* Python Fundamentals Revision *

Python Revision:-

↳ High-level, interpreted programming language, known for its readability & simplicity.

↳ Guido van Rossum - First release 1991.

↳ Web development, data analysis, AI, ML etc.

Python Features:-

↳ Easy-to-learn

↳ Easy-to-read

↳ Easy-to-maintain

↳ Broad standard library

↳ Interactive Mode

↳ Portable

↳ Extendable

↳ Interpreted

↳ Database connectivity

↳ GUI Programming

↳ Scalable

↳ Dynamically-typed

↳ Both POP & OOP support.

Get Started:-

① First, install Python interpreter.

② Use a text Editor or IDE to write code.

⊙ Identifiers in Python:-

↳ Names used to identify variables, functions, classes, modules & other Python objects.

↳ Essential for naming elements meaningfully.

↳ Case-sensitive.

- **Rules for Naming Identifiers:**

① Must start with letter (a-z, A-Z) or underscore (_).

② Contains letters, digits / underscores.

③ Can't use keywords (e.g. 'if', 'for', 'while').

④ Can't start with digit (0-9)

⑤ No space or special char (except _).

- **Naming conventions for Identifiers:**

① Camel case: 'className', 'functionName'

② Snake case: 'class_name', 'function_name'.

③ Uppercase for constants: 'PI', 'MAX_LENGTH'

④ Descriptive and meaningful names recommended.

⊙ Python Keywords:-

↳ Reserved words that cannot be used as names for identifiers.

↳ They have special meaning.

Python Variables:-

↳ Used to store data values (ex. num, str, objects).

↳ Names should be descriptive & follow naming conventions.

- **Create Variable** - Variable created automatically, when assign a value to it using (=) sign.
  - ↳ Ex: count = 100
- **Print Variable** - Print variable using print() function.
  - ↳ Ex: print(count)
- **Delete a Variable** - can delete reference to object using del statement.
  - ↳ Ex: del count
- **Multiple assignment** - Can assign values to multiple objects in single line.
  - ↳ Ex: a = b = c = 10 # Assign same value
    a, b, c = 10, 20, 10 # Assign multiple values
- **Variables Names** - should follow Naming Rules & conventions.
- **Local Variables** - Defined inside a fun^n, can't access outside the fun^n. Local scope to fun^n only.
- **Global Variables** - Defined outside fun^n, can access anywhere in program. so, have global scope.
  - ↳ can create global variable inside function using global keyword.

Python Constants:- Fixed values, do not change during program's execution.

- Naming convention is to use uppercase letters.
- Doesn't have fixed constants like other languages.

Operators in Python:-

- Used to perform operⁿ on var & values.
  - **OPERATORS** - Special symbols. Ex. +, -, *, /.
  - **OPERANDS** - Value on which operator applied.

1) Arithmetic Operators:

Operator	Name	Description	Example
+	Addition	Add 2 operands	a+b
-	Subtraction	Subtract 2 operand	a-b
*	Multiplicaⁿ	Multiply 2 operand	a*b
/	Division (float)	Divide 1st by 2nd. (Return float)	a/b
//	Division (floor)	(Return int)	a//b
%	Modulus	Return remainder when 1st divided by 2nd.	a%b
**	Power	Return 1st raised to power 2nd.	a**b

2) Comparison Operators:- Compare values & return True or False.

Operator	Description	Example
>	Greater than- True, if LHS greater than RHS.	a>b
<	Less than- True, if LHS less than RHS.	a<b
==	Equal to- True, if both equal.	a==b
!=	Not equal to- True, if both are not equal.	a!=b
>=	Greater than or equal to- LHS greater than equal to RHS.	a>=b
<=	Less than or equal to- LHS less than equal to RHS.	a<=b

3) Logical Operators :- Used to combine conditional statements.

Operator	Description	Example
and	Logical AND: True, if both operands are True.	a and b
or	Logical OR: True, if either operand is True.	a or b
not	Logical NOT: True, if operand is False.	not a

4) Assignment Operators:-

Operator	Description	Example
=	Assign RHS exp value to left.	a = b+c
+=	Add right to left & assign value to left.	a += b a = a + b
-=	Subtract right from left & assign value to left.	a -= b a = a - b
*=	Multiply right with left & assign value to left.	a *= b a = a * b
/=	Divide left by right & assign value to left.	a /= b a = a / b
%=	Mod of left to right & assign to left.	a %= b a = a % b
//=	Divide left by right & assign floor value to left.	a //= b a = a // b
**=	Find left raised to rign & assign to left.	a **= b a = a ** b
&=	Perform Bitwise AND & assign to left.	a &= b a = a & b
|=	Perform Bitwise OR & assign to left.	a |= b a = a | b
^=	Perform Bitwise XOR & assign to left.	a ^= b a = a ^ b
>>=	Bitwise right shift & assign to left.	a >>= b a = a >> b
<<=	Bitwise left shift & assign to left.	a <<= b a = a << b

5) Bitwise Operators:-

Operator	Description	Example
&	Bitwise AND	a&b
|	Bitwise OR	a|b
~	Bitwise NOT	~a
^	Bitwise XOR	a^b
>>	Bitwise right shift	a>>
<<	Bitwise left shift	a<<

6) Identity Operators:- Used to check, if 2 values are located on same part of the memory.

is : True if operands identical.

is not : True if operands not identical.

7) Membership Operators:- check if value / variable present in the sequence

in : True if value found in sequence.

not in : True if value not found in sequence.

8) Ternary Operator :- Testing condition in single line, instead of multiline if-else.

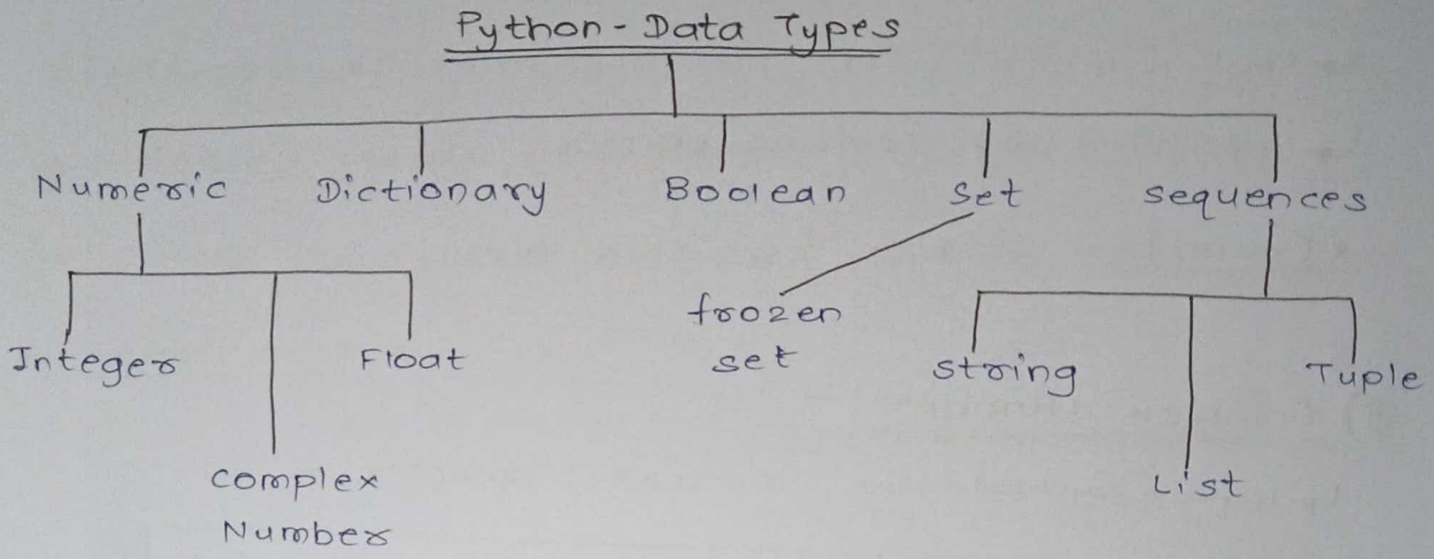
- Syntax- [on-true] if [expression] else [on-false]
- Example-

```
>>> print ('Even' if a%2==0 else 'Odd')
```

Operator Precedence :-

Operator	Description
( )	Parentheses
* *	Exponentiation
+a, -a ~a	Unary plus, unary minus & bitwise NOT.
* / // %	Multiplicaⁿ, Division, Floor Division & Modulus.
+ -	Addiⁿ & Subtracⁿ
<< >>	Bitwise left & right shift.
&	Bitwise AND.
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is is not in not in	comparison, identity & membership operators.
not	Logical NOT
and	Logical AND
or	Logical OR

⊙ Data Types in Python:- Represent kind of value stored in an operator.



- **type() Function:** check data type of a variable.

```
>>> a = 10
>>> type(a)  ⟹  <class 'int'>
```

∴ Data types are classes & variables instances.

- **id() Function:** Used to get memory address of any variable.

- **Numeric Data Types:-**

1) Integers:

↳ Represented by <class 'int'>.

↳ contains +ve / -ve whole numbers (w/o fraction/decimals).

↳ No limit how long int can be.

- **Example-** a = 10, b = -20, c = 0 } type() = <class 'int'>

2) Float :-

↳ Represented by <class 'float'>.

↳ Real-numbers with floating-point representaⁿ.

↳ Specified by decimal point.

• Example - a = 10.5, b = 9.25 etc.

3) Complex Numbers :-

↳ Represented by <class 'complex'>.

↳ Specified as: (real part) + (imaginary) j.

• Example - -2+3j , 2+4j.

• Sequence Data Types :-

↳ Ordered collⁿ of similar / diff. data types.

↳ Allow storing multi values in organized / efficient manners.

1) String Data Type : Immutable.

↳ Collⁿ of 1/more chars put in single, double / triple quotes.

↳ Represented by : <class 'str'>

↳ No separate character data type, char is string of length 1.

• Example - s = 'Snehal'

t = 'a'

2) List Data Type :- Mutable (Add/remove elements)

↳ Just like arrays in other programming lang.

↳ Ordered collecⁿ of same / different data.

↳ Flexible - items not needed of same type.

↳ comma-separated elements inside [].

• Example - a = [1, 2, 3, 4, 5]

b = [1, 2.8, 's', 'abc', True, [1, 2]].

3) Tuple Data Type :- Immutable.

↳ Ordered like list.

↳ Cannot modified after created.

↳ Represented by : <class 'tuple'>.

↳ comma-separated elements with or w/o use of parentheses - ().

• Example - t = (1, 2, 3, 4, 5)

d = 23, 55, 78

• Boolean Data Type :-

↳ One of 2 built-in values : True OR False.

↳ Denoted by <class 'bool'>.

• Example - a = True

- **Set data type in Python :-**

↳ **Set:** Unordered collec^n of data.

↳ Iterable, mutable & has no duplicates.

↳ Order of elements is undefined.

↳ Comma-separated elements in {}.

- **Example-** s = {1, 2, 5, 7, 8, 2, 3, 5}.

t = set ([1, 2, 3, 4])

create set using set() constr.

② **frozen set** - Immutable unordered collection of unique elements.

↳ created using frozenset() constructor.

- **Example-** f = frozenset ({1, 2, 3, 4}).

- **Dictionary Data type :-**

↳ Unorder collection.

↳ Hold values in key: value pair.

↳ No indexing, access by keys only.

↳ Keys must be unique.

↳ Immutable keys (string, int, tuple), No list, set etc

- **Example-** d = {1 : 5, 'name' : 'snehal', 'b' : '250'}.

comma-separated

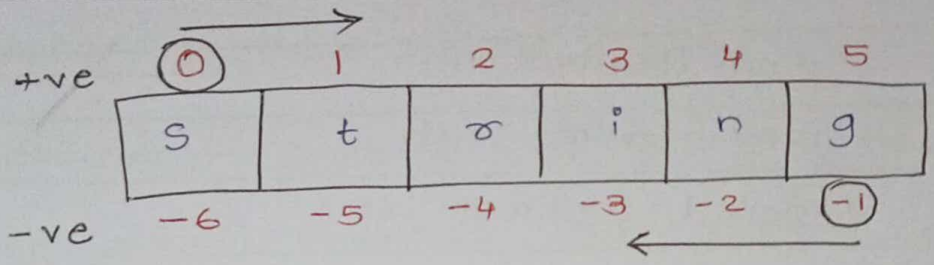
key : value pairs.

1) String Data Type :-

↳ Coll^{n} of char enclosed in single/double/triple quotes.

↳ Represented by - <class 'str'>.

⊙ String Indexing :-



var [start : end : step]

↳ Default set to 1.

- Reverse string: var [::-1]
- String Update: String Immutable, cannot update.
- Special String Operators:

Operator	Description
+	Concatenation - Add/Join strings.
*	Repetition - Add multiple str copies.
[]	Slice - Find char at index.
[:]	Range Slice - Slice bet^{n} range.
in	Membership - True, if char present.
not in	Membership - True, if not present.
r/R	Raw String - Raw form of string. Suppress escape char meaning.
%	Format - Perform string formatting.

- **String Formatting:-**

```
>>> s = 'I am %s and age %d' % ('Snehal', 22)
```

Symbol	Description
%c	Character
%s	string
%i	Signed decimal int.
%d	Signed decimal int.
%u	Unsigned decimal int
%o	Octal int.
%x	Hexadecimal int (Lowercase letter)
%X	Hexadecimal int (Uppercase letter)
%e	Exponential Nota^n (lowercase 'e')
%E	Exponential Nota^n (Upper case 'E')
%f	Floating point real-numbers.
%G	Shorter of %f & %E.
%g	Shorter of %f & %e.

- **Character case conversion Functions -**

1. capitalize() - Capitalize 1st letter of string.
2. title() - Each word begin with uppercase.
3. lower() - Convert all chars to lowercase.
4. upper() - Convert all chars to uppercase.
5. swapcase() - Invert cases of all chars.
   Convert lower → upper & upper → lower.

- Check for Special char Functions :-

① isalpha - True, if all char letters, not even space.

② istitle() - True if str 'title-cased'.

③ isupper() - True, if all letter uppercase.

④ islower() - True, if all chars lowercase.

⑤ isspace() - True, if all chars empty space.

⑥ isalnum() - combina" of alpha & nums.

⑦ isdecimal() - support only decimal numbers.

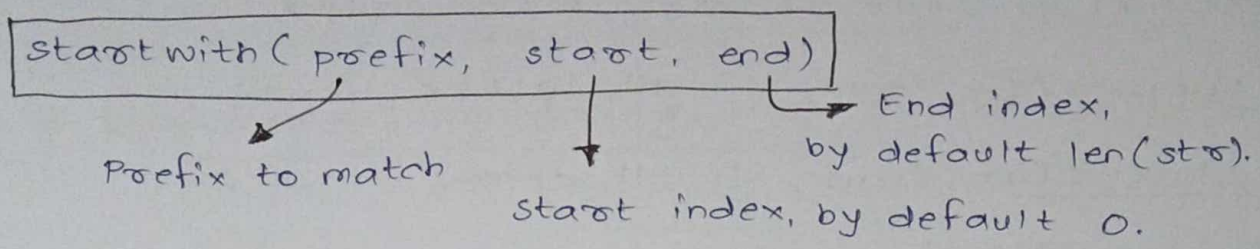
⑧ isdigit() - Support decimal, subscript & superscript

⑨ isnumeric() - Support digits, vulgar functions, subscripts, superscripts, Roman numerals, currency numerators.

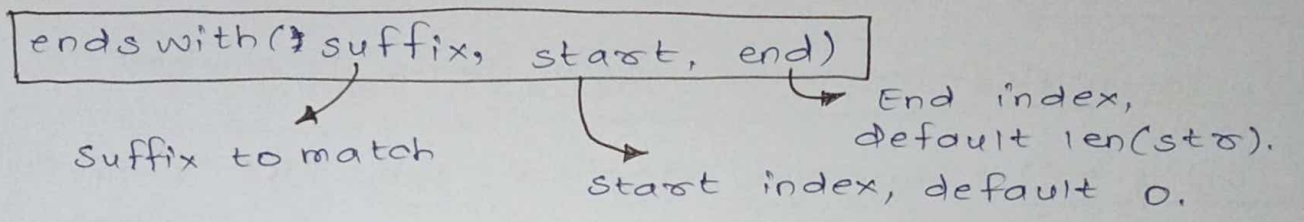
string Type	Example	isdecimal()	isdigit()	isnumeric()
Base 10 numbers	'0123'	True	. True	True
Fraction & Subscripts	'2/3' '2^2'	False	True	True
Roman numerals	'L', 'J', 'v', 'xJ'.	False	False	True
Others	'a', 'bc'	False	False	False

- **Check start & end of string Functions:-**

① **startswith()** - True, if start with given prefix.



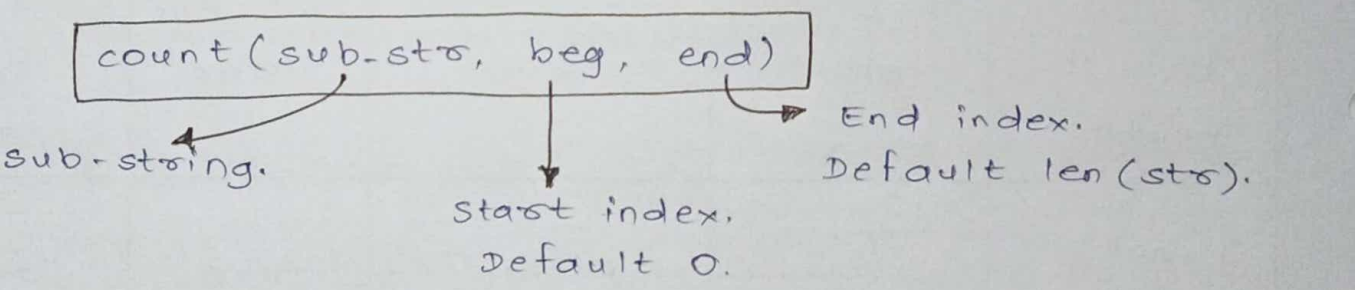
② **endswith()** - True, if ends with given suffix.



- ~~Finding~~ **Counting Functions:-**

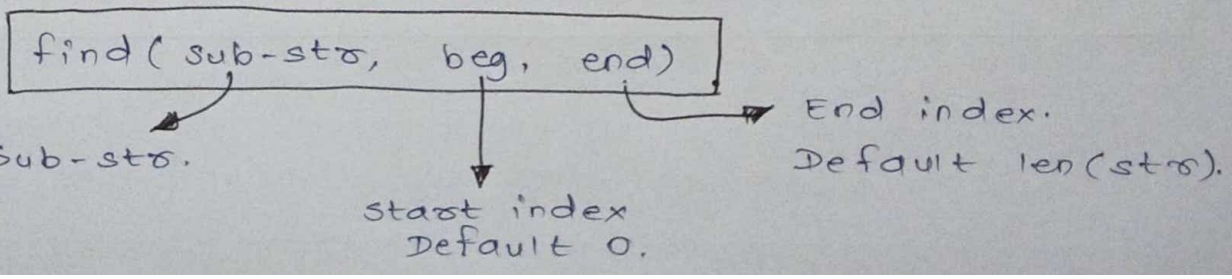
① **len()** - Return length of string.
↳ count empty spaces as well.

② **count()** - Find count of all occurrences of given sub-str.



- **Finding char occurrences Functions:-**

① **find()** - Return index if sub-str found, & -1 otherwise.



② rfind() - search backward from right.

```
rfind(sub-str, beg, end)
```

③ index() - Find index, raise error if not found.

↳ ValueError: substring not found

```
index (sub-str, beg, end)
```

④ rindex() - search from right, raise ValueError if not found.

```
rindex (sub_str, beg, end)
```

⑤ replace() - Replace all old-substr occurrences with new_substr.

```
replace (old, new, count)
```

- old → Old sub-string
- new → New sub-str
- count → Max no. of counts to replace, starting from left.

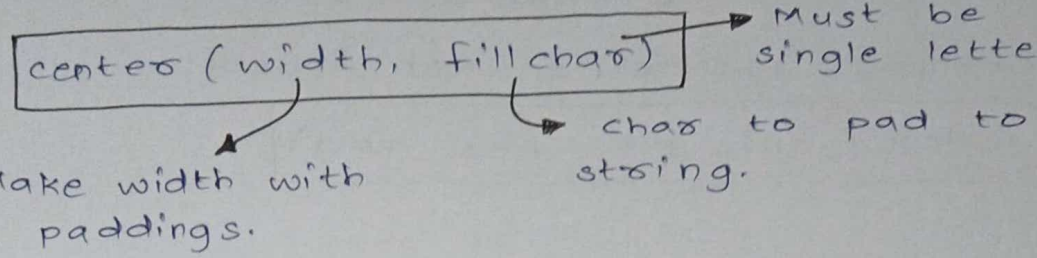
- **Find max/min from the string:-**

① max(s) - Find char with max ASCII value.

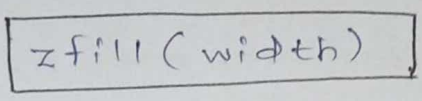
② min(s) - Find char with min ASCII value.

- **Add / Remove char on both ends :-**

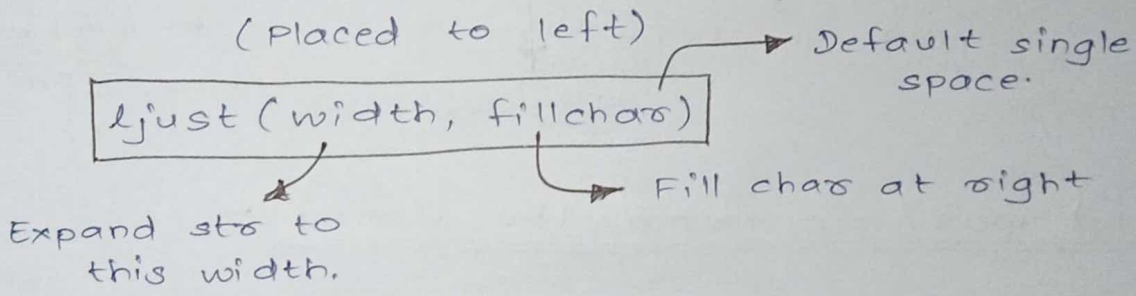
① **center()** - Return space-padded string with original string placed at center of width.



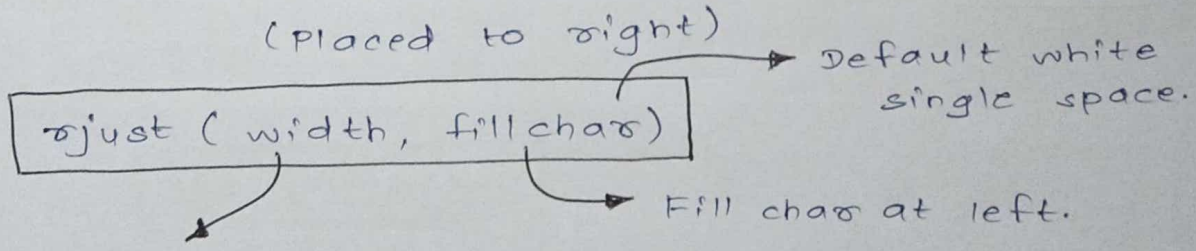
② **zfill()** - Place 0's at start to make the str to total given width.



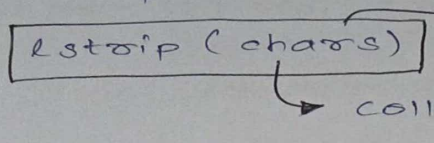
③ **ljust()** - Return space/char-padded string with original string left-justified to total width.



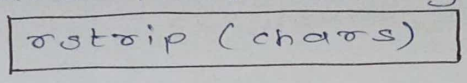
④ **rjust()** - Return space/char-padded string with original string right-justified to total width.



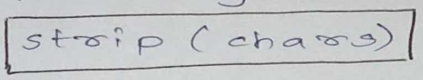
⑤ lstrip() - Remove all leading (from left side of string) char / spaces from string.



⑥ rstrip() - Remove all trailing (on right side) space / chars from string.

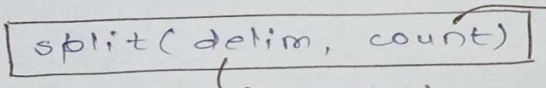


⑦ strip() - Remove all leading & trailing spaces/ chars from string.



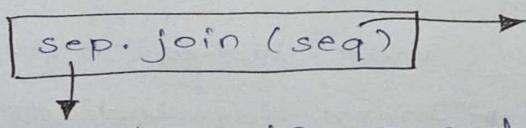
- Split & Join Functions :-

① split() - Split string with delimiter given.



② splitlines() - Splite string by NEWLINE ('\n').

③ join() - Merge (concat) strings from given sequence into string, by appending separator in-bet°.



2) Lists Data Type :-

- Comma-separated, enclosed in [].
- Mutable (changeable).
- Allow duplicates.
- Iterable
- Diff data types - int, float, tuple, list.
- Support indexing (start from 0).

- **List Declaration :-**

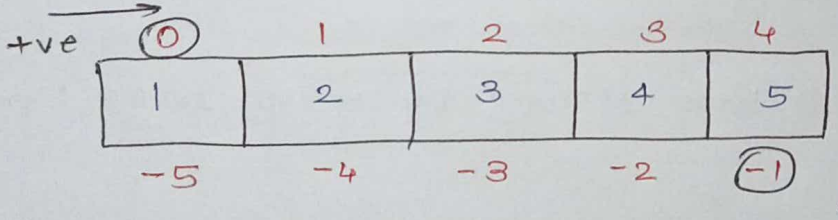
Declare in one line

```
Time complexity: O(1)
Space complexity: O(n)
```

→ Number of elements.

- **List Accessing :-**

① **Indexing & slicing :**




```
lst [start : end : step]
```

→ Default 1.

② **Multi-dimensional List** : List inside list.

```
lst [ • ] [ • ]
```

→ Inner list indices
→ Outer list indices.

```
Time complexity:  O(1)
Space complexity: O(1)
```

- **Taking List Input :-**

① **Using loop** - Take list size prior.

```
>>> n = int(input()) # size
>>> s = [] # Declare empty list
>>> for i in range(n):
>>>     s.append(input())
```
} Append n no. of i/p.

② **Single line input** - Split the input by space.

```
>>> s = input() # Take space separated list i/p.
>>> l = s.split() # split by space, by default.
```

③ **Take single line int list input** - convert each split element to int using list compre.

```
>>> s = input() # Space separated list i/p.
>>> l = [int(i) for i in s.split()]
```
↳ convert each to list.

- **Operations of List :**

① **len()** - Used to get length of list.

```
len(lst)
```

② **index()** - Used to find index of item in list.

↳ Raise ValueError, if not present.

```
l.index(item, start, end)
```
- item: Value to find index in str.
- start: Index to start searching. (Default (0).
- end: Index to stop searching (Default len(lst)).

- **Add Elements Functions:-**

① **append()** - Add item at end of list.
(-1 / len(l) index).

Space: O(1)
Time: O(1)

② **extend()** - Items from sequence added to list individually. By default, at end.

Space: O(1)
Time: O(n)

③ **insert()** - Insert value at particular index.

```
insert (index, value)
```

Index to insert value at.

Value: int, float, list etc.

Space: O(n)
Time: O(1)

- **Delete Elements Functions:-**

① **remove()** - Remove 1st occur. of element.

```
l.remove(item)
```

Item to remove from list.

Space: O(n)
Time: O(1)

② **pop()** - By default remove last, but can give index.

```
l.pop()          # Remove last
l.pop(index)     # Remove from index.
```

Space: O(1)
Time: O(1) Last
O(n) other

③ **delete** - Delete slice of list.
↳ Free memory allocated to var.

```
del l[:]     # Delete slice
del l        # Free memory
```

Space: O(n)
Time: O(1)

④ clear() - Make list empty, delete all items.

```
l.clear()   →   []
```

- **List sorting functions:-**

① sort() - Sort original list in-place.

```
s.sort()                          # Ascending
s.sort(reverse = True)            # Descending
```

② sorted() - Return sorted list, original not modified.

```
sorted(l)                         # Ascending
sorted(l, reverse = True)         # Descending
```

- **List Reversing Functions:-**

① slicing - Slice whole list with -1 step.

```
l[::-1]
```

② reverse() - Reverse actual list, in place.

```
l.reverse()
```

③ reversed() - Return reversed list, original list is not modified.

```
reversed(l)
```

- **Create list with Item Repetition:-**

① * Operator: create list with smaller list elements.

```
>>> l = [5] * 3   ⇒   [5, 5, 5]
```

Repeat element to form a list.

- **Membership Operators:-**

① **'in' Operator-** True, if item present & False otherwise.

② **'not in' Operator-** True, if item not present & False otherwise.

- **Create copy of List:-**

① **copy()-**

```
>>> c == d      # Both var point to same list.
                Modifica^n in 1, reflected to other.
```

```
>>> c = d.copy()   # Return copy of list.
                   Independent for modifica^n.
```

- **count occurs. of item:-**

① **count()-** Return count of given element in the list.

```
l.count(item)
```

- **Statistic Functions:-**

① **sum()-** Find sum of elements in ~~list~~ numeric list only.

```
sum(l)
```

② **min()-** Find min value from list (number list only)

```
min(l)
```

③ **max()-** Find max from list (number list only).

```
max(l)
```

- **Other list Functions:-**

① **any()** – Return True, if any 1 of list item 'True/1'.

↳ Return False, if all False / empty list.

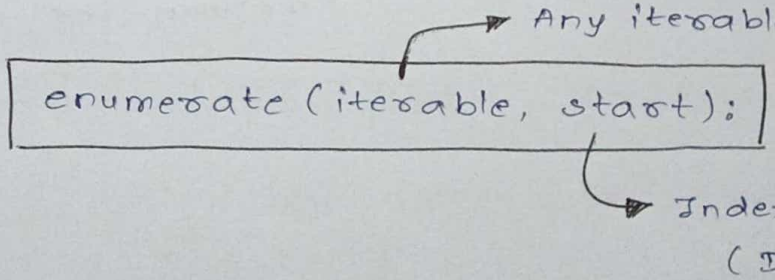
↳ **Short-circuit the execu^n**: Stop execu^n as soon as result-known. i.e. Return True, as 1st True reached.

② **all()** – Return True, if all True / list empty.

↳ Also **short-ckt execu^n** (stop as soon as result known, i.e. 1st False / 0 reached ⇒ Return False).

	any()	all()
All Truthy Values	True	True
All Falsy values	False	False
1 Truthy (others Falsy)	True	False
1 Falsy (others Truthy)	True	False
Empty iterable	False	True

③ **enumerate()** – Return list of tuples containing – [(counter, list_item), ...]



- **List Comprehension :-** Generate list from range() / other list in single line.

① **Create list using range() fun^n -**

```
[i for i in range(num)]
```

② **range() with fun^n on element -**

ex: `[i**2 for i in range(1,6)]`

```
[exp for i in iterable]
```

Apply exp on each element i in iterable.

③ **List compre with condition -**

```
[exp for i in iterable if condition]
```

Apply exp on each element i in iterable.

Condi^n to be executed.
↳ If true, then only return the element.

3) Tuple Data Type :-

↳ comma-separated items, enclosed in ().

↳ Ordered & Immutable (Unchangable).

↳ Allow duplicates.

↳ Support indexing (start from 0).

↳ Represented by - <class 'tuple'>

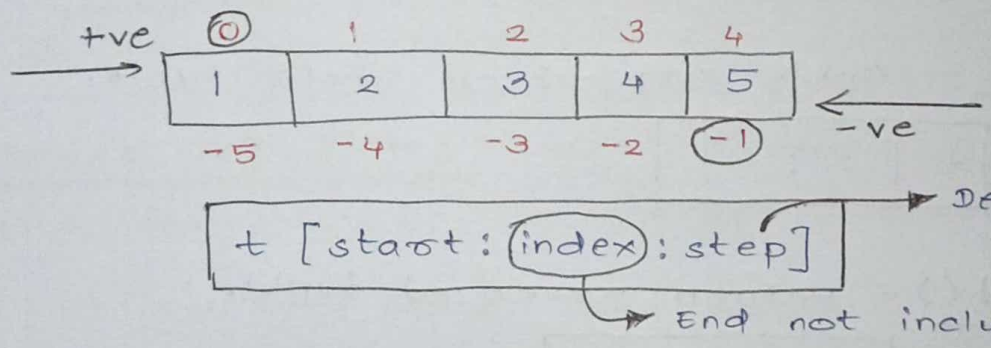
- **Operations on Tuple:-** Since immutable, cannot perform add, update, remove elements.

① len (t) - Return length of tuple.

② type (t) - <class 'tuple'>

- **Access tuple Elements :-**

① Indexing & slicing -



- **Statistical Operations :-** Operate on number tuples only.

① min() - Find min value from tuple.

```
min(t)
```

② max() - Find max value from tuple.

```
max(t)
```

③ sum() - Return sum of values in tuple.

```
sum(t)
```

- **Find count & index :-**

① **count()** - Find no. of occurs. of element in tuple.

```
t.count(item)
```

② **index()** - Return index, if found & raise ValueError otherwise.

```
t.index(value, start, end)
```

Default len(t)

Default 0.

Value to search

- **Membership Operators :-**

① **in** - Return True, if item present.

② **not in** - Return True, if item not present.

- **Tuple Reverse Operations :-**

① **-ve step** - Apply step -1 on whole tuple.

```
t[::-1]
```

② **reversed()** - Return reversed tuple.

```
reversed(t)
```

- **Tuple sorting Operations :-**

① **sorted()** - Return sorted tuple.

```
sorted(t)
```

- **Delete the tuple :-**

① **del** - Use del keyword to free tuple memory.

```
del t
```

⊙ Set Data Type :-

- **Set:** Unordered, comma-separated, enclosed in { }.

↳ No duplicates allowed.

↳ Hashable (Unordered), No indexing follows.

↳ Initialize - set() constr.

↳ Items unchangable (cannot update), but can remove & add items.

↳ Unordered - print any order (unique only).

- **Note:** 0 & False treated as same.
  1 & True treated as same.

- set cannot contain list [], dict {:}, set {0}, but can contain tuples (,,,,).

  TypeError: unhashable type: 'set'

- **Access Set Items :-** Not support keys / indexing.

↳ Use for loop to access set items.

```
>>> s = {1, 2, 5, 's', True, False}
>>> for i in s:
>>>     print(i)
```

O/p → Print items in any order.

- **Operations on sets :-**

- **Add set items Operations :-**

① **add()** - One item at a time. No index passed.

```
s.add (item)
```

② **update()** - Multiple items at a time.
↳ If pass one item, enclose in set / bracket.
↳ can add any iterable: set, list, tuple etc.

```
s.add ({iterable})
```

③ **union()** - can combine any iterable (set, list, tuple) into set.
↳ Return new set, original not modified.
↳ can provide multiple args.

```
s.union (seq1, seq2, ....)
```

- **Removing items from the set :-**

① **remove()** - Remove 1 item at a time.
↳ KeyError : If item not found.

```
s.remove (item)
```

↳ Remove all occurrences, bcz treated as single only.

② **pop()** - Remove 1 item, bcz unordered, remove any items randomly, can't predict which.

```
s.pop ()
```

③ **discard()** - Remove given 1 item.
↳ No error, if not present.

```
s.discard (item)
```

④ clear() - Remove all items & make set empty.

```
s.clear()
```
⇒ set({})

⑤ del - Delete set & also free up memory.

```
del s
```

- **Mathematical Operations on Sets :-**

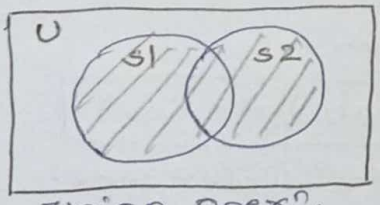
① union - combine 2 or more sets into one.
↳ Return merged set, original not modified.
↳ Can give multiple args.

#1: Using | Operator -

```
u1 | u2 | u3 | u4
```
→ Give union of sets.

#2: union() Fun^n -

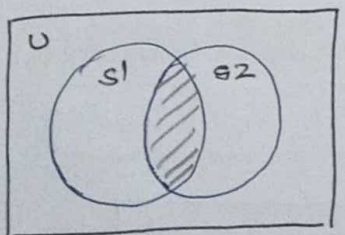
```
s1.union(s2, s3, s4)
```
→ can give any iterable - list, tuple.



Union oper^n.

x : x ∈ s1 OR
x ∈ s2 OR
x ∈ s1 & s2
(All from s1 & s2).

② Intersection :- Find common items from sets.
↳ Return interse^n set, but original not modify.



Intersection
x : x ∈ s1 & s2
(commons from s1 & s2)

#1: Using & operator -

```
s1 & s2 & s3
```

#2: Using intersection() -

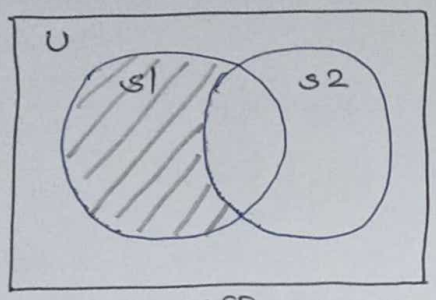
```
s1.intersection(s2, s3)
```
↳ Can give multi args of any sequence type.

⑨ intersection_update() – Find intersec^ & unpate the original set in-place.

```
s1.intersection_update(s2,s3)
```

→ Update s1 with intersection set.

④ Difference of sets – Find diff. of 2 sets. i.e. Items of s1, not present in s2.



Set Difference

$x : x \in s1 \;\&\; x \notin s2$

(x in s1, but not in s2)

Using '-' Operator –

```
s1-s2
```

#2: difference() –

```
s1.difference(s2)
```

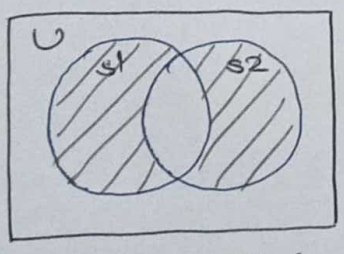
$(s1 \cup s2) - [(s1 \cap s2) \cup s2]$

⑤ difference_update() – Find difference & update it to original set in-place.

```
s1.difference_update(s2)
```

→ Update s1 to difference.

⑥ Symmetric Difference – Uncommon from both sets.



$x : x \in s1 \;\&\; x \in s2 \;\&\; x \notin s1 \,\&\, s2$

(x in s1, x in s2, but not in $s1 \cap s2$)

Using ^ Operator –

```
s1 ^ s2
```

#2: sym_diff() –

```
s1.symmetric_difference(s2)
```

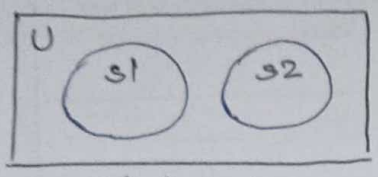
$sym\_diff = (s1 \cup s2) - (s1 \cap s2)$

⑦ symmetric_difference_update() – sym-diff reflected back to the original set, in-place.

```
s1.sym_diff_update(s2)
```

- Other Functions :–

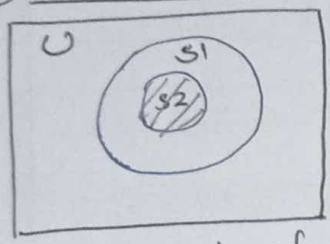
① isdisjoint() – Find if, no common in both sets.



Disjoint

```
s1.isdisjoint(s2)
```
↳ Return True, if no common bet^n s1 & s2.

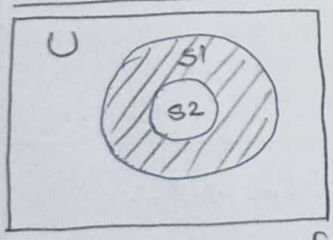
② issubset() – Find if, subset of another set.



s2 subset of s1

```
s2.issubset(s1)
```
↳ Return True, if s2 subset of s1.

③ issuperset() – Find if, set is superset of another.



s1 superset of s2

```
s1.issuperset(s2)
```
↳ True → s1 superset of s2.

- Aggregation Functions :– consider unique items only.

① len() – Find length of items.

② sum() – Find sum of items.

③ max() – Find max from items.

④ min() – Find min from items.

- **Special Operators on sets:-**

Operator	Description
item in s	check membership.
item not in s	check membership.
s1 == s2	s1 equivalent to s2.
s1 != s2	s1 not equivalent to s2.
s1 <= s2	s1 subset of s2.
s1 < s2	s1 proper subset of s2.
s1 >= s2	s1 superset of s2.
s1 > s2	s1 proper superset of s2.
s1 | s2	Union of s1 & s2.
s1 & s2	Intersection of s1 & s2.
s1 - s2	Items in s1, not in s2.
s1 ^ s2	Items in s1 & s2, but not in both. (Union - Intersection)

- Frozenset :-

  - Unordered.
  - Duplicates not allowed.
  - Immutable (No update operations).
  - Same as set, but immutable.

- Operations on Frozenset :-

  1. copy() - create & return copy of set.
  2. diff() - Find difference of 2 sets.
  3. intersection() - Intersec^n of 2 sets.
  4. isdisjoint() - check if sets disjoint.
  5. issubset() - check if, set subset of another.
  6. issuperset() - check if set superset of another.
  7. sym_diff() - Find symmetric difference of sets.
  8. union() - Find union of sets.

⊙ Dictionary Data Type :-

- key-value pairs stored in {key: value, ..}.
- Not support indexing, use keys.
- Unordered before 3.7, now ordered.
- Keys cannot duplicated.
- Mutable.
- Keys: Immutable python objects - int, float, string, tuple.
- values: Mutable / Immutable.

• Create Dictionary :-

① Empty Dictionary - {} is empty dict, not set.

```
>>> type({})   ⇒ <class 'dict'>
```

② Create using {} -

```
>>> a = [1, 2, 3]
>>> b = {2, 3, 5}
>>> d = {'a': a, 'b': b}
```

③ Using dict() Fun^n -

```
>>> d = dict([(1, 25), (2, 23)])
```

• Properties of Keys :-

① Unique - No 2 keys with same name, but diff values.

② Immutable - Use str, nums / tuple, bcz their value cannot be changed.
- Cannot use - list, dict, set etc. mutable.

- **Access Dictionary Items :-**

① **Get value by keys** - Return **KeyError : 'key'**, if not present.

```
d [key]
```

② **Using get() Function** - Return value of the given key.

↳ No error, if not present, return **None**.

```
d.get (key)
```

③ **items()** - Return **dict_items()** data type.

Return tuples like : [(key, value)]

```
d.items ()
```

O/P → dict_items ([(key, val), (key, val), ...])

- **Loop through Dictionary :-**

① **d.keys()** - Return all keys from dict.

```
>>> for i in d.keys():
>>>     print (i, d[i])
```

② **d.items()** - (key, value) tuples.

```
>>> for k, v in d.items():
>>>     print (k, v)
```

③ **d.values()** - Print values only.

```
>>> for i in d.values ():
>>>     print (i)
```

④ **Print in keys sorted orded** -

```
>>> for i in sorted (d.keys()):
>>>     print (i, d[i])
```

- **Changing item in the Dictionary:-**

① **Change value** - Re-assign new value to key.

```
d[key] = value
```

↳ This overwrites prev value of key.

② **change key** - Cannot change key, so assign prev key value to new key & del prev key.

```
d[new] = d[prev]
del d[prev]
```

- **update() Function:-** Merge 2 dicts.

↳ Assigning add 1 item at a time.

↳ update() - Add multiple items in dict.

```
d.update(d1)
```

d1 keys:val added to d, in-place.

- **setdefault() Function:-**

```
dict.setdefault(key, value)
```

↳ **key present** - Return prev value of key.

↳ **No key & pass (key)** - Create key & assign None.

↳ **No key & pass (k, v)** - Create key & assign value.

- **Remove from Dictionary :-**

① **pop()** - Delete one key:value, whose key passed.

```
d.pop(key)
```

② **del** - Used to delete 1 key (free its space).
↳ Free complete memory by deleting dict.

```
del d[key]
del d
```

③ **clear()** - Deletes all items & make dict empty.

```
d.clear()
```
⇒ {}

- **fromkeys() Function :-** create dict from provided keys & values sequences.
↳ If only keys given, all assign to **None**.

```
dict.fromkeys(keys, vals)
```
Default None.

keys → sequence for dict keys.

vals → It is 1 value assigned to all keys for initializn.

- **zip() Function :-** combine 2 seq in zip object, then converted dictionary.

```
dict(zip(keys, vals))
```
vals → seq of values

keys → Sequence of keys.

• Sorting in Dictionary :-

① Print dict in keys sorted order -

```
>>> for i in sorted(d.keys()):
>>>     print(i, d[i])
```

② Print only keys in sorted order -

```
>>> sorted(d.keys())
```

③ Print only values in sorted order -

```
>>> sorted(d.values())
```

④ Print dict_items in keys sorted orded -

```
>>> print(sorted(d.items()))
```

↳ Keys sorted in Ascending order.

⑤ Print sorted keys -

```
>>> sorted(d)
```

⑥ Sort by values -

```
>>> new = dict(zip(d.values(), d.keys()))
>>> b = dict(sorted(new.items()))
>>> sort = dict(zip(b.values(), b.keys()))
# Thus, dict is sorted by values.
```

⑦ Easy sorting -

sort by key

```
dict(sorted(d.items(), key = lambda item : item[0]))
```

```
diat(sorted(d.items(), key = lambda item : item[1]))
```

sort by value.

⊙ Dictionary Comprehension :-

↳ Create dict from another dict / sequences in single line.

↳ Syntax :

can also give condition after iterable.

```
{key: value   for (key, value) in iterable}
```

List of tuples, like [(key, val), (key, val), ...]

- Example -

```
>>> {key: value   for (key, value) in zip(keys, vals)}
```

- Example with condition -

```
{key: val for (key, val) in zip(keys, vals)
                    if type(key) == int}
```

condition.

↳ select keys of type int only.

① Dict from list :-

```
>>> {i: i**2   for i in l}
```

② range() & condition :-

```
>>> {i: i**2   for i in range(10)
                    if i%2 == 0}
```

condition for even numbers.

- **Create Nested Dictionary:-**

① **Dict with List values:-**

generate list by list compr.

```
>>> d = {i: [i*d for d in range (1, 11)]
        for i in range (11, 21)}
```

O/P → `{11: [11, 22, 33, 44, 55, 66, 77, 88, 99, 110],`

`:`

`}`

② **Dict with Dict values:-**

dictionary comprehension

```
>>> d = {i: {d: i*d for d in range (1, 11)}
        for i in range (11, 21)}
```

O/P → `{11: {1: 11, 2: 22, ..... , 10: 110},`

`:`

`}`

- **popitem () Method:-** Remove most recent key:value pair.

↳ i.e. remove the last one.

```
d.popitem ()
```

⊙ Control statements :- combinar of decision making, looping statements & looping control statements.

1) Control Flow / Decision Making statements :-

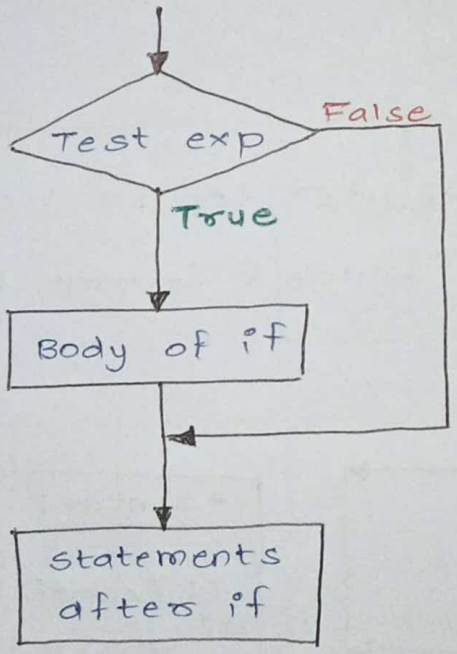
↳ Decide dirn (control flow) of flow of program execun.

1. if
2. if-else
3. if-elif-else
4. nested-if

↳ Enable to make choices & execute the block, depending on condition True / False.

- ① **if statement** – Most simple decision-making strnt.

↳ Decide whether block executed or not.



Flowchart of 'if'

• Syntax :

```
if condition:
    if body
Body after if
```

Test exp to be evaluated.

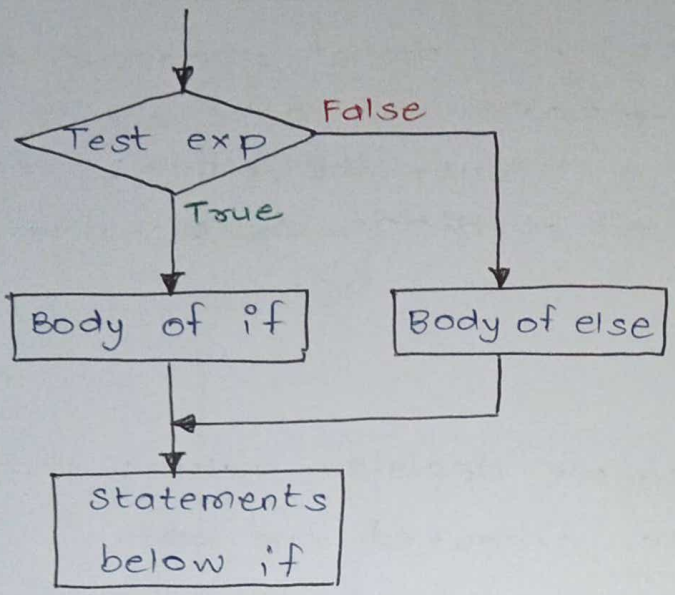
Execute if condin True

↳ Body of 'if' block executed, only when condin evaluated True.

② if-else Statements :-

↳ If condⁿ true ⇒ Execute 'if' block.

↳ condⁿ False ⇒ Execute 'else' block.



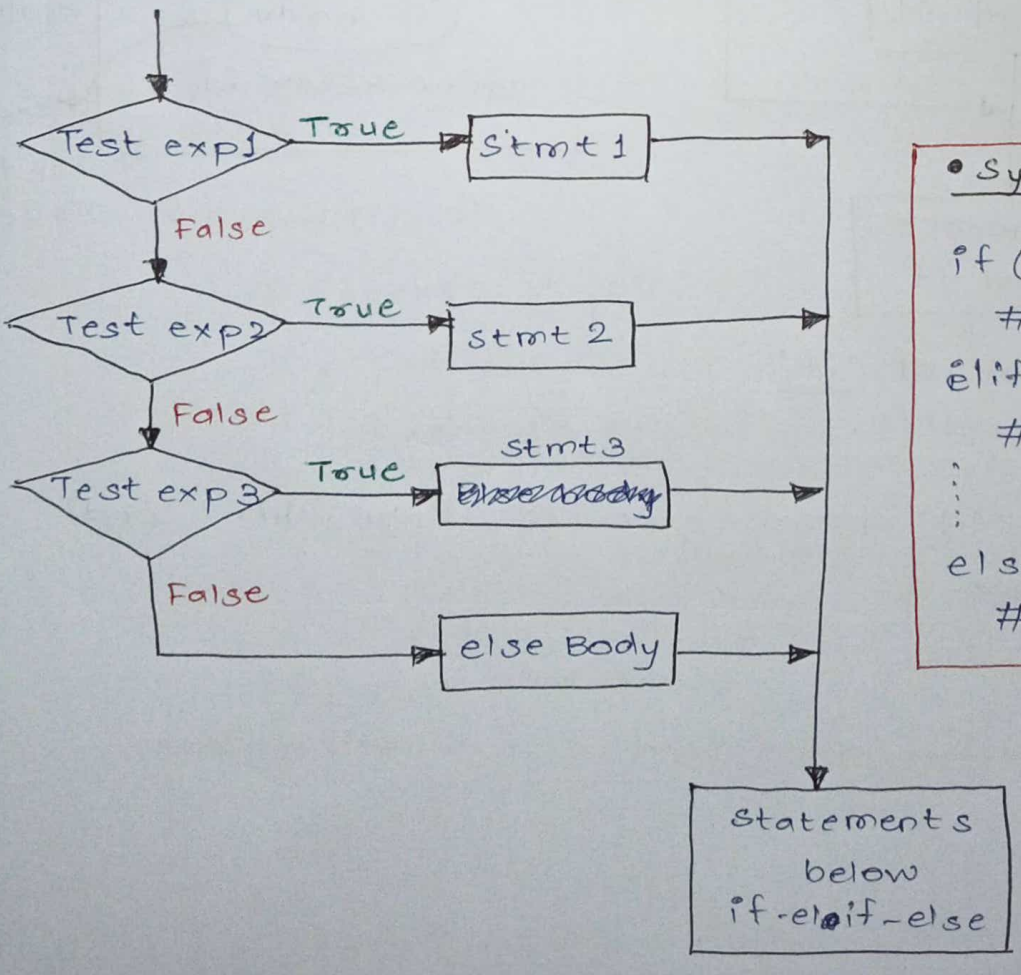
• Syntax –

```
if (condition):
    # condition True
    # Execute if
else:
    # condition False
    # Execute else
```

③ if-elif-else Statements :-

↳ Ladders of conditions, execute top to bottom.

↳ condition True ⇒ Execute Block & ignore below.

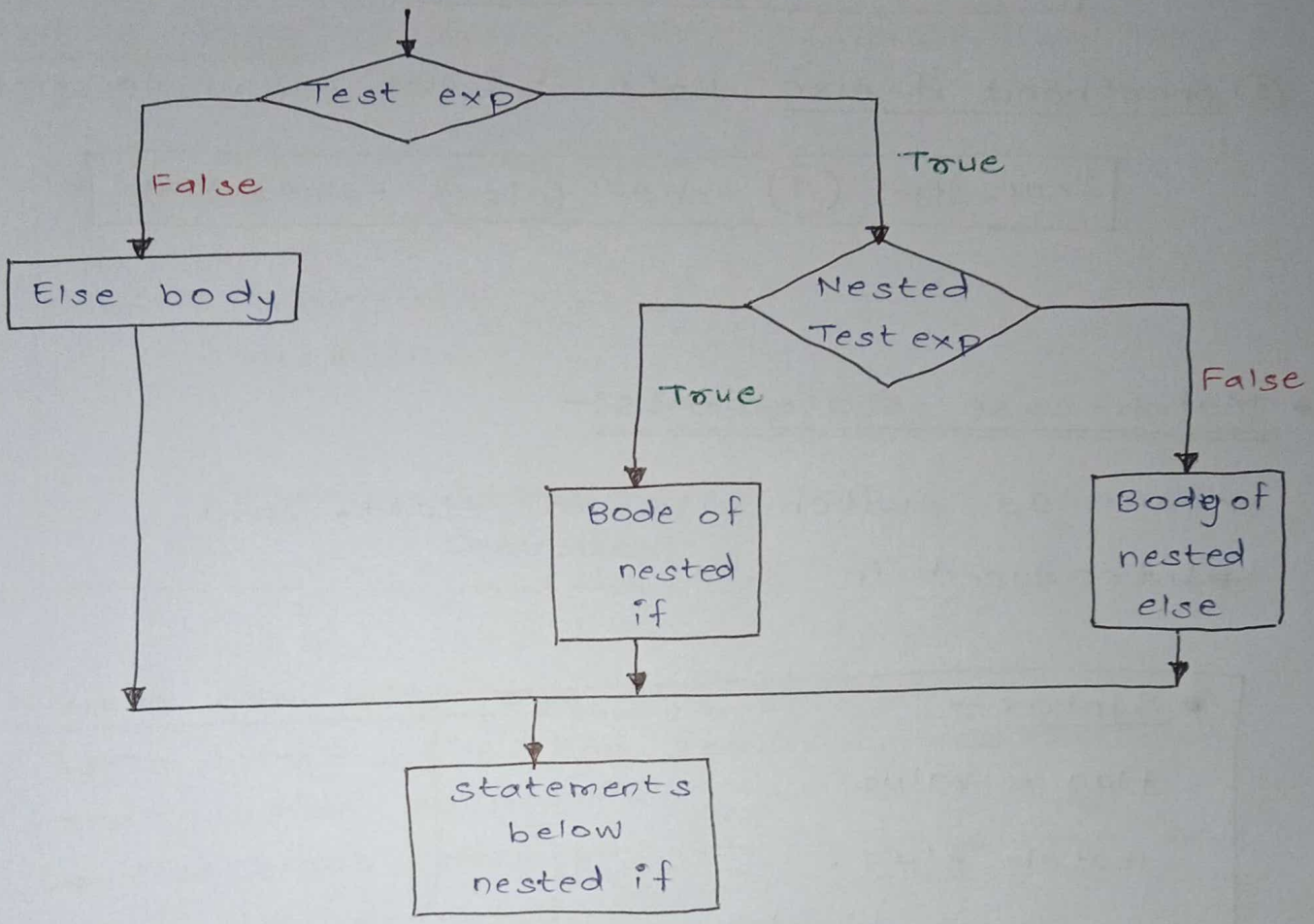


• Syntax :

```
if (condi1):
    # stmt 1
elif(condi2):
    # stmt 2
    .
    .
    .
else:
    #else Body
```

④ Nested-if statements:-

↳ Nest the 'if' stmts within another 'if' stmts, i.e., place 'if' inside another 'if'.



- Syntax:

```
if (condition 1):
    # Execute if block
    if (condition 2):
        # Execute nested if Block
    else:
        # Execute nested else Block
else:
    # Execute else Block
```

- **Shorthand statements:-**

① **Shorthand if** - If 'if' has only 1 statement.

```
if condition: statement
```

② **Shorthand if-else**: Both if & else has single stmts.

```
True_stmt if condi^n else False_stmt
```

- **Match-case statements:-**

↳ Same as switch-case in c/c++, Java.

↳ Introduced in 3.10.

- **Syntax -**

```
flag = value

match flag:
    case first:
        'first' block
    :
    :
    Case n:
        'n' block
    case _:          → Execute, when no other case matched.
        Default Block
```

⊙ **Looping statements :-** Used, when want to repeat a block of code, by giving some condⁿ to meet for stopping loop.

① **While loop :-** Repeat block of code, as long as condⁿ evaluated to be True.

- **Syntax -**

```
Initialization
while condition:
    statements
    :
    :
    Updation - Increment/
               Decrement
```

- **while loop with else and break :-**
  - ↳ To break while loop forcefully, we can give break inside condition.
  - ↳ else block executed, only if while loop ~~not~~ break not executed.

- **Syntax -**

```
Initialization
while expression:
    statements
    if condition:
        break
    :
    Updation - Increment/
               Decrement
else:
    else block execution
```

If break executed, else does not execute.

Execute only if, loop not exit with break- statement

② for loop: Execute for limited num of times.

- **for loop with range() –**

```
for i in range(num):
    print(i)
```

- **Sequence Iterations –**

```
>>> for i in lst:
>>>     print(i)
```
} Print items one-by-one.

```
>>> for i in range(len(lst)):
>>>     print(lst[i])
```
} Print items with help of indices

```
>>> for i,e in enumerate(lst):
>>>     print(i, e)
```
} enumerate() return index along with item.

- **Dictionary Iterations –**

```
>>> for i in d.keys():
>>>     print(i)
```
} Print keys only

```
>>> for v in d.values():
>>>     print(v)
```
} Print values only.

```
>>> for t in d.items():
>>>     print(t)
```
} Print (key, val) tuples.

```
>>> for k in d.keys():
>>>     print(k, d[k])
```
} Print values by keys.

Looping control statements:-

↳ Change / control execution of the block.

① **break**: Bring control out of the loop.

↳ Breaks execun of loop & resume program execun below loop.

- Syntax:

```
for i in range(num):
    if condition:
        break
    statements
```

→ break & exit loop. Stop further loop execution.

② **continue**: Bring control back to start of loop.

↳ current execution of loop skipped & resume further loop execution.

- Syntax:

```
for i in range(num):
    if condition:
        continue
    statements
```

→ skip loop execun, when condin True

③ **pass**: can be written in empty loops / funcn.

↳ Used in empty control stmts, funn & classes.

↳ Thus, not give error for empty block.

- Syntax:

```
for exp:
    pass

def func():
    pass
```

→ skip the block.

List Comprehension:-

↳ offer shorter syntax, to create new list based on existing list / range() function.

- **Syntax -**

```
new_list = [ exp for item in iterable
                          if condition ]
```

↳ List compre consist of brackets - [] containing the exp, executed with loop to iterate over each element in iterable.

- **Benefits of List Compre -**

① More time & space efficient than loops.

② Require few lines of code.

③ Transform a block into single line.

- **List compre example with If-else:**

```
>>> lst = ['Even' if i%2==0 else 'Odd'
           for i in range(5)]
```