Alfonso Hidalgo
Data Analyst
Python Study Guide

Focused on highest-impact topics for data analysis in Python

Contents

1. Pandas — Data Analysis

Pandas is the core library for data manipulation in Python. Data Analyst roles typically involve reading, filtering, merging, grouping, and transforming DataFrames. All examples assume `import pandas as pd`.

1.1 — Create & Inspect DataFrames

Problem: Create a DataFrame from raw data and explore its structure. Always the first step before any analysis.

```
import pandas as pd

# Create from dictionary — each key becomes a column
df = pd.DataFrame({
    'name':   ['Ana', 'Bob', 'Cara', 'Dan', 'Eve'],
    'dept':   ['Eng', 'Sales', 'Eng', 'Sales', 'Eng'],
    'salary': [90000, 75000, 95000, 72000, 88000],
    'age':    [28, 34, 31, 29, 26]
})

# ■■ Essential inspection methods ■■
df.shape          # (5, 4) — rows, columns
df.columns        # column names as Index
df.dtypes         # data type per column
df.head(3)        # first 3 rows
df.tail(2)        # last 2 rows
df.describe()     # count, mean, std, min, quartiles, max
df.info()         # dtypes + non-null counts (spot missing data)
df.nunique()      # unique value count per column
df.isnull().sum() # count of NaN per column
df.value_counts('dept')  # frequency of each dept value
```

> ***Key Takeaway:*** *Always start with .shape, .dtypes, .isnull().sum() to understand data before analysis. These three calls reveal size, types, and data quality in seconds.*

1.2 — Filtering & Selection

Problem: Select specific rows and columns based on conditions. The most commonly tested Pandas skill — appears in nearly every technical exercise.

```
# ■■ Column selection ■■
df['name']                  # single column -> Series
df[['name', 'salary']]      # multiple columns -> DataFrame

# ■■ Row filtering by condition ■■
df[df['dept'] == 'Eng']             # exact match
df[df['salary'] > 80000]            # numeric comparison
df[df['name'].str.startswith('A')]  # string condition

# ■■ Multiple conditions ■■
# MUST use & (and), | (or), ~ (not)
# MUST wrap each condition in parentheses
df[(df['dept'] == 'Eng') & (df['salary'] > 80000)]
df[(df['dept'] == 'Eng') | (df['dept'] == 'HR')]
df[~(df['dept'] == 'Sales')]  # everyone except Sales

# ■■ isin() for matching against a list of values ■■
df[df['dept'].isin(['Eng', 'HR', 'Data'])]

# ■■ between() for numeric ranges (inclusive) ■■
df[df['age'].between(25, 30)]  # 25 <= age <= 30

# ■■ String contains (case-insensitive) ■■
df[df['name'].str.contains('an', case=False, na=False)]

# ■■ loc vs iloc ■■
df.loc[0:2, 'name':'salary']   # label-based, INCLUSIVE end
df.iloc[0:2, 0:3]              # position-based, EXCLUSIVE end
```

> ***Watch Out:*** *Use & | ~ for Pandas boolean ops, NOT Python's 'and' 'or' 'not'. Always wrap each condition in parentheses or you'll get operator precedence errors.*

1.3 — GroupBy & Aggregation

Problem: Group rows by one or more columns and compute aggregate statistics. This is the Pandas equivalent of SQL's GROUP BY — core to any data analyst task.

```
# ■■ Basic: one column, one aggregation ■■
df.groupby('dept')['salary'].mean()
# dept
# Eng      91000.0
# Sales    73500.0

# ■■ Multiple aggregations on same column ■■
df.groupby('dept')['salary'].agg(['mean', 'min', 'max', 'sum', 'count'])

# ■■ Different aggregations for different columns ■■
df.groupby('dept').agg({
    'salary': ['mean', 'sum'],
    'age': 'mean',
    'name': 'count'
})

# ■■ Named aggregations (clean column names in output) ■■
result = df.groupby('dept').agg(
    avg_salary=('salary', 'mean'),
    total_salary=('salary', 'sum'),
    headcount=('name', 'count'),
    youngest=('age', 'min'),
    oldest=('age', 'max')
).reset_index()  # group key becomes a regular column

# ■■ Group by multiple columns ■■
df.groupby(['dept', 'age'])['salary'].mean()

# ■■ Filter groups: keep depts with avg salary > 80k ■■
df.groupby('dept').filter(lambda g: g['salary'].mean() > 80000)
```

> ***Key Takeaway:*** *.reset_index() after groupby converts the group key from index to column. Named agg (new_name=('col', 'func')) produces clean output. .filter() keeps/removes entire groups based on a condition.*

1.4 — Merge / Join DataFrames

Problem: Combine two DataFrames on a shared key — SQL JOIN in Pandas. Essential for enriching data from multiple tables.

```
users = pd.DataFrame({
    'user_id': [1, 2, 3, 4],
    'name': ['Ana', 'Bob', 'Cara', 'Dan']
})
txns = pd.DataFrame({
    'user_id': [1, 1, 2, 5],
    'amount': [100, 200, 50, 300]
})

# INNER: only rows with matching keys in BOTH tables
pd.merge(users, txns, on='user_id', how='inner')
# Users 1 & 2 only. Cara, Dan (no txns) and user 5 (no profile) excluded

# LEFT: all rows from left table, NaN where no match on right
pd.merge(users, txns, on='user_id', how='left')
# All 4 users. Cara & Dan have NaN amounts

# RIGHT: all rows from right table, NaN where no match on left
pd.merge(users, txns, on='user_id', how='right')
# user_id 5 appears with NaN name

# OUTER: all rows from both sides, NaN where no match
pd.merge(users, txns, on='user_id', how='outer')
```

```
# Different column names? Use left_on / right_on
pd.merge(df1, df2, left_on='id', right_on='user_id')

# Merge on multiple keys (same column names in both tables)
pd.merge(df1, df2, on=['user_id', 'date'], how='left')

# Merge on multiple keys (different column names across tables)
# left_on and right_on match columns positionally:
# user_id <-> customer_id, date <-> order_date
pd.merge(df1, df2,
    left_on=['user_id', 'date'],
    right_on=['customer_id', 'order_date'],
    how='left')
```

> ***Key Takeaway:*** *how='inner' keeps only matches (default). how='left' keeps all left rows. After a left join, check for NaN in the right-side columns to find unmatched rows. This is how you find 'users with no transactions'.*

1.5 — Pivot Tables

Problem: Reshape data into a summary table with one category as rows, another as columns, and aggregated values in cells.

```
sales = pd.DataFrame({
    'region':  ['North','North','South','South','North','South'],
    'product': ['A','B','A','B','A','A'],
    'revenue': [100, 150, 200, 120, 180, 90]
})

# Average revenue by region x product
pivot = sales.pivot_table(
    values='revenue',     # what to aggregate
    index='region',       # row labels
    columns='product',    # column labels
    aggfunc='mean'        # aggregation function
)
# product       A      B
# region
# North     140.0  150.0
# South     145.0  120.0

# Add margins (totals row + column)
sales.pivot_table(
    values='revenue', index='region',
    columns='product', aggfunc='sum',
    margins=True, fill_value=0  # fill missing combos with 0
)
```

1.6 — Apply & Lambda

Problem: Apply custom transformations when built-in methods aren't enough. Use .apply() for row-wise or column-wise logic.

```
# ■■ Column-level apply (Series.apply) ■■
# Categorize salary into bands
df['band'] = df['salary'].apply(
    lambda x: 'Senior' if x >= 90000 else 'Junior'
)

# ■■ Row-level apply (axis=1 means each row) ■■
def categorize(row):
    # Access multiple columns for complex logic
    if row['dept'] == 'Eng' and row['salary'] > 90000:
        return 'Senior Eng'
    elif row['dept'] == 'Eng':
        return 'Junior Eng'
    else:
        return row['dept']

df['category'] = df.apply(categorize, axis=1)

# ■■ Vectorized alternative (faster, preferred) ■■
import numpy as np
df['band'] = np.where(df['salary'] >= 90000, 'Senior', 'Junior')

# ■■ Multiple conditions with np.select ■■
conditions = [
    df['salary'] >= 95000,
    df['salary'] >= 85000,
]
choices = ['Senior', 'Mid']
df['level'] = np.select(conditions, choices, default='Junior')
```

> ***Key Takeaway:*** *Prefer np.where() for simple if/else and np.select() for multiple conditions — both are much faster than .apply(). Use .apply(axis=1) only when you need complex multi-column logic that can't be vectorized.*

1.7 — Handling Missing Data

Problem: Detect, fill, or remove NaN values. Real-world data always has gaps, and real-world data will too.

```
# ■■ Detect ■■
df.isnull()             # Boolean DataFrame (True = missing)
```

```
df.isnull().sum()       # count NaN per column
df.isnull().any()       # True/False: does column have any NaN?
df.notnull()            # opposite of isnull()

# ■■ Drop rows with missing values ■■
df.dropna()                    # drop rows with ANY NaN
df.dropna(subset=['salary'])   # only drop if salary is NaN
df.dropna(thresh=3)            # keep rows with >= 3 non-NaN values

# ■■ Fill missing values ■■
df['salary'].fillna(0)                     # fill with constant
df['salary'].fillna(df['salary'].mean())   # fill with column mean
df['salary'].fillna(df['salary'].median()) # fill with median
df.fillna(method='ffill')  # forward fill (carry last valid value)
df.fillna(method='bfill')  # backward fill

# ■■ Fill NaN per group (e.g., fill with dept average) ■■
df['salary'] = df.groupby('dept')['salary'].transform(
    lambda x: x.fillna(x.mean())
)

# ■■ Replace specific values with NaN ■■
df.replace({'N/A': np.nan, 'missing': np.nan, -1: np.nan})
```

> ***Watch Out:*** *fillna() does NOT modify the original DataFrame unless you pass inplace=True or reassign: df['col'] = df['col'].fillna(0). Same applies to dropna().*

1.8 — String Operations on Columns

Problem: Clean and transform text columns. The .str accessor gives you access to all Python string methods on entire columns at once.

```
# ■■ Common .str methods ■■
df['name'].str.lower()           # 'Ana' -> 'ana'
df['name'].str.upper()           # 'Ana' -> 'ANA'
df['name'].str.strip()           # remove leading/trailing spaces
df['name'].str.replace(' ', '_') # replace characters
df['name'].str.len()             # length of each string

# ■■ Boolean string checks (return masks for filtering) ■■
df['name'].str.contains('an', case=False)  # contains substring?
df['name'].str.startswith('A')             # starts with?
df['name'].str.endswith('a')               # ends with?
df['name'].str.isnumeric()                 # all digits?

# ■■ Split into multiple columns ■■
# 'Ana Smith' -> first='Ana', last='Smith'
df[['first', 'last']] = df['full_name'].str.split(' ', expand=True)

# ■■ Extract with regex ■■
df['digits'] = df['text'].str.extract(r'(\d+)')  # first number
df['all_digits'] = df['text'].str.findall(r'\d+')  # all numbers

# ■■ Chaining string operations ■■
df['clean'] = df['name'].str.strip().str.lower().str.replace(' ', '_')
```

> ***Key Takeaway:*** *.str methods are vectorized — they apply to the entire column at once. Always add na=False to .str.contains() if the column might have NaN values.*

1.9 — Sorting & Ranking

Problem: Sort rows or assign rank numbers within groups.

```
# ■■ Sort by one column ■■
df.sort_values('salary', ascending=False)  # highest first

# ■■ Sort by multiple columns (different directions) ■■
df.sort_values(
    ['dept', 'salary'],
    ascending=[True, False]  # dept A-Z, then salary high-to-low
)

# ■■ Rank within groups ■■
# Rank employees by salary within each department
df['dept_rank'] = df.groupby('dept')['salary'].rank(
    ascending=False,  # highest salary = rank 1
    method='dense'    # no gaps: 1, 2, 3 (not 1, 2, 4)
)

# ■■ Top N per group ■■
top2 = df.sort_values('salary', ascending=False) \
         .groupby('dept').head(2)  # top 2 earners per dept

# ■■ Bottom N per group ■■
bottom1 = df.sort_values('salary') \
            .groupby('dept').head(1)  # lowest earner per dept

# ■■ nlargest / nsmallest (more explicit) ■■
df.nlargest(5, 'salary')   # top 5 by salary
df.nsmallest(3, 'age')     # 3 youngest
```

> ***Key Takeaway:*** *rank() method options: 'dense' (no gaps), 'min' (SQL-style, ties get lowest rank), 'average' (ties get mean rank). For top-N per group, sort then groupby().head(n).*

1.10 — Window Functions (Rolling, Cumulative, Shift)

Problem: Compute running totals, moving averages, or compare rows to group stats. Pandas equivalents of SQL window functions like SUM() OVER, LAG(), LEAD().

```
# ■■ Cumulative sum (running total) ■■
df['running_total'] = df['revenue'].cumsum()

# ■■ Cumulative max / min ■■
df['best_so_far'] = df['revenue'].cummax()

# ■■ Rolling average (moving window) ■■
df['ma_3'] = df['revenue'].rolling(window=3).mean()  # 3-period MA
df['ma_7'] = df['revenue'].rolling(window=7, min_periods=1).mean()
# min_periods=1 avoids NaN at the start of the series

# ■■ Percentage of group total ■■
df['pct_of_dept'] = (
    df['salary'] /
    df.groupby('dept')['salary'].transform('sum')
) * 100

# ■■ Difference from group mean ■■
df['vs_avg'] = (
    df['salary'] -
    df.groupby('dept')['salary'].transform('mean')
)

# ■■ Lag / Lead (shift) ■■
df['prev_revenue'] = df['revenue'].shift(1)    # previous row (LAG)
df['next_revenue'] = df['revenue'].shift(-1)   # next row (LEAD)

# ■■ Period-over-period growth ■■
df['growth'] = df['revenue'].pct_change() * 100  # % change
df['abs_change'] = df['revenue'].diff()           # absolute change
```

Key Takeaway: *.transform() returns same-sized output aligned to the original DataFrame — use it to broadcast group stats to each row. .shift(1) = SQL LAG, .shift(-1) = SQL LEAD. .pct_change() computes percentage growth between consecutive rows.*

1.11 — Multi-Condition Filtering & query()

Problem: Complex row selection combining multiple conditions, negation, string matching, and null checks.

```
# ■■ Standard boolean indexing ■■
mask = (
    (df['dept'] == 'Eng') &
    (df['salary'] > 90000) &
    (df['age'] < 35)
)
result = df[mask]

# ■■ Negation ■■
df[~(df['dept'] == 'Sales')]    # ~ inverts the mask
df[df['dept'] != 'Sales']       # equivalent, simpler

# ■■ .query() — SQL-like syntax (cleaner for complex filters) ■■
df.query('dept == "Eng" and salary > 90000 and age < 35')

# query with variables using @
min_salary = 80000
df.query('salary >= @min_salary')

# ■■ Null-aware filtering ■■
df[df['salary'].notna()]            # rows where salary is NOT null
df[df['salary'].isna()]             # rows where salary IS null

# ■■ Combine string + numeric filters ■■
df[
    (df['name'].str.contains('a', case=False, na=False)) &
    (df['salary'].between(70000, 95000))
]
```

1.12 — Full Pipeline: Revenue Analysis

Problem: Given users and transactions tables, find: total revenue per country, top spending user, users with no transactions, and users below a spending threshold. This combines everything above and mirrors common Data Analyst interview questions.

```
import pandas as pd
import numpy as np

# ■■ Sample data ■■
countries = pd.DataFrame({
    'country_id': [1, 2, 3],
    'country': ['UK', 'DE', 'FR']
})
users = pd.DataFrame({
    'user_id': [1, 2, 3, 4, 5],
    'country_id': [1, 1, 2, 2, 3]
})
txns = pd.DataFrame({
    'user_id': [1, 1, 2, 3, 3, 3],
    'amount': [100, 250, 50, 300, 150, 200]
})

# ■■ Q1: Users with NO transactions ■■
# Left join users to txns, find NaN amounts
merged = pd.merge(users, txns, on='user_id', how='left')
no_txn_users = merged[merged['amount'].isna()]['user_id'].nunique()
# Answer: 2 (users 4 and 5)

# ■■ Q2: Users with total spend below $10 ■■
user_totals = txns.groupby('user_id')['amount'].sum().reset_index()
below_10 = user_totals[user_totals['amount'] < 10].shape[0]

# ■■ Q3: Countries with no users ■■
# Left join countries to users, find NaN user_ids
c_merged = pd.merge(countries, users, on='country_id', how='left')
no_user_countries = c_merged[c_merged['user_id'].isna()]['country_id'].nunique()

# ■■ Q4: Revenue per country ■■
```

```
# Chain merges: users -> txns -> countries
full = pd.merge(users, txns, on='user_id', how='inner')  # only active users
full = pd.merge(full, countries, on='country_id', how='left')
revenue = full.groupby('country')['amount'].sum().reset_index()
revenue.columns = ['country', 'total_revenue']

# ■■ Q5: Top spender per country ■■
user_country = pd.merge(
    full.groupby(['country', 'user_id'])['amount'].sum().reset_index(),
    countries, on='country_id', how='left'
) if 'country_id' in full.columns else \
    full.groupby(['country', 'user_id'])['amount'].sum().reset_index()

top_spenders = user_country.sort_values('amount', ascending=False) \
    .groupby('country').first().reset_index()
```

> ***Key Takeaway:*** *This pipeline covers common Data Analyst interview SQL questions translated to Pandas. The pattern is always: merge tables -> filter or group -> aggregate -> sort. Practice both SQL and Pandas versions.*

1.13 — Highest Salary per Department

Problem: Given an employee table and a department table, find the employees who earn the highest salary in each department. If multiple employees share the top salary in a department, return all of them.

Approach: Merge both tables, use transform('max') to compute each department's max salary as a new column on every row, then filter rows where salary equals that max.

```
import pandas as pd

def department_highest_salary(employee, department):
    # Merge on different column names: departmentId (left) = id (right)
    # Both tables have a 'name' column, so Pandas adds _x and _y suffixes
    # name_x = employee name, name_y = department name
    df = pd.merge(employee, department,
                  left_on='departmentId', right_on='id', how='left')

    # Compute max salary PER department, broadcast to every row
    # transform keeps the same row count (unlike .max() which collapses)
    df['max_salary'] = df.groupby('departmentId')['salary'].transform('max')

    # Keep only employees whose salary matches their dept's max
    # This correctly handles TIES (multiple employees with same top salary)
    highest = df[df['salary'] == df['max_salary']]

    # Select and rename columns for clean output
    result = highest[['name_y', 'name_x', 'salary']]
    result.columns = ['Department', 'Employee', 'Salary']
    return result
```

> ***Key Takeaway:*** *When merging tables that share a column name (e.g., both have 'name'), Pandas adds _x (left table) and _y (right table) suffixes. Use the suffixes=('_emp','_dept') parameter to make them readable. transform('max') is the key here — it broadcasts the group max to every row so you can filter with a simple == comparison.*

1.14 — Customers Who Never Order

Problem: Given a customers table and an orders table, find all customers who have never placed an order. Return their names.

Approach: Left merge keeps all customers. Rows with NaN in the orders column are customers without any order — the classic anti-join pattern.

```
def find_customers(customers, orders):
    # Left merge: keep ALL customers, even those with no orders
    df = customers.merge(orders, left_on='id',
                         right_on='customerId', how='left')

    # Rows where customerId is NaN = customers with zero orders
    no_orders = df[df['customerId'].isna()]

    # Return only the name column
    result = no_orders[['name']]
    result.columns = ['Customers']
    return result
```

> ***Key Takeaway:*** *The anti-join pattern in Pandas: left merge → filter .isna() on the right-side key. This is equivalent to SQL's LEFT JOIN + WHERE ... IS NULL.*

1.15 — Duplicate Emails

Problem: Find all email addresses that appear more than once in a table.

```
def duplicate_emails(person):
    # value_counts() returns email -> count, indexed by email
    counts = person['email'].value_counts()

    # Filter for count > 1, .index gives the email values
    duplicates = counts[counts > 1].index

    return pd.DataFrame({'Email': duplicates})
```

> ***Key Takeaway:*** *value_counts() is the fastest way to count duplicates. The result is a Series where the index is the value and the data is the count. Filter then grab .index.*

1.16 — Employees Earning More Than Their Manager

Problem: Find employees whose salary is higher than their manager's salary. Both employee and manager are in the same table.

Approach: Self-merge — join the table to itself. The left side is the employee, the right side is the manager (matched via managerId = id).

```
def find_employees(employee):
    # Self-merge: employee.managerId matches to manager's id
    # suffixes distinguish the two copies of each column
    df = employee.merge(
        employee,
        left_on='managerId',
        right_on='id',
        suffixes=('_emp', '_mgr')  # _emp = employee, _mgr = manager
    )

    # Keep rows where employee earns more than their manager
    overpaid = df[df['salary_emp'] > df['salary_mgr']]

    return overpaid[['name_emp']].rename(columns={'name_emp': 'Employee'})
```

> ***Key Takeaway:*** *Self-merge = merge a DataFrame with itself using different key columns. Always use suffixes to tell apart the two copies. This is the Pandas equivalent of a SQL self-join.*

1.17 — Consecutive Numbers (shift pattern)

Problem: Find all numbers that appear at least three times consecutively in a log table with columns (id, num).

Approach: Use shift() to look at the next rows, then compare.

```
def consecutive_numbers(logs):
    # shift(-1) = next row's value, shift(-2) = two rows ahead
    logs['next_1'] = logs['num'].shift(-1)
```

```
    logs['next_2'] = logs['num'].shift(-2)

    # All three consecutive values must be equal
    mask = (logs['num'] == logs['next_1']) & (logs['num'] == logs['next_2'])

    # drop_duplicates handles numbers that appear 4+ times
    result = logs[mask][['num']].drop_duplicates()
    return result.rename(columns={'num': 'ConsecutiveNums'})
```

Key Takeaway: *shift(-n) looks n rows AHEAD (LEAD in SQL). shift(n) looks n rows BACK (LAG). Combine multiple shifts with & to detect consecutive patterns.*

1.18 — Rank Scores (dense rank)

Problem: Rank scores from highest to lowest. Ties share the same rank. No gaps between ranks (dense ranking). Return ordered by score descending.

```
def order_scores(scores):
    # dense rank: ties share rank, no gaps (1, 1, 2 not 1, 1, 3)
    # ascending=False: highest score = rank 1
    scores['rank'] = scores['score'].rank(
        method='dense', ascending=False
    )

    # Sort by rank and return clean output
    return scores[['score', 'rank']] \
        .sort_values('rank').reset_index(drop=True)
```

> ***Key Takeaway:*** *rank() methods: 'dense' (no gaps: 1,1,2), 'min' (SQL default: 1,1,3), 'average' (ties get mean: 1.5,1.5,3). ascending=False flips the direction.*

1.19 — Nth Highest Distinct Salary

Problem: Find the Nth highest distinct salary. If fewer than N distinct salaries exist, return null.

```
def nth_highest_salary(employee, N):
    # Get unique salaries sorted descending
    unique = employee['salary'].drop_duplicates() \
        .sort_values(ascending=False)

    # Edge case: N invalid or exceeds available salaries
    if N <= 0 or N > len(unique):
        result = None
    else:
        # iloc is 0-based, so Nth element is at index N-1
        result = unique.iloc[N - 1]

    return pd.DataFrame({f'getNthHighestSalary({N})': [result]})
```

> ***Key Takeaway:*** *drop_duplicates() removes repeated values before sorting. iloc[N-1] accesses by position (0-based). Always handle the edge case where N exceeds the number of unique values — return None, not an IndexError.*

2. Classic Algorithms & Loops

Technical interviews often include 1-2 pure Python problems testing loops, logic, and common patterns. These are quick wins if you know the templates.

2.1 — For Loop vs While Loop

When to use which: `for` when you know how many iterations or are traversing a collection. `while` when you loop until a condition changes with unknown iteration count.

```
# ■■ FOR: known iterations or iterating over a collection ■■
for i in range(5):              # 0, 1, 2, 3, 4
    print(i)

for char in 'hello':            # h, e, l, l, o
    print(char)

for i, val in enumerate(lst):   # index + value
    print(f'{i}: {val}')

for key, val in my_dict.items():  # dict key-value pairs
    print(key, val)

# ■■ WHILE: condition-based, unknown iterations ■■
x = 100
while x > 1:
    x = x // 2  # halve until x <= 1
    print(x)     # runs 6 times: 50, 25, 12, 6, 3, 1
```

> ***Key Takeaway:*** *for + range() for counted loops. for + collection for traversal. while for unknown-count loops. Prefer for — it's safer (no infinite loop risk).*

2.2 — Nested Loops & Print Patterns

Problem: Print geometric patterns. Outer loop = rows, inner loop = columns. Very common warm-up in technical exercises.

```
n = int(input())

# ■■ Right-aligned staircase ■■
# n=4:    #
#        ##
#       ###
#      ####
for i in range(1, n + 1):
    # rjust pads '#'*i with spaces on the left
    print(('#' * i).rjust(n))

# ■■ Number triangle ■■
# 1
# 1 2
# 1 2 3
for i in range(1, n + 1):
    for j in range(1, i + 1):
        print(j, end=' ')  # same line
    print()  # newline after each row

# ■■ Diamond (upper half) ■■
for i in range(n):
    spaces = ' ' * (n - i - 1)  # decreasing left padding
    stars = '*' * (2 * i + 1)    # 1, 3, 5, 7... stars
    print(spaces + stars)

# ■■ Multiplication table ■■
for i in range(1, n + 1):
    for j in range(1, n + 1):
        print(f'{i*j:4}', end='')  # 4-char wide columns
    print()
```

2.3 — Loop with Accumulators

Problem: Build up a result inside a loop — running total, max tracking, list building, or string construction.

```
numbers = [10, 20, 30, 40, 50]

# ■■ Running sum ■■
total = 0
for n in numbers:
    total += n
print(total)  # 150

# ■■ Track maximum value and its index ■■
best_val = float('-inf')  # start with smallest possible
best_idx = -1
for i, val in enumerate(numbers):
    if val > best_val:
        best_val = val
        best_idx = i
# best_val=50, best_idx=4

# ■■ Count occurrences manually ■■
text = 'abracadabra'
freq = {}  # empty dict as accumulator
for c in text:
    freq[c] = freq.get(c, 0) + 1
# {'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}

# ■■ Build filtered list ■■
evens = []
for x in range(20):
    if x % 2 == 0:
        evens.append(x)
# Equivalent comprehension: [x for x in range(20) if x % 2 == 0]
```

> ***Key Takeaway:*** *Accumulator pattern: initialize before the loop, update inside, use after. Works for sum, max, count, list building, string joining, and dict building.*

2.4 — Break, Continue & For-Else

Problem: Control loop execution — exit early, skip iterations, or detect whether a loop ran to completion.

```
# ■■ BREAK: exit loop immediately ■■
for x in range(100):
    if x == 5:
        break        # stops loop, x stays at 5
    print(x)         # prints 0, 1, 2, 3, 4

# ■■ CONTINUE: skip to next iteration ■■
for x in range(6):
    if x == 3:
        continue     # skips print for x=3 only
    print(x)         # prints 0, 1, 2, 4, 5

# ■■ FOR-ELSE: else block runs ONLY if no break occurred ■■
# Perfect for search-and-report problems
target = 7
for x in [2, 4, 6, 8]:
    if x == target:
        print(f'Found {target}')
        break
else:
    # Only executes if break was never triggered
    print(f'{target} not found')
# Output: '7 not found'
```

> ***Watch Out:*** *for-else is a unique Python feature that confuses many people. The else block runs when the loop finishes normally (exhausts the iterable). If break fires, else is skipped.*

2.5 — FizzBuzz

Problem: For numbers 1 to N: print 'FizzBuzz' if divisible by both 3 and 5, 'Fizz' if by 3, 'Buzz' if by 5, else the number itself. The most classic coding interview warm-up.

```
n = int(input())

for i in range(1, n + 1):
    # Check most restrictive condition FIRST
    if i % 15 == 0:       # divisible by both 3 and 5
        print('FizzBuzz')
    elif i % 3 == 0:      # divisible by 3 only
        print('Fizz')
    elif i % 5 == 0:      # divisible by 5 only
        print('Buzz')
    else:
        print(i)

# ■■ String-building alternative (elegant) ■■
for i in range(1, n + 1):
    out = ''
    if i % 3 == 0: out += 'Fizz'
    if i % 5 == 0: out += 'Buzz'
    print(out or i)  # empty string is falsy, so 'or i' fires
```

> ***Watch Out:*** *Check divisibility by 15 (or 3 AND 5) BEFORE checking 3 or 5 individually. If you check 3 first, numbers like 15 match the elif and print 'Fizz' instead of 'FizzBuzz'.*

2.6 — Pascal's Triangle

Problem: Print N rows of Pascal's Triangle. Each number is the sum of the two numbers above it. Tests nested loop logic and list manipulation.

```
n = int(input())
row = [1]  # first row

for i in range(n):
    # Print current row, centered for visual alignment
    print(' '.join(map(str, row)).center(n * 4))

    # Build next row from current row
    # zip(row, row[1:]) pairs adjacent elements
    next_row = [1]                      # always starts with 1
    for a, b in zip(row, row[1:]):      # each adjacent pair
        next_row.append(a + b)          # sum of pair above
    next_row.append(1)                  # always ends with 1
    row = next_row

# Output for n=5:
#        1
#       1 1
#      1 2 1
#     1 3 3 1
#    1 4 6 4 1
```

2.7 — Collatz Sequence

Problem: Given N, generate the Collatz sequence: if even, divide by 2; if odd, multiply by 3 and add 1. Repeat until 1. Count steps. Classic while-loop exercise.

```
n = int(input())
steps = 0
sequence = [n]  # track the full sequence

while n != 1:
    if n % 2 == 0:
        n = n // 2       # even: halve
    else:
        n = 3 * n + 1    # odd: triple + 1
    sequence.append(n)
    steps += 1
```

```
print(' -> '.join(map(str, sequence)))
print(f'Steps: {steps}')
# n=6: 6 -> 3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1 (8 steps)
```

2.8 — Two Sum (Hash Map Lookup)

Problem: Given an array and a target, find two elements that sum to the target. Return their indices. The most frequently asked algorithm question.

Approach: For each element, compute complement = target - element. Check if complement exists in a dictionary. This reduces $O(n^2)$ brute force to O(n).

```
def two_sum(arr, target):
    seen = {}  # maps value -> index

    for i, x in enumerate(arr):
        complement = target - x  # what we need to find

        if complement in seen:
            # Found a pair: complement was seen earlier
            return [seen[complement], i]

        # Store current value for future lookups
        seen[x] = i

    return None  # no valid pair

# Example:
two_sum([2, 7, 11, 15], 9)  # [0, 1]
# arr[0]=2, arr[1]=7, 2+7=9
```

> ***Key Takeaway:*** *The hash map lookup pattern is the #1 optimization to know. Any time you're doing nested loops to find pairs, ask: can I store seen values in a dict and check complements?*

2.9 — Balanced Brackets (Stack)

Problem: Check if a string of brackets — (), [], {} — is properly balanced. Uses a stack: push on open, pop on close, verify the match.

```
def is_balanced(s):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}  # close -> open

    for c in s:
        if c in '([{':
            stack.append(c)           # push opening bracket
        elif c in pairs:
            if not stack:             # nothing to match
                return False
            if stack[-1] != pairs[c]: # top doesn't match
                return False
            stack.pop()               # matched, remove

    return len(stack) == 0  # all brackets consumed?

# Tests:
is_balanced('({[]})')  # True
is_balanced('({[}])')  # False — ] expected before }
is_balanced('((')      # False — unmatched opening
```

2.10 — Matrix Transpose

Problem: Swap rows and columns of a matrix, or rotate it 90 degrees.

```
matrix = [[1,2,3], [4,5,6], [7,8,9]]

# Transpose: zip(*matrix) unpacks rows, zip pairs by position
transposed = [list(row) for row in zip(*matrix)]
# [[1,4,7], [2,5,8], [3,6,9]]

# Rotate 90 clockwise = transpose + reverse each row
```

```
rotated_cw = [list(row)[::-1] for row in zip(*matrix)]
# [[7,4,1], [8,5,2], [9,6,3]]

# Rotate 90 counter-clockwise = reverse rows then transpose
rotated_ccw = [list(row) for row in zip(*matrix[::-1])]
# [[3,6,9], [2,5,8], [1,4,7]]
```

2.11 — Frequency Sort

Problem: Sort elements by frequency (most common first), break ties by value.

```
from collections import Counter

arr = [4, 5, 6, 5, 4, 3, 4]
freq = Counter(arr)

# Sort: frequency descending (-freq), then value ascending
result = sorted(arr, key=lambda x: (-freq[x], x))
# [4, 4, 4, 5, 5, 3, 6]
# 4 appears 3x (most), then 5 (2x), then 3 and 6 (1x each)
```

2.12 — Palindrome Check

```
# ■■ Simple check using slicing ■■
s = 'racecar'
is_palindrome = s == s[::-1]  # True

# ■■ Case-insensitive, ignore non-alphanumeric ■■
def is_clean_palindrome(s):
    # Remove non-alphanumeric, lowercase
    clean = ''.join(c.lower() for c in s if c.isalnum())
    return clean == clean[::-1]

is_clean_palindrome('A man, a plan, a canal: Panama')  # True

# ■■ Two-pointer approach (no extra memory) ■■
def is_palindrome_2p(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True
```

2.13 — Longest Common Prefix

Problem: Given an array of strings, find the longest common prefix shared by all of them. If there is no common prefix, return an empty string.

Approach: Start with the first string as the prefix. For each subsequent string, shrink the prefix from the right until it matches. If the prefix becomes empty, return ''.

```
def longest_common_prefix(strs):
    if not strs:
        return ''

    # Assume the first string is the full prefix
    prefix = strs[0]

    # Compare prefix against each remaining string
    for i in range(1, len(strs)):
        # Shrink prefix until current string starts with it
        while not strs[i].startswith(prefix):
            prefix = prefix[:-1]  # chop last char
            if not prefix:
                return ''  # no common prefix exists

    return prefix
```

```
# Examples:
# ['flower','flow','flight'] -> 'fl'
# ['dog','racecar','car']    -> ''
```

> ***Key Takeaway:*** *The shrinking prefix approach is O(S) where S is the total characters across all strings. startswith() is cleaner than manual character comparison. prefix[:-1] removes the last character — keep shrinking until it matches or becomes empty.*

2.14 — Reverse a Linked List

Problem: Given the head of a singly linked list, reverse it in-place and return the new head. Input: [1,2,3,4,5] → Output: [5,4,3,2,1].

Approach: Walk through the list with two pointers (prev and curr). At each node, flip its .next pointer to point backward instead of forward. After the loop, prev holds the new head.

```
# Node definition (given in the problem)
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next

def reverse_list(head):
    prev = None   # will become the new head
    curr = head   # start at the original head

    while curr:
        next_node = curr.next  # save the next node before we break the link
        curr.next = prev       # flip pointer: point backward
        prev = curr            # advance prev one step
        curr = next_node       # advance curr one step

    return prev  # prev is now the first node of the reversed list

# Step-by-step for [1 -> 2 -> 3]:
# Start: prev=None, curr=1
# Step 1: 1.next=None,  prev=1, curr=2   -> [1] [2->3]
# Step 2: 2.next=1,     prev=2, curr=3   -> [2->1] [3]
# Step 3: 3.next=2,     prev=3, curr=None -> [3->2->1]
```

> ***Key Takeaway:*** *The three-pointer pattern (prev, curr, next_node) is the standard way to reverse a linked list in O(n) time and O(1) space. Save next before flipping, or you lose the rest of the list. This is one of the most frequently asked interview questions.*

3. Lists, Sorting & Collections

Lists and collections are the building blocks of almost every Python problem. Custom sorting with lambda, Counter, and defaultdict appear constantly.

3.1 — Nested Lists (Group & Filter)

Problem: Given N students with names and grades, find students with the second-lowest grade and print their names alphabetically.

```
records = []
for _ in range(int(input())):
    name = input()
    score = float(input())
    records.append([name, score])

# Step 1: Get sorted unique scores
unique_scores = sorted(set(s for _, s in records))

# Step 2: The second-lowest score
second_lowest = unique_scores[1]

# Step 3: Filter names with that score, sort alphabetically
result = sorted(name for name, score in records
                if score == second_lowest)

for name in result:
    print(name)
```

> ***Key Takeaway:*** *Pattern: extract unique values with set(), sort them, index the one you need, then filter original data. Avoids complex multi-key sorting.*

3.2 — Custom Multi-Key Sorting

Problem: Sort records by multiple criteria with mixed ascending/descending. This pattern appears in almost every data-related exercise.

```
people = [
    ('Ana', 30, 'Eng'),
    ('Bob', 25, 'Sales'),
    ('Cara', 30, 'Eng'),
    ('Dan', 25, 'Sales')
]

# Sort by age DESCENDING, then name ASCENDING for ties
# Negate numeric fields for descending order
people.sort(key=lambda p: (-p[1], p[0]))
# [('Ana',30,'Eng'), ('Cara',30,'Eng'), ('Bob',25,'Sales'), ('Dan',25,'Sales')]

# ■■ Why negation works ■■
# Sorting is ascending by default
# -30 < -25, so age 30 sorts before 25 (descending effect)

# ■■ For strings descending: use two separate sorts ■■
# Python sort is STABLE: equal elements keep their relative order
# Sort by secondary key first, then primary key
people.sort(key=lambda p: p[0])          # 1st: name ascending
people.sort(key=lambda p: -p[1])         # 2nd: age descending
# Stability preserves the name order within same-age groups

# ■■ Sort dicts by nested value ■■
users = [{'name': 'Ana', 'score': 90}, {'name': 'Bob', 'score': 85}]
sorted(users, key=lambda u: u['score'], reverse=True)
```

> ***Key Takeaway:*** *Negate numeric fields for descending in a tuple key. For mixed string directions, use two stable sorts (secondary first, primary second). sorted() returns a new list; .sort() modifies in place.*

3.3 — Second Largest Element

```
arr = list(map(int, input().split()))

# Approach 1: set + sort (O(n log n), safe)
unique = sorted(set(arr), reverse=True)
print(unique[1])

# Approach 2: two-pass max (O(n), faster)
max_val = max(arr)
second = max(x for x in arr if x != max_val)
print(second)

# Approach 3: single-pass (most efficient)
first = second = float('-inf')
for x in arr:
    if x > first:
        second = first  # demote current first
        first = x
    elif x > second and x != first:
        second = x
```

3.4 — List Comprehensions

Problem: Create lists with transformation and filtering in a single expression. Faster and more Pythonic than appending in a loop.

```
# ■■ Basic: transform each element ■■
squares = [x**2 for x in range(10)]  # [0, 1, 4, 9, ...81]

# ■■ With filter condition ■■
even_sq = [x**2 for x in range(10) if x % 2 == 0]  # [0, 4, 16, 36, 64]

# ■■ Nested comprehension (flatten 2D list) ■■
matrix = [[1,2,3], [4,5,6]]
flat = [x for row in matrix for x in row]  # [1,2,3,4,5,6]
# Read as: for row in matrix, then for x in row

# ■■ Conditional expression (if-else) ■■
labels = ['even' if x % 2 == 0 else 'odd' for x in range(5)]
# ['even', 'odd', 'even', 'odd', 'even']

# ■■ Dict comprehension ■■
word_lengths = {w: len(w) for w in ['hello', 'world', 'hi']}
# {'hello': 5, 'world': 5, 'hi': 2}

# ■■ Set comprehension ■■
unique_lengths = {len(w) for w in ['hello', 'world', 'hi']}
# {2, 5}

# ■■ 3D coordinates where i+j+k != N ■■
x, y, z, n = 1, 1, 2, 3
coords = [[i,j,k] for i in range(x+1)
                  for j in range(y+1)
                  for k in range(z+1)
                  if i+j+k != n]
```

> ***Key Takeaway:*** *List comprehensions are 20-30% faster than equivalent for-loops with append. Filter goes AFTER the for clause. Conditional expression (if-else) goes BEFORE the for.*

3.5 — Counter (Frequency Counting)

Problem: Count element frequency. Used for: most common element, histogram, frequency ranking, checking anagrams.

```
from collections import Counter

# ■■ Count anything iterable ■■
freq = Counter(['a', 'b', 'a', 'c', 'b', 'a'])
# Counter({'a': 3, 'b': 2, 'c': 1})

# ■■ Character frequency ■■
Counter('abracadabra')  # {'a':5, 'b':2, 'r':2, 'c':1, 'd':1}

# ■■ Top N most common ■■
freq.most_common(2)     # [('a', 3), ('b', 2)]

# ■■ Access count (missing key returns 0, no KeyError) ■■
freq['a']       # 3
freq['z']       # 0

# ■■ Counter arithmetic ■■
c1 = Counter('aabb')
c2 = Counter('abcc')
c1 + c2   # Counter({'a':3, 'b':3, 'c':2})
c1 - c2   # Counter({'a':1, 'b':1})  — only positive counts

# ■■ Check if anagram ■■
def is_anagram(s1, s2):
    return Counter(s1.lower()) == Counter(s2.lower())

is_anagram('listen', 'silent')  # True
```

3.6 — defaultdict (Auto-Initialize)

Problem: Group items by key without checking if the key exists first. Eliminates boilerplate 'if key not in dict' checks.

```
from collections import defaultdict

# ■■ Group employees by department ■■
employees = [('Ana','Eng'), ('Bob','Sales'), ('Cara','Eng'), ('Dan','Sales')]

# Without defaultdict — verbose
d = {}
for name, dept in employees:
    if dept not in d:
        d[dept] = []
    d[dept].append(name)

# With defaultdict — clean
d = defaultdict(list)  # missing keys auto-create empty list
for name, dept in employees:
    d[dept].append(name)  # never raises KeyError
# {'Eng': ['Ana','Cara'], 'Sales': ['Bob','Dan']}

# ■■ Common factories ■■
defaultdict(list)    # default: []
defaultdict(int)     # default: 0  (great for counting)
defaultdict(set)     # default: set()
defaultdict(dict)    # default: {}

# ■■ Counting with defaultdict(int) ■■
word_count = defaultdict(int)
for word in text.split():
    word_count[word] += 1  # starts at 0, increments
```

3.7 — OrderedDict & namedtuple

```
from collections import OrderedDict, namedtuple

# ■■ OrderedDict: equality depends on insertion order ■■
od1 = OrderedDict([('a', 1), ('b', 2)])
```

```
od2 = OrderedDict([('b', 2), ('a', 1)])
od1 == od2  # False (different order)

# Regular dict: {'a':1,'b':2} == {'b':2,'a':1} -> True

# ■■ namedtuple: lightweight class for structured data ■■
Student = namedtuple('Student', ['name', 'age', 'grade'])
s = Student('Alice', 22, 95.5)
s.name    # 'Alice'  — readable field access
s[0]      # 'Alice'  — also supports indexing

# Useful for reading structured input
students = []
for _ in range(n):
    name, age, score = input().split()
    students.append(Student(name, int(age), float(score)))

# Compute average grade cleanly
avg = sum(s.grade for s in students) / len(students)
```

3.8 — Word Order Problem

Problem: Given N words, output the count of distinct words, then each word's frequency in order of first appearance.

```
from collections import OrderedDict

n = int(input())
d = OrderedDict()

for _ in range(n):
    word = input()
    # .get(key, 0) returns 0 if key is new
    d[word] = d.get(word, 0) + 1

# Number of distinct words
print(len(d))

# Frequency of each word in order of first appearance
print(' '.join(map(str, d.values())))

# Example input: 'cat cat dog bird cat dog'
# Output: 3       (3 distinct words)
#         3 2 1   (cat=3, dog=2, bird=1)
```

> ***Key Takeaway:*** *OrderedDict + .get(key, default) is a clean pattern for counting while preserving first-appearance order. In Python 3.7+, regular dicts also maintain insertion order, so dict + .get() works too.*

Bonus: Complexity, Built-ins & Strategy

Time Complexity Reference

Operation	Time	Notes
List append / pop (end)	O(1)	Use as stack (LIFO)
List insert(0, x) / pop(0)	O(n)	Use deque for O(1) at both ends
List index access	O(1)	Direct memory access
List search (x in list)	O(n)	Convert to set for O(1) lookup
List sort	O(n log n)	Timsort — stable, efficient
Dict / Set get/set/in	O(1) avg	Hash table — fastest lookups
String concat in loop	O(n²)	Use ''.join(list) instead
heapq push/pop	O(log n)	Use for top-K problems
sorted()	O(n log n)	Returns new list; .sort() modifies in place

Essential Built-in Functions

```
# ■■ Type conversion ■■
int('42')        # 42        str(42)         # '42'
float('3.14')    # 3.14      bool(0)         # False
list('abc')      # ['a','b','c']
tuple([1,2])     # (1, 2)    set([1,1,2])    # {1, 2}

# ■■ Aggregation ■■
sum([1,2,3])     # 6         min(1,2,3)      # 1
max(1,2,3)       # 3         len('hello')    # 5
abs(-5)          # 5         round(3.456, 2) # 3.46

# ■■ Iteration helpers ■■
enumerate('ab')       # (0,'a'), (1,'b')  — index + value
zip([1,2], [3,4])     # (1,3), (2,4)      — parallel iter
reversed([1,2,3])     # 3, 2, 1           — reverse iterator
sorted([3,1,2])       # [1,2,3]           — new sorted list

# ■■ Boolean checks ■■
any([False, True, False])  # True  — at least one True
all([True, True, True])    # True  — every element True
isinstance(x, (int, float))  # type check
```

Common Input Patterns

```
# Single integer
n = int(input())

# Two integers on one line
a, b = map(int, input().split())

# List of integers on one line
arr = list(map(int, input().split()))

# N lines of input
lines = [input() for _ in range(n)]

# N lines of name + score
records = []
for _ in range(n):
    name = input()
    score = float(input())
    records.append((name, score))

# Print list as space-separated string
```

```
print(' '.join(map(str, result)))

# Print without newline
print(x, end='')
```

Pandas Quick Reference

Task	Code
Read CSV	`pd.read_csv('file.csv')`
Filter rows	`df[df['col'] > val]`
Select columns	`df[['col1', 'col2']]`
Group + aggregate	`df.groupby('col')['val'].mean()`
Merge tables	`pd.merge(df1, df2, on='key', how='left')`
Sort	`df.sort_values('col', ascending=False)`
Drop NaN	`df.dropna(subset=['col'])`
Fill NaN	`df['col'].fillna(df['col'].mean())`
New column	`df['new'] = df['a'] + df['b']`
Value counts	`df['col'].value_counts()`
Rank in group	`df.groupby('g')['v'].rank(method='dense')`
Top N per group	`df.sort_values('v').groupby('g').tail(n)`
Running total	`df['v'].cumsum()`
Lag / shift	`df['v'].shift(1)`
% change	`df['v'].pct_change()`
Pivot table	`df.pivot_table(values, index, columns, aggfunc)`
Apply function	`df['col'].apply(lambda x: x*2)`
String clean	`df['col'].str.strip().str.lower()`

Exam Strategy

#	Strategy	Why
1	Read the FULL problem + constraints first	Constraints tell you required time complexity
2	Start with the sample test case	Verify basic logic before edge cases
3	Handle edge cases: empty input, single element, negatives	Many points lost to edges, not logic
4	If stuck, brute force first then optimize	Partial credit > zero credit
5	Prefer dict/set for lookups over nested loops	O(1) lookup prevents TLE
6	input() returns string — cast with int() / float()	Most common beginner mistake
7	Use Pandas vectorized ops over .apply() loops	10-100x faster, fewer bugs
8	After groupby, always .reset_index()	Keeps output clean, avoids MultiIndex issues
9	For merges: check which 'how' you need	Wrong join type = wrong answer silently
10	Skip hard problems, collect easy points first	Time management wins tests

Golden Rule: *Final rule of thumb: $n \leq 1{,}000$ means brute force is fine. $n \leq 10^5$ needs O(n log n) or better. $n \leq 10^6$ needs O(n) only. For Pandas: if your DataFrame has under 100k rows, almost anything works.*