obj.display_name() # accessible
print(obj.name) # accessible
```

```
class protected:
    def __init__(self):
        self._age = 30 # protected attr → prefix with single '_', so treated as protected. not accessible outside class & subclass
class Subclass:
    def display_age(self):
        print(self._age) # Accessible in subclass

obj = Subclass()
obj.display_age()
```

```
class private:
    def __init__(self):
        self.__salary = 50000 # private attribute → two "__"
    def Salary(self):
        return self.__salary # access through public method →

obj = private()
print(obj.Salary()) # works
print(obj.__Salary()) # error.
```

Abstraction

* Concept of hiding the complex method / unnecessary method to another class and gives access for only required
* we can achieve data abstraction by using abstract classes and they can be created using 'abc' module
* abstract class is where one or more abstract method is defined.

```
from abc import ABC, abstractmethod

from abc import ABC, abstractmethod
                   ↳ class        ↳ method.
```

```
class Test(ABC):
    def method(self):
        pass    → public.

    def method2(self):
        pass    → abstract.
```

Polymorphism

* It lets different classes share same method name.

Overloading

* Same name, diff sign.
* all previous def will be ignored

Overriding

* Same name, same sign.
* previous definition can be used by super()

File Handling

Storage of data

- read
- write
- create

File

- Text file
  - → .txt, .csv, .xlsx, .xls, .json
  - handling
    - → create
    - → open()
    - → write()
    - → read()
    - → append()
    - → close()
  - Note book
    - → purchase
    - → open
    - → write
    - → read
    - → adding
    - → close()
- binary file
  - → .bin

'r' → to read data from file

'w' → to write / add data to file

'x' → It creates file

'a' → add to existing data.

'r+' → read & write.

'w+' → write & read

'a+' → append & read.

- read
  - → read()
  - → readlines()
  - → readline()
- write
  - → write()
  - → writelines()

close() → to make changes & close file