Python Cheat Sheet

A high-level, dynamically-typed, multi-paradigm language.
Verified against docs.python.org/3/ · current release line assumed

INSTALL & RUN

```
python3 --version
python3 -m venv .venv
source .venv/bin/activate    # Win:
.venv\Scripts\activate
python3 script.py            # run a
script
python3                      # REPL
(interactive mode)
python3 -c "print(1+1)"      # run an
expression
```

Script (compiled to bytecode, cached in `__pycache__/`) or interactive REPL, both run on the PVM.

CORE TYPES & MUTABILITY

int / float / complex	immutable numbers
str / bytes	immutable text / binary
list	mutable sequence
tuple	immutable sequence
dict	mutable mapping, ordered
set / frozenset	mutable / immutable set
bool	int subclass
None	the null singleton

```
x = 5; y = x; x = 6      # rebind: y
still 5
a = [1]; b = a; a.append(2)  # mutate: b
== [1,2]
x is y   # identity   x == y   # equality
```

NUMBERS

```
7 // 2       # 3  floor division
7 / 2        # 3.5 true division
7 % 2        # 1
2 ** 10      # 1024
0 < x < 10   # chained comparison
1_000_000    # underscores in literals
from decimal import Decimal
from fractions import Fraction
```

STRINGS & F-STRINGS

```
name = "Ada"
f"Hello, {name}!"          # f-string
(default)
f"{value:.2f}"             # format spec
f"{value=}"                # debugging:
value=...
"a,b,c".split(",")
",".join(["a","b"])
s.strip(); s.replace(a,b)
b = "text".encode("utf-8")
s = b.decode("utf-8")
```

COLLECTIONS & COMPREHENSIONS

```
lst = [1, 2, 3]; lst.append(4);
lst.sort()
sorted(lst, reverse=True)      # new list
t = (1, 2); a, b = t           # tuple +
unpack
s = {1, 2} | {2, 3}            # union
d = {"a": 1} | {"b": 2}        # merge
(3.9+)
d.get("a", 0); d.items()
[x*x for x in range(5) if x % 2 == 0]
{x for x in range(5)}
{k: v for k, v in pairs}
```

CONTROL FLOW

```
if x > 0: ...
elif x == 0: ...
else: ...

for item in seq:       # or: while cond:
    if skip: continue
    if done: break
else:
    print("loop finished without break")

match command:
    case "go" | "start": ...
    case Point(x=0, y=0): ...
    case [x, y, *rest]: ...
    case {"type": t}: ...
    case _ if x > 0: ...
    case _: ...
```

FUNCTIONS & SCOPE

```
def f(a, b=0, *args, c, d=1, **kwargs) ->
int:
    """Docstring."""
    return a + b

def g(pos_only, /, normal, *, kw_only):
...
add = lambda a, b: a + b

def f(x, items=None):      # mutable-
default fix
    items = items if items is not None
else []
```

LEGB: Local → Enclosing → Global → Built-in. `global` / `nonlocal` rebind outer names.

ITERATORS, GENERATORS

```
it = iter([1, 2]); next(it)      #
StopIteration at end

def countdown(n):
    while n > 0:
        yield n
        n -= 1

gen = (x*x for x in range(1000000))  #
lazy, O(1) memory
```

MODULES & PACKAGES

```
import pkg.mod
from pkg import mod as m
from . import sibling              #
relative, inside a package

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

pkg/__init__.py marks a package. Import: check `sys.modules` cache → search `sys.path` → compile → run once.

FILES, WITH & PATHLIB

```
with open("f.txt", encoding="utf-8") as
fh:
    text = fh.read()

from pathlib import Path
p = Path("data") / "f.txt"
p.exists(); p.read_text();
p.write_text("x")
```

EXCEPTIONS

```
try:
    risky()
except (ValueError, KeyError) as e:
    handle(e)
except Exception:
    raise
else:
    ok()
finally:
    cleanup()

raise RuntimeError("x") from original_exc

class AppError(Exception): ...
except* TypeError as eg:   # exception
groups (3.11+)
    ...
assert x > 0, "x must be positive"
```

CLASSES & COMMON DUNDERS

```
class Point:
    def __init__(self, x, y): self.x,
self.y = x, y
    def __repr__(self): return
f"Point({self.x},{self.y})"
    def __eq__(self, other): return
(self.x,self.y)==(other.x,other.y)
    def __add__(self, other): return
Point(self.x+other.x, self.y+other.y)
    def __len__(self): return 2
    def __getitem__(self, i): return
(self.x, self.y)[i]
    def __call__(self, *a): ...
    def __enter__(self): return self
    def __exit__(self, *exc): return
False
    @staticmethod
    def origin(): return Point(0, 0)
    @classmethod
    def from_tuple(cls, t): return
cls(*t)
```

INHERITANCE, MRO & SUPER()

```
class Child(Parent):
    def method(self):
        return super().method() + 1

class D(B, C): ...
D.__mro__      # D -> B -> C -> A ->
object (C3)
```

Composition ("has-a") over inheritance ("is-a") when behaviour, not identity, is shared. Mixins add behaviour via multiple inheritance.

PROPERTIES & DESCRIPTORS

```
class Temp:
    def __init__(self, c=0): self._c = c
    @property
    def celsius(self): return self._c
    @celsius.setter
    def celsius(self, v):
        if v < -273.15: raise
ValueError("too cold")
        self._c = v

class Validated:                #
descriptor protocol
    def __set_name__(self, owner, name):
self.name = "_" + name
    def __get__(self, obj, owner): return
obj.__dict__[self.name]
    def __set__(self, obj, value):
obj.__dict__[self.name] = value
```

DECORATORS

```
from functools import wraps

def logged(func):
    @wraps(func)
    def wrapper(*a, **kw):
        print(func.__name__)
        return func(*a, **kw)
    return wrapper

@logged
def greet(name): return f"hi {name}"
```

Metaclasses (`class Meta(type)`) exist but are rarely needed — prefer a class decorator.

TYPE HINTS

```
def add(a: int, b: int = 0) -> int:
return a + b
names: list[str] = []
mapping: dict[str, int] = {}
maybe: int | None = None          # 3.10+
from typing import Optional, Callable,
TypeVar, Protocol
T = TypeVar("T")
Fn = Callable[[int], str]
```

Hints are erased at runtime — checked only by mypy/pyright.

DATACLASSES & ENUM

```
from dataclasses import dataclass, field

@dataclass(frozen=True)
class Point:
    x: int
    y: int = 0
    tags: list =
field(default_factory=list)

from enum import Enum, auto
class Color(Enum):
    RED = auto(); GREEN = auto()
Color.RED.name, Color.RED.value
```

STANDARD LIBRARY INDEX

os, sys, pathlib	OS, argv, paths
json, re, datetime	data & text
collections	Counter, defaultdict
itertools, functools	iteration, reduce, cache
math, random, statistics	numeric helpers
argparse, logging	CLIs, observability

CONCURRENCY & ASYNC

```
import asyncio
async def fetch(): await
asyncio.sleep(1); return "x"
async def main():
    results = await
asyncio.gather(fetch(), fetch())
asyncio.run(main())
```

Threads: I/O-bound (GIL limits CPU-bound). `multiprocessing` / `ProcessPoolExecutor`: CPU-bound. `asyncio`: many concurrent I/O tasks, one thread.

VENV & PIP

```
python3 -m venv .venv
pip install requests
pip freeze > requirements.txt
pip install -r requirements.txt
```

TESTING WITH PYTEST & MOCKING

```
pip install pytest
def test_add(): assert add(1, 2) == 3

@pytest.fixture
def client():
    yield make_client()   # setup /
teardown

@pytest.mark.parametrize("a,b,exp",
[(1,1,2),(2,3,5)])
def test_add_many(a, b, exp): assert
add(a,b) == exp

from unittest.mock import Mock, patch
m = Mock(return_value=42)
with patch("mod.func", return_value=1) as
p:
    mod.func(); p.assert_called_once()

def test_env(monkeypatch):
    monkeypatch.setenv("KEY", "val")
```

pytest -v -k expr · conftest.py for shared fixtures · mock (fake), stub (canned), spy (records real calls)