Python Idioms & Mastery Cheat Sheet

List Comprehensions

Syntax: `[expr for item in iterable if condition]`
Returns NEW list. Eager evaluation (creates list immediately).

```
# Basic - squares
[x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# With condition (filter)
[x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]

# With if-else (transform)
[x if x > 0 else 0 for x in nums]

# Nested loops (outer first, inner second)
[(x, y) for x in range(3) for y in range(2)]
# [(0,0), (0,1), (1,0), (1,1), (2,0), (2,1)]

# 2D matrix
[[i*j for j in range(5)] for i in range(5)]

# Flatten 2D list
[x for row in matrix for x in row]

# Multiple conditions
[x for x in range(20) if x % 2 == 0 if x % 3 == 0]
```

String Manipulation & Slicing

Strings are immutable. Methods return NEW strings.

```
# Join list to string
''.join(['a', 'b', 'c'])  # 'abc'
','.join(['a', 'b'])      # 'a,b'

# Split string into list
'a,b,c'.split(',')        # ['a', 'b', 'c']
'hello world'.split()     # ['hello', 'world'] (
    default on whitespace)

# Formatting (f-strings, Python 3.6+)
name = "world"
f"Hello, {name}"
f"{3.14159:.2f}"          # Format float to 2 decimal
    places: "3.14"
f"{42:05d}"               # Pad integer with zeros to
    width 5: "00042"

# Slicing [start:stop:step] (works on lists too)
s = "abcdefg"
s[2:5]    # "cde" (from index 2 up to 5)
s[:4]     # "abcd" (from start up to 4)
s[3:]     # "defg" (from index 3 to end)
s[-3:]    # "efg" (last 3 characters)
s[:-1]    # "abcdef" (all but the last)
s[::2]    # "aceg" (every 2nd character)
s[::-1]   # "gfedcba" (reverse the string)

# Common string methods
s.strip()           # remove leading/trailing
    whitespace
s.lstrip()          # remove leading whitespace only
s.rstrip()          # remove trailing whitespace only
s.startswith('pre') # check if starts with prefix
s.endswith('.txt')  # check if ends with suffix
s.replace('old', 'new')  # replace all occurrences
s.lower()           # convert to lowercase
s.upper()           # convert to uppercase
```

Set Operations

Fast, unordered collections of unique, hashable elements.

```
# O(1) average time complexity for add, remove,
    contains
a = {1, 2, 3}
b = {2, 3, 4}
# Intersection
a & b  # {2, 3}
# Union
a | b  # {1, 2, 3, 4}
# Difference
a - b  # {1}
# Symmetric Difference (XOR)
a ^ b  # {1, 4}
```

Generator Expressions

Syntax: `(expr for item in iterable if condition)`
Returns generator object. Lazy evaluation (computes on demand). Memory efficient for large data.

```
# Generator (not evaluated until iterated)
gen = (x**2 for x in range(1000000))
# No memory used until you iterate

# Convert to list
list(gen)  # Now evaluates and uses memory

# Use directly in functions (no parentheses needed)
sum(x**2 for x in range(100))
max(x for x in nums if x > 0)
any(x > 10 for x in nums)

# One-time use (generator exhausted after iteration)
gen = (x for x in range(5))
list(gen)  # [0, 1, 2, 3, 4]
list(gen)  # [] - already exhausted!
```

Dict Comprehensions

Syntax: `{key: val for item in iterable}`
Returns NEW dict. Keys must be unique (duplicates overwritten).

```
# Square mapping
{x: x**2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Filter by value
{k: v for k, v in d.items() if v > 10}

# Swap keys and values (keys must be hashable)
{v: k for k, v in original.items()}

# From two lists
{k: v for k, v in zip(keys, values)}

# Conditional values
{k: ('even' if v % 2 == 0 else 'odd')
 for k, v in d.items()}
```

Set Comprehensions

Syntax: `{expr for item in iterable}`
Returns NEW set. Automatically removes duplicates. Unordered.

```
# Unique squares
{x**2 for x in [-2, -1, 0, 1, 2]}
# {0, 1, 4}

# Unique characters (case-insensitive)
{c.lower() for c in "Hello World"}
# {'h', 'e', 'l', 'o', ' ', 'w', 'r', 'd'}

# Elements must be hashable
{x for x in nums if x > 0}
```

min() / max()

Syntax: `min/max(iterable, key=func, default=val)`
Returns single value. Raises ValueError if empty (unless default).

```
# Basic
min([3, 1, 4, 1, 5])  # 1
max([3, 1, 4, 1, 5])  # 5

# With generator (memory efficient)
min(x**2 for x in range(1000000))

# Find minimum over range (DP pattern)
min(dp[i][mid][1] + dp[mid+1][j][p-1]
    for mid in range(i, j))

# With condition
min(x for x in nums if x > 0)
# Raises ValueError if no x > 0

# Default for empty iterable
min(nums, default=float('inf'))
min((x for x in nums if x > 0), default=0)

# Custom key function (doesn't change return value)
min(words, key=len)  # returns shortest word itself
max(students, key=lambda s: s.grade)  # returns
    student

# Get index of min/max
min_idx = min(range(len(nums)), key=lambda i: nums[i])
max_idx = max(range(len(nums)), key=lambda i: nums[i])

# Multiple arguments (not iterable)
min(3, 1, 4)  # 1
max(3, 1, 4)  # 4
```

sum()

Syntax: `sum(iterable, start=0)`
Returns sum of all elements + start value. Only for numbers.

```
# Basic
sum([1, 2, 3, 4])  # 10

# With condition (count even numbers)
sum(x for x in range(100) if x % 2 == 0)

# Count occurrences (True=1, False=0)
sum(1 for x in nums if x > 10)
sum(x > 10 for x in nums)  # same thing
```

```python
# Dot product
sum(a * b for a, b in zip(list1, list2))

# Start value (initial accumulator)
sum([1, 2, 3], start=100)  # 106

# Don't use for strings (use ''.join() instead)
# sum(['a', 'b'], start='')  # ERROR!
''.join(['a', 'b'])  # Correct: 'ab'
```

any() / all()

Syntax: `any/all(iterable)`
Returns bool. Short-circuits (stops early when possible).

```python
# any: True if AT LEAST ONE element is True
any([False, False, True, False])  # True
any([False, False, False])  # False
any([])  # False (empty iterable)

# Check if any element satisfies condition
any(x > 10 for x in nums)
any(word.startswith('A') for word in words)

# all: True if ALL elements are True
all([True, True, True])  # True
all([True, False, True])  # False
all([])  # True (vacuous truth - empty iterable)

# Check if all elements satisfy condition
all(x > 0 for x in nums)
all(len(s) > 5 for s in strings)

# Short-circuit example
any(expensive_check(x) for x in huge_list)
# Stops at first True, doesn't check rest

# Check if iterable is non-empty
if any(nums):  # True if nums has elements
if not any(nums):  # True if nums is empty
```

sorted() vs list.sort()

sorted(): Function. Returns NEW list. Works on any iterable.
list.sort(): Method. Modifies IN-PLACE. Only for lists. Returns None.

```python
# sorted() - returns new list, original unchanged
nums = [3, 1, 2]
result = sorted(nums)  # [1, 2, 3]
print(nums)  # [3, 1, 2] - unchanged

# Works on any iterable
sorted({3, 1, 2})  # [1, 2, 3] - list from set
sorted("cab")  # ['a', 'b', 'c'] - list from string
sorted({'z': 1, 'a': 2})  # ['a', 'z'] - keys only

# list.sort() - modifies in place, returns None
nums = [3, 1, 2]
result = nums.sort()  # result is None
print(nums)  # [1, 2, 3] - modified

# Both accept same arguments
sorted(data, key=lambda x: x[1], reverse=True)
data.sort(key=lambda x: x[1], reverse=True)
```

```python
# Custom key function
sorted(words, key=len)  # by length
sorted(words, key=str.lower)  # case-insensitive
sorted(students, key=lambda s: s.grade)

# Using operator module (often faster than lambda)
from operator import itemgetter, attrgetter
# Sort list of tuples by 2nd element
sorted(data, key=itemgetter(1))
# Sort list of objects by 'grade' attribute
sorted(students, key=attrgetter('grade'))
# Multiple criteria
sorted(students, key=attrgetter('grade', 'name'))

# Multiple criteria (tuples compared element-wise)
sorted(people, key=lambda p: (p.age, p.name))
sorted(people, key=lambda p: (-p.age, p.name))  # desc
    age

# Sort dict by value
sorted(d.items(), key=lambda x: x[1])

# Reverse
sorted(nums, reverse=True)
nums.sort(reverse=True)

# Stability: equal elements keep original order
```

map() and filter()

map(): Apply function to all elements. Returns map object (iterator).
filter(): Keep elements where function returns True. Returns filter object.

```python
# map - apply function to each element
result = map(str.upper, words)  # map object
list(result)  # ['HELLO', 'WORLD']

# With lambda
list(map(lambda x: x**2, [1, 2, 3]))  # [1, 4, 9]

# Multiple iterables (stops at shortest)
list(map(lambda x, y: x + y, [1,2,3], [10,20,30]))
# [11, 22, 33]

# filter - keep elements where function is True
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))
# [1, 2]

list(filter(str.isdigit, ['1', 'a', '2', 'b']))
# ['1', '2']

# None as function: filters out falsy values
list(filter(None, [0, 1, '', 'hi', False, True]))
# [1, 'hi', True]

# Modern Python: prefer comprehensions (more readable)
[x**2 for x in nums]  # instead of map
[x for x in nums if x > 0]  # instead of filter

# But map/filter useful for passing existing functions
list(map(int, ['1', '2', '3']))  # [1, 2, 3]
list(filter(str.isalpha, chars))
```

zip()

Syntax: `zip(*iterables, strict=False)`
Returns zip object (iterator). Stops at shortest iterable (unless strict=True in Python 3.10+).

```python
# Combine multiple iterables
list(zip([1,2,3], ['a','b','c']))
# [(1,'a'), (2,'b'), (3,'c')]

# Stops at shortest
list(zip([1,2,3,4], ['a','b']))  # [(1,'a'), (2,'b')]

# Python 3.10+: strict mode (raises if lengths differ)
list(zip([1,2,3], ['a','b'], strict=True))  #
    ValueError

# Unzip (transpose) - use * unpacking
pairs = [(1,'a'), (2,'b'), (3,'c')]
nums, chars = zip(*pairs)
# nums = (1, 2, 3), chars = ('a', 'b', 'c')

# Parallel iteration
for x, y in zip(list1, list2):
    print(x + y)

# Create dict from two lists
dict(zip(keys, values))

# Enumerate alternative (with indices)
for i, val in zip(range(len(nums)), nums):
    print(i, val)

# Zip three or more iterables
list(zip([1,2], ['a','b'], [10,20]))
# [(1,'a',10), (2,'b',20)]
```

enumerate()

Syntax: `enumerate(iterable, start=0)`
Returns enumerate object. Yields (index, value) tuples.

```python
# Get index and value
for i, val in enumerate(['a', 'b', 'c']):
    print(i, val)
# 0 a
# 1 b
# 2 c

# Start from different index
for i, val in enumerate(['a', 'b', 'c'], start=1):
    print(i, val)
# 1 a
# 2 b
# 3 c

# In comprehension - create dict
{i: val for i, val in enumerate(nums)}

# Find indices where condition is true
[i for i, x in enumerate(nums) if x > 10]

# Get both index and value in one loop
for i, (key, val) in enumerate(d.items()):
    print(f"{i}: {key} = {val}")
```

Lambda Functions

Syntax: `lambda args: expression`
Anonymous function. Single expression only (no statements). Returns expression value.

```
# Basic lambda
square = lambda x: x**2
square(5)  # 25

# Multiple arguments
add = lambda x, y: x + y
add(3, 4)  # 7

# No arguments
get_pi = lambda: 3.14159

# In sorting
sorted(data, key=lambda x: x[1])
sorted(students, key=lambda s: (s.grade, s.name))

# In map/filter
list(map(lambda x: x*2, nums))
list(filter(lambda x: x > 0, nums))

# Conditional expression (ternary)
f = lambda x: 'even' if x % 2 == 0 else 'odd'

# CANNOT do multi-line or statements
# lambda x: print(x)  # ERROR - print is statement
# lambda x: x = x + 1  # ERROR - assignment

# Use def for anything complex
def complex_func(x):
    if x > 0:
        return x**2
    else:
        return -x
```

Unpacking

Syntax: `a, *b, c = iterable`
*Extract values from iterables. * captures remaining elements as list.*

```
# Basic unpacking
a, b, c = [1, 2, 3]  # a=1, b=2, c=3
a, b = b, a  # swap without temp variable

# Must match count (or use *)
# a, b = [1, 2, 3]  # ERROR - too many values

# Extended unpacking with *
first, *middle, last = [1,2,3,4,5]
# first=1, middle=[2,3,4], last=5

first, *rest = [1,2,3]  # first=1, rest=[2,3]
*most, last = [1,2,3]  # most=[1,2], last=3

# Only one * allowed per unpacking
# *a, *b = [1,2,3]  # ERROR

# Ignore values with _ (convention)
a, _, c = [1, 2, 3]  # ignore middle value
_, b, _ = [1, 2, 3]  # only want middle

# Nested unpacking
(a, b), (c, d) = [(1, 2), (3, 4)]

# In function calls - unpack arguments
args = [1, 2, 3]
func(*args)  # same as func(1, 2, 3)

kwargs = {'x': 1, 'y': 2}
func(**kwargs)  # same as func(x=1, y=2)

# In function definitions - collect arguments
def func(*args, **kwargs):
    print(args)    # tuple of positional args
    print(kwargs)  # dict of keyword args

# Positional-only and keyword-only arguments
def f(pos1, /, pos_or_kw, *, kw1):
    # pos1 must be positional
    # pos_or_kw can be either
    # kw1 must be keyword
    pass

# Dict unpacking (merge dicts)
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'b': 4}  # 'b' overwrites
d3 = {**d1, **d2}  # {'a': 1, 'b': 4, 'c': 3}

# List unpacking
[1, 2, *[3, 4], 5]  # [1, 2, 3, 4, 5]
```

Walrus Operator :=

Syntax: `name := expression`
Assignment expression (Python 3.8+). Assigns AND returns value.

```
# Assign and use in same expression
if (n := len(nums)) > 10:
    print(f"Large list: {n} elements")

# Avoid repeated computation
if (result := expensive_func()) > 0:
    print(f"Result: {result}")

# In comprehension - reuse computed value
[y for x in nums if (y := x**2) > 100]
# Without :=, would need: [x**2 for x in nums if x**2
    > 100]

# While loop with assignment (replaces assignment +
    check)
while (line := file.readline()):
    process(line)
# Old way:
# line = file.readline()
# while line:
#     process(line)
#     line = file.readline()

# In list comprehension for filtering + transforming
[y for x in data if (y := transform(x)) is not None]

# Cannot use in regular assignment context
# (x := 5) + 2  # OK - returns 7
# x := 5  # ERROR - use x = 5 for plain assignment
```

itertools (must import)

Import: `from itertools import *`
Efficient iterators for combinatorics and iteration patterns.

```
from itertools import *

# combinations - r-length tuples, in sorted order, no
    repeats
list(combinations([1,2,3], 2))
# [(1,2), (1,3), (2,3)]
list(combinations('ABC', 2))
# [('A','B'), ('A','C'), ('B','C')]

# permutations - r-length tuples, all orderings
list(permutations([1,2,3], 2))
# [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]
list(permutations([1,2,3]))  # r=3 (full length)
# [(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2),
    (3,2,1)]

# product - Cartesian product (nested loops)
list(product([1,2], ['a','b']))
# [(1,'a'), (1,'b'), (2,'a'), (2,'b')]
list(product(range(2), repeat=3))  # 3 nested loops
# [(0,0,0), (0,0,1), (0,1,0), ...]

# combinations_with_replacement - combinations
    allowing repeats
list(combinations_with_replacement([1,2], 2))
# [(1,1), (1,2), (2,2)]

# chain - flatten/concatenate iterables
list(chain([1,2], [3,4], [5]))  # [1,2,3,4,5]
list(chain.from_iterable([[1,2], [3,4]]))  # same

# accumulate - running totals (or other operations)
list(accumulate([1,2,3,4]))  # [1, 3, 6, 10]
list(accumulate([1,2,3,4], lambda x,y: x*y))  #
    [1,2,6,24]

# groupby - group consecutive equal elements (MUST
    sort first!)
data = [('a',1), ('a',2), ('b',1), ('a',3)]
for key, group in groupby(sorted(data, key=lambda x: x
    [0]),
                          key=lambda x: x[0]):
    print(key, list(group))
# a [('a',1), ('a',2), ('a',3)]
# b [('b',1)]

# islice - slice an iterator
list(islice(range(100), 5))  # [0,1,2,3,4]
list(islice(range(100), 5, 10))  # [5,6,7,8,9]

# cycle - repeat indefinitely
# list(cycle([1,2,3]))  # infinite!
list(islice(cycle([1,2,3]), 7))  # [1,2,3,1,2,3,1]

# repeat - repeat value n times
list(repeat(10, 3))  # [10, 10, 10]

# takewhile - take while condition is true
list(takewhile(lambda x: x < 5, [1,3,5,2,4]))  # [1,3]

# dropwhile - drop while condition is true
list(dropwhile(lambda x: x < 5, [1,3,5,2,4]))  #
    [5,2,4]

# pairwise - sliding window of size 2 (Python 3.10+)
list(pairwise('ABCDE')) # [('A','B'), ('B','C'), ('C
    ','D'), ('D','E')]
```

functools

Import: `from functools import *`
Higher-order functions and operations on callables.

```
from functools import reduce, partial

# reduce - apply function cumulatively (fold)
reduce(lambda x, y: x + y, [1,2,3,4])
# ((1+2)+3)+4 = 10

# With initial value
reduce(lambda x, y: x + y, [1,2,3], 100)  # 106

# Product of all elements
reduce(lambda x, y: x * y, nums)

# Maximum (equivalent to max())
reduce(lambda x, y: x if x > y else y, nums)

# Note: often better to use sum(), math.prod(), max()
import math
math.prod([1,2,3,4])  # 24 (Python 3.8+)

# partial - fix some arguments of a function
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)
square(5)  # 25
cube(5)   # 125

# Useful with map
list(map(partial(int, base=2), ['101', '110']))
# [5, 6] - binary to decimal

# lru_cache - memoization decorator
from functools import lru_cache

@lru_cache(maxsize=128)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
# Caches results, much faster for repeated calls

# cmp_to_key - for complex comparison logic (Py3)
from functools import cmp_to_key

def compare_items(a, b):
    # return -1 if a < b, 0 if a == b, 1 if a > b
    if a.priority < b.priority: return -1
    if a.priority > b.priority: return 1
    return 0

sorted(items, key=cmp_to_key(compare_items))
```

Common Patterns

```
# Flatten 2D list
flat = [x for row in matrix for x in row]

# Transpose matrix (zip with unpacking)
transposed = list(zip(*matrix))
# [[1,2,3], [4,5,6]] -> [(1,4), (2,5), (3,6)]

# Count occurrences
from collections import Counter
counts = Counter(items)  # dict-like {item: count}
counts.most_common(3)  # top 3 most common

# Frequency dict manually
freq = {}
for item in items:
    freq[item] = freq.get(item, 0) + 1

# Group by key (using defaultdict)
from collections import defaultdict
groups = defaultdict(list)
for item in items:
    groups[key_func(item)].append(item)

# Group by key (using itertools - must sort first!)
from itertools import groupby
groups = {k: list(v) for k, v
          in groupby(sorted(items, key=fn), key=fn)}

# Find index of element
idx = nums.index(42)  # finds first occurrence, raises
      ValueError if not found

# Find index of max/min
max_idx = max(range(len(nums)), key=lambda i: nums[i])
min_idx = min(range(len(nums)), key=lambda i: nums[i])

# All indices where condition is true
indices = [i for i, x in enumerate(nums) if x > 10]

# Conditional expression (ternary operator)
result = x if condition else y
result = x if cond1 else y if cond2 else z  # chaining

# Multiple assignment
a = b = c = 0  # all equal to 0
a, b, c = 1, 2, 3  # different values

# Check if all elements same
all(x == lst[0] for x in lst)

# Remove duplicates while preserving order
seen = set()
result = [x for x in items if not (x in seen or seen.
     add(x))]
# Or: list(dict.fromkeys(items))

# Chunking list into n-sized pieces
chunks = [lst[i:i+n] for i in range(0, len(lst), n)]

# Sliding window
windows = [lst[i:i+k] for i in range(len(lst)-k+1)]

# Sliding window (efficient with deque)
from collections import deque
dq = deque(maxlen=k)
for x in lst:
    dq.append(x) # process the window
    if len(dq) == k:
        process(list(dq))

# Reverse string/list
reversed_str = s[::-1]
reversed_list = lst[::-1]
# Or: ''.join(reversed(s)), list(reversed(lst))

# Invert dictionary (values become keys)
# Note: if values are not unique, some data is lost
inv_map = {v: k for k, v in my_map.items()}

# Invert dictionary with non-unique values
from collections import defaultdict
inv_map = defaultdict(list)
for k, v in my_map.items():
    inv_map[v].append(k)

# Dictionary get with default
val = d.get(key, 'default_value') # returns default if
      key not found

# Type checking
isinstance(x, int) # check if x is an integer
isinstance(x, (list, tuple)) # check against multiple
     types

# Char to ASCII and back
ord('A')    # 65
chr(65)     # 'A'

# Script structure (__name__ == "__main__" guard)
def main():
    # Main logic here
    print("Running as a script")

if __name__ == "__main__":
    # This code only runs when file is executed
     directly
    # It does NOT run when the file is imported as a
     module
    main()
```

collections module

```
from collections import *

# defaultdict - dict with default value
d = defaultdict(int)  # default 0
d['key'] += 1  # no KeyError

d = defaultdict(list)  # default []
d['key'].append(1)

d = defaultdict(lambda: 'N/A')  # custom default

# Counter - count occurrences
c = Counter(['a', 'b', 'a', 'c', 'b', 'a'])
# Counter({'a': 3, 'b': 2, 'c': 1})
c.most_common(2)  # [('a', 3), ('b', 2)]
c['d']  # 0 (missing keys return 0)

# deque - double-ended queue (efficient at both ends)
from collections import deque
dq = deque([1, 2, 3])
dq.append(4)       # add right: [1,2,3,4]
dq.appendleft(0)   # add left: [0,1,2,3,4]
dq.pop()           # remove right: returns 4
dq.popleft()       # remove left: returns 0

# Max length (automatically discards old items)
last_five = deque(maxlen=5)
last_five.append(1) # deque([1], maxlen=5)
last_five.extend([2,3,4,5,6]) # deque([2,3,4,5,6])

# namedtuple - tuple with named fields
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

```
p = Point(1, 2)
print(p.x, p.y)  # 1 2
```

heapq module (min-heap)

Import: `import heapq`
Implements a min-heap. Operates on a standard list IN-PLACE.

```
import heapq
nums = [3, 1, 4, 1, 5, 9, 2, 6]

# Convert list to heap (in-place, O(n))
heapq.heapify(nums)  # nums is now a min-heap

# Push item onto heap (O(log n))
heapq.heappush(nums, 0)

# Pop smallest item from heap (O(log n))
smallest = heapq.heappop(nums)  # returns 0

# Peek at smallest item (O(1))
smallest = nums[0]

# Push and pop in one step (more efficient)
replaced = heapq.heappushpop(nums, 7)

# Find n largest/smallest elements (O(n log k))
heapq.nlargest(3, nums)   # e.g. [9, 6, 5]
heapq.nsmallest(3, nums)  # e.g. [1, 1, 2]

# Common use: Top K elements
# Keep a min-heap of size k. For each new element,
# if it's larger than the smallest in heap (heap[0]),
# pop and push the new element.
k = 3
heap = []
for num in [10, 2, 8, 1, 9, 4, 12]:
    if len(heap) < k:
        heapq.heappush(heap, num)
    elif num > heap[0]:
        heapq.heappushpop(heap, num)
# heap now contains the top k elements

# Max heap pattern: negate values
max_heap = []
heapq.heappush(max_heap, -val)  # push negated value
max_val = -heapq.heappop(max_heap)  # pop and negate
    back
```

bisect module (binary search)

Import: `import bisect`
Maintains a list in sorted order without expensive resort operations.

```
import bisect
a = [1, 2, 4, 5]

# Find insertion point to maintain order (O(log n))
bisect.bisect_left(a, 3)   # returns 2 (for value 3)
bisect.bisect_right(a, 2)  # returns 2 (after existing
    2)
bisect.bisect(a, 2)        # same as bisect_right

# Insert item while maintaining order (O(n))
# The search is O(log n), but list.insert is O(n)
bisect.insort_left(a, 3)   # a is now [1, 2, 3, 4, 5]
bisect.insort_right(a, 2)  # a is now [1, 2, 2, 3, 4,
    5]
bisect.insort(a, 2)        # same as insort_right

# Example: Grade ranges
def grade(score, breakpoints=[60, 70, 80, 90],
          grades='FDCBA'):
    i = bisect.bisect(breakpoints, score)
    return grades[i]

grade(85) # 'B'
```

regex (re module)

Import: `import re`
Pattern matching and text manipulation with regular expressions.

```
import re

# Find all matches
re.findall(r'\d+', 'Order 123, Item 456')  # ['123',
    '456']
re.findall(r'\w+', 'hello world')  # ['hello', 'world
    ']

# Search for pattern (returns Match object or None)
match = re.search(r'\d+', 'Item 123')
if match:
    print(match.group())  # '123'

# Replace patterns
re.sub(r'\s+', ' ', 'hello   world')  # 'hello world'
re.sub(r'\d+', 'X', 'Room 101')  # 'Room X'

# Split by pattern
re.split(r'[,;]', 'a,b;c')  # ['a', 'b', 'c']

# Match at start of string
if re.match(r'\d+', '123abc'):  # matches
    print("Starts with digits")

# Compile for reuse (more efficient)
pattern = re.compile(r'\d+')
pattern.findall('123 and 456')  # ['123', '456']
```

Essential I/O

Patterns for file handling, data interchange, and script input.

```
import json, sys
from pathlib import Path

# JSON (data interchange)
data = {'name': 'Alice', 'id': 123}
json_str = json.dumps(data)      # dict to string
parsed_dict = json.loads(json_str) # string to dict

# pathlib (modern file paths, Python 3.4+)
path = Path("my_dir/my_file.txt")
# path.read_text()
# path.write_text("New content")
parent_dir = path.parent # Path("my_dir")
new_path = path.with_suffix(".md") # my_dir/my_file.md

# sys (stdin/argv for competitive programming)
# In terminal: python script.py arg1 arg2
script_name = sys.argv[0] # "script.py"
first_arg = sys.argv[1]   # "arg1"

# Read from standard input
for line in sys.stdin:
    print(line.strip())
```

math module

Import: `import math`
Mathematical functions and constants.

```
import math

# Rounding
math.ceil(3.2)    # 4 (round up)
math.floor(3.8)   # 3 (round down)
round(3.5)        # 4, round(2.5) # 2 (rounds to
    nearest even)

# Power and roots
math.pow(2, 3)    # 8.0 (returns float)
math.sqrt(16)     # 4.0
2 ** 3            # 8 (built-in operator, preserves int
    )

# GCD and LCM
math.gcd(12, 8)   # 4
math.lcm(12, 8)   # 24 (Python 3.9+)

# Product (Python 3.8+)
math.prod([1, 2, 3, 4])  # 24

# Constants
math.pi           # 3.141592653589793
math.e            # 2.718281828459045
math.inf          # infinity
math.isfinite(x) # check if finite number

# Quotient and remainder (built-in)
divmod(17, 5)  # (3, 2)
```

Generator Functions

Syntax: `def func(): yield val`
A function that returns a generator. Uses 'yield' to produce a sequence of values lazily.

```
# A simple generator function
def count_up_to(n):
    i = 0
    while i < n:
        yield i
        i += 1

# Using the generator
counter = count_up_to(5)
# <generator object count_up_to at ...>
next(counter)  # 0
next(counter)  # 1
list(counter)  # [2, 3, 4] (consumes the rest)

# 'yield from' to delegate to another generator
def chain_generators(gen1, gen2):
    yield from gen1
    yield from gen2

# Infinite sequence generator
```

```python
def fib_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Decorators

Syntax: `@decorator`
A function that takes another function, adds functionality, and returns it.

```python
# A simple decorator for timing a function
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end-start:.2f}s"
    )
        return result
    return wrapper

@timer
def do_something(n):
    time.sleep(n)

do_something(1)
# prints: do_something took 1.00s

# functools.wraps preserves original function metadata
from functools import wraps
def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # ...
        return func(*args, **kwargs)
    return wrapper
```

Context Managers (‘with‘)

Syntax: `with expression as var:` *Ensures resources are properly managed (e.g., files closed, locks released).*

```python
# File handling (guarantees file is closed)
with open('file.txt', 'r') as f:
    content = f.read()
# f is automatically closed here

# Threading locks
import threading
lock = threading.Lock()
with lock:
    # critical section, lock is acquired/released
    pass

# Creating a custom context manager
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwargs):
    # Code to acquire resource (setup)
    resource = (args, kwargs) # dummy resource
    try:
        yield resource
    finally:
        # Code to release resource (teardown)
        print("Resource released")

with managed_resource(1) as res:
    print(f"Used resource: {res}")
```

Dunder Methods (Magic Methods)

Special methods with double underscores. Allow custom classes to integrate with Python's syntax.

```python
class MyObject:
    def __init__(self, value):
        self.value = value

    # String representation for users
    def __str__(self):
        return f"MyObject(value={self.value})"

    # Unambiguous representation for developers
    def __repr__(self):
        return f"MyObject({self.value!r})"

    # Equality check
    def __eq__(self, other):
        return self.value == other.value

    # For sorting / comparison (<)
    def __lt__(self, other):
        return self.value < other.value

    # Make object callable like a function
    def __call__(self, x):
        return self.value + x

    # Get length
    def __len__(self):
        return len(str(self.value))

# Usage examples:
obj1 = MyObject(10)
obj2 = MyObject(20)
print(obj1)       # MyObject(value=10) (calls __str__)
print(obj1 < obj2)    # True (calls __lt__)
print(obj1(5))    # 15 (calls __call__)
```

Advanced OOP

Core patterns for building robust and efficient classes.

```python
# super() for parent class initialization
class Base:
    def __init__(self):
        self.name = "Base"

class Derived(Base):
    def __init__(self):
        super().__init__() # Correctly calls Base.
    __init__
        self.extra = "Derived"

# classmethod, staticmethod
class MyClass:
    @classmethod
    def from_string(cls, s):
        # Factory method, receives the class ‘cls‘
        # Used for alternative constructors
        return cls()

    @staticmethod
    def utility_func(x):
        # Helper/utility, no access to ‘self‘ or ‘cls‘
        # Doesn't depend on instance or class state
        return x * 2

# __slots__ for memory optimization
class Point:
    __slots__ = ['x', 'y'] # Pre-declares attributes
    def __init__(self, x, y):
        self.x, self.y = x, y
# Instances have no __dict__, saving memory and
# providing faster attribute access.

# @property decorator (Pythonic getters/setters)
class Circle:
    def __init__(self, radius):
        self._radius = radius  # "Protected" attribute

    @property
    def radius(self):
        """Get the radius."""
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive"
    )
        self._radius = value

c = Circle(5)
print(c.radius)  # 5 (calls getter)
c.radius = 10    # calls setter
# c.radius = -1  # ValueError: Radius must be positive
```

Type Hinting (PEP 484)

Annotations for variables and functions. Not enforced at runtime, but used by static analysis tools (mypy).

```python
from typing import List, Dict, Tuple, Optional, Any

def process_data(name: str, data: List[int]) -> float:
    if not data: return 0.0
    return sum(data) / len(data)

# Variable annotation
pi: float = 3.14159

# Complex types
user: Dict[str, Any] = {'name': 'Alice', 'id': 1}
coords: Tuple[int, int] = (10, 20)

# Optional type (can be None)
def greet(name: Optional[str] = None) -> str:
    if name is None:
        return "Hello, stranger."
    return f"Hello, {name}."
```

Exception Handling

Robustly handle errors and control program flow.

```
# Try-except-else-finally pattern
try:
    # Code that might raise an exception
    result = 10 / int('5')
except ValueError:
    print("Invalid number provided!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    # Runs only if no exception was raised in try
     block
    print(f"Result was {result}")
finally:
    # Always runs, regardless of exception
    print("Execution finished.")

# Assert for preconditions
assert len(nums) > 0, "List cannot be empty"
# Raises AssertionError if condition is False
```

Dataclasses & Match-Case

Modern features for concise data structures and pattern matching.

```
# Dataclasses (Python 3.7+)
from dataclasses import dataclass

@dataclass(frozen=True) # frozen=True makes it
    immutable
class Point:
    x: int
    y: int

p = Point(1, 2)
# p.x = 3 # ERROR if frozen=True
print(p) # Point(x=1, y=2) -- auto __repr__
print(p == Point(1, 2)) # True -- auto __eq__

# Match-case (Python 3.10+)
def handle_command(command):
    match command.split():
        case ["move", direction]:
            print(f"Moving {direction}")
        case ["look", obj, *details]:
            print(f"Looking at {obj} ({', '.join(
    details)})")
        case ["quit" | "exit"]:
            print("Goodbye!")
        case _: # Default case
            print("Unknown command")
```

Common Pitfalls & Scope

Key concepts for avoiding common bugs and managing variable scope.

```
# Mutable default arguments (BUG!)
def bad_append(n, lst=[]): # lst is created ONCE
    lst.append(n)
    return lst
bad_append(1) # [1]
bad_append(2) # [1, 2] -- DANGEROUS!

# Correct way:
def good_append(n, lst=None):
    if lst is None:
        lst = []
    lst.append(n)
    return lst

# Scope: nonlocal and global
g_var = 100 # global scope
def outer():
    o_var = 10 # enclosing scope
    def inner():
        nonlocal o_var # modify parent function's var
        o_var += 1
        global g_var # modify global var
        g_var += 1
    inner()
    print(o_var, g_var) # 11, 101

# Closure example
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter() # c is a closure
c() # 1
c() # 2
```

Important Notes

- **Mutability:** Lists are mutable (can change), tuples/strings are immutable.
- **Hashability:** Immutable types (int, str, tuple) can be dict keys/set elements. Lists cannot.
- **Iteration:** Most functions return iterators (lazy). Use list() to convert.
- **Short-circuit:** any() and all() stop early when result is determined.
- **Chaining:** Can chain methods: `result = func1(func2(func3(x)))`
- **Readability:** Comprehensions usually more readable than map/filter. Use what's clearest.
- **Performance:** Generators save memory for large data. List comprehensions faster for small data.
- **PEP 8:** Keep lines under 79-88 chars. Break long comprehensions into multiple lines.
- **Imports:** Avoid star imports (`from module import *`). Prefer specific imports (`from module import name`).