DSA Revision

Python

Adarsh Roy

Contents

1. Syntax

1.1. Better I/O

```
import sys
input = sys.stdin.readline            # faster input -> function
print(*args)                           # print space-separated -> None
numbers = list(map(int, input().split())) # read ints from line -> list[int]
words = list(map(str, input().split()))   # read strs from line -> list[str]
```

1.2. Bits

```
x << n        # shift left by n bits -> int (x*2**n)
x >> n        # shift right by n bits -> int (x//2**n)
x & y         # bitwise AND -> int
x | y         # bitwise OR -> int
x ^ y         # bitwise XOR -> int
~x            # bitwise NOT -> int (-(x+1))
x & 1         # check LSB -> int (0 or 1)
x | 1         # set LSB to 1 -> int
x & ~1        # set LSB to 0 -> int
x ^ 1         # toggle LSB -> int
x.bit_length()# number of bits to represent x -> int
x.bit_count() # number of set bits (popcount) -> int
bin(x)        # binary string -> str
int(s, 2)     # parse binary string s -> int
```

1.3. List

```
lst = []                              # empty list -> list
lst = [1, 2, 3]                       # with elements -> list
lst = list(iterable)                  # from iterable -> new list
lst.append(x)                         # add to end -> None
lst.insert(i, x)                      # insert at index i -> None
lst.extend(iterable)                  # append all from iterable -> None
elem = lst.pop()                      # remove last -> element
elem = lst.pop(i)                     # remove at index i -> element
lst.remove(x)                         # remove first occurrence of x -> None (ValueError
if missing)
lst.clear()                           # remove all elements -> None
elem = lst[i]                         # access by index -> element
sub = lst[start:end:step]             # slicing -> new list
ok = (x in lst)                       # membership -> bool
idx = lst.index(x)                    # first index of x -> int (ValueError if missing)
cnt = lst.count(x)                    # occurrences of x -> int
lst.sort()                            # sort asc -> None
lst.sort(reverse=True)                # sort desc -> None
new = sorted(lst)                     # sorted copy -> new list
lst.reverse()                         # reverse in place -> None
it = reversed(lst)                    # reversed iterator -> iterator
n = len(lst)                          # length -> int
lo, hi = min(lst), max(lst)           # min/max -> element
total = sum(lst)                      # sum -> number
```

1.4. Set

```
s = set()                            # empty set -> set
s = {1, 2, 3}                         # literal -> set
s = set(iterable)                     # from iterable -> new set
```

```
s.add(x)                              # add element -> None
s.update(iterable)                    # add multiple elements -> None
s.remove(x)                           # remove element -> None (KeyError if missing)
s.discard(x)                          # remove if present -> None
x = s.pop()                           # remove & return arbitrary element -> element
(KeyError if empty)
s.clear()                             # remove all -> None
u = s.union(t)                        # union -> new set
u = s | t                             # union -> new set
i = s.intersection(t)                  # intersection -> new set
i = s & t                             # intersection -> new set
d = s.difference(t)                   # difference -> new set
d = s - t                             # difference -> new set
sd = s.symmetric_difference(t)        # symmetric difference -> new set
sd = s ^ t                            # symmetric difference -> new set
n = len(s)                            # size -> int
ok = (x in s)                         # membership -> bool
```

1.5. Tuple

```
t = ()                                # empty tuple -> tuple
t = (1, 2, 3)                         # with elements -> tuple
t = tuple(iterable)                   # from iterable -> new tuple
t = (1,)                              # single-element tuple -> tuple
elem = t[i]                           # access by index -> element
sub = t[start:end:step]               # slicing -> new tuple
ok = (x in t)                         # membership -> bool
idx = t.index(x)                      # first index of x -> int (ValueError if missing)
cnt = t.count(x)                      # occurrences of x -> int
n = len(t)                            # length -> int
lo, hi = min(t), max(t)                # min/max -> element
total = sum(t)                        # sum -> number
```

1.6. String

```
s = "abc"                             # literal -> str
s2 = str(obj)                         # convert to string -> str
ch = s[i]                             # char at index -> str (len=1)
sub = s[start:end:step]                # slicing -> new str
s3 = s + t                            # concat -> new str
s3 = s * n                            # repeat -> new str
n = len(s)                            # length -> int
lo, hi = min(s), max(s)                # min/max char -> str
cnt = s.count(sub)                    # count substring -> int
idx = s.index(sub)                    # index of sub -> int (ValueError if missing)
pos = s.find(sub)                     # find substring -> int or -1
pos = s.rfind(sub)                    # find from right -> int or -1
ok = s.startswith(prefix)             # startswith -> bool
ok = s.endswith(suffix)               # endswith -> bool
ok = s.isalpha()                      # all letters -> bool
ok = s.isdigit()                      # all digits -> bool
ok = s.isalnum()                      # letters or digits -> bool
ok = s.isspace()                      # all whitespace -> bool
loUp = (s.islower(), s.isupper())     # case checks -> (bool, bool)
s2 = s.lower()                        # to lowercase -> new str
s2 = s.upper()                        # to uppercase -> new str
s2 = s.title()                        # title case -> new str
s2 = s.swapcase()                     # swap case -> new str
s2 = s.strip()                        # trim both ends -> new str
```

```
s2 = s.lstrip()                         # trim left -> new str
s2 = s.rstrip()                         # trim right -> new str
s2 = s.replace(a, b)                    # replace a with b -> new str
arr = s.split()                         # split whitespace -> list[str]
arr = s.split(sep)                      # split on sep -> list[str]
arr = s.rsplit(sep)                     # split on sep from right -> list[str]
s2 = sep.join(iterable)                 # join with sep -> str
trp = s.partition(sep)                  # split once -> (head, sep, tail)
trp = s.rpartition(sep)                  # split once from right -> (head, sep, tail)
```

1.7. Dict

```
d = {}                                  # empty dict -> dict
d = {"a": 1, "b": 2}                    # literal -> dict
d = dict(key=value, x=1)                # from kwargs -> new dict
d = dict(iterable)                      # from (key, value) pairs -> new dict
d[key] = value                          # set/update key -> None
d.update(other)                         # merge from other -> None
val = d.pop(key)                        # remove key -> value (KeyError if missing)
val = d.pop(key, default)               # remove or default -> value (default if missing)
kv = d.popitem()                        # remove last inserted -> (key, value)
del d[key]                              # delete key -> None (KeyError if missing)
d.clear()                               # remove all -> None
val = d[key]                            # access -> value (KeyError if missing)
val = d.get(key)                        # access with default None -> value or None
val = d.get(key, default)               # access with custom default -> value or default
ks = d.keys()                           # keys view -> dict_keys
vs = d.values()                         # values view -> dict_values
it = d.items()                          # items view -> dict_items
n = len(d)                              # number of keys -> int
ok = (key in d)                         # key existence -> bool
```

1.8. Deque

```
from collections import deque
dq = deque()                            # empty deque -> deque
dq = deque(iterable)                    # from iterable -> deque
dq.append(x)                            # push right -> None
dq.appendleft(x)                        # push left -> None
dq.extend(iterable)                     # extend right -> None
dq.extendleft(iterable)                 # extend left (reversed order) -> None
x = dq.pop()                            # pop right -> element
x = dq.popleft()                        # pop left -> element
dq.clear()                              # remove all -> None
dq.rotate(k)                            # rotate right by k -> None
```

1.9. Heapq

Min Heap by default. Use negative insertions for Max Heap.

```
import heapq
heap = []                               # heap storage -> list
heapq.heappush(heap, x)                  # push x -> None
x = heapq.heappop(heap)                  # pop smallest -> element
heapq.heapify(heap)                      # list to heap in place -> None
x = heapq.heappushpop(heap, x)           # push then pop smallest -> element
x = heapq.heapreplace(heap, x)           # pop smallest then push x -> element
top = heapq.nlargest(k, iterable)        # k largest -> list
bot = heapq.nsmallest(k, iterable)       # k smallest -> list
```

1.10. Math

```
import heapq
heap = []                              # heap storage -> list
heapq.heappush(heap, x)                 # push x -> None
x = heapq.heappop(heap)                 # pop smallest -> element
heapq.heapify(heap)                     # list to heap in place -> None
x = heapq.heappushpop(heap, x)          # push then pop smallest -> element
x = heapq.heapreplace(heap, x)          # pop smallest then push x -> element
top = heapq.nlargest(k, iterable)       # k largest -> list
bot = heapq.nsmallest(k, iterable)      # k smallest -> list
```

1.11. Itertools

```
import itertools as it
it.accumulate(iterable)                  # partial sums/products -> iterator
it.chain(a, b)                           # concatenate iterables -> iterator
it.chain.from_iterable(iterable)         # flatten one level -> iterator
it.combinations(iterable, r)             # r-length combos -> iterator[tuple]
it.combinations_with_replacement(iterable, r) # combos w/ repeat -> iterator[tuple]
it.permutations(iterable, r=None)        # r-length perms -> iterator[tuple]
it.product(a, b, repeat=1)               # cartesian product -> iterator[tuple]
it.groupby(iterable, key=None)           # consecutive groups -> iterator[(key, group)]
it.islice(iterable, start, stop, step=1) # slice of iterable -> iterator
```

1.12. Bisect

```
import bisect
i = bisect.bisect_left(a, x)             # leftmost insertion index -> int
i = bisect.bisect_right(a, x)            # rightmost insertion index -> int
bisect.insort_left(a, x)                 # insert left keeping sort -> None
bisect.insort_right(a, x)                # insert right keeping sort -> None
```

1.13. Collections Utilities

```
from collections import Counter, defaultdict
c = Counter(iterable)                    # counts per element -> Counter
c.update(iterable_or_mapping)            # add counts -> None
mc = c.most_common(n=None)                # top n or all -> list[(elem, count)]
dd = defaultdict(list)                   # dict with default factory -> defaultdict
val = dd[key]                             # get or create default -> value
```