Python Foundations

Industrial Reference Guide

Strings · Lists · Dictionaries · Sets · Functions · Loops · Modules

Edge Cases · Performance · Best Practices · Pitfalls · Optimization

BEGINNER FRIENDLY | INDUSTRY STANDARD | EDGE CASES COVERED | PERFORMANCE TIPS

Table of Contents

CHAPTER 01

Strings & String Methods

Python strings are immutable sequences of Unicode characters. Every operation returns a new string — the original is never modified.

What is a String?

A string in Python is a sequence of characters enclosed in single quotes, double quotes, or triple quotes. Triple-quoted strings can span multiple lines and preserve whitespace and newlines exactly.

String Creation

```
# Three ways to create strings
s1 = 'Hello'
s2 = "World"
s3 = '''Multi
line'''

# f-strings (Python 3.6+) — preferred for interpolation
name = "Waseem"
greeting = f"Hello {name}!" # Hello Waseem!
```

String Indexing & Slicing

Strings support zero-based indexing. Negative indices count from the end. Slicing uses the syntax **string[start:end:step]** — none of the bounds are mandatory.

Indexing & Slicing

```
s = "Hello World"

# Positive indexing
s[0] # 'H' (first char)
s[4] # 'o'

# Negative indexing
s[-1] # 'd' (last char)
s[-5] # 'W'

# Slicing [start:end:step]
s[0:5] # 'Hello' (0 to 4, end is exclusive)
s[6:] # 'World' (6 to end)
s[:5] # 'Hello' (start to 4)
s[::2] # 'HloWrd' (every 2nd char)
s[::-1] # 'dlroW olleH' (reversed!)
s[0:10:3] # 'HlWl'
```

■■ WARNING Remember: slicing NEVER raises an IndexError even if the index is out of range. "Hello"[100:200] returns "" (empty string). Direct indexing DOES raise IndexError.

Common String Methods

These are the most frequently used string methods in industry code:

Method	What It Does	Returns	Mutates?
.upper()	Converts to uppercase	New str	No
.lower()	Converts to lowercase	New str	No
.strip()	Removes leading/trailing whitespace	New str	No
.lstrip() / .rstrip()	Strip from left or right only	New str	No
.replace(old, new)	Replaces all occurrences	New str	No
.find(sub)	First index of sub, -1 if not found	int	No
.index(sub)	First index, raises ValueError if missing	int	No
.count(sub)	Number of non-overlapping occurrences	int	No
.split(sep)	Splits into a list (default: whitespace)	list	No
.join(iterable)	Joins iterable with string as separator	New str	No
.startswith(prefix)	True if string starts with prefix	bool	No
.endswith(suffix)	True if string ends with suffix	bool	No
.capitalize()	First char upper, rest lower	New str	No
.title()	Title Case Every Word	New str	No
.isdigit()	True if all chars are digits	bool	No
.isalpha()	True if all chars are letters	bool	No
.isalnum()	True if letters or digits only	bool	No
.zfill(width)	Pads with zeros on the left	New str	No
.center(w, char)	Centers string within width	New str	No

String Methods in Action

```
# Practical examples
email = " Waseem@Email.COM "
clean = email.strip().lower() # "waseem@email.com"

# find vs index — know the difference!
txt = "Hello World"
txt.find("xyz") # -1 (safe — no crash)
# txt.index("xyz") # raises ValueError!

# split and join — powerful combo
csv = "apple,banana,cherry"
fruits = csv.split(",") # ["apple", "banana", "cherry"]
rejoined = " | ".join(fruits) # "apple | banana | cherry"

# startswith/endswith accept tuples!
filename = "report.pdf"
filename.endswith((".pdf", ".docx")) # True
```

Edge Cases & Gotchas

Common Pitfalls

```
# Bug: startsWithString was checking endsWithString variable
# (from your code — classic copy-paste bug)
endsWithString = letter.endswith("|>")
startsWithString = letter.startswith("Dear")
print(endsWithString) # This was the bug — wrong variable!
print(startsWithString) # This is correct

# capitalize() lowercases everything except first char
"hELLO WORLD".capitalize() # "Hello world" (not "Hello World"!)
"hELLO WORLD".title() # "Hello World" (correct for titles)

# count() is non-overlapping
"aaa".count("aa") # 1, not 2!

# Immutability means reassignment is needed
name = "waseem"
name.upper() # Does nothing useful without assignment!
name = name.upper() # Correct
```

■ AVOID Never chain string operations without assigning: "hello".upper().replace("E","e") works, but "hello".upper() alone produces output you discard. Always capture the return value.

■ TIP For building strings in a loop, use a list and join at the end. result = []; for x in items: result.append(str(x)); final = "".join(result) — this is O(n) vs O(n²) with += string concatenation in a loop.

Performance & Optimization

Operation	Performance	Recommendation
s1 + s2 in loop	O(n²) — slow	Use list + join()
f-string formatting	Fastest	■ Preferred
% formatting	Slow, legacy	Avoid
.format()	Moderate	OK, verbose
in operator (search)	O(n)	Use for simple checks
re module (regex)	Slower	Use for complex patterns

CHAPTER 02

Lists & List Methods

Lists are ordered, mutable sequences. They can hold mixed types and support in-place modification. Understanding which methods mutate vs return new lists is critical.

Creating Lists

List Creation

```
# Various ways to create lists
myList = [10, 12, 34, 4, 5]
empty = []
mixed = [1, "hello", 3.14, True, None]

# List comprehension (Pythonic & fast)
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]
flat = [item for sub in [[1,2],[3,4]] for item in sub]

# From other iterables
from_str = list("hello") # ["h","e","l","l","o"]
from_range= list(range(5)) # [0, 1, 2, 3, 4]
```

Mutating Methods (modify in-place, return None)

Mutating List Methods

```
myList = [10, 12, 34, 4, 5]

myList.sort() # [4, 5, 10, 12, 34] — ascending sort in-place
myList.sort(reverse=True) # [34, 12, 10, 5, 4] — descending
myList.sort(key=lambda x: -x) # custom sort key

myList.reverse() # reverses in-place — returns None!

myList.append(3) # adds to END: O(1) amortised
myList.extend([6, 7]) # adds multiple items: O(k)

myList.insert(0, 1) # inserts at index 0: O(n) — SLOW for large lists

myList.remove(10) # removes FIRST occurrence of VALUE 10
# raises ValueError if not found!

myList.pop() # removes & returns LAST element: O(1)
myList.pop(1) # removes & returns element at INDEX 1: O(n)

myList.clear() # empties the list
```

■■ WARNING All mutating methods return None. A very common bug: myList = myList.sort() → myList becomes None! Call sort() on its own line, never assign its return value.

Non-Mutating Methods (return new value)

Non-Mutating List Operations

```
myList = [4, 5, 10, 12, 34]
```

```
sorted_copy = sorted(myList) # new sorted list
sorted_desc = sorted(myList, reverse=True)
reversed_copy = list(reversed(myList)) # new reversed list

idx = myList.index(10) # returns index of value (raises ValueError if absent)
cnt = myList.count(5) # number of occurrences
n = len(myList) # length of list

copy1 = myList.copy() # shallow copy
copy2 = myList[:] # also shallow copy via slice

import copy
deep = copy.deepcopy(myList) # deep copy (for nested lists)
```

Shallow vs Deep Copy — Critical Edge Case

Shallow vs Deep Copy

```
# Shallow copy — nested objects are SHARED
original = [[1, 2], [3, 4]]
shallow = original.copy()
shallow[0].append(99)
print(original) # [[1, 2, 99], [3, 4]] ← original is affected!

# Deep copy — fully independent
import copy
deep = copy.deepcopy(original)
deep[0].append(100)
print(original) # [[1, 2, 99], [3, 4]] ← original NOT affected
```

■ AVOID Never use a mutable default argument like def func(lst=[]). The list is created ONCE and shared across all calls. Use def func(lst=None): if lst is None: lst = []

Performance Reference

Operation	Time Complexity	Notes
append(x)	O(1) amortized	Very fast
pop() — last	O(1)	Very fast
pop(i) — middle	O(n)	Shifts elements
insert(i, x)	O(n)	Avoid in hot loops
remove(x)	O(n)	Searches linearly
x in list	O(n)	Use set for O(1)
list[i]	O(1)	Random access is fast
sort()	O(n log n)	Timsort — very efficient
len()	O(1)	Stored, not computed
list + list	O(n+m)	Creates new list

■ BEST PRACTICE Use collections.deque for O(1) append AND popleft. Lists are O(n) for operations at the front. For membership testing on large data, convert to a set first.

CHAPTER 03

Dictionaries

Dictionaries are ordered (Python 3.7+), mutable key-value stores. They provide O(1) average-case lookups using hash tables.

Creating & Accessing

Dictionary Access

```
myDictionary = {
"name": "Hafiz Waseem Ahmed",
"email": "waseem@email.com",
"tasks": ["Work Hard", "Develop Skills"]
}

# Access — two ways
myDictionary["name"] # "Hafiz Waseem Ahmed" (raises KeyError if missing)
myDictionary.get("name") # "Hafiz Waseem Ahmed" (returns None if missing)
myDictionary.get("age", 0) # 0 (default value if key missing)

# Nested access
myDictionary["tasks"][0] # "Work Hard"

# Safe nested access
age = myDictionary.get("age") # None — no crash
```

■■ WARNING Using dict["key"] raises a KeyError if the key does not exist. In production code, always prefer .get() with a default, or check with the in operator first.

CRUD Operations

Dictionary CRUD

```
# CREATE / UPDATE
myDictionary["age"] = 22 # set new key
myDictionary.update({"city": "Quetta"}) # update with dict
myDictionary.update(country="Pakistan") # keyword syntax

# READ
myDictionary.keys() # dict_keys([...]) — all keys
myDictionary.values() # dict_values([...]) — all values
myDictionary.items() # dict_items([...]) — (key, value) pairs

# DELETE
myDictionary.pop("password") # removes & returns value (safe)
del myDictionary["tasks"] # removes key (raises KeyError if missing)
myDictionary.popitem() # removes & returns last inserted pair
myDictionary.clear() # empties dictionary
```

Iteration Patterns

Dictionary Iteration

```
user = {"name": "Waseem", "age": 22, "city": "Quetta"}

# Iterate over keys (default)
for key in user:
print(key)

# Iterate over values
for value in user.values():
print(value)

# Iterate over key-value pairs — MOST COMMON
for key, value in user.items():
print(f"{key}: {value}")

# Sorted iteration
for key in sorted(user):
print(key, user[key])

# Reversed iteration (Python 3.8+)
for key in reversed(user):
print(key, user[key])
```

Nested Dictionaries & Lists in Dicts

Nested Structures

```
data = {
1: {"name": "Waseem"},
2: {"scores": [85, 90, 78]}
}

# Add to a nested list
data[2]["scores"].append(100) # [85, 90, 78, 100]

# Add a new nested dict
data.update({3: {"age": 23}})

# List of dicts — very common pattern
users = [
{"name": "Waseem", "age": 22},
{"name": "Lamiah", "age": 25},
]
users.append({"name": "Summiya", "age": 26})

for user in users:
print(f'Name: {user["name"]}, Age: {user["age"]}')
```

dict Methods Reference

Method	Returns	Raises if key missing?
d[key]	value	Yes — KeyError
d.get(key, default)	value or default	No — returns default
d.pop(key)	value, removes it	Yes — KeyError
d.pop(key, default)	value or default, removes it	No

d.update({...})	None	No
d.keys()	dict_keys view	No
d.values()	dict_values view	No
d.items()	dict_items view (pairs)	No
d.setdefault(k, v)	value (sets if missing)	No
d.copy()	shallow copy	No
del d[key]	None	Yes — KeyError

■ TIP Use dict.setdefault(key, []).append(value) to group items without checking if a key exists first. Or use collections.defaultdict(list) for even cleaner grouping.

CHAPTER 04

Sets

Sets are unordered collections of unique elements backed by a hash table. They offer O(1) membership testing and powerful mathematical set operations.

Creating Sets

Set Creation

```
# Creating sets
mySet = {1, 2, 3, 4, 5}
empty = set() # NOT {} — that creates an empty dict!

# From other iterables
from_list = set([1, 2, 2, 3, 3]) # {1, 2, 3} — duplicates removed
from_str = set("hello") # {"h", "e", "l", "o"}

# Mixed types are allowed (must be hashable)
newSet = set()
newSet.add(18)
newSet.add("18") # {18, "18"} — int and str are different!
```

Set Methods

Set Operations

```
mySet = {1, 2, 3, 4, 5}

mySet.add(6) # adds element (no-op if already present)
mySet.remove(4) # removes element — raises KeyError if missing!
mySet.discard(99) # removes element — NO error if missing (safer)
mySet.pop() # removes & returns ARBITRARY element (order undefined)
mySet.clear() # empties the set
n = len(mySet) # number of elements

# Set operations
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

A.union(B) # {1,2,3,4,5,6} — all elements from both
A | B # same as union (operator syntax)

A.intersection(B) # {3, 4} — elements in BOTH
A & B # same as intersection

A.difference(B) # {1, 2} — in A but NOT in B
A - B # same as difference

A.symmetric_difference(B) # {1,2,5,6} — in one but not both
A ^ B # same

A.issubset(B) # Is every element of A in B?
A.issuperset(B) # Does A contain all elements of B?
A.isdisjoint(B) # Do A and B share no elements?
```

■■ WARNING Sets are UNORDERED. mySet.pop() removes a random element — never rely on which one. Do not use sets when order matters. Also: unhashable types (lists, dicts) CANNOT be added to a set.

When to Use Sets — Industry Guide

Use Case	Why Sets Win	Example
Deduplication	O(n) vs O(n²) with list	unique = list(set(data))
Membership test	O(1) vs O(n) for list	if user in active_users_set
Finding common items	One line with &	common = setA & setB
Removing specific values	discard() is safe	ids.discard(old_id)
Tag / category systems	Natural fit	article.tags = {"python", "code"}

CHAPTER 05

Functions

Functions are reusable, named blocks of code. Python functions are first-class objects — they can be passed as arguments, returned from other functions, and stored in variables.

Function Anatomy

Function Syntax

```
# Basic function
def nameFunc(name):
greeting = "Hello " + name
return greeting

# Default parameters
def nameFunc(name="user"): # called with no arg → "Hello user"
return f"Hello {name}"

# Multiple parameters
def add(a, b=0):
return a + b

add(3) # 3 (b defaults to 0)
add(3, 4) # 7
add(b=4, a=3) # 7 (keyword arguments, order does not matter)

# *args — variable positional arguments
def sumAll(*args):
return sum(args)
sumAll(1, 2, 3, 4) # 10

# **kwargs — variable keyword arguments
def printInfo(**kwargs):
for k, v in kwargs.items():
print(f"{k}: {v}")
printInfo(name="Waseem", age=22)
```

Recursion

Recursion is when a function calls itself. Every recursive function needs a base case (the condition that stops recursion) and a recursive case (the call that reduces the problem).

Recursion Example — Factorial

```
def factorialFunction(number):
if number == 1 or number == 0: # BASE CASE — stops recursion
return 1
else:
return number * factorialFunction(number - 1) # RECURSIVE CASE

# Execution trace for factorialFunction(4):
# 4 * factorialFunction(3)
```

```
# 4 * 3 * factorialFunction(2)
# 4 * 3 * 2 * factorialFunction(1)
# 4 * 3 * 2 * 1 → 24
```

■■ WARNING Python has a default recursion limit of 1000 calls (sys.getrecursionlimit()). Deep recursion causes a RecursionError. For large inputs, prefer iterative solutions or use functools.lru_cache to memoize recursive results.

Scope — LEGB Rule

Variable Scope

```
# L — Local: inside current function
# E — Enclosing: outer function (closures)
# G — Global: module level
# B — Built-in: Python built-ins (len, print...)

x = "global"

def outer():
x = "enclosing"
def inner():
x = "local"
print(x) # "local"
inner()
print(x) # "enclosing"

outer()
print(x) # "global"

# global keyword — modify global from inside function
count = 0
def increment():
global count
count += 1
```

■ AVOID Avoid using global variables. They create hidden dependencies, make testing hard, and cause bugs in concurrent code. Pass values as parameters and return results instead.

Lambda Functions

Lambda Functions

```
# Lambda = anonymous single-expression function
square = lambda x: x ** 2
add = lambda a, b: a + b

# Common use: key for sorting
users = [{"name": "Zara", "age": 30}, {"name": "Ali", "age": 22}]
users.sort(key=lambda u: u["age"]) # sort by age

# With map/filter
nums = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x*2, nums)) # [2,4,6,8,10]
evens = list(filter(lambda x: x%2==0, nums)) # [2, 4]
```

```
# Tip: list comprehensions are usually clearer
doubled = [x*2 for x in nums] # preferred over map+lambda
```

CHAPTER 06

Loops & Control Flow

Python has two loop types: for (iteration) and while (condition-based). Both support break, continue, pass, and an else clause.

for Loops

for Loop Patterns

```
# Iterate over a list
for item in [1, 2, 3]:
print(item)

# range() — most common for index-based loops
for i in range(5): # 0, 1, 2, 3, 4
print(i)

for i in range(1, 11): # 1 to 10
print(i)

for i in range(0, 10, 2): # 0, 2, 4, 6, 8 (step=2)
print(i)

for i in reversed(range(1, 11)): # 10, 9, 8... 1
print(i)

# enumerate() — index + value
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits, start=1):
print(f"{index}. {fruit}")

# zip() — iterate two lists in parallel
names = ["Ali", "Sara"]
scores = [90, 85]
for name, score in zip(names, scores):
print(f"{name}: {score}")
```

while Loops

while & Control Flow

```
# Basic while
i = 1
while i < 11:
print(i)
i += 1 # CRITICAL: must update condition, else infinite loop!

# while True with break — common for menus/user input
while True:
choice = input("Enter Y or N: ")
if choice in ("Y", "N"):
break
```

```
print("Invalid, try again")

# continue — skip rest of loop body, go to next iteration
for i in range(10):
if i % 2 == 0:
continue # skip even numbers
print(i) # prints 1, 3, 5, 7, 9

# pass — do nothing placeholder
for i in range(100):
pass # useful when syntax requires a body but you have none yet
```

The for/while else Clause — Python's Unique Feature

Loop else Clause

```
# else runs if loop completed WITHOUT hitting break
number = 17
for i in range(2, int(number**0.5) + 1):
if number % i == 0:
print(f"{number} is NOT prime")
break
else:
print(f"{number} is prime") # runs because break was never hit
```

■ TIP The for/else pattern is very useful for search operations. If you find what you are looking for, break. If the loop exhausts without finding it, the else block tells you it was not found. No need for a separate "found" flag variable.

CHAPTER 07

The random Module

Python's random module provides pseudo-random number generation. It is suitable for simulations and games, but NOT for cryptographic security.

random Module Methods

```
import random

random.random() # float in [0.0, 1.0)
random.uniform(1.5, 3.5) # float in [a, b]
random.randint(1, 100) # int in [1, 100] — BOTH endpoints included
random.randrange(0, 10) # int in [0, 10) — like range()
random.randrange(0, 10, 2) # even numbers: 0,2,4,6,8

colors = ["red", "green", "blue"]
random.choice(colors) # single random element
random.choices(colors, k=5) # 5 picks WITH repetition
random.choices(colors, weights=[1,2,1], k=10) # weighted choices
random.sample(colors, k=2) # 2 picks WITHOUT repetition

cards = [1, 2, 3, 4, 5]
random.shuffle(cards) # shuffles IN-PLACE (like list.sort)

# Seeding — makes results reproducible
random.seed(10)
print(random.randint(1, 10)) # always same number for same seed
# Useful for: testing, debugging, reproducible experiments
```

Method	Type	Range	Repetition?
random()	float	[0.0, 1.0)	N/A
uniform(a, b)	float	[a, b]	N/A
randint(a, b)	int	[a, b] inclusive	N/A
randrange(a,b,s)	int	[a, b) with step	N/A
choice(seq)	element	from sequence	N/A
choices(seq, k=n)	list	from sequence	Yes
sample(seq, k=n)	list	from sequence	No
shuffle(seq)	None	in-place	N/A

■ AVOID NEVER use the random module for passwords, tokens, or security keys. Use the secrets module instead: secrets.token_hex(32) or secrets.randbelow(n). The random module is deterministic and can be predicted.

CHAPTER 08

The string Module

The string module provides pre-built character set constants — perfect for password generators, validators, and text processors.

string Module Constants & Password Generator

```
import string as st

st.ascii_lowercase # "abcdefghijklmnopqrstuvwxyz"
st.ascii_uppercase # "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
st.ascii_letters # lowercase + uppercase combined
st.digits # "0123456789"
st.punctuation # '!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
st.whitespace # space, tab, newline, carriage return, etc.
st.printable # all printable characters combined

# Password generator using string + random
import random

def generatePassword(length, uppercase=True, digits=True, special=True):
pool = st.ascii_lowercase
if uppercase: pool += st.ascii_uppercase
if digits: pool += st.digits
if special: pool += st.punctuation
return "".join(random.choices(pool, k=length))

print(generatePassword(16)) # e.g. "aB3!xZ9#qW2&mR5;@"
```

■ TIP

Your password generator uses random.choices which allows repetition — good for passwords. The issue in your code: "Y".lower() evaluates to "y", so askUppercase == "Y".lower() means you are comparing against "y", not "Y". This is intentional to accept both cases, but it is cleaner to write: if askUppercase.lower() == "y"

CHAPTER 09

Mutable vs Immutable

One of the most important concepts in Python. Immutable objects cannot be changed after creation. Mutable objects can be modified in-place.

The Core Distinction

Type	Mutable?	Can be dict key?	JS Equivalent
int, float, bool	No	Yes	Number (primitive)
str	No	Yes	String (primitive)
tuple	No	Yes (if contents immutable)	Object.freeze([...])
frozenset	No	Yes	No direct equivalent
list	Yes	No	Array (object)
dict	Yes	No	Object / Map
set	Yes	No	Set

Mutability in Action

```
# IMMUTABLE — every "change" creates a new object
name = "waseem"
id_before = id(name)
name = name.upper() # new object created, name now points to it
id_after = id(name)
print(id_before == id_after) # False — different objects!

# MUTABLE — changes happen in-place, same object
lst = [1, 2, 3]
id_before = id(lst)
lst.append(4) # modifies existing object
id_after = id(lst)
print(id_before == id_after) # True — same object!

# This is why mutable default args are dangerous
def bad(lst=[]): # lst created ONCE at definition time
lst.append(1)
return lst
bad() # [1]
bad() # [1, 1] — accumulated!
bad() # [1, 1, 1]
```

Python vs JavaScript Comparison

Concept	Python	JavaScript
Immutable string ops	Always return new str	Always return new str

Array sort (in-place)	list.sort() — returns None	array.sort() — returns array
Array reverse	list.reverse() — None	array.reverse() — returns array
Non-mutating sort	sorted(list)	[...array].sort()
Non-mutating reverse	list[::-1]	[...array].reverse()
Check membership	x in list (O(n))	array.includes(x) (O(n))
Fast membership	x in set (O(1))	Set.has(x) (O(1))
Dict/Object methods	.items(), .keys()	Object.entries(), Object.keys()

■■ WARNING Critical difference from JS: Python's list.sort() and list.reverse() return None, not the list. In JS, array.sort() returns the sorted array. This is one of the most common bugs when switching between the two languages.

CHAPTER 10

Real-World Mini Projects

Analysis and improvements of your actual code — calculator, To-Do app, number analyzer.

Calculator — Analysis & Improvements

Improved Calculator

```
# Your calculator has an infinite loop with no exit.
# Industry improvement: add exit option + error handling

while True:
try:
a = float(input("Enter first number (or q to quit): "))
except ValueError:
print("Bye!")
break

op = input("+ - * /: ").strip()
b = float(input("Enter second number: "))

ops = {",": a+b, "-": a-b, "*": a*b}
if op in ops:
print(f"Result: {ops[op]}")
elif op == "/":
if b == 0:
print("Error: Division by zero!")
else:
print(f"Result: {a/b}")
else:
print("Invalid operator")
```

To-Do App — Analysis

Your To-Do app is well-structured! Here are observations and improvements:

To-Do App Review

```
# Good practices you used:
# ■ .strip().lower() for clean input comparison
# ■ Checking for empty list before operations
# ■ Returning early to avoid nested ifs
# ■ enumerate() for numbered display

# Issues to fix:
# ■■ Modifying a list while iterating over it (removeTask)
# ■■ Using tasks.remove(task) inside a for loop over tasks is unsafe

# Safer remove pattern:
def removeTask(tasks):
task_name = input("Task to remove: ").strip().lower()
tasks[:] = [t for t in tasks if t["task"] != task_name]
```

```
# tasks[:] modifies in-place (same list object, important for callers)
```

Number Analyzer — Bug Fix

Number Analyzer Fix

```
# Bug in your code: analyzeNumbers result is not printed
# Fix in the main loop:
elif startingMessage == 2:
if not numList:
print("No numbers yet!")
else:
result = analyzeNumbers(numList) # capture return value!
print("===== Analysis =====")
for key, value in result.items():
print(f" {key}: {value}")
```

General Best Practices Summary

Area	Do	Avoid
Input validation	try/except + strip()	Bare int(input())
Dictionary access	.get(key, default)	dict[key] without try/except
List modification	List comprehension or copy	Modify list while iterating
String building	"".join(list)	str += str in a loop
Membership test	Use set for O(1)	if x in large_list
Function defaults	def f(lst=None)	def f(lst=[])
Variable naming	snake_case	camelCase (JS style)
Error handling	Specific exceptions	Bare except:
Recursion	Add base case first	Forget termination condition

■ BEST PRACTICE You are building great habits: using f-strings, .strip().lower() for input normalization, checking list length before operations, and using enumerate() for indexed loops. Keep going — you are progressing very well!

Keep Building. Keep Breaking Things.

Every expert was once a beginner who refused to give up.

Python Foundations — Industrial Reference Guide
Strings · Lists · Dicts · Sets · Functions · Loops · Modules