Cheat Sheets of Revision Tour 1 and 2

PYTHON CHEAT SHEET

Quick Reference Guide with Syntax & Examples

1 DATA TYPES IN PYTHON

Data types are used to identify the type of data and set of valid operations which can be performed on it.

1. Numbers — Used to store numeric values.

Type	Description	Syntax	Example
Integer (int)	Whole numbers, positive or negative	`x = 10`	`x = -123, 1234`
Boolean (bool)	True or False (1 or 0)	`b = True`	`True, False`
Float (float)	Decimal numbers (15 digit precision)	`f = 12.345`	`12.345, 0.17`
Complex (complex)	Numbers in the form A + Bj (j = √-1)	`z = 5 + 4j`	`5+4j, 2-2j`

2. String (str) — Sequence of Unicode characters.

Syntax	Example
`s = 'Hello'`	`'Hello'`
`s = "World"`	`"World"`
`s = '''Multi Line'''`	`'''This is multi-line string'''`
`s = "Line1\ Line2"`	`"Line1 Line2"  # Using \`

- Indexing starts from 0.
- Supports slicing, concatenation, repetition etc.

Example

```
s = "Python"
s[0]    # 'P'
s[-1]   # 'n'
s[1:4]  # 'yth'
```

3. List (list) — Ordered, mutable collection of items.

Syntax

```
lst = [1, 2, 3, 4]
lst = ['a', 101, 90.5]
lst = [1, 'a', 3.5, True]
```

- Indexing starts from 0.
- Items can be of mixed data types.

Example

```
lst = [10, 20, 30, 40]
lst[0]    # 10
lst[-1]   # 40
lst[1:3]  # [20, 30]
```

4. Tuple (tuple) — Ordered, immutable collection of items.

Syntax

```
tup = (1, 2, 3)
tup = ('a', 101, 90.5)
tup = (1, 'a' 3.5, True)
```

- Indexing starts from 0.
- Cannot be changed after creation.

Example

```
tup = (10, 20, 30, 40)
tup[0]    # 10
tup[-1]   # 40
tup[1:3]  # (20, 30)
```

5. Dictionary (dict) — Unordered collection of key:value pairs.

Example

```
d = {'a': 1, 'e': 2, 'i': 3}
d = {'name': 'Ankit',
     'age': 21}
```

- Keys must be unique and immutable.
- Values can be of any data type.

Example

```
d = {'a': 1, 'e': 2, 'i': 3}
d['a']      # 1
d.keys()    # dict_keys(['a','e','i'])
d.values()  # dict_values([1,2,3])
```

2 TOKENS IN PYTHON

The smallest unit/element in the Python program/script is known as a Token or a Lexical unit.

1. Keywords — Reserved words with special meaning.

```
False   True    None    def     if      lambda  class   yield
continue else   assert  or      while   break   elif    del
from    is      not     pass    for     global  finally import  as
in      nonlocal return  with    and     int     except  raise   print
```

2. Identifiers — (User Defined Names)

Names given to programming elements like variables, functions, classes etc.

Rule 1	Rule 2	Rule 3
Must be made up of letters, numbers and underscore (_).	Cannot begin with a number, although underscore (_) may.	Cannot be a keyword and cannot contain space.

Valid Identifiers	Invalid Identifiers
name, _age, total_marks, student1, value2, __x	2name, class, total marks, name-1, emp.code

3. Literals — Data items that have a fixed/constant value.

String Literals	Numeric Literals	Boolean Literals
`'Hello'  "World"`	• Integer : 123, -45, 0o32, 0xAF	True
`'''Multi-line'''`	• Float : 12.3, -0.17, 3E2	False
`"Line1\nLine2"`	• Complex : 2+3j, -1.5-4j	**Special Literal**
		None

4. Operators — Symbols or words that perform operations.

(See section 3 for details)

5. Punctuators — Symbols used to organize the structure of the program.

```
( )   { }   [ ]   ,   :   .   '   "   #   \   @
```

3 OPERATORS IN PYTHON

Perform operations on operands and return the result.

Type	Operators	Example	Result
Arithmetic	+ - * / % ** //	10 + 3	13
		10 // 3	3
Relational	> < >= <= == !=	5 >= 3	True
Logical	and or not	(5>3) and (2<1)	False
Bitwise	& | ^ ~ << >>	5 & 3	1
Assignment	=	x = 10	10
Arithmetic Assignment	+= -= *= /= //= **= %=	x += 5	15
Membership	in not in	'a' in 'apple'	True
Identity	is is not	x is y	False
Shift	<< >>	8 >> 1	4

4 MUTABLE VS IMMUTABLE DATA TYPES

Mutable (can be changed): list, dict, bytearray, set

Immutable (cannot be changed): int, float, complex, str, tuple, bytes, bool, frozenset

Mutability can be checked using *id()* function.

```
x = 10
print(id(x))
x = 20
print(id(x))
```

```
# Different id → Immutable (int)
# Different id → Immutable (int)
```

```
lst = [1, 2, 3]
print(id(lst))
lst[0] = 99
print(id(lst))
```

```
# Same id → Mutable (list)
# Same id → Mutable (list)
```

5 TYPE CONVERSION IN PYTHON

Implicit Type Conversion (Automatic)

Python automatically converts one data type to another.

```
n = 12                          # int
f = 10.23                       # float
num_new = n + f                 # int converted to float
                                  due to float operation
print(type(n))                  # <class 'int'>
print(type(f))                  # <class 'float'>
print(num_new)                  # 22.23
print(type(num_new))            # <class 'float'>
```

Explicit Type Conversion (Manual)

User converts data type using built-in functions.

```
n = 12
s = "45"
print(type(n))                  # <class 'int'>
print(type(s))                  # <class 'str'>
s = int(s)                      # str to int
num_sum = n + s
print(type(s))                  # <class 'int'>
print(num_sum)                  # 57
print(type(num_sum))            # <class 'int'>
```

Common Conversion Functions

```
int(x)    →  to integer
float(x)  →  to float
str(x)    →  to string
list(x)   →  to list
tuple(x)  →  to tuple
dict(x)   →  to dict
set(x)    →  to set
bool(x)   →  to boolean
```

6 EXAMPLES OF TYPE CONVERSION

To Integer (int())

```
a = "101"       # str
a_int = int(a)
b_int = int(122.4)
print(a_int)    # 101
print(b_int)    # 122
```

To Float (float())

```
a = '301.4'     # str
a_float = float(a)
b_float = float(121)  # int to float
print(a_float)  # 301.4
print(b_float)  # 121.0
```

To String (str())

```
a = 100         # int
a_str = str(a)
b_str = str(3.14)  # float
print(a_str)    # '100'
print(b_str)    # '3.14'
```

To Boolean (bool())

```
print(bool(1))          # True
print(bool(0))          # False
print(bool('Hello'))    # True
print(bool(''))         # False
```

To List (list())

```
t = (1, 2, 3)
lst = list(t)
print(lst)      # [1, 2, 3]
```

Tip: Use type() function to check the data type of any value.

```
type(10.5)  # <class 'float'>
```

PYTHON DATATYPES CHEAT SHEET

Overview • Explanation • Syntax • Examples

Python is Dynamically Typed

You don't need to declare the type of a variable.
Python infers the type automatically.

```
x = 10      # int
y = 3.14    # float
s = "Hello" # str
```

1 NUMBERS

Numeric data types.

Type	Description	Example
int	Integer (whole numbers, positive or negative)	10, -25, 0
float	Floating point numbers (decimal values)	3.14, -0.001, 2.0
complex	Complex numbers (real + imaginary part)	2+3j, -1.5-2.5j
bool	Boolean (True / False) Subclass of int	True, False (1, 0)

SYNTAX (Examples)

```
a = 10          # int
b = 3.14        # float
c = 2 + 3j      # complex
d = True        # bool
```

TYPE CHECK

```
type(a)  # <class 'int'>
type(b)  # <class 'float'>
type(c)  # <class 'complex'>
type(d)  # <class 'bool'>
```

2 STRING

Text data – sequence of characters.

Explanation

- Strings are immutable.
- Enclosed in single (' '), double (" ") or triple quotes (''' ''').
- Each character can be accessed using index.

SYNTAX

```
s = 'Hello World'    # or  "Python"  or  '''Multi Line'''
```

EXAMPLES

```
s1 = 'Python'
s2 = "Data Science"
s3 = '''This is
   a multiline string'''
print(s1[0])       # P
print(s2[5:10])    # Science
print(len(s1))     # 6
```

COMMON OPERATIONS

```
s + '!'      # Concatenation
s * 3        # Repetition
'Py' in s    # Membership
s.upper()    # Upper case
s.lower()    # Lower case
```

3 LIST

Explanation

- Lists can store items of different data types.
- Enclosed in square brackets [ ].
- Items are comma-separated.
- Items can be changed, added or removed.

SYNTAX

```
lst = [item1, item2, item3, ...]
```

EXAMPLES

```
lst = [1, 'Apple', 3.5, True]
print(lst[0])      # 1
print(lst[1:3])    # ['Apple', 3.5]
lst.append(99)     # Add item
lst[0] = 100       # Modify item
print(lst)         # [100, 'Apple', 3.5, True, 99]
```

COMMON OPERATIONS

```
append(x)    # Add item
extend(it)   # Add items
insert(i,x)  # Insert item
remove(x)    # Remove item
pop(i)       # Remove & return
len(lst)     # Length
```

4 TUPLE

Ordered, immutable collection.

Explanation

- Similar to list but immutable (cannot change).
- Enclosed in parentheses ( ).
- Used for fixed data.

SYNTAX

```
tup = (item1, item2, item3, ...)
```

EXAMPLES

```
t = (10, 'Python', 3.14, False)
print(t[0])        # 10
print(t[1:3])      # ('Python', 3.14)
print(len(t))      # 4
# t[0] = 20        # Error: tuple is immutable
```

COMMON OPERATIONS

```
count(x)   # Count occurrences
index(x)   # Index of first occurrence
len(t)     # Length
t + t2     # Concatenation
t * 2      # Repetition
```

5 DICTIONARY

Unordered collection of key-value pairs.

Explanation

- Each item has a unique key and a value.
- Keys must be immutable (e.g., int, str, tuple).
- Enclosed in curly braces { }.

SYNTAX

```
d = {key1: value1, key2: value2, ...}
```

EXAMPLES

```
d = {'name': 'Aman', 'age': 20, 'city': 'Delhi'}
print(d['name'])        # Aman
print(d.get('age'))     # 20
d['age'] = 21           # Modify
d['grade'] = 'A'        # Add
print(d.keys())         # dict_keys([...])
print(d.values())       # dict_values([...])
print(d.items())        # dict_items([...])
```

COMMON OPERATIONS

```
get(key)     # Get value
keys()       # All keys
values()     # All values
items()      # All key-value pairs
update(d2)   # Merge dicts
pop(key)     # Remove by key
del d[key]   # Delete by key
len(d)       # Length
```

6 SET

Unordered collection of unique items.

Explanation

- Sets store only unique elements.
- Unordered (no indexing).
- Enclosed in curly braces { }.
- Mutable (can add/remove items).

SYNTAX

```
s = {item1, item2, item3, ...}
```

EXAMPLES

```
s = {1, 2, 3, 3, 4, 'A', 'A'}  # Duplicates removed
print(s)          # {1, 2, 3, 4, 'A'}
s.add(5)          # Add item
s.remove(2)       # Remove item
print(3 in s)     # True
```

COMMON OPERATIONS

```
add(x)       # Add item
remove(x)    # Remove item
discard(x)   # Remove if exists
pop()        # Remove & return an item
len(s)       # Length
s1 | s2      # Union
s1 & s2      # Intersection
s1 - s2      # Difference
```

MUTABLE vs IMMUTABLE

Mutable (can change)

- list
- dict
- set
- bytearray

Immutable (cannot change)

- int
- float
- complex
- str
- tuple
- bool
- bytes
- frozenset

TYPE CHECKING

Use type() or isinstance()

```
x = [1, 2, 3]
type(x)                 # <class 'list'>
isinstance(x, list)     # True
```

QUICK FACTS

- ✓ Python is dynamically typed.
- ✓ Variables can hold any data type.
- ✓ Use type() to check data type.
- ✓ Choose the right data type for memory efficiency and performance.

TYPE CONVERSION (COMMON)

Convert To	Function	Example	Result
int	int(x)	int('123')	123
float	float(x)	float('3.14')	3.14
str	str(x)	str(100)	'100'
list	list(x)	list('abc')	['a','b','c']
tuple	tuple(x)	tuple([1,2])	(1, 2)
set	set(x)	set([1,2,2,3])	{1,2,3}

TIP

Pick the right data type:

- Use list for ordered, changeable collections.
- Use tuple for fixed data.
- Use dict for key-value lookups.
- Use set for unique items.

PYTHON CONDITIONAL STATEMENTS CHEAT SHEET

Decision Making in Python – Syntax, Flow & Examples

1. OVERVIEW

Conditional statements allow a program to make decisions and execute different blocks of code based on conditions. Python provides four main types:

- if statement
- if...else statement
- if...elif...else statement
- Nested if...else statement

COMPARISON OPERATORS

Operator	Meaning	Example
==	Equal to	5 == 5 → True
!=	Not equal to	5 != 3 → True
>	Greater than	7 > 4 → True
<	Less than	2 < 9 → True
>=	Greater than or equal to	5 >= 5 → True
<=	Less than or equal to	4 <= 6 → True

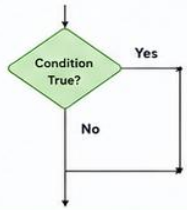
3. if STATEMENT

Executes a block of code only if the condition is True.

SYNTAX

```
if condition:
    statement(s)
```

FLOW DIAGRAM



EXAMPLE

```
n = int(input("Enter number: "))
if n <= 10:
    print("Condition is True")
```

OUTPUT (if input = 6)

```
Enter number: 6
Condition is True
```

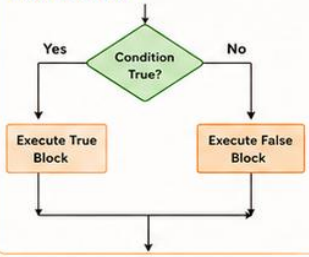
4. if...else STATEMENT

Executes one block if condition is True, another if False.

SYNTAX

```
if condition:
    statement(s) for True
else:
    statement(s) for False
```

FLOW DIAGRAM



EXAMPLE

```
n = int(input("Enter number: "))
if n <= 10:
    print("Condition is True")
else:
    print("Condition is False")
```

OUTPUT (if input = 12)

```
Enter number: 12
Condition is False
```

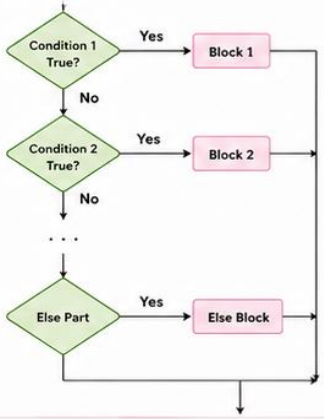
5. if...elif...else STATEMENT

Used when there are multiple conditions to check.

SYNTAX

```
if condition1:
    statement(s)
elif condition2:
    statement(s)
...
else:
    statement(s)
```

FLOW DIAGRAM



EXAMPLE

```
n = int(input("Enter number: "))
if n <= 5:
    print("Less than or equal to 5")
elif n <= 10:
    print("Between 6 and 10")
else:
    print("Greater than 10")
```

OUTPUT (if input = 12)

```
Enter number: 12
Greater than 10
```

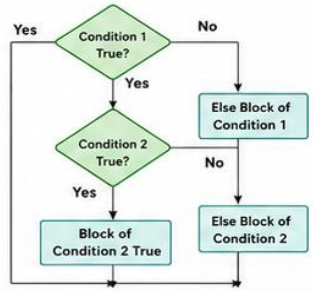
6. NESTED if...else STATEMENT

An if...else statement inside another if...else statement.

SYNTAX

```
if condition1:
    if condition2:
        statement(s)
    else:
        statement(s)
else:
        statement(s)
```

FLOW DIAGRAM



EXAMPLE

```
n = int(input("Enter number: "))
if n <= 15:
    if n == 10:
        print("Number is 10")
    else:
        print("Less than or not equal to 15, but not 10")
else:
    print("Greater than 15")
```

OUTPUT (if input = 11)

```
Enter number: 11
Less than or not equal to 15, but not 10
```

7. QUICK TIPS

- Indentation is mandatory in Python (usually 4 spaces).
- Use meaningful conditions for better readability.
- You can have multiple elif blocks.
- The else block is optional.
- Conditional statements are case sensitive.

8. COMMON EXAMPLE: GRADING SYSTEM

```
marks = int(input("Enter marks (0-100): "))
if marks >= 90:
    grade = 'A'
elif marks >= 75:
    grade = 'B'
elif marks >= 50:
    grade = 'C'
else:
    grade = 'F'
print("Grade:", grade)
```

SAMPLE OUTPUT

```
Enter marks (0-100): 82
Grade: B
-------------------------
Enter marks (0-100): 45
Grade: F
-------------------------
Enter marks (0-100): 95
Grade: A
```

NOTE: Conditions evaluate to True or False. The block under the first True condition is executed.

CONDITION EXAMPLES

True: 5 > 3, x != 0, name == "John"
False: 2 > 5, x == 0, age < 18

BUILT-IN FUNCTIONS USED

`input()`, `int()`, `print()`

MORE TO EXPLORE

Combine with loops for powerful programs! (e.g., for, while loops with conditions)

PYTHON LOOPS & JUMP STATEMENTS CHEAT SHEET

Repeat • Control • Jump • Master Your Flow

1. LOOPS (CONTROL STATEMENTS)

Loops allow us to execute a block of code multiple times.

Python provides two main loop constructs:

- for loop
- while loop
- Nested loops

2. FOR LOOP

- Used to iterate over a sequence (string, list, tuple, dictionary or any iterable object).
- Iterating over a sequence is called **traversal**.

SYNTAX

```
for val in expression:
    # Body of the for loop
```

Example 1: Iterating over a list

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

Example 2: Iterating over a string

```
for ch in "Python":
    print(ch, end=' ')
```

Output:

```
P y t h o n
```

Example 3: Using range()

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

2.1 range() FUNCTION

range() generates a sequence of numbers.

SYNTAX

```
range([begin], end, [step_value])
```

Parameters	Description	Default
begin	Starting number	0
end	Stop before this number	Required
step_value	Increment (positive or negative)	1

Examples:

```
list(range(5))          # [0, 1, 2, 3, 4]
list(range(1, 6))       # [1, 2, 34, 5]
list(range(2, 10, 2))   # [2, 4, 6, 8]
list(range(5, 0, -1))   # [5, 4, 3, 2, 1]
```

3. WHILE LOOP

- Repeats a block of code as long as the condition is True.
- The condition is checked before executing the body.

SYNTAX

```
while condition:
    # Body of while loop
```

Example:

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

Output:

```
1
2
3
4
5
```

4. NESTED LOOPS

A loop inside another loop.

Example:

```
for i in range(1, 3):
    for j in range(1, 4):
        print(f"i={i}, j={j}")
```

Output:

```
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
i=2, j=3
```

QUICK TIPS

- Use for loop when you know the number of iterations.
- Use while loop when the number of iterations is unknown.
- Use else with loops to execute code when loop ends normally.

5. FOR LOOP WITH ELSE

- The else block runs when the loop completes normally (without break).

Syntax:

```
for val in expression:
    # statements
else:
    # statements executed after loop
      completes normally
```

Example:

```
for i in range(3):
    print(i)
else:
    print("Loop completed successfully!")
```

Output:

```
0
1
2
Loop completed
successfully!
```

6. JUMP STATEMENTS

Jump statements are used to alter the normal flow of a loop.

Python provides three jump statements:

- break
- continue
- pass

6.1 break STATEMENT

Terminates the loop entirely.

Example:

```
for val in "string":
    if val == 'i':
        break
    print(val)
print( "The end" )
```

Output:

```
s
t
r
The end
```

Note: Loop stops when condition is met.

6.2 continue STATEMENT

Skips the remaining statements in the loop for the current iteration and continues with the next iteration.

Example:

```
for val in "string":
    if val == 'i':
        continue
    print(val)
print( "The end" )
```

Output:

```
s
t
r
n
g
The end
```

Note: 'i' is skipped, rest are printed.

6.4 pass vs continue (EXAMPLE)

Using pass

```
for i in 'initial':
    if i == 'i':
        pass
    else:
        print(i)
```

Output:

```
n
t
a
l
```

VS

Using continue

```
for i in 'initial':
    if i == 'i':
        continue
    else:
        print(i)
```

Output:

```
i
n
i
t
i
a
l
```

continue skips to next iteration.
pass allows the loop to continue normally.

6.3 pass STATEMENT

Does nothing. It is used when a statement is required syntactically but no action is needed.

Example:

```
for val in "string":
    if val == 'i':
        pass
    print(val)
print( "The end" )
```

Output:

```
s
t
r
i
n
g
The end
```

Note: pass performs no action.

7. pass STATEMENT – OTHER USES

In while loop (busy wait):

```
while True:
    pass   # Busy-wait (Ctrl+C to stop)
```

In function definition:

Used as a placeholder when no code is written yet.

```
def my_function():
    pass # Placeholder
```

Output: Function defined)

AT A GLANCE

Loop Types	Jump Statements	Common Use
• for loop	• break → exits loop	• Repeat tasks
• while loop	• continue → next iteration	• Process collections
• Nested loops	• pass → no operation	• Control loop flow

BEST PRACTICES

- Avoid infinite loops.
- Use meaningful loop control.
- Prefer for loop for sequences.
- Use jump statements wisely.

PYTHON STRING – CHEAT SHEET (CLASS XII)

Quick Revision Guide for CBSE Board Exam

Key Point
- Strings are immutable.
- Index starts from 0.
- Negative indexing is allowed.

1. WHAT IS STRING?

A string is a sequence of characters enclosed in single quotes (' '), double quotes (" ") or triple quotes (''' ''' or """ """). Python treats single and double quotes same.

Syntax :

```
str = 'Hello'
str = "Hello"
str = '''This is a
      multi-line string'''
str = """This is also a
      multi-line string"""
```

Example :

```
s1 = 'Python'
s2 = "Computer Science"
s3 = '''Welcome
      to Python'''
```

2. TRIPLE QUOTES

Triple quotes are used to create strings with multiple lines or to write documentation in functions.

Syntax :

```
str = """This course will introduce
the learner to text mining and
text manipulation basics.
The course begins with an
understanding of how text is
handled by python"""
```

Example :

```
print(str)
```

(Output) →

```
This course will introduce the learner
to text mining and text manipulation
basics.
The course begins with an
understanding of how text is
handled by python
```

3. FEATURES OF STRING

1. It is used to store text type of data.
2. Strings are immutable (no change of value in place).
3. Empty string has zero characters.
4. In string each character has a unique position id / index.

4. DECLARE EMPTY STRING

Syntax :

```
str = ''  (or)  str = ""
```

Example :

```
empty1 = ''
empty2 = ""
```

Length of empty string is 0

5. ACCESSING CHARACTERS (INDEXING)

String items can be accessed using its index position.

Example : `s = "who"`

Positive Indexing

0	1	2
w	h	o

Negative Indexing

-3	-2	-1
w	h	o

```
print(s[0])   # w
print(s[1])   # h
print(s[2])   # o
print(s[-1])  # o
print(s[-2])  # h
print(s[-3])  # w
```

6. SLICING IN STRING

To extract limited information from a string or to access a range of characters.

Syntax :

```
string_name[start : stop : step]
```

- start : starting index (default = 0)
- stop : ending index (default = len(string))
- step : step size (default = 1)

Example : `s = "INDIA"`

Expression	Result	Explanation
s[0:3]	'IND'	from index 0 to 2
s[3:]	'IA'	from index 3 to end
s[:]	'INDIA'	entire string
s[::2]	'INA'	step of 2
s[::-1]	'AIDNI'	reverse string

7. TRAVERSING / ITERATING THROUGH STRING

Each character of the string can be accessed sequentially using for loop.

Example 1 :

```
str = 'Computer Science'
for ch in str:
    print(ch)
```

(Output)

```
C
o
m
p
u
t
e
r

S
c
i
e
n
c
e
```

Example 2 :

```
str = 'Computer Science'
for i in range(len(str)):
    print(str[i])
```

(Output)

```
C
o
m
p
u
t
e
r

S
c
i
e
n
c
e
```

8. DISPLAY CHARACTER AND ITS POSITION

Print index and character.

Example :

```
str = 'COMPUTER SCIENCE'
for i in range(len(str)):
    print(i, str[i])
```

(Output)

0	C
1	O
2	M
3	P
4	U
5	T
6	E
7	R
8	
9	S
10	C
11	I
12	E
13	N
14	C
15	E

Space also has an index!

9. STRING COMPARISON

We can use (>, <, >=, <=, ==, !=) to compare two strings. Strings are compared lexicographically (ASCII value).

Example :

```
"Reena" == "Reeta"    # False
"Reena" != "Reeta"    # True
"Reena" >  "Reeta"    # False
"Reena" >= "Reeta"    # False
"Reena" <  "Reeta"    # True
"Reena" <= "Reeta"    # True
"Reena" >  ""         # True
```

ord() is used to find the Unicode (ASCII) value.

Syntax : `ord('A') → 65`

10. UPDATING STRINGS

Strings are immutable but we can update by reassigning new value (using concatenation).

Example :

```
a = 'Hello World'
a = a[:7] + ' with file'
print(a)
```

(Output) → `Hello with file`

Hello World → (Change 'World' to 'with file') → Hello with file

11. STRING SPECIAL BASIC OPERATORS

Operator	Description	Example
+	Concatenation (add two strings)	a='comp', b='sc' → a+b = 'compsc'
*	Replication (repeat string)	a*2 → 'compcomp'
[]	Character at index	a[1] → 'o'
[:]	Slice (range of characters)	a[1:4] → 'omp'
in	Membership (check availability)	'p' in a → True
not in	Membership (check non-availability)	'M' not in a → False
%	Format the string (placeholder)	print("My file name is %s\n location is %d" % ('story',10))

12. STRING METHODS (PART – 1)

Method	Description	Example	Output
len()	Returns length of string	len('comp')	4
capitalize()	Capitalize first character	'comp'.capitalize()	'Comp'
title()	Title case string	'my comp'.title()	'My Comp'
upper()	Upper case	'comp'.upper()	'COMP'
lower()	Lower case	'COMP'.lower()	'comp'
count(sub)	Count occurrences of substring	'comp'.count('o')	1
find(sub)	Find index of first occurrence	'comp'.find('m')	2
replace(old,new)	Replace substring	'my comp'.replace('my','your')	'your comp'
index(sub)	Return index, if not found -> error	'comp'.index('om')	1

13. STRING METHODS (PART – 2)

Method	Description	Example	Output
isalnum()	Only alphanumeric characters	'abc123'.isalnum()	True
isalpha()	Only alphabetic characters	'abc'.isalpha()	True
islower()	All letters lowercase	'abc'.islower()	True
isnumeric()	Only numeric characters	'1234'.isnumeric()	True
isspace()	Only whitespace	'I '.isspace()	True
istitle()	Title case	'My Comp'.istitle()	True
isupper()	All letters uppercase	'ABC'.isupper()	True

14. STRING METHODS (PART – 3)

Method	Description	Example	Output
lstrip(char)	Remove leading characters	' comp'.lstrip()	'comp'
rstrip(char)	Remove trailing characters	'comp '.rstrip()	'comp'
strip(char)	Remove both leading & trailing	' comp '.strip()	'comp'
split()	Split string into list	'my comp'.split()	['my','comp']
partition(sub)	Split at first occurrence	'my comp'.partition('comp')	('my ','comp','')

QUICK TIPS

- ☑ Strings are immutable.
- ☑ Use slicing to extract parts.
- ☑ Use methods to work easily.
- ☑ Index starts from 0.
- ☑ Negative indexing allowed.
- ☑ Use find() to avoid errors (if not found -> -1).

15. SEARCHING FOR SUBSTRINGS

Method	Description	Example on s = "welcome to python"	Output
endswith(s1)	True if string ends with s1	s.endswith('thon')	True
startswith(s1)	True if string starts with s1	s.startswith('good')	False
find(s1)	Index of first occurrence	s.find('come') s.find('become')	3 -1
rfind(s1)	Index of last occurrence	s.rfind('o')	15
count(s1)	Number of occurrences	s.count('o')	3

16. COUNTING DIGITS IN STRING (PROGRAM)

```
string = input("Enter string: ")
c = 0
for i in string:
    if(i.isdigit()):     # or if '0' <= i <= '9':
        c = c + 1
print("The number of digits is:", c)
```

17. STRING FORMATTING (OLD STYLE % OPERATOR)

Format	Use	Format	Use
%s	String conversion via str()	%x	Hexadecimal (lowercase)
		%X	Hexadecimal (UPPERCASE)
%i	Signed decimal integer	%e	Exponential (lowercase e)
%d	Signed decimal integer	%E	Exponential (UPPERCASE E)
%u	Unsigned decimal integer	%f	Floating point number
%o	Octal integer	%c	Character
		%G	Shorter of %f and %E

Example :

```
name  = 'John'
age   = 20
price =  75.25
print( 'Name: %s, Age: %d, Price: %.2f
      % (name, age, price))
```

(Output)

```
Name: John, Age: 20, Price: 75.25
```

Lists are ordered, mutable collections that can store multiple items.

PYTHON LIST – CHEAT SHEET (CLASS XII)

⭐ Quick Revision Guide for CBSE Board Exam ⭐

Indexing starts from 0 to n-1 where n is the number of items.

1. WHAT IS LIST?

- A list is a collection of items.
- Items can be of different data types.
- Lists are written inside square brackets [ ].
- Lists are ordered, changeable (mutable) and allow duplicate values.
- Index of first item is 0 and last item is n-1 (where n is number of items)

Example:

```
l1 = ['English', 'Hindi', 1997, 2000]
```

```
l1 → [ 'English', 'Hindi', 1997, 2000 ]
Index →    0         1       2     3
```

2. CREATING LIST

Lists are created by placing items inside square brackets [ ] separated by commas.

Syntax:

```
list_name = [item1, item2, item3, ...]
```

Examples:

1. list1 = ['English', 'Hindi', 1997, 2000]
2. list2 = [11, 22, 33, 44, 55]
3. list3 = ['a', 'b', 'c', 'd']
4. Blank List (empty list)

```
list4 = []
```

3. INDEXING (ACCESSING ITEMS)

Items in a list can be accessed using their index position.

Positive Indexing

0	1	2	3	4
I	N	D	I	A

Negative Indexing

-5	-4	-3	-2	-1
I	N	D	I	A

Example:

```
a = ['I','N','D','I','A']
print(a[0])    # I
print(a[2])    # D
print(a[-1])   # A
print(a[-3])   # D
```

Output:

```
I
D
A
D
```

4. TRAVERSING (ITERATING) THROUGH A LIST

Elements of a list can be accessed one by one using loop.

Example:

```
a = [3, 5, 9]
for i in range(len(a)):
    print(a[i])
```

Output:

```
3
5
9
```

5. SLICING OF A LIST

We can access a part (sub-list) of a list using slicing.

Syntax: `list[start : end]` (end-1 is the last index included)

Example	Explanation
`a = ['I','N','D','I','A']`	
`print(a[0:3])  # ['I','N','D']`	From index 0 to 2
`print(a[3:])  # ['I','A']`	From index 3 to end
`print(a[:])  # ['I','N','D','I','A']`	Entire list

6. UPDATING / MANIPULATING LISTS

We can update single or multiple elements in a list.

Syntax:

```
list[index] = new_value              (single update)
list[start : end] = [v1, v2, ...]    (multiple update)
```

Example:

```
l = ['English', 'Hindi', 1997, 2000]
print(l[2])              # 1997
l[2] = 2001              # single update
l[2:4] = [2001, 2002]    # multiple update
print(l)                 # ['English', 'Hindi', 2001, 2002]
```

7. ADDING ITEMS TO A LIST

(i) append() – Add single item at the end

Example:

```
l = [1, 2]
l.append(3)
print(l)      # [1, 2, 3]
```

(ii) extend() – Add multiple items at the end

Example:

```
l.extend([4, 5])
print(l)      # [1, 2, 3, 4, 5]
```

(iii) insert() – Add item at a specific index

Example:

```
l.insert(1, 10)
print(l)      # [1, 10, 2, 3, 4, 5]
```

8. ADD TWO LISTS (CONCATENATION)

Two lists can be combined using + operator.

Example:

```
l1 = [1, 2]
l2 = [3, 4]
l3 = l1 + l2
print(l3)    # [1, 2, 3, 4]
```

9. DELETING ITEMS FROM A LIST

(i) Delete an item using del and index

Example:

```
l = [1, 2, 3]
del l[1]
print(l)      # [1, 3]
```

(ii) Delete multiple items (using slice)

Example: `del l[0:2]  # deletes first two items`

(iii) Delete entire list

Example: `del l`

10. BASIC LIST OPERATIONS

Operation	Example	Result / Explanation
Length	`len([4, 2, 3])`	3
Concatenation	`[4,2,3] + [1,5,6]`	`[4,2,3,1,5,6]`
Repetition	`['CS!'] * 4`	`['CS!','CS!','CS!','CS!']`
Membership	`3 in [4,2,3]`	True
Iteration	`for x in [4,2,3]: print(x,end=' ')`	4 2 3

11. IMPORTANT METHODS & FUNCTIONS OF LIST

Method / Function	Description	Method / Function	Description
append(x)	Add an item at end of list	index(x)	Return index of first matched item
extend(iterable)	Add multiple items at end	count(x)	Count number of occurrences of x
insert(i, x)	Insert item x at index i	sort()	Sort list in ascending order
remove(x)	Remove first occurrence of x	sort(reverse=True)	Sort list in descending order
del list[i]	Delete item at index i	reverse()	Reverse the list
clear()	Remove all items (empty list)	len(list)	Return total number of items
pop([i])	Remove & return item at index i (Default last item)	max(list) / min(list)	Return largest / smallest item
		list(seq)	Convert tuple, string, set, dict to list

12. IMPORTANT PROGRAMS (QUICK LOOK)

(i) Largest / Max Number

```
max_list = [10, 5, 30, 20]
print(max(max_list))   # 30
```

(ii) Average (Mean)

```
lst = [10,20,30,40]
print(sum(lst)/len(lst))  # 25.0
```

(iii) Linear Search

```
x = 20
if x in lst:
    print('Found')
else:
    print('Not Found')
```

(iv) Frequency of an Element

```
lst = [101,101,201,101]
print(lst.count(101))   # 3
```

CBSE EXAM TIPS

- ✅ List is mutable (changeable).
- ✅ Use append() to add one item.
- ✅ Use extend() to add multiple items.
- ✅ Index starts from 0.
- ✅ Negative indexing is allowed.
- ✅ Use count() to find frequency.
- ✅ Use sort() for ascending order.
- ✅ Use max() and min() for largest and smallest value.

MEMORY TRICK

A	E	R	I	C	P	S
Append	Extend	Remove	Insert	Count	Pop	Sort

Practice More, Score More! 💯

PYTHON TUPLES CHEAT SHEET

Quick Reference Guide with Explanation, Syntax & Examples

1 WHAT IS A TUPLE?

- A tuple is a sequence of **immutable** objects.
- Tuples are just like lists **but cannot be changed** once created.
- Elements are ordered and allow duplicates.
- Tuples use **parentheses ()** while lists use **square brackets []**.

(1 , 'a' , 3.5)

CREATING A TUPLE

```
# Syntax
tup = (item1, item2, item3, ...)
```

Examples

```
t1 = ('comp sc', 'info practices', 2017, 2018)
t2 = (5, 11, 22, 44)
t3 = ( )          # empty tuple
```

Indexing of tuple is similar to list.

Index	0	1	2	3
	'a'	11	22	33
Negative Index	-4	-3	-2	-1

2 ACCESSING & SLICING

Use square brackets [] to access elements.

EXAMPLE

```
tup1 = ("comp sc", "info", 2017, 2018)
tup2 = (5, 11, 22, 44, 9, 66)
print("tup1[0]:", tup1[0])
print("tup2[1:5]:", tup2[1:5])
```

OUTPUT

```
tup1[0]:  comp sc
tup2[1:5]: (11, 22, 44, 9)
```

3 ITERATING THROUGH A TUPLE

Access elements sequentially using loops.

SYNTAX

```
for i in range(len(tup)):
    print(tup[i])
```

EXAMPLE

```
tup = (5, 11, 22)
for i in range(len(tup)):
    print(tup[i])
```

OUTPUT

```
5
11
22
```

4 TUPLE IMMUTABILITY & UPDATE

- Tuples are **immutable**.
- We cannot change content directly.
- To update, create a new tuple by **combining existing tuples.**

EXAMPLE

```
tup1 = (1, 2)
tup2 = ('a', 'b')
tup3 = tup1 + tup2      # create new tuple
print(tup3)
```

OUTPUT

```
(1, 2, 'a', 'b')
```

5 DELETING TUPLE ELEMENTS

- Direct deletion of an element is **not possible**.
- Create a new tuple excluding unwanted items.

EXAMPLE

```
tup1 = (1, 2, 3)
tup3 = tup1[0:1] + tup1[2:]
print(tup3)
```

OUTPUT

```
(1, 3)
```

Delete entire tuple using **del** statement.

```
del tup1
```

6 BASIC TUPLE OPERATIONS

Operation	Python Expression	Result	Description
len()	`len((1, 2))`	2	Length of tuple
Concatenation	`(1, 2) + (4, 5)`	(1, 2, 4, 5)	Join two tuples
Repetition	`('CS',) * 2`	('CS', 'CS')	Repeat tuple
Membership	`5 in (1, 2, 3)`	False	Check item exists
Iteration	`for x in (4,2,3): print(x)`	4 2 3	Iterate elements

7 TUPLE FUNCTIONS (Most Useful)

Function	Description	Example	Output
tuple(seq)	Convert list to tuple	tuple([1,2,3])	(1, 2, 3)
min(tup)	Minimum value	min((1,4,6))	1
max(tup)	Maximum value	max((1,4,6))	6
len(tup)	Length of tuple	len((1,2,3))	3
count(value)	Count occurrences	(1,3,7,8,7,5).count(7)	2
index(value)	Index of first occurrence	('a','e','i').index('e')	1
sum(tup)	Sum of numeric elements	sum((1,4,6))	11
sorted(tup)	Return sorted list	sorted((3,1,2))	[1, 2, 3]
Mean/Average	Average of numeric items	sum((1,4,6))/len((1,4,6))	3.66

TIP — Immutable → Cannot be changed | Ordered → Keeps insertion order | ( ) Uses () parentheses

8 LINEAR SEARCH IN TUPLE

Find an element using linear search.

EXAMPLE

```
tup = (1, 2, 3, 4, 5, 6)
n = 5
found = False
for i in range(len(tup)):
    if tup[i] == n:
        found = True
        break
if found:
    print("Found")
else:
    print("Not Found")
```

OUTPUT

```
Found
```

(1, 2, 3, 4, 5, 6)

9 QUICK NOTES

- Tuples allow **duplicate values**.
- Access by **positive or negative index**.
- Tuples are **faster** than lists.
- A single item tuple needs a **comma** → (5,)
- Used for fixed data (e.g., coordinates, records).

TUPLE = Fixed • Ordered • Immutable

PYTHON DICTIONARY CHEAT SHEET

{ key : value }

Unordered • Mutable • Key-Value Pairs

1. WHAT IS A DICTIONARY?

- A dictionary is an unordered collection of items.
- Each item consists of a key and a value.
- It is mutable (can modify its contents), but keys must be unique and immutable.

Key	Value
'name'	'Aman'
'age'	25

2. CREATING A DICTIONARY

- Enclosed in curly braces {}.
- Each key and value are separated by a colon (:).
- Each item is separated by a comma (,).

SYNTAX

```
dict_name = {key1: value1, key2: value2, ...}
```

EXAMPLE

```
student = {'name': 'Aman', 'age': 17, 'class': 11}
print(student)
```

OUTPUT

```
{'name': 'Aman',
 'age': 17, 'class': 11}
```

3. ACCESSING ITEMS

Using Key ([])

```
student = {'name': 'Aman',
           'age': 17, 'class': 11}
print(student['name'])
```

OUTPUT

```
Aman
```

Using get()

```
print(student.get('age'))    # 17
print(student.get('address'))
```

OUTPUT

```
None  (if key not found)
```

Note: get() is safer than [] because it doesn't raise an error if the key is missing.

4. ITERATING / TRAVERSING

Iterate over Keys

```
for key in student:
    print(key)
```

OUTPUT

```
name
age
class
```

Iterate over Values

```
for value in student.values():
    print(value)
```

OUTPUT

```
Aman
17
11
```

Iterate over Key-Value Pairs

```
for k, v in student.items():
    print(k, ":", v)
```

OUTPUT

```
name : Aman
age : 17
class : 11
```

5. UPDATING / MODIFYING

Change a Value

```
student = {'name': 'Aman', 'age': 17}
student['age'] = 18
print(student)
```

OUTPUT

```
{'name': 'Aman', 'age': 18}
```

Add a New Key-Value Pair

```
student['class'] = 11
print(student)
```

OUTPUT

```
{'name': 'Aman', 'age': 18, 'class': 11}
```

6. DELETING ITEMS

del - Delete by Key

```
student = {'name': 'Aman',
  'age': 17, 'class': 11}
del student['age']
print(student)
```

OUTPUT

```
{'name': 'Aman','class': 11}
```

pop() - Remove & Return Value

```
val = student.pop('class')
print(val)
print(student)
```

OUTPUT

```
11
{'name': 'Aman'}
```

clear() - Remove All Items

```
student.clear()
print(student)
```

OUTPUT

```
{}
```

Note: del removes an item by key. del dict_name deletes the entire dictionary.

7. DICTIONARY OPERATIONS

Operation	Example	Result	Description
len(d)	len(student)	3	Number of key-value pairs
'name' in d	'name' in student	True	Check if key exists
d1.update(d2)	d1.update(d2)	-	Merge d2 into d1
d.copy()	student.copy()	New dict	Shallow copy
d1 == d2	d1 == d2	True/False	Compare dictionaries

Key Points

- Unordered • Mutable • Keys are Unique & Immutable
- Values can be any data type • Fast lookup by key

8. BUILT-IN FUNCTIONS (Dictionary)

Function	Example	Result	Description
len(d)	len(student)	3	Length of dictionary
str(d)	str(student)	"{...}"	String representation
type(d)	type(student)	dict	Type of variable

9. DICTIONARY METHODS

Method	Example	Description	Example Output
dict()	x = dict(name='Aman', age=17)	Create dictionary	{'name': 'Aman', 'age': 17}
keys()	student.keys()	Return all keys	dict_keys(['name', 'age', 'class'])
values()	student.values()	Return all values	dict_values(['Aman', 17, 11])
items()	student.items()	Return key-value pairs	dict_items([('name', 'Aman'), ('age', 17), ('class', 11)])
update(d)	student.update(d2)	Update/merge dictionaries	Updated dict
pop(key)	student.pop('age')	Remove key & return value	17
popitem()	student.popitem()	Remove last inserted key-value pair	('class', 11)
clear()	student.clear()	Remove all items	{}
copy()	student.copy()	Return a shallow copy	New dictionary
fromkeys(seq, val)	dict.fromkeys(['a','b'],0)	Create dict from keys with value	{'a': 0, 'b': 0}
setdefault(k, val)	student.setdefault('city', 'Pune')	Return value if key exists; else insert key with value	'Pune' (and inserts if not exists)
max(d, key=d.get)	max(d, key=d.get)	Key with max value	Key
min(d, key=d.get)	min(d, key=d.get)	Key with min value	Key
sorted(d)	sorted(d.items())	Return sorted list of items	Sorted list

10. COMMON EXAMPLE

Count frequency of characters in a string

```
s = "python"
freq = {}
for ch in s:
    if ch in freq:
        freq[ch] += 1
    else:
        freq[ch] = 1
print(freq)
```

OUTPUT

```
{'p': 1, 'y': 1, 't': 1,
 'h': 1, 'o': 1, 'n': 1}
```

Store employee names and salary

```
employees = {'Aman': {'salary': 10000},
             'Mayur': {'salary': 51000}}
emp1 = employees['Aman']
print(emp1)
```

OUTPUT

```
{'salary': 10000}
```

Month and number of days

```
months = {'Jan':31, 'Feb':28, 'Mar':31}
m = input('Enter month: ')
print('Days:', months.get(m, 'Invalid'))
```

OUTPUT

```
Enter month: Feb
Days:  28
```

11. QUICK NOTES

- Dictionaries store data in key-value pairs.
- Keys must be unique and immutable (str, int, tuple, etc.).
- Values can be any data type.
- Use get() to avoid KeyError for missing keys.
- Highly efficient for lookups and updates by key.

12. QUESTIONS (PRACTICE)

1. Create a dictionary to store 4 student details with rollno, name, age. Search a student in the list.
2. Create a dictionary for months and number of days in a year. User enters month name and system displays number of days.