Python Interview Preparation Guide

The Complete Reference for Technical Interviews

This comprehensive guide contains all essential Python functions, methods, and modules needed for technical interviews. Each section includes:
• Detailed explanations and definitions
• Practical examples with expected outputs
• Common use cases and interview patterns
• Time and space complexity considerations

Master these concepts to excel in your Python interviews!

Table of Contents

Built-in Functions

Built-in Functions Overview:
Python's built-in functions are the foundation of the language. These functions are available without importing any modules and provide essential functionality for data conversion, mathematical operations, and working with iterables. Understanding these functions is crucial for technical interviews as they appear in most coding problems.

Type Conversion

```
int('5')            # → 5
float('3.14')       # → 3.14
str(100)            # → '100'
bool(0)             # → False
list('abc')         # → ['a', 'b', 'c']
tuple([1,2])        # → (1, 2)
set([1,2,2])        # → {1, 2}
dict([('a', 1)])    # → {'a': 1}
```

Math Functions

```
abs(-5)             # → 5
round(3.6)          # → 4
pow(2,3)            # → 8
divmod(5,2)         # → (2,1)
max(1, 2, 3)        # → 3
min(1, 2, 3)        # → 1
sum([1,2,3])        # → 6
```

Iterable Functions

```
len('abc')                     # → 3
enumerate(['a','b'])           # → [(0,'a'),(1,'b')]
```

```
zip([1,2],['a','b'])          # → [(1,'a'),(2,'b')]
reversed([1,2])               # → [2,1]
sorted([3,1,2])               # → [1,2,3]
range(3)                      # → [0,1,2]
```

Input/Output

```
input('Enter: ')              # takes user input
print('Hello')                # prints Hello
```

Type Checking

```
type(5)                       # → <class 'int'>
isinstance(5, int)            # → True
```

Functional Programming

```
map(str, [1,2])               # → ['1','2']
filter(lambda x:x>1, [1,2]) # → [2]
any([0,1,0])                  # → True
all([1,2,3])                  # → True
```

Others

```
help(str)                     # displays help
id(5)                         # memory address
dir(str)                      # lists attributes
eval('2+3')                   # → 5
exec('a=5; print(a)')         # executes code
```

String Methods

String Methods Overview:
String methods are essential for text processing and manipulation. Since strings are immutable in Python, these methods return new strings rather than modifying the original. They're frequently used in parsing, validation, and data cleaning problems in interviews.

```
s = "Hello World"

# Case conversion
s.lower()                  # → 'hello world'
s.upper()                  # → 'HELLO WORLD'
s.capitalize()             # → 'Hello world'
s.title()                  # → 'Hello World'
s.swapcase()               # → 'hELLO wORLD'

# Whitespace handling
s.strip()                  # removes whitespace from both ends
s.lstrip()                 # removes left whitespace
s.rstrip()                 # removes right whitespace

# Search and replace
s.replace("H", "J")        # → 'Jello World'
s.find("o")                # → 4 (first occurrence)
s.rfind("o")               # → 7 (last occurrence)
s.index("o")               # → 4 (raises error if not found)
s.count("l")               # → 3

# Boolean checks
s.startswith("Hel")        # → True
s.endswith("ld")           # → True
s.isalpha()                # → False (has space)
s.isdigit()                # → False
s.isalnum()                # → False
s.islower()                # → False
s.isupper()                # → False
s.isspace()                # → False

# Split and join
s.split(" ")               # → ['Hello', 'World']
s.split()                  # → ['Hello', 'World'] (default whitespace)
" ".join(['Hello', 'World']) # → 'Hello World'
s.partition(" ")           # → ('Hello', ' ', 'World')
```

List Methods

```
lst = [1, 2, 3]

# Adding elements
lst.append(4)                # → [1, 2, 3, 4]
lst.insert(1, 5)             # → [1, 5, 2, 3, 4]
lst.extend([6, 7])           # → [1, 5, 2, 3, 4, 6, 7]

# Removing elements
lst.pop()                    # → removes and returns last element
lst.pop(1)                   # → removes and returns element at index 1
lst.remove(2)                # → removes first occurrence of 2
lst.clear()                  # → empties the list

# Sorting and reversing
lst.sort()                   # → sorts in place
lst.sort(reverse=True)       # → sorts in descending order
sorted(lst)                  # → returns new sorted list
lst.reverse()                # → reverses in place
list(reversed(lst))          # → returns new reversed list

# Searching
lst.index(3)                 # → returns index of first occurrence
lst.count(1)                 # → returns count of occurrences

# Copying
lst.copy()                   # → shallow copy
```

Tuple Methods

```
t = (1, 2, 2, 3)

t.count(2)                  # → 2
t.index(3)                  # → 3
```

Set Methods

```
a = {1, 2, 3}
b = {3, 4, 5}

# Adding elements
a.add(4)                    # → {1, 2, 3, 4}
a.update([5, 6])            # → {1, 2, 3, 4, 5, 6}

# Removing elements
a.remove(2)                 # → removes 2 (raises error if not found)
a.discard(10)               # → removes 10 (no error if not found)
a.pop()                     # → removes and returns arbitrary element
a.clear()                   # → empties the set

# Set operations
a.union(b)                  # → {1, 2, 3, 4, 5}
a | b                       # → {1, 2, 3, 4, 5}
a.intersection(b)           # → {3}
a & b                       # → {3}
a.difference(b)             # → {1, 2}
a - b                       # → {1, 2}
a.symmetric_difference(b)   # → {1, 2, 4, 5}
a ^ b                       # → {1, 2, 4, 5}

# Boolean operations
a.issubset(b)               # → False
a.issuperset(b)             # → False
a.isdisjoint(b)             # → False
```

Dictionary Methods

```
d = {"a": 1, "b": 2}

# Accessing
d.keys()                     # → dict_keys(['a', 'b'])
d.values()                   # → dict_values([1, 2])
d.items()                    # → dict_items([('a', 1), ('b', 2)])
d.get("a")                   # → 1
d.get("c", 0)                # → 0 (default value)

# Modifying
d.update({"c": 3})           # → {"a": 1, "b": 2, "c": 3}
d.setdefault("d", 0)         # → sets "d": 0 if not present
d.pop("a")                   # → removes and returns value of "a"
d.popitem()                  # → removes and returns arbitrary (key, value)
d.clear()                    # → empties dictionary

# Copying
d.copy()                     # → shallow copy
```

Math Module

```
import math

# Basic functions
math.sqrt(16)                # → 4.0
math.pow(2, 3)               # → 8.0
math.abs(-5)                 # → 5 (same as built-in abs)

# Rounding
math.ceil(4.2)               # → 5
math.floor(4.9)              # → 4
math.trunc(4.9)              # → 4

# Logarithms
math.log(10)                 # → 2.302... (natural log)
math.log10(100)              # → 2.0 (base 10)
math.log2(8)                 # → 3.0 (base 2)

# Trigonometry
math.sin(math.pi/2)          # → 1.0
math.cos(0)                  # → 1.0
math.tan(math.pi/4)          # → 1.0
math.degrees(math.pi)        # → 180.0
math.radians(180)            # → 3.14159...

# Constants
math.pi                      # → 3.14159...
math.e                       # → 2.71828...

# Other functions
math.factorial(5)            # → 120
math.gcd(12, 18)             # → 6
math.isnan(float('nan'))     # → True
math.isinf(float('inf'))     # → True
```

Random Module

```
import random

# Random numbers
random.random()             # → float between 0.0 and 1.0
random.uniform(1, 10)       # → float between 1 and 10
random.randint(1, 10)       # → integer between 1 and 10 (inclusive)
random.randrange(1, 10)     # → integer between 1 and 9

# Random choice
random.choice([1, 2, 3])     # → randomly picks one element
random.choices([1, 2, 3], k=2) # → picks 2 elements with replacement
random.sample([1, 2, 3], 2)  # → picks 2 elements without replacement

# Shuffling
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)     # → shuffles list in place

# Seed for reproducibility
random.seed(42)             # → sets random seed
```

Collections Module

Collections Module Overview:
The collections module provides specialized data structures that are more efficient than built-in types for specific use cases. Counter, defaultdict, and deque are particularly important for interview problems involving frequency counting, graph algorithms, and sliding window techniques.

```
from collections import Counter, defaultdict, OrderedDict, deque, namedtuple, Ch
```

Definition: A dict subclass for counting hashable objects. **Example:** from collections import Counter c = Counter('hello world') c.most_common(3) # [('l', 3), ('o', 2), ('h', 1)] c['l'] # 3 **Use Case:** Counting frequencies, finding most common elements.

Counter

```
from collections import Counter

# Create Counter
c = Counter('banana')         # → Counter({'a': 3, 'n': 2, 'b': 1})
c = Counter([1, 2, 2, 3, 3, 3]) # → Counter({3: 3, 2: 2, 1: 1})

# Most common elements
c.most_common()               # → [('a', 3), ('n', 2), ('b', 1)]
c.most_common(1)              # → [('a', 3)]
c.most_common(1)[0][0]        # → 'a' (most frequent element)

# Operations
c['x']                        # → 0 (missing elements have count 0)
c.update('apple')             # → adds counts
c.subtract('apple')           # → subtracts counts
c.total()                     # → sum of all counts
```

Definition: A dict subclass for counting hashable objects. **Example:** from collections import Counter c = Counter('hello world') c.most_common(3) # [('l', 3), ('o', 2), ('h', 1)] c['l'] # 3 **Use Case:** Counting frequencies, finding most common elements.

defaultdict

```
from collections import defaultdict

# Create defaultdict
dd = defaultdict(int)        # → default value is 0
dd = defaultdict(list)       # → default value is []
dd = defaultdict(set)        # → default value is set()

# Usage
dd['key1'] += 1              # → automatically creates key with default value
dd['key2'].append(1)         # → automatically creates key with empty list
```

Definition: A dict subclass that calls a factory function to supply missing values. **Example:** from collections import defaultdict dd = defaultdict(list) dd['key1'].append('value1') # No KeyError dd = defaultdict(int) dd['counter'] += 1 # No KeyError **Use Case:** Avoiding KeyError, grouping data.

OrderedDict

```
from collections import OrderedDict

# Create OrderedDict (maintains insertion order)
od = OrderedDict([('a', 1), ('b', 2)])

# Move to end
od.move_to_end('a')          # → moves 'a' to end
od.move_to_end('b', last=False) # → moves 'b' to beginning

# Pop from end
od.popitem()                 # → removes and returns last item
od.popitem(last=False)       # → removes and returns first item
```

deque (Double-ended queue)

```
from collections import deque
```

```
# Create deque
dq = deque([1, 2, 3])          # → deque([1, 2, 3])
dq = deque([1, 2, 3], maxlen=3 # → fixed size deque

# Add elements
dq.append(4)                   # → add to right
dq.appendleft(0)               # → add to left
dq.extend([5, 6])              # → extend right
dq.extendleft([7, 8])          # → extend left

# Remove elements
dq.pop()                       # → remove from right
dq.popleft()                   # → remove from left

# Rotate
dq.rotate(1)                   # → rotate right by 1
dq.rotate(-1)                  # → rotate left by 1
```

Definition: A double-ended queue supporting efficient operations at both ends. **Example:** from collections import deque dq = deque([1, 2, 3]) dq.appendleft(0) # deque([0, 1, 2, 3]) dq.append(4) # deque([0, 1, 2, 3, 4]) dq.popleft() # 0 **Use Case:** Sliding window problems, BFS traversal.

namedtuple

```
from collections import namedtuple

# Create namedtuple class
Point = namedtuple('Point', ['x', 'y'])
Point = namedtuple('Point', 'x y')  # alternative syntax

# Create instance
p = Point(1, 2)                # → Point(x=1, y=2)
p.x                            # → 1
p.y                            # → 2

# Convert to dict
p._asdict()                    # → {'x': 1, 'y': 2}

# Replace values
p._replace(x=3)                # → Point(x=3, y=2)
```

ChainMap

```
from collections import ChainMap

# Create ChainMap
cm = ChainMap({'a': 1}, {'b': 2}, {'c': 3})

# Access
cm['a']                        # → 1
cm.get('d', 'default')         # → 'default'

# Add new mapping
cm = cm.new_child({'d': 4})    # → adds new mapping at front
```

Itertools Module

Itertools Module Overview:
Itertools provides memory-efficient tools for creating iterators. The permutations, combinations, and product functions are especially useful for combinatorial problems commonly seen in interviews. These functions can solve complex problems with minimal code.

```
import itertools
```

Infinite Iterators

```
# count(start, step)
for i in itertools.count(10, 2):  # → 10, 12, 14, 16, ...
    if i > 20: break

# cycle(iterable)
for i in itertools.cycle([1, 2, 3]):  # → 1, 2, 3, 1, 2, 3, ...
    if i > 100: break

# repeat(object, times)
list(itertools.repeat(5, 3))  # → [5, 5, 5]
```

Iterators on Shortest Input

```
# accumulate(iterable, func)
list(itertools.accumulate([1, 2, 3, 4]))  # → [1, 3, 6, 10]
list(itertools.accumulate([1, 2, 3, 4], operator.mul))  # → [1, 2, 6, 24]

# chain(*iterables)
list(itertools.chain([1, 2], [3, 4]))  # → [1, 2, 3, 4]

# compress(data, selectors)
list(itertools.compress('ABCDEF', [1, 0, 1, 0, 1, 1]))  # → ['A', 'C', 'E', 'F'

# dropwhile(predicate, iterable)
```

```
list(itertools.dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]))  # → [6, 4, 1]

# takewhile(predicate, iterable)
list(itertools.takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]))  # → [1, 4]

# filterfalse(predicate, iterable)
list(itertools.filterfalse(lambda x: x % 2, range(10)))  # → [0, 2, 4, 6, 8]

# groupby(iterable, key)
groups = itertools.groupby('AAABBBCCDAABBB')
[(k, list(g)) for k, g in groups]  # → [('A', ['A', 'A', 'A']), ('B', ['B', 'B'

# islice(iterable, start, stop, step)
list(itertools.islice('ABCDEFG', 2, 5))  # → ['C', 'D', 'E']
```

Combinatorial Iterators

```
# product(*iterables, repeat=1)
list(itertools.product('AB', repeat=2))  # → [('A', 'A'), ('A', 'B'), ('B', 'A'
list(itertools.product('AB', '12'))      # → [('A', '1'), ('A', '2'), ('B', '1'

# permutations(iterable, r=None)
list(itertools.permutations('ABC'))      # → [('A', 'B', 'C'), ('A', 'C', 'B'),
list(itertools.permutations('ABC', 2))   # → [('A', 'B'), ('A', 'C'), ('B', 'A'

# combinations(iterable, r)
list(itertools.combinations('ABC', 2))   # → [('A', 'B'), ('A', 'C'), ('B', 'C'

# combinations_with_replacement(iterable, r)
list(itertools.combinations_with_replacement('ABC', 2))  # → [('A', 'A'), ('A',
```

Definition: Returns successive r-length permutations of elements. **Example:** import itertools list(itertools.permutations([1, 2, 3])) # [(1,2,3), (1,3,2), (2,1,3), ...] list(itertools.permutations('ABC', 2)) # [('A','B'), ('A','C'), ('B','A'), ...] **Use Case:** Generating all possible arrangements.

Functools Module

```
import functools
```

reduce

```
from functools import reduce

# reduce(function, iterable, initializer)
reduce(lambda x, y: x + y, [1, 2, 3, 4])  # → 10
reduce(lambda x, y: x * y, [1, 2, 3, 4])  # → 24
reduce(lambda x, y: x if x > y else y, [1, 5, 2, 9, 3])  # → 9
```

partial

```
from functools import partial

# partial(func, *args, **keywords)
def multiply(x, y):
    return x * y

double = partial(multiply, 2)
double(5)  # → 10

# With keyword arguments
def greet(greeting, name):
    return f"{greeting}, {name}!"

say_hello = partial(greet, "Hello")
say_hello("Alice")  # → "Hello, Alice!"
```

lru_cache

```
from functools import lru_cache

# lru_cache(maxsize=128, typed=False)
@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Check cache info
fibonacci.cache_info()  # → CacheInfo(hits=0, misses=0, maxsize=None, currsize=

# Clear cache
fibonacci.cache_clear()
```

wraps

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def say_hello():
    """Say hello function"""
    print("Hello!")
```

Datetime Module

```
from datetime import datetime, date, time, timedelta
```

datetime

```
# Current date and time
now = datetime.now()
utc_now = datetime.utcnow()

# Create specific datetime
dt = datetime(2023, 12, 25, 10, 30, 45)  # → 2023-12-25 10:30:45

# Parse from string
dt = datetime.strptime('2023-12-25 10:30:45', '%Y-%m-%d %H:%M:%S')

# Format to string
dt.strftime('%Y-%m-%d %H:%M:%S')  # → '2023-12-25 10:30:45'
dt.strftime('%B %d, %Y')          # → 'December 25, 2023'

# Access components
dt.year                           # → 2023
dt.month                          # → 12
dt.day                            # → 25
dt.hour                           # → 10
dt.minute                         # → 30
dt.second                         # → 45
```

date

```
# Current date
today = date.today()

# Create specific date
d = date(2023, 12, 25)

# Format
d.strftime('%Y-%m-%d')            # → '2023-12-25'
```

time

```
# Create time
t = time(10, 30, 45)                 # → 10:30:45

# Format
t.strftime('%H:%M:%S')               # → '10:30:45'
```

timedelta

```
# Create timedelta
td = timedelta(days=7, hours=3, minutes=30)

# Date arithmetic
tomorrow = date.today() + timedelta(days=1)
last_week = datetime.now() - timedelta(weeks=1)

# Get components
td.days                              # → 7
td.seconds                           # → 12600 (3 hours 30 minutes in seconds)
td.total_seconds()                   # → 617400.0
```

Essential Interview Functions

Most Frequent Element

```
from collections import Counter

def most_frequent(lst):
    return Counter(lst).most_common(1)[0][0]

# Example
nums = [1, 2, 2, 3, 3, 3, 4]
print(most_frequent(nums))  # → 3
```

Definition: A dict subclass for counting hashable objects. **Example:** from collections import Counter c = Counter('hello world') c.most_common(3) # [('l', 3), ('o', 2), ('h', 1)] c['l'] # 3 **Use Case:** Counting frequencies, finding most common elements.

Fibonacci Sequence

```
def fibonacci(n):
    """Generate first n Fibonacci numbers"""
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Example
list(fibonacci(10))  # → [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

# Recursive version with memoization
@lru_cache(maxsize=None)
def fib_recursive(n):
    if n < 2:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)
```

Prime Number Check

```python
def is_prime(n):
    """Check if number is prime"""
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True

# Generate primes up to n
def sieve_of_eratosthenes(n):
    """Generate all primes up to n"""
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False

    for i in range(2, int(n**0.5) + 1):
        if primes[i]:
            for j in range(i*i, n + 1, i):
                primes[j] = False

    return [i for i in range(2, n + 1) if primes[i]]
```

Palindrome Check

```python
def is_palindrome(s):
    """Check if string is palindrome"""
    return s == s[::-1]

def is_palindrome_ignore_case(s):
    """Check palindrome ignoring case and spaces"""
    s = ''.join(c.lower() for c in s if c.isalnum())
    return s == s[::-1]
```

Two Sum Problem

```
def two_sum(nums, target):
    """Find indices of two numbers that add to target"""
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []

# Example
nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # → [0, 1]
```

Anagram Check

```
def are_anagrams(s1, s2):
    """Check if two strings are anagrams"""
    return sorted(s1.lower()) == sorted(s2.lower())

# Using Counter
from collections import Counter
def are_anagrams_counter(s1, s2):
    return Counter(s1.lower()) == Counter(s2.lower())
```

Definition: A dict subclass for counting hashable objects. **Example:** from collections import Counter c = Counter('hello world') c.most_common(3) # [('l', 3), ('o', 2), ('h', 1)] c['l'] # 3 **Use Case:** Counting frequencies, finding most common elements.

Remove Duplicates

```
def remove_duplicates(lst):
    """Remove duplicates while preserving order"""
    seen = set()
```

```python
    result = []
    for item in lst:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result

# Using dict.fromkeys() (Python 3.7+)
def remove_duplicates_dict(lst):
    return list(dict.fromkeys(lst))
```

Factorial

```python
def factorial(n):
    """Calculate factorial recursively"""
    return 1 if n <= 1 else n * factorial(n - 1)

# Iterative version
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Using math module
import math
print(math.factorial(5))  # → 120
```

Binary Search

```python
def binary_search(arr, target):
    """Binary search in sorted array"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # Not found
```

Reverse String/List

```
    def reverse_string(s):
        """Reverse a string"""
        return s[::-1]

    def reverse_words(s):
        """Reverse words in a string"""
        return ' '.join(s.split()[::-1])

    def reverse_list(lst):
        """Reverse a list"""
        return lst[::-1]
```

Operator Module

```
import operator

# Arithmetic operators as functions
operator.add(2, 3)           # → 5
operator.sub(5, 3)           # → 2
operator.mul(2, 3)           # → 6
operator.truediv(6, 2)       # → 3.0
operator.floordiv(7, 2)      # → 3
operator.mod(7, 3)           # → 1
operator.pow(2, 3)           # → 8

# Comparison operators
operator.eq(2, 2)            # → True
operator.ne(2, 3)            # → True
operator.lt(2, 3)            # → True
operator.le(2, 2)            # → True
operator.gt(3, 2)            # → True
operator.ge(3, 2)            # → True

# Logical operators
operator.and_(True, False)   # → False
operator.or_(True, False)    # → True
operator.not_(True)          # → False

# Item and attribute access
operator.getitem([1, 2, 3], 1)     # → 2
operator.itemgetter(1)([1, 2, 3])  # → 2
operator.attrgetter('real')(3+4j)   # → 3.0

# Useful for sorting
students = [('Alice', 85), ('Bob', 90), ('Charlie', 78)]
sorted(students, key=operator.itemgetter(1))  # Sort by grade
```

Heapq Module (Priority Queue)

```
import heapq

# Min heap operations
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
heapq.heappush(heap, 2)
print(heap)                  # → [1, 2, 4, 3]

# Pop smallest element
smallest = heapq.heappop(heap)  # → 1
print(heap)                  # → [2, 3, 4]

# Peek at smallest without removing
smallest = heap[0]           # → 2

# Convert list to heap
nums = [3, 1, 4, 1, 5, 9, 2, 6]
heapq.heapify(nums)          # → modifies nums in place
print(nums)                  # → [1, 1, 2, 3, 5, 9, 4, 6]

# N largest/smallest elements
heapq.nlargest(3, nums)      # → [9, 6, 5]
heapq.nsmallest(3, nums)     # → [1, 1, 2]

# With custom key
students = [('Alice', 85), ('Bob', 90), ('Charlie', 78)]
heapq.nlargest(2, students, key=lambda x: x[1])  # → [('Bob', 90), ('Alice', 85

# Push and pop in one operation
heapq.heappushpop(heap, 0)   # → pushes 0 and pops smallest
heapq.heapreplace(heap, 10)  # → pops smallest and pushes 10
```

Definition: Provides heap queue algorithm (priority queue). **Example:** import heapq heap = [3, 1, 4, 1, 5] heapq.heapify(heap) # [1, 1, 4, 3, 5] heapq.heappush(heap, 2) # [1, 1, 2, 3, 5, 4] heapq.heappop(heap) # 1 **Use Case:** Finding k largest/smallest elements, Dijkstra's algorithm.

Bisect Module (Binary Search)

```
import bisect

# Binary search in sorted list
arr = [1, 3, 5, 7, 9]

# Find insertion point
bisect.bisect_left(arr, 5)   # → 2 (leftmost position)
bisect.bisect_right(arr, 5)  # → 3 (rightmost position)
bisect.bisect(arr, 5)        # → 3 (same as bisect_right)

# Insert element maintaining sorted order
bisect.insort(arr, 6)        # → [1, 3, 5, 6, 7, 9]
bisect.insort_left(arr, 4)   # → [1, 3, 4, 5, 6, 7, 9]

# Find range of equal elements
def find_range(arr, target):
    left = bisect.bisect_left(arr, target)
    right = bisect.bisect_right(arr, target)
    return (left, right - 1) if left < len(arr) and arr[left] == target else (-

# Binary search for existence
def binary_search_exists(arr, target):
    pos = bisect.bisect_left(arr, target)
    return pos < len(arr) and arr[pos] == target
```

Definition: Provides support for maintaining sorted lists. **Example:** import bisect arr = [1, 3, 5, 7, 9] bisect.bisect_left(arr, 5) # 2 bisect.insort(arr, 6) # [1, 3, 5, 6, 7, 9] **Use Case:** Binary search, maintaining sorted order.

Copy Module (Deep/Shallow Copy)

```
import copy

# Shallow copy
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
shallow[0][0] = 'X'
print(original)              # → [['X', 2], [3, 4]] (modified!)

# Deep copy
original = [[1, 2], [3, 4]]
deep = copy.deepcopy(original)
deep[0][0] = 'X'
print(original)              # → [[1, 2], [3, 4]] (unchanged)

# Copy with different methods
lst = [1, 2, 3]
lst_copy1 = lst.copy()       # shallow copy
lst_copy2 = lst[:]           # shallow copy
lst_copy3 = list(lst)        # shallow copy
```

String Module

```
import string

# Constants
string.ascii_lowercase       # → 'abcdefghijklmnopqrstuvwxyz'
string.ascii_uppercase       # → 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
string.ascii_letters         # → 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
string.digits                # → '0123456789'
string.hexdigits             # → '0123456789abcdefABCDEF'
string.octdigits             # → '01234567'
string.punctuation           # → '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
string.printable             # → all printable ASCII characters
string.whitespace            # → ' \t\n\r\x0b\x0c'

# Template strings
from string import Template
template = Template('Hello, $name! You have $count messages.')
result = template.substitute(name='Alice', count=3)
# → 'Hello, Alice! You have 3 messages.'

# Safe substitute (doesn't raise error for missing keys)
result = template.safe_substitute(name='Alice')
# → 'Hello, Alice! You have $count messages.'
```

Sys Module

```
import sys

# Command line arguments
sys.argv                      # → list of command line arguments

# Python version
sys.version                   # → Python version string
sys.version_info              # → version info tuple

# Maximum values
sys.maxsize                   # → maximum value for int
sys.float_info.max            # → maximum float value

# Recursion limit
sys.getrecursionlimit()       # → current recursion limit
sys.setrecursionlimit(2000)   # → set new recursion limit

# Exit program
sys.exit(0)                   # → exit with status code

# Standard streams
sys.stdin                     # → standard input
sys.stdout                    # → standard output
sys.stderr                    # → standard error

# Path manipulation
sys.path                      # → list of paths Python searches for modules
```

Os Module

```
import os

# Current working directory
os.getcwd()                  # → current directory
os.chdir('/path/to/directory')  # → change directory

# Environment variables
os.environ                   # → dictionary of environment variables
os.environ.get('PATH')       # → get PATH environment variable
os.environ.get('NONEXISTENT', 'default')  # → with default value

# Path operations
os.path.join('folder', 'file.txt')  # → 'folder/file.txt' or 'folder\\file.txt'
os.path.exists('file.txt')          # → True if file exists
os.path.isfile('file.txt')          # → True if it's a file
os.path.isdir('folder')             # → True if it's a directory
os.path.getsize('file.txt')         # → file size in bytes
os.path.basename('/path/to/file.txt')  # → 'file.txt'
os.path.dirname('/path/to/file.txt')   # → '/path/to'
os.path.splitext('file.txt')        # → ('file', '.txt')

# File operations
os.listdir('.')              # → list files in current directory
os.makedirs('new/nested/dir', exist_ok=True)  # → create nested directories
os.remove('file.txt')        # → delete file
os.rmdir('empty_dir')        # → delete empty directory
```

Re Module (Regular Expressions)

```
import re

# Basic pattern matching
pattern = r'\d+'              # match one or more digits
text = "I have 5 apples and 3 oranges"

# Find all matches
re.findall(pattern, text)     # → ['5', '3']

# Find first match
match = re.search(pattern, text)
if match:
    print(match.group())      # → '5'

# Replace patterns
re.sub(r'\d+', 'X', text)    # → 'I have X apples and X oranges'

# Split by pattern
re.split(r'\d+', text)       # → ['I have ', ' apples and ', ' oranges']

# Common patterns
r'\d'                         # digit
r'\w'                         # word character
r'\s'                         # whitespace
r'[a-z]'                      # lowercase letter
r'[A-Z]'                      # uppercase letter
r'[0-9]'                      # digit
r'.'                          # any character
r'^'                          # start of string
r'$'                          # end of string
r'*'                          # zero or more
r'+'                          # one or more
r'?'                          # zero or one
r'{n}'                        # exactly n times
r'{n,m}'                      # between n and m times

# Email validation example
email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
re.match(email_pattern, 'user@example.com')  # → Match object if valid
```

Definition: Searches for a pattern in a string and returns a match object. **Example:** import re match = re.search(r'\d+', 'I have 5 apples') if match: print(match.group()) # '5' **Use Case:** Finding patterns in text, data extraction.

Json Module

```
import json

# Convert Python object to JSON string
data = {'name': 'Alice', 'age': 30, 'city': 'New York'}
json_string = json.dumps(data)           # → '{"name": "Alice", "age": 30, "cit
json_pretty = json.dumps(data, indent=2) # → formatted JSON

# Convert JSON string to Python object
parsed_data = json.loads(json_string)    # → {'name': 'Alice', 'age': 30, 'city

# Read/write JSON files
with open('data.json', 'w') as f:
    json.dump(data, f)

with open('data.json', 'r') as f:
    loaded_data = json.load(f)

# Handle special cases
json.dumps([1, 2, 3])        # → '[1, 2, 3]'
json.dumps(None)             # → 'null'
json.dumps(True)             # → 'true'
json.dumps(False)            # → 'false'
```

Definition: Serializes a Python object to a JSON formatted string. **Example:** import json data = {'name': 'Alice', 'age': 30} json_str = json.dumps(data) # '{"name": "Alice", "age": 30}' **Use Case:** API responses, data serialization.

Itertools Advanced Functions

```
import itertools
import operator

# More combinatorial functions
list(itertools.product([1, 2], [3, 4]))  # → [(1, 3), (1, 4), (2, 3), (2, 4)]

# Flatten nested lists
def flatten(nested_list):
    return list(itertools.chain.from_iterable(nested_list))

flatten([[1, 2], [3, 4], [5]])  # → [1, 2, 3, 4, 5]

# Pairwise iteration (Python 3.10+)
# For older versions, use this implementation:
def pairwise(iterable):
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)

list(pairwise([1, 2, 3, 4]))  # → [(1, 2), (2, 3), (3, 4)]

# Sliding window
def sliding_window(iterable, n):
    it = iter(iterable)
    window = list(itertools.islice(it, n))
    if len(window) == n:
        yield tuple(window)
    for x in it:
        window.pop(0)
        window.append(x)
        yield tuple(window)

list(sliding_window([1, 2, 3, 4, 5], 3))  # → [(1, 2, 3), (2, 3, 4), (3, 4, 5)]

# Batching
def batched(iterable, n):
    iterator = iter(iterable)
    while True:
        batch = list(itertools.islice(iterator, n))
        if not batch:
            break
        yield batch

list(batched([1, 2, 3, 4, 5, 6, 7], 3))  # → [[1, 2, 3], [4, 5, 6], [7]]
```

More Essential Interview Functions

Union Find (Disjoint Set)

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

Trie (Prefix Tree)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

Segment Tree

```
    class SegmentTree:
        def __init__(self, arr):
            self.n = len(arr)
            self.tree = [0] * (4 * self.n)
            self.build(arr, 0, 0, self.n - 1)

        def build(self, arr, node, start, end):
            if start == end:
                self.tree[node] = arr[start]
            else:
                mid = (start + end) // 2
                self.build(arr, 2*node+1, start, mid)
                self.build(arr, 2*node+2, mid+1, end)
                self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

        def update(self, node, start, end, idx, val):
            if start == end:
                self.tree[node] = val
            else:
                mid = (start + end) // 2
                if idx <= mid:
```

```
                self.update(2*node+1, start, mid, idx, val)
            else:
                self.update(2*node+2, mid+1, end, idx, val)
            self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return 0
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return (self.query(2*node+1, start, mid, l, r) +
                self.query(2*node+2, mid+1, end, l, r))
```

Graph Algorithms

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

        return result

    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
        visited.add(start)
        result = [start]

        for neighbor in self.graph[start]:
            if neighbor not in visited:
                result.extend(self.dfs(neighbor, visited))

        return result

    def has_cycle(self):
        """Detect cycle in directed graph"""
        WHITE, GRAY, BLACK = 0, 1, 2
        color = defaultdict(int)

        def dfs(node):
            if color[node] == GRAY:
                return True
            if color[node] == BLACK:
                return False

            color[node] = GRAY
            for neighbor in self.graph[node]:
                if dfs(neighbor):
                    return True
            color[node] = BLACK
            return False

        for node in self.graph:
            if color[node] == WHITE:
                if dfs(node):
                    return True
        return False

    def topological_sort(self):
        """Topological sort using DFS"""
        visited = set()
        stack = []

        def dfs(node):
            visited.add(node)
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    dfs(neighbor)
            stack.append(node)

        for node in self.graph:
            if node not in visited:
                dfs(node)

        return stack[::-1]
```

Definition: A dict subclass that calls a factory function to supply missing values. **Example:** from collections import defaultdict dd = defaultdict(list) dd['key1'].append('value1') # No KeyError dd = defaultdict(int) dd['counter'] += 1 # No KeyError **Use Case:** Avoiding KeyError, grouping data.

Advanced Sorting Algorithms

```
def merge_sort(arr):
    """Merge sort implementation"""
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

def quick_sort(arr):
    """Quick sort implementation"""
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

```python
def heap_sort(arr):
    """Heap sort implementation"""
    import heapq
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]
```

Definition: Provides heap queue algorithm (priority queue). **Example:** import heapq heap = [3, 1, 4, 1, 5] heapq.heapify(heap) # [1, 1, 4, 3, 5] heapq.heappush(heap, 2) # [1, 1, 2, 3, 5, 4] heapq.heappop(heap) # 1 **Use Case:** Finding k largest/smallest elements, Dijkstra's algorithm.

Dynamic Programming Patterns

```python
def knapsack_01(weights, values, capacity):
    """0/1 Knapsack problem"""
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    dp[i-1][w],
                    dp[i-1][w-weights[i-1]] + values[i-1]
                )
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]

def longest_common_subsequence(text1, text2):
    """LCS problem"""
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

def coin_change(coins, amount):
    """Coin change problem"""
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1
```

Bit Manipulation

```
def count_set_bits(n):
    """Count number of set bits"""
    count = 0
    while n:
        count += 1
        n &= n - 1  # Remove rightmost set bit
    return count

def is_power_of_two(n):
    """Check if number is power of 2"""
    return n > 0 and (n & (n - 1)) == 0

def find_single_number(nums):
    """Find single number in array where all others appear twice"""
    result = 0
    for num in nums:
        result ^= num
    return result

def reverse_bits(n):
    """Reverse bits of a 32-bit integer"""
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result

def get_bit(n, i):
    """Get i-th bit"""
    return (n >> i) & 1

def set_bit(n, i):
    """Set i-th bit"""
    return n | (1 << i)
```

```
def clear_bit(n, i):
    """Clear i-th bit"""
    return n & ~(1 << i)
```

Advanced Python Features for Interviews

Context Managers

```
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, *args):
        import time
        self.end = time.time()
        print(f"Elapsed time: {self.end - self.start:.4f} seconds")

# Usage
with Timer():
    # Some code to time
    sum(range(1000000))

# File context manager
with open('file.txt', 'r') as f:
    content = f.read()
```

Decorators

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci(n):
    if n < 2:
        return n
        return fibonacci(n-1) + fibonacci(n-2)

# Property decorator
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

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

    @property
    def area(self):
        return 3.14159 * self._radius ** 2
```

Generators

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

def sliding_window_generator(iterable, n):
    it = iter(iterable)
    window = list(itertools.islice(it, n))
    if len(window) == n:
        yield tuple(window)
    for x in it:
        window.pop(0)
        window.append(x)
        yield tuple(window)

# Generator expression
squares = (x**2 for x in range(10))
```

This comprehensive guide covers all the essential Python functions, methods, and modules you'll need for technical interviews. Practice these patterns and functions regularly to build muscle memory!

This comprehensive addition includes all the essential Python modules and advanced functions you'll need for technical interviews. The guide now covers data structures, algorithms, bit manipulation, dynamic programming, and advanced Python features!