Heapq

```
import heapq

# MinHeap Operations - All O(log n) except heapify
nums = [3,1,4,1,5]
heapq.heapify(nums)          # Convert to heap in-place: O(n)
heapq.heappush(nums, 2)      # Add element: O(log n)
smallest = heapq.heappop(nums)  # Remove smallest: O(log n)

# MaxHeap Trick: Multiply by -1
nums = [-x for x in nums]    # Convert to maxheap: O(n)
heapq.heapify(nums)          # O(n)
largest = -heapq.heappop(nums)  # Get largest: O(log n)

# Advanced Operations
k_largest = heapq.nlargest(k, nums)    # O(n * log k)
k_smallest = heapq.nsmallest(k, nums)  # O(n * log k)

# Custom Priority Queue
heap = []
heapq.heappush(heap, (priority, item))  # Sort by priority
```

Counter

```
from collections import Counter

# Initialize
c = Counter(['a','a','b'])    # From iterable
c = Counter("hello")          # From string

# Operations
c.most_common(2)      # Top 2 frequent elements
c['a'] += 1           # Increment count
c.update("more")      # Add counts from iterable
c.total()             # Sum of all counts
```

Deque

```
from collections import deque

# Perfect for BFS - O(1) operations on both ends
d = deque()
d.append(1)          # Add right
d.appendleft(2)      # Add left
d.pop()              # Remove right
d.popleft()          # Remove left
d.extend([1,2,3])    # Extend right
d.extendleft([1,2,3])# Extend left
d.rotate(n)          # Rotate n steps right (negative for left)
```

Testing

```
# Use assertions for edge cases
assert binary_search([], 1) == -1, "Empty array should return -1"
assert binary_search([1], 1) == 0, "Single element array should work"
```

APIs

GET

```
import requests
response = requests.get(URL, timeout=10) # good practice
if response.status_code == 200:
    users = response.json()
    for user in users:
        print(user)
else:
    print("Failed to retrieve users:", response.status_code)

# query parameters (arguments for an api call)
params = {
    'userId': 1
}
response = requests.get(URL, params=params)

# headers
headers = {
    'Authorization': 'Bearer YOUR_ACCESS_TOKEN'
}
response = requests.get(URL, headers=headers)
```

POST

```
new_user = {
    "name": "John Doe",
    "address": {
        "street": "123 Main St",
        "city": "Anytown",
    }
}
response = requests.post(URL, json=new_user)
if response.status_code == 201:
    user = response.json()
    print("Created new user:", user)
```

DELETE

```
response = requests.delete(URL)
```

Sets

```
s = {1,2,3}

# Common Operations
s.add(4)              # Add element
s.remove(4)           # Remove (raises error if missing)
s.discard(4)          # Remove (no error if missing)
s.pop()               # Remove and return arbitrary element

# Set Operations
a.union(b)            # Elements in a OR b
a.intersection(b)     # Elements in a AND b
a.difference(b)       # Elements in a but NOT in b
a.symmetric_difference(b)  # Elements in a OR b but NOT both
a.issubset(b)         # True if all elements of a are in b
a.issuperset(b)       # True if all elements of b are in a
```

Strings

```
s = "hello world"

# Essential Methods
s.split()             # Split on whitespace
s.split(',')          # Split on comma
s.strip()             # Remove leading/trailing whitespace
s.lower()             # Convert to lowercase
s.upper()             # Convert to uppercase
s.isalnum()           # Check if alphanumeric
s.isalpha()           # Check if alphabetic
s.isdigit()           # Check if all digits
s.find('sub')         # Index of substring (-1 if not found)
s.count('sub')        # Count occurrences
s.replace('old', 'new')  # Replace all occurrences

# ASCII Conversion
ord('a')              # Char to ASCII (97)
chr(97)               # ASCII to char ('a')

# Join Lists
''.join(['a','b'])    # Concatenate list elements
```

Pad a String or Number

```
>>> a = "hello"
>>> a = a.zfill(10) # pad to left side - same as rjust(10, '0')
"00000hello"

>>> padded = s.ljust(10, '0') # pad to right side
"hello00000"
```

Zip

```
>>> numbers = [1, 2, 3]
>>> letters = ['a', 'b', 'c']

# match 2 lists by index
>>> zipped = zip(numbers, letters)
>>> list(zipped)
[(1, 'a'), (2, 'b'), (3, 'c')]

# to take longest iterable
>>> from itertools import zip_longest
>>> longest = range(5)  # [0, 1, 2, 3, 4]
>>> zipped = zip_longest(numbers, letters, longest, fillvalue='?')
>>> list(zipped)
[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), ('?', '?', 3), ('?', '?', 4)]

# verify that both arguments are of equal length
>>> list(zip(range(5), range(100), strict=True))  # strict=False is default
```

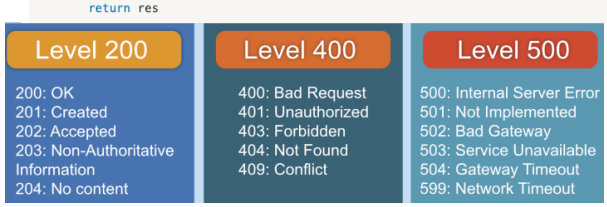
Transpose Matrix

```
# zip() function returns a zip object, which is an iterator of tuples
# *matrix unpacks the matrix into rows
>>> matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> *matrix
[1, 2, 3], [4, 5, 6], [7, 8, 9]
>>> transposed_matrix = zip(*matrix)
>>> list(transposed_matrix)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

# mathematically, transpose is the same as primary diagonal (top-left to bottom-right)
def transpose(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    res = [[0] * rows for _ in range(cols)]

    for i in range(rows):
        for j in range(cols):
            res[j][i] = matrix[i][j]

    return res
```



Built-in Functions

```
# Iteration Helpers
enumerate(lst)        # Index + value pairs
zip(lst1, lst2)       # Parallel iteration
map(fn, lst)          # Apply function to all elements
filter(fn, lst)       # Keep elements where fn returns True
any(lst)              # True if any element is True
all(lst)              # True if all elements are True

# Binary Search (import bisect)
bisect.bisect(lst, x)     # Find insertion point
bisect.bisect_left(lst, x)# Find leftmost insertion point
bisect.insort(lst, x)     # Insert maintaining sort

# Type Conversion
int('42')             # String to int
str(42)               # Int to string
list('abc')           # String to list
''.join(['a','b'])    # List to string
set([1,2,2])          # List to set

# Math
abs(-5)               # Absolute value
pow(2, 3)             # Power
round(3.14159, 2)     # Round to decimals
```

Important Python Integer Operations

```
# Binary representation
bin(10)               # '0b1010'
format(10, 'b')       # '1010' (without prefix)

# Division and Modulo
divmod(10, 3)         # (3, 1) - returns (quotient, remainder)

# Negative number handling
x = -3    x/y = -1.5
y = 2
print(x // y)         # -2 (floor division)
print(int(x/y))       # -1 (preferred for negative numbers)
print(x % y)          # 1 (Python's modulo with negative numbers)
```

Math Module Essentials

```
import math

# Constants
math.pi       # 3.141592653589793
math.e        # 2.718281828459045

# Common Functions
math.ceil(2.3)        # 3 - Smallest integer greater than x
math.floor(2.3)       # 2 - Largest integer less than x
math.gcd(a, b)        # Greatest common divisor
math.log(x, base)     # Logarithm with specified base
```

Tips & Gotchas

3. Heap Priority:

```
# For custom priority in heapq, use tuples
heap = []
heapq.heappush(heap, (priority, item))
```

4. List Comprehension:

```
# Often clearer than map/filter
squares = [x*x for x in range(10) if x % 2 == 0]
```

5. String Building:

```
# Use join() instead of += for strings
chars = ['a', 'b', 'c']
word = ''.join(chars)  # More efficient
```

7. Custom Sort Keys:

```
# Sort by length then alphabetically
words.sort(key=lambda x: (len(x), x))
```

Python Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+x, -x, ~x	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR