PYTHON CHEATSHEET

grito LABS

1. PYTHON BASICS & DATA STRUCTURES

TOPIC / COMMAND	SYNTAX & CODE	NOTES & EXAMPLE

Variables & Types	age = 28 # int
	price = 19.99 # float
	name = 'Alice' # str
	active = True # bool
	
	type(age) # <class 'int'>
	isinstance(price, float)# True
	
	# Type conversion
	int('42') # 42
	float('3.14')# 3.14
	str(100) # '100'
	bool(0) # False
		0, '', None, [], {} are falsy.
		f-string: f'Hi {name}, age {age}'

Lists	nums = [10, 20, 30, 40, 50]
	nums[0] # 10 (first)
	nums[-1] # 50 (last)
	nums[1:3] # [20, 30]
	nums[::-1] # reversed
	nums.append(60) # add end
	nums.insert(0,5) # add at index
	nums.remove(20) # remove first 20
	nums.pop() # remove & return last
	nums.sort() # in-place sort
	
	# List comprehension
	squares = [x**2 for x in range(1,6)]
	evens = [x for x in nums if x%2==0]
		Comprehensions are faster than loops.
		len(nums) returns count.
		sorted(nums) returns new list.

Dictionaries	d = {'id':101, 'name':'Bob', 'score':88.5}
	
	d['name'] # 'Bob'
	d.get('age', 0) # 0 (safe default)
	d['score'] = 92.0 # update
	d['dept'] = 'Sales'# add key
	del d['score'] # remove key
	
	d.keys() # dict_keys([...])
	d.values() # dict_values([...])
	d.items() # key-value pairs
	
	for k, v in d.items():
	    print(f'{k}: {v}')
		Use .get(k, default) to avoid KeyError.
		Dict comprehension:
		{k.upper(): v for k,v in d.items()}

Tuples	coords = (19.07, 72.87) # immutable lat, lon = coords # unpack # Named tuples (readable) from collections import namedtuple Point = namedtuple('Point',['x','y']) p = Point(3, 4) p.x # 3 p.y # 4	Tuples are faster than lists. Use for fixed data: coords, RGB, shape. Parentheses optional for packing.

Control Flow	score = 74
	if score >= 90: grade = 'A'
	elif score >= 75: grade = 'B'
	elif score >= 60: grade = 'C'
	else: grade = 'F'
	
	# Ternary
	label = 'Pass' if score>=60 else 'Fail'
	
	# for loop
	for i in range(5): print(i)
	
	# enumerate
	for i, val in enumerate(['a','b'], 1):
	    print(i, val) # 1 a, 2 b
	
	# while
	n = 0
	while n < 5: n += 1
		AND > OR precedence. Use parentheses.
		range(start, stop, step)
		break exits loop; continue skips.

Functions

```
def clean_revenue(val, default=0.0):
    '''Docstring goes here.'''
    try:
        return float(str(val)
            .replace('$','').replace(',',''))
    except (ValueError, TypeError):
        return default

# Lambda (anonymous)
double = lambda x: x * 2

# *args / **kwargs
def report(*cols, **opts):
    decimals = opts.get('decimals', 2)

# map / filter
cleaned = list(map(clean_revenue, raw))
valid   = list(filter(lambda x:x>0, cleaned))
```

Always write docstrings.
map/filter return iterators; wrap in list().
Use default args for optional parameters.

2. NUMPY — NUMERICAL COMPUTING

TOPIC / COMMAND | **SYNTAX & CODE** | **NOTES & EXAMPLE**

Array Creation

```
import numpy as np

np.array([1, 2, 3, 4, 5])
np.array([[1,2],[3,4]])  # 2D
np.zeros((3, 4))        # all zeros
np.ones((3, 4))         # all ones
np.eye(4)               # identity
np.arange(0, 10, 2)     # [0,2,4,6,8]
np.linspace(0, 1, 5)    # 5 even pts
np.random.randn(3, 4)   # std normal
np.random.randint(0,10,(3,4))
```

arange: like range() → array.
linspace: useful for plot axes.
randn: standard normal distribution.

Properties & Reshaping

```
a.shape      # (3, 4)
a.dtype      # int64 / float64
a.ndim       # 1 or 2
a.size       # total elements

a.reshape(2, 6)    # new shape
a.flatten()        # 1D copy
a.T                # transpose
a.astype(float)    # cast dtype

np.concatenate([a,b], axis=0)
np.vstack([a, b])  # stack rows
np.hstack([a, b])  # stack cols
```

axis=0 = column-wise.
axis=1 = row-wise.
reshape(-1, 1) = column vector.

Math & Statistics

```
np.sum(a, axis=0)   # col sums
np.mean(a)
np.median(a)
np.std(a)
np.min(a) / np.max(a)
np.argmin(a)        # index of min
np.cumsum(a)        # running total
np.percentile(a, [25,50,75])
np.corrcoef(x, y)   # correlation
np.dot(a, b)        # matrix multiply
np.sqrt(a)          # element-wise
```

Vectorised ops avoid Python loops.
NumPy ops are 10-100x faster than loops.

Boolean Indexing & Where

```
a[a > 5]            # filter elements
a[(a>2) & (a<8)]    # AND condition
a[(a<2) | (a>8)]    # OR condition

# Vectorised if/else
np.where(a > 5, 'big', 'small')

# Multiple conditions
np.select(
    [a < 3, a < 7],
    ['low', 'mid'],
    default='high')

np.isin(a, [1,3,5])   # membership
np.isnan(a)           # detect NaN
np.unique(a, return_counts=True)
```

np.where is much faster than iterating.
np.select replaces nested if/elif.

3. PANDAS FOUNDATIONS

TOPIC / COMMAND | **SYNTAX & CODE** | **NOTES & EXAMPLE**

Creating DataFrames

```
import pandas as pd

# From dict
df = pd.DataFrame({
    'product': ['A','B','C'],
    'price':   [100, 200, 150],
    'qty':     [30,  15,  25]
})

# From CSV
df = pd.read_csv('sales.csv',
    parse_dates=['date'],
    dtype={'id': str})

# From Excel
df = pd.read_excel('data.xlsx',
    sheet_name='Sheet1')

# Save
df.to_csv('out.csv', index=False)
df.to_excel('out.xlsx', index=False)
```

Always index=False when saving CSV.
parse_dates converts strings to datetime.
dtype={'id':str} prevents leading-zero loss.

First Look

```
df.head(5)       # first 5 rows
df.tail(5)       # last 5 rows
df.sample(10)    # random rows
df.shape         # (rows, cols)
df.columns       # column names
df.dtypes        # data types
df.info()        # dtypes + null counts

df.describe()               # numeric
df.describe(include='all') # all cols
df.describe().T             # transposed

# Missing data
df.isnull().sum()
(df.isnull().sum()/len(df)*100).round(1)
```

Run head+info+describe on EVERY dataset.
These 3 reveal dtypes, nulls, distribution.

loc & iloc

```
# Column selection
df['price']                # Series
df[['price','qty']]        # DataFrame

# .loc — label-based
df.loc[0]                  # row by label
df.loc[0:4, 'price']       # rows 0-4
df.loc[df['qty']>20,
       ['product','price']]

# .iloc — position-based
df.iloc[0]                 # first row
df.iloc[-1]                # last row
df.iloc[0:5, 1:3]          # slice

# Boolean filter
df[df['price'] > 100]
df[(df['price']>100)&(df['qty']>20)]
df[df['product'].isin(['A','C'])]
df.query("price > 100 and qty > 20")
```

loc inclusive on both ends.
iloc excludes end index.
query() is more readable for complex filters.

Transforming Columns

```
# Vectorised (fast)
df['rev'] = df['price'] * df['qty']

# assign() — chainable
df = df.assign(
    rev    = lambda x: x.price * x.qty,
    margin = lambda x: x.rev * 0.3
)

# np.where — conditional
import numpy as np
df['pass'] = np.where(
    df['score']>=60,'Pass','Fail')
```

Vectorised is 10-100x faster than apply().
apply(axis=1) for row-wise custom logic.
assign() is chainable; returns new df.

```
# pd.cut — binning
df['band'] = pd.cut(df['score'],
    bins=[0,60,80,100],
    labels=['Low','Mid','High'])

# apply() — custom function
df['clean'] = df['rev'].apply(fn)
```

Sort, Rename, Drop

```
# Sort
df.sort_values('score', ascending=False)
df.sort_values(['region','score'],
    ascending=[True, False])
df.nlargest(10, 'revenue')   # top 10
df.nsmallest(5, 'cost')      # bottom 5

# Rename
df.rename(columns={'old':'new'})
df.columns = df.columns.str.lower()
    .str.replace(' ','_')

# Drop
df.drop(columns=['col1','col2'])
df.drop(index=[0,1,2])

# Reorder columns
df = df[['id','name','score']]

# Always copy before modifying!
df2 = df.copy()
```

nlargest/nsmallest faster than sort+head.
df.copy() avoids SettingWithCopyWarning.
Normalise column names after every load.

4. DATA READING, WRITING & MANIPULATION

TOPIC / COMMAND	SYNTAX & CODE	NOTES & EXAMPLE

Reading Files

```
# CSV with options
df = pd.read_csv('file.csv',
    sep=';', encoding='utf-8',
    parse_dates=['date'],
    usecols=['a','b','c'],
    dtype={'code': str},
    nrows=1000,
    na_values=['N/A','--'])

# Parquet (fast, compressed)
df = pd.read_parquet('file.parquet')

# JSON
df = pd.read_json('file.json')

# SQL
df = pd.read_sql(query, conn)
```

Parquet is 5-10x faster than CSV.
usecols saves memory — load only needed cols.
errors='coerce' turns bad values to NaN.

Writing & Chunked Reading

```
# CSV
df.to_csv('out.csv', index=False)

# Excel multi-sheet
with pd.ExcelWriter('out.xlsx',
        engine='openpyxl') as w:
    df1.to_excel(w, sheet_name='Data')
    df2.to_excel(w, sheet_name='Summary')

# Parquet
df.to_parquet('out.parquet',
    compression='snappy')

# Chunked reading (big files)
chunks = pd.read_csv('big.csv',
    chunksize=10_000)
results = []
for chunk in chunks:
    results.append(chunk[chunk['val']>0])
df = pd.concat(results, ignore_index=True)
```

Use utf-8-sig for Excel-compatible CSV.
Parquet + snappy: best size/speed tradeoff.
chunksize when file > available RAM.

GroupBy & Aggregation

```
df.groupby('region')['sales'].sum()
df.groupby('region')['sales'].mean()
```

Named agg gives clean column names.
Always .reset_index() after groupby.

<table>
<tr><td></td><td><pre>
# Multiple aggs
df.groupby('region').agg({
    'sales': ['sum','mean'],
    'orders': 'count'
})

Named agg (clean output) *
df.groupby('region').agg(
    total_sales = ('sales', 'sum'),
    avg_price   = ('price', 'mean'),
    n_orders   = ('order_id','count')
).reset_index()

# Multi-level groupby
df.groupby(['region','category'])
  ['sales'].sum()
</pre></td><td>Use size() (incl. NaN) vs count() (excl.).</td></tr>
<tr><td>Merging & Joining</td><td><pre>
Inner join
pd.merge(orders, customers,
    on='customer_id')

Left join
pd.merge(orders, customers,
    on='customer_id', how='left')

Different column names
pd.merge(orders, customers,
    left_on='cust_id', right_on='id')

Stack vertically
pd.concat([df1, df2, df3],
    ignore_index=True)

# Verify after merge!
print(f'Before:{len(a)} After:{len(r)}')
</pre></td><td>how: inner, left, right, outer.
Always verify row count after merge.
Unexpected row multiplication is common bug.</td></tr>
<tr><td>Reshaping Data</td><td><pre>
melt: wide -> long (unpivot)
pd.melt(df,
    id_vars=['id','name'],
    value_vars=['Q1','Q2','Q3'],
    var_name='quarter',
    value_name='sales')

pivot: long -> wide
df.pivot(index='date',
    columns='product', values='sales')

Pivot table (like Excel)
pd.pivot_table(df,
    values='sales', index='region',
    columns='quarter', aggfunc='sum',
    fill_value=0, margins=True)

Cross-tab with percentages
pd.crosstab(df['region'], df['status'],
    normalize='index').round(2)
</pre></td><td>melt converts wide dashboards to tidy format.
margins=True adds Grand Total row/column.
normalize='index' gives row percentages.</td></tr>
</table>

PYTHON CHEATSHEET

5. DATA QUALITY & CLEANING

TOPIC / COMMAND | **SYNTAX & CODE** | **NOTES & EXAMPLE**

DQ Dimensions

```
# Five dimensions to check:
# Completeness  - required fields filled?
# Accuracy      - values make sense?
# Consistency   - same format across rows?
# Uniqueness    - no duplicate IDs?
# Validity      - within expected ranges?

def dq_report(df):
    return pd.DataFrame({
        'dtype':  df.dtypes,
        'nulls':  df.isnull().sum(),
        'null_%': (df.isnull().mean()
                   *100).round(1),
        'unique': df.nunique(),
    })
```

Run dq_report() on every new dataset.
Flag columns with >30% nulls for review.
Spend 50% of EDA time on data quality.

Missing Data

```
# Detect
df.isnull().sum()
df.isnull().mean() * 100  # %

# Drop
df.dropna()                   # any null
df.dropna(subset=['sales'])# specific col
df.dropna(thresh=3)           # keep >=3 vals

# Fill / Impute
df['age'].fillna(df['age'].median())
df['city'].fillna('Unknown')
df.fillna(method='ffill')  # forward fill
df.fillna(method='bfill')  # backward fill

# Interpolate
df['temp'].interpolate(method='linear')
```

Use median (not mean) for skewed columns.
ffill works well for time-series gaps.
Consider flag column: df['age_missing']=...

Duplicates & Validation

```
df.duplicated().sum()
df[df.duplicated(keep=False)]  # all dupes
df.drop_duplicates(inplace=True)
df.drop_duplicates(
    subset=['id'], keep='last')

# Sanity checks (put in pipeline)
assert df['id'].is_unique, 'Dup IDs!'
assert df.notnull().all().all(),'Nulls!'
assert (df['age'] >= 0).all(),'Neg age!'

# Check expected categories
valid = {'active','inactive','pending'}
assert set(df['status'].unique())
    .issubset(valid)
```

keep='last': keep most recent (if date sorted).
Assert statements catch bugs early.
Put validation after every data load step.

Outlier Detection

```
# IQR Method (robust) *
Q1  = df['sales'].quantile(0.25)
Q3  = df['sales'].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

outliers = df[(df['sales']<lower) |
              (df['sales']>upper)]
clean_df = df[df['sales']
    .between(lower, upper)]

# Z-Score (assumes normality)
from scipy import stats
z = np.abs(stats.zscore(df['sales']))
df_clean = df[z < 3]

# Cap instead of remove
df['sales'] = df['sales'].clip(lower, upper)
```

IQR preferred for skewed data (sales, revenue).
Z-score only works for normal distributions.
clip() keeps sample size intact.

Type Fixing & String Cleaning

```
# Fix messy numeric
df['rev'] = (
    df['rev']
    .str.replace('[$,]','',regex=True)
```

errors='coerce' → bad values become NaN.
Category dtype saves 50-90% memory.

	<pre> .pipe(pd.to_numeric, errors='coerce')
	)
	
	# Fix dates
	df['date'] = pd.to_datetime(
	    df['date'], format='%Y-%m-%d',
	    errors='coerce')
	
	# Normalise text
	df['city'] = (df['city']
	    .str.strip().str.lower()
	    .str.replace(r'\s+',' ',regex=True))
	
	# Normalise column names
	df.columns = (df.columns
	    .str.lower().str.replace(' ','_'))
	
	# Save memory with category
	df['region'] = df['region'].astype('category')</pre>
		Normalise column names after every load.

Encoding & Transformation	<pre># One-Hot Encoding
	dummies = pd.get_dummies(
	    df['region'], prefix='reg',
	    drop_first=True)
	df = pd.concat([df, dummies], axis=1)
	
	# Label Encoding (ordinal data)
	tier_map = {'Bronze':1,'Silver':2,'Gold':3}
	df['tier_code'] = df['tier'].map(tier_map)
	
	# Binning
	df['age_group'] = pd.cut(df['age'],
	    bins=[0,25,40,60,100],
	    labels=['Youth','Adult','Sr','Elder'])
	
	# Min-Max Normalise (0 to 1)
	df['s_norm'] = (
	    (df['s'] - df['s'].min()) /
	    (df['s'].max() - df['s'].min())
	)</pre>
		drop_first=True avoids dummy variable trap.
		map() applies a lookup dict to a Series.
		pd.cut: bins are closed on the right.

6. EXPLORATORY DATA ANALYSIS (EDA)

TOPIC / COMMAND	SYNTAX & CODE	NOTES & EXAMPLE

EDA Workflow	<pre># 6-Step EDA Framework:
	# 1. Load & Inspect - head, info, describe
	# 2. Data Quality - nulls, types, dupes
	# 3. Univariate - per-column distribution
	# 4. Bivariate - correlations, crosstabs
	# 5. Multivariate - patterns across dims
	# 6. Insights - answer business Qs
	
	# EDA Objectives:
	# - Understand distributions
	# - Find relationships between variables
	# - Spot anomalies and outliers
	# - Generate hypotheses to test</pre>
		Never skip step 2 (data quality).
		Spend 50% of EDA time on cleaning.
		End EDA with a list of hypotheses.

Descriptive Statistics	<pre># Central Tendency
	df['sales'].mean() # average
	df['sales'].median() # middle value *
	df['sales'].mode()[0] # most frequent
	
	# Spread
	df['sales'].std()
	df['sales'].var()
	df['sales'].min() / df['sales'].max()
	
	# Percentiles
	df['sales'].quantile(0.25) # Q1
	df['sales'].quantile(0.75) # Q3
	df['sales'].quantile([.1,.25,.5,.75,.9])
	
	# Distribution shape
	df['sales'].skew() # >0 right-skewed</pre>
		Skew > 1 → right-skewed → use MEDIAN.
		IQR = Q3 - Q1.
		Values outside Q1-1.5*IQR are outliers.

PYTHON CHEATSHEET

	`df['sales'].kurt()  # kurtosis (peakedness)`	

Bivariate & Correlation	`# Correlation matrix`
	`df.corr()                  # Pearson`
	`df.corr(method='spearman') # rank-based`
	
	`# Interpretation:`
	`# |r| > 0.7  = Strong`
	`# |r| 0.4-0.7= Moderate`
	`# |r| < 0.4  = Weak`
	
	`# Cross-tab (cat vs cat)`
	`pd.crosstab(df['region'], df['status'])`
	`pd.crosstab(df['region'], df['status'],`
	`    normalize='index').round(2)`
	
	`# Group means (num vs cat)`
	`df.groupby('segment')['spend'].agg(`
	`    ['mean','median','std'])`
		Correlation does NOT imply causation. Always look for confounding variables. Spearman works better for non-linear.

Frequency & Pareto	`df['cat'].value_counts()`
	`df['cat'].value_counts(normalize=True)`
	
	`# Top 10 products by revenue`
	`(df.groupby('product')['revenue']`
	`   .sum()`
	`   .sort_values(ascending=False)`
	`   .head(10))`
	
	`# Pareto: 80/20 analysis`
	`rev = (df.groupby('product')['rev']`
	`       .sum()`
	`       .sort_values(ascending=False))`
	`cum_pct = rev.cumsum()/rev.sum()*100`
	
	`# Products driving 80% of revenue`
	`pareto_prods = rev[cum_pct <= 80]`
		Pareto: 20% of SKUs drive 80% of revenue.
		normalize=True gives percentage frequency.
		Use for prioritising decisions.

Time Series	`# Parse dates (always first)`
	`df['date'] = pd.to_datetime(df['date'])`
	
	`# Extract date parts`
	`df['year']    = df['date'].dt.year`
	`df['month']   = df['date'].dt.month`
	`df['weekday'] = df['date'].dt.day_name()`
	`df['quarter'] = df['date'].dt.quarter`
	
	`# Monthly aggregation`
	`monthly = (df.groupby(`
	`    df['date'].dt.to_period('M'))`
	`    ['revenue'].sum())`
	
	`# Rolling average (smooth noise)`
	`df = df.sort_values('date')`
	`df['ma7'] = df['sales'].rolling(7).mean()`
	
	`# MoM % change`
	`monthly_pct = monthly.pct_change()*100`
		Rolling 7-day average removes weekday patterns.
		ALWAYS sort by date before rolling!
		pct_change() calculates % growth.

7. VISUALIZATION — MATPLOTLIB & SEABORN

TOPIC / COMMAND	SYNTAX & CODE	NOTES & EXAMPLE

Matplotlib Setup	`import matplotlib.pyplot as plt`
	
	`# Single plot`
	`fig, ax = plt.subplots(figsize=(10,5))`
	
	`# Multiple subplots`
	`fig, axes = plt.subplots(1, 2,`
	`    figsize=(14,5), sharey=True)`
	
	`# Labels`
	`ax.set_title('Title', fontsize=14,`
	`    fontweight='bold')`
	`ax.set_xlabel('X Label')`
	`ax.set_ylabel('Y Label')`
	`ax.tick_params(axis='x', rotation=45)`
	`ax.legend(loc='upper right')`
	
	`fig.tight_layout()`
	`plt.savefig('chart.png', dpi=150,`
	`    bbox_inches='tight')`
	`plt.show()`
		figsize=(width, height) in inches.
		tight_layout() prevents label overlap.
		bbox_inches='tight' avoids cut-off labels.

Line & Bar Charts	`# Line (trend over time)`
	`ax.plot(x, y, color='steelblue',`
	`    linewidth=2, marker='o', label='Sales')`
	`ax.fill_between(x, y, alpha=0.15)`
	
	`# Bar (category comparison)`
	`ax.bar(cats, vals,`
	`    color='coral', edgecolor='white')`
	
	`# Horizontal bar (long labels)`
	`ax.barh(cats, vals, color='steelblue')`
	
	`# Grouped bar`
	`x = np.arange(len(regions))`
	`ax.bar(x-0.2, q1, width=0.4, label='Q1')`
	`ax.bar(x+0.2, q2, width=0.4, label='Q2')`
	`ax.set_xticks(x)`
	`ax.set_xticklabels(regions)`
		Use horizontal bars for long labels.
		fill_between shows confidence intervals.
		edgecolor='white' separates bars visually.

Scatter & Histogram	`# Scatter (two numeric variables)`
	`ax.scatter(df['ad_spend'],df['revenue'],`
	`    c='steelblue', alpha=0.6, s=30)`
	
	`# Colour by category`
	`for seg, g in df.groupby('segment'):`
	`    ax.scatter(g['x'], g['y'],`
	`               label=seg, alpha=0.7)`
	`ax.legend()`
	
	`# Histogram (distribution)`
	`ax.hist(df['age'], bins=20,`
	`    color='teal', edgecolor='white')`
	
	`# Reference lines`
	`ax.axhline(y=mean_val,`
	`    linestyle='--', color='red')`
	`ax.axvline(x=threshold,`
	`    linestyle=':', color='gray')`
		alpha 0-1: 0=invisible, 0.6 good for scatter.
		axhline/axvline add context to charts.
		bins=20-50 usually works well.

Seaborn Statistical Plots	`import seaborn as sns`
	`sns.set_theme(style='whitegrid')`
	
	`# Distribution`
	`sns.histplot(df['sales'], kde=True)`
	`sns.kdeplot(df['sales'], fill=True)`
	
	`# Box plot (dist by group)`
	`sns.boxplot(data=df, x='region',`
	`    y='sales', hue='quarter')`
	
	`# Violin plot`
		Seaborn = Matplotlib with better defaults.
		violinplot shows full distribution shape.
		pairplot useful for EDA on numerical cols.

<table>
<tr><td></td><td><pre>sns.violinplot(data=df, x='segment',
    y='spend', inner='quart')

Scatter + regression
sns.regplot(data=df,
    x='ad_spend', y='revenue')

All-vs-all
sns.pairplot(df, hue='segment')</pre></td><td></td></tr>
<tr><td>Heatmap & Chart Guide</td><td><pre># Correlation heatmap *
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap(
    df.corr(),
    annot=True, fmt='.2f',
    cmap='coolwarm', center=0,
    square=True, linewidths=0.5,
    ax=ax)

Bar plot with error bars
sns.barplot(data=df, x='region',
    y='revenue', estimator='sum',
    palette='Blues_d')

Count plot
sns.countplot(data=df, x='category',
    order=df['category']
        .value_counts().index)</pre></td><td>Chart guide:
Trend over time -> Line
Compare categories -> Bar
Distribution -> Histogram/Boxplot
Relationship -> Scatter
Correlation matrix -> Heatmap
Part-to-whole -> Stacked bar</td></tr>
<tr><td>Chart Annotation</td><td><pre>fig, ax = plt.subplots(figsize=(12,5))
ax.plot(dates, sales,
    color='steelblue', lw=2)

Mark event
ax.axvline(x=launch,
    linestyle='--',
    color='green', alpha=0.7)

ax.annotate('Campaign +34%',
    xy=(launch, peak),
    xytext=(launch+7, peak*0.9),
    arrowprops=dict(
        arrowstyle='->',
        color='green'),
    fontsize=10, color='green')

Insight as title
ax.set_title(
    'Campaign drove 34% spike Week 3',
    fontsize=12, loc='left')</pre></td><td>Title = the CONCLUSION, not the axis names.
'Revenue by Region' is weak.
'North Underperforms by 18%' is strong.</td></tr>
</table>

PYTHON CHEATSHEET

8. DESCRIPTIVE STATISTICS & DATA STORYTELLING

TOPIC / COMMAND | **SYNTAX & CODE** | **NOTES & EXAMPLE**

Mean vs Median

```
# Symmetric data -> use Mean
income = [40000,45000,50000,55000,60000]
np.mean(income)      # 50000 - ok

# Skewed data -> use Median
income2 = [40000,45000,50000,55000,2000000]
np.mean(income2)     # 438000 - misleading!
np.median(income2)   # 50000  - accurate

# Decision rule:
# mean >> median -> right-skewed -> MEDIAN
# mean << median -> left-skewed  -> MEDIAN
# mean ~= median -> symmetric    -> either

# Report both for first EDA
print(f'Mean: {df.sales.mean():.0f}')
print(f'Med:  {df.sales.median():.0f}')
```

House prices, income, sales -> use MEDIAN.
Test scores (normalised) -> mean is fine.
Always check skewness first.

Std Dev & Percentiles

```
df['sales'].std()      # spread

# Coefficient of Variation
cv = df['s'].std() / df['s'].mean()
# CV > 1 -> high variability

# Percentile table
df['sales'].quantile(
    [.10,.25,.50,.75,.90])

# IQR (robust spread)
IQR = (df['s'].quantile(.75) -
       df['s'].quantile(.25))

# Z-score (standardise)
df['s_z'] = (
    (df['s'] - df['s'].mean()) /
    df['s'].std()
)
# |z| > 3 -> potential outlier
```

IQR is robust to outliers; std dev is not.
Z-scores compare values across different scales.
CV > 1 means std dev > mean (high variation).

Distribution Types

```
# Check skewness
df['sales'].skew()
# > 0  -> right-skewed
# < 0  -> left-skewed
# ~= 0 -> symmetric / normal

# Normality test
from scipy.stats import shapiro
stat, p = shapiro(df['sales'].sample(50))
print('Normal?', p > 0.05)

# Plot always first!
df['sales'].hist(bins=50)

# Common distributions:
# Normal       -> test scores, heights
# Right-skewed -> income, sales, prices
# Left-skewed  -> age at retirement
# Bimodal      -> two mixed populations
```

ALWAYS plot histogram before statistics.
Bimodal = two distinct subgroups; investigate!
Shapiro-Wilk: p > 0.05 = likely normal.

Storytelling Framework

```
# Context -> Insight -> So What -> Action

# 1. CONTEXT
#    Who is the audience?
#    What decision do they need to make?
#    What data was used?

# 2. INSIGHT (one sentence)
#    'North region revenue fell 18% in Q3'

# 3. SO WHAT?
#    Why does this matter to the business?
#    'Q3 target at risk by Rs 2.4 Cr'
```

Start presentation with RECOMMENDATION.
Then show the data that supports it.
Executives read action first.

	`# 4. ACTION`
	`#    What? By whom? By when?`
	`#    'Regional VP reviews distributor`
	`#     contracts by Oct 15th'`	

Dashboard Principles	`# 8 Principles of Effective Dashboards:`
	`#`
	`# 1. One message per chart`
	`# 2. Most important insight -> top-left`
	`# 3. No 3D charts, no pie charts`
	`# 4. Use colour intentionally (1-2 accents)`
	`# 5. Label directly - avoid far-off legends`
	`# 6. Include context: targets, benchmarks`
	`# 7. Every metric -> suggests an action`
	`# 8. Exec -> KPIs only; Analyst -> details`
	
	`# Common mistakes to AVOID:`
	`# x Truncated Y-axis`
	`# x Dual axes without labels`
	`# x Too many colours (>5 categories)`
	`# x Chart titles = axis descriptions`
	`# OK 'Revenue up 23% YoY' (insight title)`
		'Q3 Sales by Region' is weak title. 'North Underperforms by 18%' is strong. Truncated Y-axis makes small diffs look big.

Common EDA Mistakes	`# 1. Correlation != Causation`
	`#    Ice cream sales and drownings both`
	`#    rise in summer - not causal`
	
	`# 2. Removing outliers blindly`
	`#    Outliers = errors OR key business events`
	`#    Always investigate before removing`
	
	`# 3. Mean on skewed data`
	`#    Average salary hides distribution`
	`#    Use median + percentiles`
	
	`# 4. Survivorship bias`
	`#    Analysing only successful customers`
	`#    misses what drove churn`
	
	`# 5. Skipping data quality`
	`#    Bad data -> confident wrong conclusions`
	`#    Validate every column after loading`
		EDA goal = generate hypotheses, not answer them. Spend 30-50% of analysis time on data quality. Document every assumption and data issue found.