Core Python Examples Cheatsheet (Python 3.13 · VS Code)

Chris N John · Cleveland, OH

January 12, 2026

Refresher/Intermediate

Environment: Python 3.13 · VS Code

Includes environment-aware setup, expanded collections, and a pandas mini-project with VS Code debugging.

0) VS Code Setup (Environment-Aware)

Configure tooling so code is consistent, debuggable, and type-safe.

```
// .vscode/settings.json
{
  "python.defaultInterpreterPath": "python",
  "editor.formatOnSave": true,
  "python.formatting.provider": "black",
  "python.analysis.typeCheckingMode": "basic",
  "python.linting.flake8Enabled": true
}
```

1) Syntax & Flow Basics

Build readable control flow with clear conditions and loops.

```
x = 10                                # assignment: store a value for reuse
y = x * 2 + 5                         # expression: compute new value from existing ones

items = []                            # empty list: falsy in condition checks
if not items:                         # truthiness: empty containers evaluate to False
    print("No items")                 # print: basic output for quick checks

value = None                          # None: explicit “no value”
if value is None:                     # identity check: the Pythonic way to test None
    print("Missing")

for n in range(5):                    # range: generate sequence 0..4 for iteration
    if n == 3:
        break                        # break: exit loop early when condition met
else:
    print("Loop completed")           # loop-else: runs only if no break occurred

i = 0
while i < 5:                          # while: loop until condition becomes False
    i += 1                            # increment: move progress forward
    if i % 2 == 0:
        continue                     # continue: skip to next iteration
    print(i)                          # print: inspect odd values

name = "Chris"
```

```
count = 7
print(f"Count: {count:03d}")          # format spec: pad integers to width 3 with zeros
print(f"{name=}")                     # debug syntax: show variable name and value
print(f"Pi: {3.14159:.2f}")           # precision: round/fmt floats to 2 decimals
```

2) Strings & Slicing

Strings represent text; methods help clean, search, and transform.

```
s = "  Cleveland, OH  "
print(s.strip())                      # strip: remove leading/trailing whitespace
print(s.lower().replace(" ", ""))     # lower: normalize case; replace: remove spaces
print(s.casefold())                   # casefold: robust case-insensitive normalize (better than lower
print(s.startswith(" "))              # startswith: prefix check for validation or parsing
print(s.endswith("OH  "))             # endswith: suffix check, often used in filters
print("a,b,c".split(","))             # split: break string into list by delimiter
print("-".join(["a","b","c"]))        # join: combine list of strings with separator
print("mississippi".count("s"))       # count: occurrences of substring

t = "abcdef"
print(t[1:4])                         # slicing: get substring by index range
print(t[::-1])                        # slicing step -1: reverse string quickly
```

3) Collections (Lists, Tuples, Dicts, Sets)

Core data structures: ordered sequences, fixed records, mappings, and unique sets.

```
# --- Lists ---
nums = [3, 1, 2]                      # list literal: ordered, mutable collection
nums.append(4)                        # append: add single item to end
nums.extend([5, 6])                   # extend: add all items from another iterable
nums.insert(0, 99)                    # insert: add at index shifting right
popped = nums.pop()                   # pop: remove & return last item (or index if provided)
nums.remove(2)                        # remove: delete first matching value (ValueError if absent)
print(nums.index(1))                  # index: find first index of value
print(nums.count(3))                  # count: occurrences of a value
nums.sort(reverse=True)               # sort(reverse): descending order
nums.reverse()                        # reverse: in-place reversal
copy_nums = nums.copy()               # copy: shallow copy for independent edits
print(list(reversed(nums)))           # reversed: iterator over items in reverse
print(len(nums), sum([1,2,3]))        # len/sum: built-ins for size and aggregation
print(min([3,1,2]), max([3,1,2]))     # min/max: boundaries
print(sorted([3,1,2]))                # sorted: non-destructive sorted copy

# --- Tuples ---
point = (10, 20)                      # tuple: immutable record
a, b = point                          # unpack: assign components quickly
print(point.count(10), point.index(20)) # count/index: value stats in tuple
cfg = ("localhost", 5432, True)       # tuple: protect fixed configuration

# --- Dicts ---
cfg = {"host": "localhost", "port": 5432}     # dict: key-value mapping
print(cfg.get("user", "postgres"))            # get: safe access with default
cfg["timeout"] = 2.0                          # assignment: set/overwrite value
cfg.update({"ssl": True})                     # update: merge new keys/values
cfg |= {"retries": 3}                         # |=: union update (merge)
print(list(cfg.keys()), list(cfg.values()))    # keys/values: dynamic views
print(list(cfg.items()))                       # items: (key, value) pairs
print(cfg.pop("ssl"))                          # pop: remove key & return value
print(cfg.popitem())                           # popitem: remove & return last inserted pair
```

```
cfg.setdefault("region", "us-east")          # setdefault: get or set default

# --- Sets ---
a = {1, 2, 3}                                # set: unique items, fast membership
a.add(4)                                     # add: insert single value
a.update([4,5,6])                            # update: add many values
a.discard(99)                                # discard: remove if present (no error)
# a.remove(99)                               # remove: raises KeyError if missing
print(a.pop())                               # pop: remove & return arbitrary element
b = {3, 4, 5}
print({1,2,3} | b)                           # union: combine distinct elements
print({1,2,3} & b)                           # intersection: common elements only
print({1,2,3} - b)                           # difference: elements in left not in right
print({1,2,3} ^ b)                           # symmetric difference: elements in exactly one set
print({1,2} <= {1,2,3})                      # subset test
print({1,2,3} >= {1,2})                      # superset test
```

4) collections Helpers (Counter, defaultdict, deque, namedtuple)

Power tools for counting, auto-initialized maps, fast queues, and lightweight records.

```
from collections import Counter, defaultdict, deque, namedtuple

c = Counter("mississippi")                   # Counter: tally occurrences per element
print(c.most_common(2))                      # most_common: top N frequent elements
print(c["s"])                                # direct key count lookup

d = defaultdict(list)                        # defaultdict: create lists for missing keys
d["errors"].append("timeout")                # append: accumulate messages
d["errors"].append("unauthorized")

q = deque([1,2,3], maxlen=4)                 # deque: efficient double-ended queue
q.appendleft(0)                              # appendleft: push to left end
q.append(4)                                  # append: push to right end (auto-drop oldest if full)

Record = namedtuple("Record", ["id", "name"]) # namedtuple: typed, named fields
rec = Record(1, "alice")                     # instantiate: create record
print(rec.id, rec.name)                      # field access
```

5) Comprehensions (List/Dict/Set)

Compact, readable transformations for sequences and mappings.

```
evens_squared = [n*n for n in range(20) if n % 2 == 0]    # list comp: filter & transform
sizes = {name: len(name) for name in ["alice", "bob"]}     # dict comp: build lookup
uniq_lengths = {len(w) for w in ["a", "bb", "ccc", "bb"]} # set comp: unique transformed values
```

6) Functions & Parameters

Encapsulate logic, reuse code, and communicate intent via signatures.

```
def greet(name: str, loud: bool = False) -> str:  # define: callable with typed params
    msg = f"Hello, {name}"                        # f-string: compose greeting
    return msg.upper() if loud else msg           # conditional expression

def add_all(*nums: int) -> int:                   # *args: variable number of positional ints
    return sum(nums)                              # sum: aggregate numeric sequence

def configure(**options):                         # **kwargs: collect arbitrary keyword options
```

```
    return options                                  # return: expose configured settings

def connect(host: str, *, port: int = 5432, ssl: bool = True):  # keyword-only: enforce clarity
    return (host, port, ssl)

print(greet("Chris", loud=True))                    # call: execute function with arguments
print(add_all(1, 2, 3, 4))                          # call: pass variadic inputs
print(configure(retries=3, timeout=2.0))            # call: keyword args become dict entries
print(connect("db", port=6543))                     # call: keyword-only params must be named
```

7) Lambdas & Built-ins

Concise helpers for transforming, filtering, and ordering sequences.

```
records = [{"user": "alice", "score": 10}, {"user": "bob", "score": 7}]
top = sorted(records, key=lambda r: r["score"], reverse=True) # sorted: order items by computed key

nums = [0, 1, 2, 3]
print(any(nums))     # any: True if any element is truthy
print(all(n > 0 for n in nums))  # all: True if all match predicate
pairs = list(zip(["a", "b"], [1, 2]))  # zip: pair elements from multiple iterables
for i, item in enumerate(["x", "y"], start=1):  # enumerate: index items with offset
    print(i, item)
```

8) Modules & Imports

Organize code across files; prefer absolute imports for clarity.

```
import math                   # import: load module
from pathlib import Path      # from-import: bring specific names
# from .utils import slugify  # relative import: use within packages only
```

9) Error Handling & Context Managers

Handle failures gracefully and guarantee resource cleanup.

```
from contextlib import suppress
from pathlib import Path

try:
    int("x")                                    # int: convert string to integer (ValueError if inval
except ValueError as e:                         # except: catch specific exception type
    print(f"Bad value: {e}")                    # print: log error
else:
    print("No exception!")                      # else: only if try-succeeds
finally:
    pass                                        # finally: always runs

with suppress(FileNotFoundError):               # suppress: ignore expected exceptions
    Path("temp.log").unlink()                   # unlink: delete file if present

class locker:
    def __enter__(self):                        # __enter__: acquire resource
        print("Lock acquired")
    def __exit__(self, exc_type, exc, tb):      # __exit__: release resource
        print("Lock released")

with locker():                                  # with: deterministic release
    Path("work.txt").write_text("done")         # write_text: create file with content
```

10) Iterators, Generators & itertools

Lazily produce data to conserve memory and compose pipelines.

```
from itertools import islice, chain, groupby, product, tee

def read_lines(path: str):                      # generator: produce items over time
    with open(path, "r", encoding="utf-8") as f: # with: ensure file closure
        for line in f:                          # iterate: stream lines from file
            yield line.rstrip("\n")             # yield: emit next value lazily

sizes = (len(s) for s in ["a", "abcd", "xyz"])   # generator expr: lazy pipeline
print(sum(sizes))                                # sum: consume generator

first_five = list(islice(range(1000), 5))        # islice: take first N items
nested = [[1, 2], [3], [4, 5]]
flat = list(chain.from_iterable(nested))         # chain: flatten one level
pairs = list(product([1,2], ["a","b"]))        # product: Cartesian product

nums = [("a",1),("a",2),("b",3)]
nums_sorted = sorted(nums, key=lambda x: x[0])   # sorted: prep for groupby
for key, group in groupby(nums_sorted, key=lambda x: x[0]):  # groupby: group consecutive
    print(key, list(group))                      # list: materialize group

it1, it2 = tee(range(3))                         # tee: duplicate iterator
print(list(it1), list(it2))
```

11) Decorators & functools

Cross-cutting behaviors (timing, caching, partials, reduce, singledispatch).

```
import functools, operator, time

@functools.lru_cache(maxsize=128)
def fib(n: int) -> int:                             # lru_cache: memoize results
    return n if n < 2 else fib(n-1) + fib(n-2)

add10 = functools.partial(lambda a, b: a + b, 10) # partial: pre-fill params
print(add10(5))                                     # call: invoke with remaining arg

numbers = [1, 2, 3, 4]
print(functools.reduce(operator.add, numbers))     # reduce: fold sequence into one

@functools.singledispatch
def stringify(obj) -> str:                          # singledispatch: type-based overload
    return str(obj)

@stringify.register(int)
def _(obj: int) -> str:
    return f"int:{obj}"

@stringify.register(list)
def _(obj: list) -> str:
    return f"list(len={len(obj)})"

print(stringify(5), stringify([1,2,3]))           # dispatch: choose impl by type

# Timing decorator
def timed(fn):                                      # decorator: wrap function behavior
```

```
    @functools.wraps(fn)                            # wraps: preserve metadata
    def wrapper(*args, **kwargs):
        start = time.perf_counter()                 # perf_counter: high-res timer
        result = fn(*args, **kwargs)
        print(f"{fn.__name__} took {time.perf_counter() - start:.3f}s")
        return result
    return wrapper

@timed
def work(n: int) -> int:
    return sum(range(n))                            # sum: aggregate numbers

work(1_000_000)
```

12) Object-Oriented Basics & Type Hints

Model state/behavior with classes; make intent explicit with types.

```
class User:
    def __init__(self, name: str):                 # __init__: initialize instance
        self.name = name
    def greet(self) -> str:
        return f"Hello, {self.name}"                # method: instance behavior

class Admin(User):
    def greet(self) -> str:
        return f"[ADMIN] {super().greet()}"         # super: call parent method

from dataclasses import dataclass
@dataclass
class ServerConfig:
    host: str
    port: int = 5432
    ssl: bool = True

from typing import TypedDict, Protocol, TypeVar, Generic, Optional

class UserRow(TypedDict):
    user: str
    score: int

class Greeter(Protocol):
    def greet(self) -> str: ...                     # Protocol: structural typing

T = TypeVar("T")
class Box(Generic[T]):
    def __init__(self, value: T):
        self.value = value                          # Generic: parameterized type

def maybe_port(s: Optional[str]) -> int:
    return int(s) if s is not None else 0           # Optional: handle None explicitly
```

13) Files & I/O (text, CSV, JSON; pathlib)

Read/write common data formats predictably with modern path handling.

```
from pathlib import Path
import csv, json

p = Path("data/example.txt")
```

```
p.parent.mkdir(exist_ok=True)                       # mkdir: ensure directory exists
p.write_text("hello\n", encoding="utf-8")           # write_text: write string to file
print(p.read_text(encoding="utf-8"))                # read_text: read back string

csv_path = Path("data/users.csv")
with csv_path.open("w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["user", "score"])  # DictWriter: write rows from dicts
    writer.writeheader()                           # writeheader: header row
    writer.writerow({"user": "alice", "score": 10}) # writerow: write one record

jpath = Path("data/config.json")
json.dump({"host": "localhost", "port": 5432}, jpath.open("w", encoding="utf-8"))  # dump: serialize to
cfg = json.load(jpath.open("r", encoding="utf-8"))  # load: parse JSON into dict
```

14) datetime & zoneinfo

Handle time correctly across zones; format, parse, and compute durations.

```
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

now = datetime.now(ZoneInfo("America/New_York"))   # ZoneInfo: timezone-aware timestamp
fmt = now.strftime("%Y-%m-%d %H:%M")               # strftime: format datetime into string
parsed = datetime.strptime("2026-01-12 10:30", "%Y-%m-%d %H:%M")  # strptime: parse string into dateti
later = now + timedelta(days=7)                    # timedelta: compute durations/offsets
utc = now.astimezone(ZoneInfo("UTC"))              # astimezone: convert to different timezone
```

15) re (Regex)

Validate and parse text with powerful search patterns and flags.

```
import re

m = re.search(r"(\w+)@(\w+)\.(\w+)", "alice@example.com")  # search: find first match
if m:
    user, domain, tld = m.groups()                         # groups: capture subpatterns
    print(user, domain, tld)

pattern = re.compile(r"^\d{3}-\d{2}-\d{4}$")               # compile: pre-compile pattern
print(bool(pattern.match("123-45-6789")))                  # match: anchor at start

re_flags = re.compile(r"café", flags=re.IGNORECASE)        # IGNORECASE: case-insensitive
print(bool(re_flags.search("CAFÉ")))                       # search: scan entire string
```

16) Logging (handlers & structured format)

Emit consistent logs to console and files for observability.

```
import logging
logger = logging.getLogger("app")                          # getLogger: create named logger
logger.setLevel(logging.INFO)                              # setLevel: control verbosity

fmt = logging.Formatter("%(asctime)s %(levelname)s %(name)s %(message)s")  # Formatter: log structure

sh = logging.StreamHandler()                               # StreamHandler: console logs
sh.setFormatter(fmt)
fh = logging.FileHandler("app.log", encoding="utf-8")      # FileHandler: file logs
fh.setFormatter(fmt)
logger.addHandler(sh)                                      # addHandler: attach outputs
```

```
logger.addHandler(fh)

logger.info("Service started")                              # info: progress event
logger.error("Something failed", exc_info=True)             # error: include stack trace
```

17) Config & CLI (argparse, env, .env pattern)

Read runtime settings from CLI and environment; simple .env loader.

```
import argparse, os

p = argparse.ArgumentParser(description="Example")          # ArgumentParser: define CLI
p.add_argument("--host", default="localhost")               # add_argument: declare option
args = p.parse_args()                                       # parse_args: read CLI inputs

host = os.getenv("APP_HOST", args.host)                     # getenv: read env or fallback
print(host)                                                 # print: chosen host

# Minimal .env loader

def load_env(path=".env"):
    """Load KEY=VALUE lines into os.environ."""
    if not os.path.exists(path):
        return
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            k, _, v = line.partition("=")                   # partition: split into key/value
            os.environ.setdefault(k, v)                     # setdefault: set only if missing
```

18) Subprocess (safe command execution)

Run external commands safely without shell injection; capture outputs.

```
import subprocess

result = subprocess.run(
    ["python", "--version"],                                # argv list: avoid shell=True for safety
    capture_output=True,                                     # capture_output: collect stdout/stderr
    text=True,                                               # text: decode bytes to str
    check=False                                              # check: don’t raise on non-zero exit
)
print(result.stdout.strip())                                # stdout: command output

subprocess.run(["ls", "-la"], check=True)                   # run: execute command, raising on failure
```

Mini-Project Script (mini_report.py)

```
"""
CSV -> Clean -> Aggregate -> Excel
Usage:
  python mini_report.py --in data/scores.csv --out data/report.xlsx
"""

from __future__ import annotations
from pathlib import Path
```

```
import logging
import argparse
import pandas as pd
from datetime import datetime


def setup_logging() -> None:
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s %(levelname)s %(message)s",
        datefmt="%H:%M:%S",
    )


def parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser(description="Generate Excel report from CSV scores.")
    p.add_argument("--in", dest="input", type=str, default="data/scores.csv",
                   help="Input CSV path (default: data/scores.csv)")
    p.add_argument("--out", dest="output", type=str, default="data/report.xlsx",
                   help="Output Excel path (default: data/report.xlsx)")
    return p.parse_args()


def ensure_sample_csv(path: Path) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    if not path.exists():
        path.write_text("user,score\nalice,10\nbob,7\nalice,12\n", encoding="utf-8")


def clean_and_aggregate(df: pd.DataFrame) -> pd.DataFrame:
    df["user"] = df["user"].astype(str).str.strip().str.casefold()
    df["score"] = pd.to_numeric(df["score"], errors="coerce").fillna(0).astype(int)
    agg = df.groupby("user", as_index=False)["score"].sum()
    return agg.sort_values(["score", "user"], ascending=[False, True]).reset_index(drop=True)


def write_excel(df: pd.DataFrame, output_path: Path) -> None:
    output_path.parent.mkdir(parents=True, exist_ok=True)
    with pd.ExcelWriter(output_path, engine="openpyxl") as writer:
        df.to_excel(writer, sheet_name="totals", index=False)
        meta = pd.DataFrame({"generated_at": [datetime.now().strftime("%Y-%m-%d %H:%M:%S")]})
        meta.to_excel(writer, sheet_name="meta", index=False)
    logging.info("Excel report written: %s", output_path)


def main() -> None:
    setup_logging()
    args = parse_args()
    input_path = Path(args.input)
    output_path = Path(args.output)
    ensure_sample_csv(input_path)

    logging.info("Reading CSV: %s", input_path)
    df = pd.read_csv(input_path)
    logging.info("Rows: %d", len(df))

    agg = clean_and_aggregate(df)
    logging.info("Users found: %d", len(agg))

    write_excel(agg, output_path)
    print("\nSummary:")
```

```
    for _, row in agg.iterrows():
        print(f"- {row['user']}: {row['score']}")

    print(f"\nAttachment candidate: {output_path}")


if __name__ == "__main__":
    main()
```

VS Code Launch Configuration (launch.json)

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Mini Report (pandas)",
      "type": "python",
      "request": "launch",
      "program": "${workspaceFolder}/mini_report.py",
      "console": "integratedTerminal",
      "args": ["--in", "data/scores.csv", "--out", "data/report.xlsx"],
      "justMyCode": true
    }
  ]
}
```