Complete Python Mastery Guide

1. Data Structures

Lists

Ordered, mutable collection that allows duplicates.


python

```python
# Creation and basic operations
nums = [1, 2, 3, 4, 5]
nums.append(6)          # Add to end
nums.insert(0, 0)       # Insert at index
nums.extend([7, 8])     # Add multiple items
nums.remove(3)          # Remove first occurrence
popped = nums.pop()     # Remove and return last item
nums.pop(0)             # Remove at index
nums.reverse()          # Reverse in place
nums.sort()             # Sort in place
```

List Comprehensions:


python

```python
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]
matrix = [[i+j for j in range(3)] for i in range(3)]
```

Tuples

Ordered, immutable collection. Faster than lists, used for fixed data.


python

```python
point = (10, 20)
x, y = point          # Unpacking
nested = (1, 2, (3, 4))
single = (1,)         # Single element tuple needs comma
```

Sets

Unordered collection of unique elements. Fast membership testing.


python

```python
s = {1, 2, 3, 4}
s.add(5)
s.remove(2)           # Raises error if not found
s.discard(2)          # No error if not found
s.update([6, 7, 8])

# Set operations
a = {1, 2, 3}
b = {3, 4, 5}
union = a | b         # {1, 2, 3, 4, 5}
intersection = a & b  # {3}
difference = a - b    # {1, 2}
sym_diff = a ^ b      # {1, 2, 4, 5}
```

Dictionaries

Key-value pairs, fast lookup by key.


python

```python
person = {'name': 'John', 'age': 30}
person['city'] = 'NYC'
age = person.get('age', 0)        # Default if key missing
person.setdefault('country', 'USA')
person.update({'age': 31, 'job': 'Engineer'})

# Dictionary methods
keys = person.keys()
values = person.values()
items = person.items()

# Dictionary comprehension
squares = {x: x**2 for x in range(5)}
```

Stacks (using list)

LIFO - Last In First Out

python

```python
stack = []
stack.append(1)    # Push
stack.append(2)
top = stack.pop()  # Pop
peek = stack[-1]   # Peek without removing
```

Queues (using collections.deque)

FIFO - First In First Out

python

```python
from collections import deque

queue = deque([1, 2, 3])
queue.append(4)       # Enqueue
first = queue.popleft()  # Dequeue
```

Heaps (Priority Queue)


python

```python
import heapq

heap = [5, 3, 7, 1]
heapq.heapify(heap)          # Convert to min heap
heapq.heappush(heap, 4)      # Add element
smallest = heapq.heappop(heap)  # Remove smallest
three_smallest = heapq.nsmallest(3, heap)
```

Counter (from collections)


python

```python
from collections import Counter

words = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
count = Counter(words)
# Counter({'apple': 3, 'banana': 2, 'orange': 1})
most_common = count.most_common(2)
```

DefaultDict


python

```python
from collections import defaultdict

# Auto-initialize missing keys
dd = defaultdict(list)
dd['fruits'].append('apple')  # No KeyError

dd_int = defaultdict(int)
dd_int['count'] += 1  # Starts at 0
```

OrderedDict

Remembers insertion order (though regular dicts do too in Python 3.7+).

python

```python
from collections import OrderedDict

od = OrderedDict()
od['a'] = 1
od['b'] = 2
od.move_to_end('a')  # Move to end
```

---

2. String Methods and Functions

Core String Methods

python

```python
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
s.strip()          # "Hello World"
s.lstrip()         # "Hello World  "
s.rstrip()         # "  Hello World"

# Search and replace
s.find('World')    # Returns 8 (index)
s.index('World')   # Returns 8, raises error if not found
s.count('l')       # Count occurrences
s.replace('World', 'Python')
```

String Checking Methods

python

```python
"hello".isalpha()      # True (only letters)
"123".isdigit()        # True (only digits)
"hello123".isalnum()   # True (letters + digits)
"   ".isspace()        # True (only whitespace)
"Hello".isupper()      # False
"hello".islower()      # True
"Hello World".istitle() # True
```

String Splitting and Joining

python

```python
text = "apple,banana,orange"
fruits = text.split(',')       # ['apple', 'banana', 'orange']
words = "hello world".split()  # ['hello', 'world']

# Join
result = '-'.join(fruits)      # 'apple-banana-orange'
path = '/'.join(['home', 'user', 'docs'])

# Partition
"hello:world".partition(':')   # ('hello', ':', 'world')
```

String Formatting


python

```python
name = "John"
age = 30

# f-strings (modern, preferred)
msg = f"Name: {name}, Age: {age}"
formatted = f"{age:.2f}"       # 30.00

# format method
msg = "Name: {}, Age: {}".format(name, age)
msg = "Name: {n}, Age: {a}".format(n=name, a=age)

# Old style (avoid)
msg = "Name: %s, Age: %d" % (name, age)
```

String Alignment and Padding


python

```python
text = "hello"
text.center(10)     # "  hello   "
text.ljust(10, '-')  # "hello-----"
text.rjust(10, '-')  # "-----hello"
text.zfill(10)      # "00000hello"
```

Advanced String Operations

python

```python
# Start and end checking
"hello.py".startswith('hello')  # True
"hello.py".endswith('.py')      # True

# Encoding
text = "hello"
encoded = text.encode('utf-8')  # b'hello'
decoded = encoded.decode('utf-8')

# String slicing
s = "Python"
s[0:2]     # "Py"
s[::2]     # "Pto" (every 2nd char)
s[::-1]    # "nohtyP" (reverse)
```

---

3. Array/List Methods and Functions

Built-in Functions for Lists

python

```python
nums = [3, 1, 4, 1, 5, 9, 2]

len(nums)          # Length
max(nums)          # Maximum value
min(nums)          # Minimum value
sum(nums)          # Sum of all elements
sorted(nums)       # Return new sorted list (doesn't modify original)
reversed(nums)     # Return reverse iterator
all([True, True])  # True if all are True
any([False, True])  # True if any is True
```

List Methods

python

```python
arr = [1, 2, 3]

# Adding elements
arr.append(4)          # [1, 2, 3, 4]
arr.extend([5, 6])     # [1, 2, 3, 4, 5, 6]
arr.insert(0, 0)       # [0, 1, 2, 3, 4, 5, 6]

# Removing elements
arr.remove(3)          # Remove first occurrence of 3
popped = arr.pop()     # Remove and return last
popped = arr.pop(0)    # Remove at index
arr.clear()            # Empty the list

# Other operations
arr = [3, 1, 2]
arr.reverse()          # Reverse in place
arr.sort()             # Sort in place
arr.sort(reverse=True)  # Sort descending
arr.sort(key=lambda x: -x)  # Custom sort
arr.copy()             # Shallow copy
arr.count(1)           # Count occurrences
arr.index(2)           # Find index of first occurrence
```

List Slicing

python

```python
arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

arr[2:5]      # [2, 3, 4]
arr[:5]       # [0, 1, 2, 3, 4]
arr[5:]       # [5, 6, 7, 8, 9]
arr[::2]      # [0, 2, 4, 6, 8] (every 2nd)
arr[1::2]     # [1, 3, 5, 7, 9] (odd indices)
arr[::-1]     # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (reverse)
arr[:-3]      # [0, 1, 2, 3, 4, 5, 6] (all except last 3)
```

Enumerate and Zip

python

```python
# Enumerate - get index and value
fruits = ['apple', 'banana', 'orange']
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")

for i, fruit in enumerate(fruits, start=1):
    print(f"{i}: {fruit}")  # Start from 1

# Zip - combine multiple iterables
names = ['John', 'Jane', 'Bob']
ages = [30, 25, 35]
for name, age in zip(names, ages):
    print(f"{name} is {age}")

# Zip to create dict
person_dict = dict(zip(names, ages))
```

---

4. Map, Filter, and Reduce

Map

Apply a function to all items in an iterable.


python

```python
# Double all numbers
nums = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, nums))  # [2, 4, 6, 8, 10]

# Convert strings to integers
strings = ['1', '2', '3']
integers = list(map(int, strings))  # [1, 2, 3]

# Multiple iterables
a = [1, 2, 3]
b = [4, 5, 6]
result = list(map(lambda x, y: x + y, a, b))  # [5, 7, 9]
```

Filter

Keep only items that match a condition.


python

```python
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Get even numbers
evens = list(filter(lambda x: x % 2 == 0, nums))  # [2, 4, 6, 8, 10]

# Filter non-empty strings
words = ['hello', '', 'world', '', 'python']
non_empty = list(filter(None, words))  # ['hello', 'world', 'python']
```

Reduce

Reduce a list to a single value by applying a function cumulatively.

python

```python
from functools import reduce

nums = [1, 2, 3, 4, 5]

# Sum all numbers
total = reduce(lambda x, y: x + y, nums)  # 15

# Find maximum
maximum = reduce(lambda x, y: x if x > y else y, nums)  # 5

# Factorial
factorial = reduce(lambda x, y: x * y, range(1, 6))  # 120
```

Lambda Functions

Anonymous functions for short operations.

python

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

# Lambda with multiple arguments
add = lambda x, y: x + y
print(add(3, 4))  # 7

# Lambda in sorting
students = [('John', 85), ('Jane', 90), ('Bob', 75)]
students.sort(key=lambda x: x[1])  # Sort by grade
```

---

5. Object-Oriented Programming (OOP)

Classes and Objects


python

```python
class Person:
    # Class variable (shared by all instances)
    species = "Homo sapiens"

    # Constructor
    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

    # Instance method
    def greet(self):
        return f"Hello, I'm {self.name}"

    # Method with parameters
    def have_birthday(self):
        self.age += 1
        return f"Happy birthday! Now {self.age}"

# Create objects
person1 = Person("John", 30)
person2 = Person("Jane", 25)
print(person1.greet())
```

The Four Pillars of OOP

1. Encapsulation

Bundle data and methods together, control access.


python

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

    def get_balance(self):  # Getter
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

2. Inheritance

Child class inherits from parent class.

python

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

dog = Dog("Buddy")
print(dog.speak())  # Buddy says Woof!
```

3. Polymorphism

Different classes can be used through the same interface.


python

```python
def animal_sound(animal):
    print(animal.speak())

dog = Dog("Max")
cat = Cat("Whiskers")

animal_sound(dog)  # Max says Woof!
animal_sound(cat)  # Whiskers says Meow!
```

4. Abstraction

Hide complex implementation, show only essential features.


python

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

rect = Rectangle(5, 3)
print(rect.area())  # 15
```

Special Methods (Magic Methods)

python

```python
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):  # String representation
        return f"{self.title} ({self.pages} pages)"

    def __repr__(self):  # Developer representation
        return f"Book('{self.title}', {self.pages})"

    def __len__(self):  # len(book)
        return self.pages

    def __eq__(self, other):  # book1 == book2
        return self.pages == other.pages

    def __lt__(self, other):  # book1 < book2
        return self.pages < other.pages

book = Book("Python Guide", 300)
print(book)  # Python Guide (300 pages)
print(len(book))  # 300
```

Class Methods and Static Methods

python

```python
class MathOperations:
    pi = 3.14159

    @classmethod
    def from_diameter(cls, diameter):
        # Alternative constructor
        radius = diameter / 2
        return cls(radius)

    @staticmethod
    def is_even(num):
        # Doesn't need class or instance
        return num % 2 == 0

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return MathOperations.pi * self.radius ** 2

circle = MathOperations.from_diameter(10)
print(MathOperations.is_even(4))  # True
```

Property Decorators

python

```python
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Too cold!")
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

temp = Temperature(25)
print(temp.fahrenheit)  # 77.0
temp.celsius = 30
```

Multiple Inheritance

python

```python
class Flyable:
    def fly(self):
        return "Flying!"

class Swimmable:
    def swim(self):
        return "Swimming!"

class Duck(Flyable, Swimmable):
    def quack(self):
        return "Quack!"

duck = Duck()
print(duck.fly())   # Flying!
print(duck.swim())  # Swimming!
```

---

6. Design Patterns

1. Singleton Pattern

Ensure a class has only one instance.

python

```python
class Database:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.connection = "Connected to DB"
        return cls._instance

db1 = Database()
db2 = Database()
print(db1 is db2)  # True (same instance)
```

2. Factory Pattern

Create objects without specifying exact class.

python

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        return None

animal = AnimalFactory.create_animal("dog")
print(animal.speak())  # Woof!
```

3. Observer Pattern

Objects notify other objects about state changes.

python

```python
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

class Observer:
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received: {message}")

subject = Subject()
obs1 = Observer("Observer1")
obs2 = Observer("Observer2")
subject.attach(obs1)
subject.attach(obs2)
subject.notify("Hello!")
```

4. Strategy Pattern

Select algorithm at runtime.

python

```python
class PaymentStrategy:
    def pay(self, amount):
        pass

class CreditCard(PaymentStrategy):
    def pay(self, amount):
        return f"Paid ${amount} with Credit Card"

class PayPal(PaymentStrategy):
    def pay(self, amount):
        return f"Paid ${amount} with PayPal"

class ShoppingCart:
    def __init__(self, payment_strategy):
        self.payment_strategy = payment_strategy

    def checkout(self, amount):
        return self.payment_strategy.pay(amount)

cart = ShoppingCart(CreditCard())
print(cart.checkout(100))
```

5. Decorator Pattern

Add functionality to objects dynamically.

python

```python
class Coffee:
    def cost(self):
        return 5

    def description(self):
        return "Coffee"

class MilkDecorator:
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + 2

    def description(self):
        return self.coffee.description() + ", Milk"

coffee = Coffee()
coffee_with_milk = MilkDecorator(coffee)
print(coffee_with_milk.description())  # Coffee, Milk
print(coffee_with_milk.cost())  # 7
```

6. Builder Pattern

Construct complex objects step by step.

python

```python
class Pizza:
    def __init__(self):
        self.size = None
        self.cheese = False
        self.pepperoni = False

    def __str__(self):
        return f"{self.size} pizza, cheese={self.cheese}, pepperoni={self.pepperoni}"

class PizzaBuilder:
    def __init__(self):
        self.pizza = Pizza()

    def set_size(self, size):
        self.pizza.size = size
        return self

    def add_cheese(self):
        self.pizza.cheese = True
        return self

    def add_pepperoni(self):
        self.pizza.pepperoni = True
        return self

    def build(self):
        return self.pizza

pizza = PizzaBuilder().set_size("Large").add_cheese().add_pepperoni().build()
print(pizza)
```

7. Adapter Pattern

Make incompatible interfaces work together.

python

```python
class EuropeanSocket:
    def voltage(self):
        return 230

class USASocket:
    def voltage(self):
        return 120

class Adapter:
    def __init__(self, socket):
        self.socket = socket

    def voltage(self):
        return self.socket.voltage() // 2

eu_socket = EuropeanSocket()
adapter = Adapter(eu_socket)
print(adapter.voltage())  # 115
```

8. Command Pattern

Encapsulate requests as objects.

python

```
class Light:
    def on(self):
        return "Light is ON"

    def off(self):
        return "Light is OFF"

class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        return self.light.on()

class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        return self.command.execute()

light = Light()
remote = RemoteControl()
remote.set_command(LightOnCommand(light))
print(remote.press_button())  # Light is ON
```

---

7. Advanced Topics

Generators

Functions that yield values one at a time (memory efficient).

python

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(5):
    print(num)  # 5, 4, 3, 2, 1

# Generator expression
squares = (x**2 for x in range(10))
print(next(squares))  # 0
print(next(squares))  # 1
```

Decorators

Modify function behavior without changing code.

python

```python
def timer(func):
    def wrapper(*args, **kwargs):
        import time
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Time: {end - start:.4f}s")
        return result
    return wrapper

@timer
def slow_function():
    import time
    time.sleep(1)
    return "Done"

slow_function()
```

Context Managers

Manage resources properly (file handling, connections).

python

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()

with FileManager('test.txt', 'w') as f:
    f.write('Hello')
```

List vs Generator Performance

python

```python
# List - stores all values in memory
list_comp = [x**2 for x in range(1000000)]

# Generator - computes values on demand
gen_exp = (x**2 for x in range(1000000))

# Generators are better for large datasets
```

---

8. Common Algorithms

Sorting

python

```python
# Bubble Sort
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Binary Search (on sorted array)
def binary_search(arr, target):
    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
```

Two Pointers Technique

python

```python
# Find pair that sums to target
def two_sum(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        current = arr[left] + arr[right]
        if current == target:
            return [left, right]
        elif current < target:
            left += 1
        else:
            right -= 1
    return []
```

Sliding Window


python

```python
# Max sum of k consecutive elements
def max_sum_subarray(arr, k):
    max_sum = window_sum = sum(arr[:k])
    for i in range(len(arr) - k):
        window_sum = window_sum - arr[i] + arr[i + k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

---

9. Best Practices

Code Style

- Use 4 spaces for indentation
- Name variables with lowercase and underscores: `user_name`
- Name classes with CamelCase: `UserAccount`
- Name constants with UPPERCASE: `MAX_SIZE`

List Comprehension vs Loops

Prefer list comprehensions for simple operations:


python

```python
# Good
squares = [x**2 for x in range(10)]

# Avoid when complex logic is needed
# Use regular loop for readability
```

Don't Repeat Yourself (DRY)


python

```python
# Bad
print("Hello, John")
print("Hello, Jane")
print("Hello, Bob")

# Good
def greet(name):
    print(f"Hello, {name}")

for name in ["John", "Jane", "Bob"]:
    greet(name)
```

Use Built-in Functions

Python's built-in functions are optimized:


python

```python
# Use built-ins when possible
total = sum(numbers)  # Better than manual loop
maximum = max(numbers)
```

---

10. Common Pitfalls to Avoid

Mutable Default Arguments

python

```python
# Bad
def add_item(item, items=[]):
    items.append(item)
    return items

# Good
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
```

Shallow vs Deep Copy

python

```python
import copy

original = [[1, 2], [3, 4]]
shallow = original.copy()
deep = copy.deepcopy(original)

shallow[0][0] = 99  # Affects original!
deep[0][0] = 99     # Doesn't affect original
```

Variable Scope

python

```python
x = 10

def modify():
    global x  # Need global to modify
    x = 20

def local_scope():
    x = 30  # Creates new local variable
    return x
```

Quick Reference Cheat Sheet

Time Complexity

- List append: O(1)
- List insert: O(n)
- List search: O(n)
- Dict get/set: O(1)
- Set add/remove: O(1)
- Sorting: O(n log n)

Space Complexity Tips

- Use generators for large datasets
- Use itertools for memory-efficient loops
- Del variables when done with them

Common Imports


python

```python
import math          # Math functions
import random        # Random numbers
import datetime      # Date and time
import re            # Regular expressions
import json          # JSON handling
import sys           # System functions
from collections import Counter, defaultdict, deque
from itertools import permutations, combinations
from functools import reduce
```

This guide covers the core concepts you need to master Python. Practice implementing these patterns and understanding when to use each data structure. Focus on writing clean, readable code and always test your solutions with edge cases.