USER DEFINED FUNCTIONS IN PYTHON — Exam Cheat Sheet

Class XII Computer Science (083) | types, parameters, return, flow of execution and scope, with worked outputs

1. What a function is, and why we use it

A function is a block of code written once, given a name, and then used as many times as needed. Instead of repeating twenty lines in five places, you write them once and call the name five times.

Benefit	In plain words
Reusability	write the code once, call it any number of times
Readability	the program is broken into small named parts, easy to read
Easy to correct	a mistake is fixed in one place only
Easy testing	each function can be checked separately
Less code	the same lines are not repeated again and again

2. The three types of function

Type	Where it comes from	How it is used	Examples
Built-in function	already inside Python, nothing to import	just call it	len(), print(), input(), int(), max(), sum(), type()
Function of a module	kept in a module file, must be imported first	import the module, then call	math.sqrt(), random.randint(), statistics.mean()
User defined function	written by the programmer with def	define it, then call it	def area(), def isPrime(), def COUNT()

```
print(len("Python"))          # built-in    -> 6

import math                   # module function
print(math.sqrt(25))          #             -> 5.0

def square(n):                # user defined
    return n * n
print(square(5))              #             -> 25
```

3. Creating and calling a user defined function

```
def function_name(parameters):     # the def line, called the HEADER
    """ optional docstring """
    statements                     # the BODY, must be indented
    return value                   # optional

function_name(arguments)           # the CALL

# a complete example
def greet(name):
    print("Hello", name)

greet("Aarav")                     # Hello Aarav
greet("Siya")                      # Hello Siya
```

- The def line always ends with a colon, and every line of the body must be indented by the same amount.
- Defining a function does NOT run it. Nothing happens until the function is CALLED.
- A function must be defined BEFORE the line that calls it, otherwise Python reports a NameError.

4. Parameters and arguments

Term	Also called	Where it is written	Example
Parameter	formal parameter	inside the brackets of the def line	`def add(a, b):`
Argument	actual parameter	inside the brackets of the CALL	`add(10, 20)`

```
def add(a, b):          # a and b are PARAMETERS
    print(a + b)

add(10, 20)             # 10 and 20 are ARGUMENTS   -> 30
```

Easy way to remember : the DEFINITION has parameters, the CALL carries arguments.

5. Types of arguments

5.1 Positional arguments

The values are matched to the parameters by their POSITION — first to first, second to second. The number of arguments must be exactly equal to the number of parameters.

```
def student(name, cls):
    print(name, "studies in class", cls)

student("Aarav", 12)        # Aarav studies in class 12
student(12, "Aarav")        # 12 studies in class Aarav   <- wrong order, no error
student("Aarav")            # ERROR : missing 1 required positional argument
```

- Python does not check the meaning, only the position. Swapping the order gives a wrong answer, not an error.

5.2 Default arguments

A parameter can be given a value in the def line. If the call does not supply that value, the default is used.

```
def interest(p, r, t = 2):
    return (p * r * t) / 100

print(interest(1000, 5))        # t takes the default 2   -> 100.0
print(interest(1000, 5, 3))     # 3 replaces the default  -> 150.0
```

RULE : once a parameter has a default, every parameter after it must also have one. Defaults are written at the RIGHT end only.

```
def f(a, b = 2, c = 3):     # correct
def f(a = 1, b, c):         # ERROR : non-default argument follows default argument
def f(a, b = 2, c):         # ERROR : same reason
```

5.3 Keyword arguments

The argument is given with the name of the parameter, so the order no longer matters.

```
def student(name, cls):
    print(name, cls)

student(cls = 12, name = "Aarav")   # Aarav 12   — order changed, still correct
student("Aarav", cls = 12)          # Aarav 12   — allowed
student(name = "Aarav", 12)         # ERROR : positional argument follows keyword argument
```

- Positional arguments must always be written BEFORE keyword arguments.

Difference - Positional, Keyword and Default Arguments

Point	Positional Argument	Keyword Argument	Default Argument
Meaning	Value is passed according to its **position/order**	Value is passed using the **parameter name**	Parameter has a **predefined value**
Order	**Important**	Not important	Not applicable
Syntax	fun(10, 20)	fun(a=10, b=20)	def fun(a, b=20):
If value is not given	Error (if required)	Error (if required)	**Default value is used**
Example	add(10, 20)	add(b=20, a=10)	def add(a, b=10):

Important Notes:

Positional and keyword arguments can be mixed in a function call, but there is one important rule: **Positional arguments must come BEFORE keyword arguments.**

Example:

```
def student(name, age, city):
  print(name, age, city)

student("Riya", age=17, city="Delhi")
```

Default arguments must come after non-default arguments. Once a default argument is given, all arguments to its right must also have default values. If no value is passed, the default value is used.

def student(name, age=17): # ✅ Correct

def student(age=17, name): # ❌ Incorrect

def student(name, age=17, city="Delhi"): # ✅ Correct

def student(name, age=17, city): # ❌ Incorrect

6. Functions that return a value

6.1 return sends a value back

```
def square(n):
    return n * n            # sends the answer back to the caller

x = square(5)               # the returned value is stored
print(x)                    # 25
print(square(4) + 10)       # 26  — the returned value can be used directly
```

6.2 A function without return gives None

```
def show(n):
    print(n * n)            # it PRINTS, it does not RETURN

y = show(5)                 # 25 is printed by the function
print(y)                    # None  <- the favourite output question
```

print() shows a value on the screen. return sends a value back into the program. They are not the same thing, and a function that only prints returns None.

6.3 Returning more than one value

```
def calc(a, b):
    return a + b, a - b, a * b       # three values

print(calc(10, 3))                   # (13, 7, 30)   <- it is a TUPLE

s, d, m = calc(10, 3)                # unpacking into three variables
print(s, d, m)                       # 13 7 30
```

- Several values returned together always come back as ONE tuple.
- return also ENDS the function immediately. Any line written after it never runs.

```
def test(n):
    return n * 2
    print("Hello")          # this line is never executed

print(test(5))              # 10   — Hello is never printed
```

7. Flow of execution

Flow of execution means the order in which the lines actually run. Python starts at the top of the file and goes down, but a function CALL makes it jump.

```
1   def greet():
2       print("B")
3       print("C")
4
5   print("A")
6   greet()
7   print("D")
```

Step	Line	What happens
1	1	the def line is read — the function is only REMEMBERED, not run
2	5	prints A
3	6	the call — Python JUMPS into the function
4	2	prints B
5	3	prints C
6	6	the function ends — Python COMES BACK to where it was called
7	7	prints D

OUTPUT : A B C D

- Lines 2 and 3 are skipped at the start. A function body runs only when the function is called.
- After the function finishes, control returns to the NEXT line after the call — never to the top of the program.

A function calling another function

```
def second():
    print("In second")

def first():
```

```
    print("In first, start")
    second()
    print("In first, end")

print("Main start")
first()
print("Main end")
```

OUTPUT : Main start / In first, start / In second / In first, end / Main end

8. Scope of a variable

Scope	Where the variable is made	Where it can be used	Life
Local	inside a function	only inside that function	only while the function is running
Global	outside every function, in the main program	anywhere in the file, including inside functions (for reading)	as long as the program runs

8.1 A local variable cannot be seen outside

```
def test():
    a = 10                  # LOCAL to test()
    print(a)                # 10

test()
print(a)                    # ERROR : NameError, a is not defined
```

8.2 A global variable CAN be read inside a function

```
x = 50                      # GLOBAL

def show():
    print(x)                # reading is allowed  -> 50

show()
print(x)                    # 50
```

8.3 Assigning inside a function makes a NEW local variable

```
x = 50

def change():
    x = 100                 # a NEW local x, the global one is untouched
    print("Inside  :", x)   # Inside  : 100

change()
print("Outside :", x)       # Outside : 50    <- classic output question
```

8.4 The global statement — to really change the global one

```
x = 50

def change():
    global x                # now x means the global x
    x = 100
    print("Inside  :", x)   # Inside  : 100

change()
print("Outside :", x)       # Outside : 100
```

8.5 The error students meet most often

```
x = 10

def add():
    x = x + 1               # ERROR : UnboundLocalError
    print(x)

add()
```

Because x is assigned somewhere in the function, Python treats x as LOCAL everywhere in that function — so x + 1 tries to read a local x that has no value yet. Writing 'global x' as the first line fixes it.

8.6 Same name, two different variables

```
a = 5

def test(a):                # this a is local, it hides the global a
    a = a + 1
    print("Inside  :", a)   # Inside  : 6
```

```
test(a)
print("Outside :", a)       # Outside : 5
```

- A number, string or tuple passed to a function CANNOT be changed by it — the original stays as it was.
- A LIST or a DICTIONARY passed to a function CAN be changed by it, because both are mutable.

```
def addItem(L):
    L.append(99)            # the real list is changed

nums = [1, 2, 3]
addItem(nums)
print(nums)                 # [1, 2, 3, 99]
```

Difference – Global and Local variables

Point	Local Variable	Global Variable
Declared	Inside a function	Outside all functions
Scope	Can be used only inside the function	Can be used throughout the program
Lifetime	Exists while the function is executing	Exists throughout program execution
Access inside function	Directly accessible	Directly accessible
Access outside function	✗ Not accessible	☑ Accessible
Example	def fun(): x = 10	x = 10def fun():

9. Output questions of the kind that are asked

Q1

```
def change(n):
    n = n * 2
    return n

a = 5
b = change(a)
print(a, b)
```

OUTPUT : 5 10 a is an integer, so the function cannot change it. b holds the returned value.

Q2

```
def calc(x, y = 2, z = 3):
    return x + y * z

print(calc(1))
print(calc(1, 4))
print(calc(1, 4, 5))
print(calc(z = 1, x = 2))
```

OUTPUT : 7 / 13 / 21 / 4 Work out 1+2*3, 1+4*3, 1+4*5, 2+2*1.

Q3

```
x = 100

def fun():
    x = 200
    print(x, end=" ")

fun()
print(x)
```

OUTPUT : 200 100

Q4

```
def total(L):
    s = 0
    for i in L:
        s = s + i
    print(s)
```

```
ans = total([10, 20, 30])
print(ans)
```

OUTPUT : 60 then None The function prints but never returns, so ans is None.

Q5

```
def f(a, b):
    a = a + b
    b = a - b
    return a, b

p, q = f(10, 5)
print(p, q)
```

OUTPUT : 15 10 a becomes 15, then b becomes 15 − 5 = 10.

10. Ready-made functions for the practical file

```
# 1. a function that returns a value
def area(r):
    return 3.14 * r * r
print(area(5))                        # 78.5

# 2. a function with a default parameter
def power(base, exp = 2):
    return base ** exp
print(power(3), power(3, 3))          # 9 27

# 3. a function returning two values
def minMax(L):
    return min(L), max(L)
small, big = minMax([4, 9, 1])
print(small, big)                     # 1 9

# 4. a function using a global counter
count = 0
def visit():
    global count
    count = count + 1
visit(); visit(); visit()
print(count)                          # 3
```

11. Mistakes that cost marks

- Forgetting the colon at the end of the def line, or forgetting to indent the body.
- Defining the function but never calling it — nothing is printed and no marks are given.
- Calling the function before it is defined, which gives a NameError.
- Confusing print with return. A function that only prints returns None.
- Writing a non-default parameter after a default one : def f(a = 1, b) is an error.
- Writing a positional argument after a keyword argument : f(name = 'A', 12) is an error.
- Expecting a global variable to change without writing the global statement.
- Using a local variable outside its function, which gives a NameError.
- Writing statements after return inside the same block — they never run.
- Giving a different number of arguments than the parameters, when no default is provided.