Python 3 Quick Reference

Syntax, idioms, and standard library essentials. Targeting Python 3.11+ — covers what you actually use day to day.

Built-in Types & Operators

```
# Numeric
x = 5; y = 2.5; z = 1 + 2j # int, float, complex
17 // 5 # 3 (floor division)
17 % 5 # 2 (modulo)
2 ** 10 # 1024 (power)

# Strings
s = "hello"
s.upper(); s.split(); s.startswith("he")
f"x = {x:.2f}" # formatted: "x = 5.00"
"a" in "alphabet" # True
"ab" * 3 # "ababab"

# Booleans / None
True, False, None
0, "", [], {}, None → all falsy
```

Collections

```
lst = [1,2,3]; lst.append(4); lst[-1]; lst[1:3]
lst.sort(); sorted(lst, reverse=True)
[x*2 for x in lst if x>1] # comprehension

t = (1,2,3) # immutable tuple
a, b, c = t # unpacking
a, *rest = [1,2,3,4] # rest = [2,3,4]

d = {"a":1, "b":2}
d.get("c", 0) # default if missing
d.setdefault("c", []).append(1)
{k:v*2 for k,v in d.items()} # dict comprehension

set([1,2,2,3]) → {1,2,3}
{1,2} | {2,3} # union {1,2,3}
{1,2} & {2,3} # intersection {2}
```

Control Flow

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

for i, val in enumerate(lst):
if val < 0: continue
if val > 100: break
else:
# runs only if loop wasn't broken
pass

while cond:
...

# match statement (3.10+)
match cmd:
case "start": ...
```

```
case "stop" | "quit": ...
case _: ...
```

Functions

```
def greet(name: str, loud: bool = False) -> str:
"""Docstring."""
return name.upper() if loud else name

# *args, **kwargs
def f(*args, **kwargs):
print(args, kwargs)

# lambda
sq = lambda x: x*x

# decorator
def log(fn):
def wrap(*a, **kw):
print(fn.__name__, a, kw)
return fn(*a, **kw)
return wrap

@log
def add(a, b): return a+b
```

Classes

```
from dataclasses import dataclass

@dataclass
class Point:
x: float
y: float
def distance(self):
return (self.x**2 + self.y**2) ** 0.5

class Animal:
species = "unknown" # class var
def __init__(self, name):
self.name = name # instance var
def speak(self):
raise NotImplementedError

class Dog(Animal):
species = "canine"
def speak(self):
return f"{self.name} says woof"
```

Errors & Context Managers

```
try:
risky()
except (ValueError, KeyError) as e:
log.exception(e)
except Exception:
raise # re-raise current
else:
print("no error")
finally:
cleanup()

with open("f.txt") as f:
data = f.read() # auto-closed

# custom context manager
from contextlib import contextmanager
@contextmanager
def timer():
import time; t = time.perf_counter()
yield
print(time.perf_counter() - t)
```

Standard Library Highlights

`os, os.path, pathlib`	Filesystem and paths
`sys`	Interpreter access (argv, exit, stdin)
`json`	JSON serialization
`csv`	CSV reading/writing
`re`	Regular expressions
`datetime, time`	Date/time handling
`collections`	Counter, defaultdict, deque, namedtuple
`itertools`	chain, groupby, product, permutations
`functools`	lru_cache, partial, reduce

`subprocess`	Run external commands
`shutil`	High-level file ops (copy, move, rmtree)
`argparse`	CLI argument parsing
`logging`	Structured logging
`asyncio`	Async I/O loop and primitives
`typing`	Type hints (Optional, Union, Callable)
`dataclasses`	Lightweight class generation
`enum`	Enum and Flag classes
`pickle`	Object serialization
`sqlite3`	Embedded SQL database
`http.server`	Quick HTTP server
`socket`	Low-level networking
`threading, multiprocessing`	Concurrency
`unittest, doctest`	Testing frameworks

Common Idioms

```
# Swap
a, b = b, a

# Read file lines without trailing \n
lines = open("f").read().splitlines()

# Iterate dict in sorted order
for k in sorted(d):
print(k, d[k])

# Default empty list per key
from collections import defaultdict
groups = defaultdict(list)
for x in items:
groups[x.category].append(x)

# Run code only when invoked directly
if __name__ == "__main__":
main()

# Type hint a function
def parse(line: str) -> dict[str, int]:
...

# Walrus (3.8+)
while (chunk := f.read(8192)):
process(chunk)
```

Async / Await

```
import asyncio

async def fetch(url):
async with session.get(url) as r:
return await r.text()

async def main():
pages = await asyncio.gather(*(fetch(u) for u in urls))
```

```
asyncio.run(main())
```