Let's Learn Python (LLP)

Yogesh Haribhau Kulkarni

Overview

Overview

Why Python?

- Readability.
- Ease of use.
- Fits in your head.
- Gets things done.
- Good libraries.
- Lookie what I did!.

Course Overview

- Python Fundamentals
- Object-Oriented Programming
- File Handling & I/O Operations
- Advanced Python Features
- Asynchronous Programming
- Real-world Projects

Introduction

- Python is a simple, yet powerful interpreted language.
- Numerous libraries: NumPy, SciPy, Matplotlib . . . .
- Named after Monty Python.
- Open Source and Free
- Invented by Guido van Rossum.

Python's Benevolent Dictator For Life

"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."
- Guido van Rossum

(Reference: https://en.wikipedia.org/wiki/Guido_van_Rossum)

What is Python?

- Interpreted
- Object-oriented
- High-level
- Dynamic semantics
- Cross-platform
- Readability.

Compiled Languages

- Needs entire program
- Translates directly to machine codes
- Exe native and fast
- Usually statically typed
- Types known during compilation
- Change in type : recompilation
- Ideal for compute-heavy tasks
- E.g. C, C++, FORTRAN

Interpreted Languages

- Interpreted on the fly
- No need to compile: can execute right away
- Usually dynamically typed
- Non-syntax errors are detected only in run-time
- Slower than compiled languages
- Ideal for small tasks
- E.g. Python, Perl, PHP, Bash

Note: Python, at the beginning, loosely checks the program. Only at run time, line-by-line, it checks for errors. So, if the error statements are not in the running path, their error does not get reported. So, its a bit relaxed. Thats the objection for its use in production code, where strong type checking and error checking, upfront is essential.

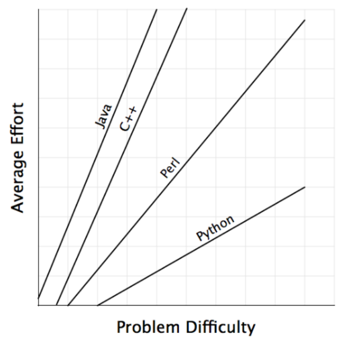
JIT-Compiled Languages

- Between compiled and interpreted
- Code is initially interpreted, hotspots compiled
- Deduce types during compilation, can change in run-time
- Slower than compiled
- E.g. Java, C#

"C" guys to take pride in

- Python interpreter written in "C"
- Source code at www.python.org
- So, Python Interpreter is compiled exe

Completely Controversial Observations about Languages



Important features

- Built-in high level data types: `strings`, `lists`, `dictionaries`, etc.
- Usual control structures: `if`, `if-else`, `if-elif-else`, `while`, `for`
- Levels of organization: `functions`, `classes`, `modules`, `packages`
- Extensions in C and C++ possible

Python 2 *vs* Python 3

- Two major versions 2.*, 3.*
- Python 2.7: Latest release in 2.x series
- Python 3.5: More polished syntax, removed inconsistencies

Variables & Data Types

- Dynamic typing - no declaration needed
- Basic types: int, float, str, bool
- Type conversion and checking
- Variable naming conventions
- Multiple assignment

```
# Variable assignment
name = ”Alice”
age = 25
height = 5.6
is_student = True

# Type checking
print(type(age))  # <class 'int'>

# Type conversion
age_str = str(age)
price = float(”19.99”)

# Multiple assignment
x, y, z = 1, 2, 3
```

Data Structures - Lists, Tuples, Dicts, Sets

- Lists: ordered, mutable sequences
- Tuples: ordered, immutable sequences
- Dictionaries: key-value pairs
- Sets: unordered, unique elements
- Indexing and slicing

```
# List
fruits = [”apple”, ”banana”, ”cherry”]
fruits.append(”date”)

# Tuple
coordinates = (10, 20)

# Dictionary
person = {”name”: ”Bob”, ”age”: 30}
person[”city”] = ”NYC”

# Set
unique_nums = {1, 2, 3, 3, 2}
print(unique_nums)  # {1, 2, 3}
```

Control Flow - Conditional Statements

- if, elif, else statements
- Comparison operators (==, !=, <, >, <=, >=)
- Logical operators (and, or, not)
- Ternary operator
- Truthiness in Python

```
score = 85

if score >= 90:
    grade = ”A”
elif score >= 80:
    grade = ”B”
elif score >= 70:
    grade = ”C”
else:
    grade = ”F”

# Ternary operator
status = ”Pass” if score >= 60 else ”Fail”

# Logical operators
if score > 70 and score < 90:
    print(”Good performance”)
```

Control Flow - Loops

- for loops - iterate over sequences
- while loops - condition-based iteration
- break and continue statements
- range() function
- enumerate() for index tracking
- Loop with else clause

```
# For loop with range
for i in range(5):
    print(i)

# Iterate over list
fruits = [”apple”, ”banana”, ”cherry”]
for fruit in fruits:
    print(fruit)

# While loop
count = 0
while count < 3:
    print(count)
    count += 1

# Enumerate
for idx, fruit in enumerate(fruits):
    print(f”{idx}: {fruit}”)
```

Functions - Basics

- Function definition with def keyword
- Parameters and arguments
- Return values
- Default parameter values
- Docstrings for documentation
- Function as first-class objects

```
# Basic function
def greet(name):
    ”””Greet a person by name”””
    return f”Hello, {name}!”

# Default parameters
def power(base, exp=2):
    return base ** exp

print(power(3))      # 9
print(power(3, 3))   # 27

# Multiple return values
def get_stats(numbers):
    return min(numbers), max(numbers)

min_val, max_val = get_stats([1, 5, 3])
```

Functions - Arguments & Scope

- Positional arguments
- Keyword arguments
- *args for variable arguments
- **kwargs for keyword arguments
- Local vs global scope
- LEGB rule (Local, Enclosing, Global, Built-in)

```
# *args and **kwargs
def calculate(*args, **kwargs):
    total = sum(args)
    operation = kwargs.get('op', 'sum')
    return f”{operation}: {total}”

result = calculate(1, 2, 3, op=”total”)

# Scope example
x = 10  # Global

def modify():
    x = 5  # Local
    print(x)  # 5

modify()
print(x)  # 10 (global unchanged)
```

Lambda Functions & Built-in Functions

- Anonymous functions with lambda
- Single expression functions
- map() - apply function to iterable
- filter() - filter items by condition
- reduce() - aggregate values
- sorted() with key parameter

```
# Lambda function
square = lambda x: x ** 2
print(square(5))  # 25

# map() − transform elements
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))

# filter() − select elements
evens = list(filter(lambda x: x % 2 == 0, numbers))

# sorted with key
students = [(”Alice”, 85), (”Bob”, 92)]
sorted_students = sorted(students,
                         key=lambda x: x[1],
                         reverse=True)
```

OOP - Classes & Objects

- Class definition with class keyword
- __init__ constructor method
- Instance variables (self)
- Instance methods
- Class vs instance attributes
- Encapsulation concept

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

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

    def bark(self):
        return f"{self.name} says Woof!"

    def description(self):
        return f"{self.name} is {self.age} years old"

# Create objects
buddy = Dog("Buddy", 3)
print(buddy.bark())
```

OOP - Inheritance

- Inheriting from parent classes
- super() to call parent methods
- Method overriding
- Multiple inheritance
- isinstance() and issubclass()
- Code reusability

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

    def speak(self):
        pass

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        return f"{self.name} barks!"

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

dog = Dog("Max", "Labrador")
print(dog.speak())
```

OOP - Polymorphism & Special Methods

- Method overriding for polymorphism
- Duck typing in Python
- Special/magic methods (__str__, __repr__)
- Operator overloading (__add__, __eq__)
- __len__, __getitem__ for collections

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)

    def __eq__(self, other):
        return self.x == other.x and \
               self.y == other.y

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2  # Calls __add__
print(v3)     # Calls __str__
```

OOP - Encapsulation & Properties

- Private attributes with __ prefix
- Protected attributes with _ prefix
- Property decorators (@property)
- Getters and setters
- Computed properties
- Data validation

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

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, amount):
        if amount < 0:
            raise ValueError("Balance cannot be negative")
        self.__balance = amount

    def deposit(self, amount):
        self.balance += amount

account = BankAccount(100)
account.deposit(50)
print(account.balance)  # 150
```

Project: Library Management System

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.is_available = True

class Member:
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.borrowed_books = []

class Library:
    def __init__(self):
        self.books = []
        self.members = []

    def add_book(self, book): # TODO: Implement adding book to
     library
        pass

    def lend_book(self, member, book): # TODO: Check availability,
     update records
        pass

    def return_book(self, member, book): # TODO: Process book return
        pass
```

Project: Shopping Cart System

```
class Product:
    def __init__(self, name, price, category):
        self.name = name
        self.price = price
        self.category = category

class ShoppingCart:
    def __init__(self):
        self.items = {}  # {product: quantity}

    def add_item(self, product, quantity=1): # TODO: Add product to
     cart
        pass

    def remove_item(self, product): # TODO: Remove product from cart
        pass

    def calculate_total(self, discount_code=None):
        # TODO: Calculate total with optional discount
        # Apply percentage discount if code valid
        pass

    def checkout(self): # TODO: Process payment and clear cart
        pass
```

File I/O - Reading & Writing Text Files

- open() function with modes (r, w, a)
- Reading: read(), readline(), readlines()
- Writing: write(), writelines()
- Context managers (with statement)
- File paths - absolute vs relative
- Automatic file closing

```
# Reading a file
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
# Reading line by line
with open('data.txt', 'r') as file:
    for line in file:
        print(line.strip())

# Writing to a file
data = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open('output.txt', 'w') as file:
    file.writelines(data)

# Appending to a file
with open('log.txt', 'a') as file:
    file.write("New log entry\n")
```

JSON Handling

- json module for JSON operations
- json.dump() - write to file
- json.dumps() - convert to string
- json.load() - read from file
- json.loads() - parse from string
- Handling nested structures

```
import json

# Python dict to JSON
data = {
    "name": "Alice",
    "age": 30,
    "skills": ["Python", "SQL"]
}

# Write to file
with open('data.json', 'w') as f:
    json.dump(data, f, indent=4)

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

# Convert to/from string
json_string = json.dumps(data)
parsed_data = json.loads(json_string)
```

CSV Processing

- csv module for CSV operations
- csv.reader() for reading
- csv.writer() for writing
- DictReader for dictionary access
- DictWriter for writing dicts
- Handling different delimiters

```
import csv

# Reading CSV
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

# Reading as dictionaries
with open('data.csv', 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row['name'], row['age'])

# Writing CSV
data = [['Name', 'Age'],
        ['Alice', 30],
        ['Bob', 25]]
with open('output.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

Exception Handling

- try-except blocks
- Multiple except clauses
- else clause - executes if no exception
- finally clause - always executes
- Raising exceptions with raise
- Custom exception classes

```
# Basic exception handling
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
else:
    print("Success!")
finally:
    print("Cleanup code")

# Custom exception
class InvalidAgeError(Exception):
    pass

def set_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative")
    return age

try:
    set_age(-5)
except InvalidAgeError as e:
    print(e)
```

Decorators Basics

- Functions as first-class objects
- Wrapper functions
- @ syntax for applying decorators
- functools.wraps for metadata
- Common use cases: logging, timing, caching
- Parameterized decorators

```
import time
from functools import wraps

def timer(func):
    @wraps(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

def log_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@timer
@log_call
def process_data(n):
    return sum(range(n))
```

Context Managers

- with statement for resource management
- __enter__ and __exit__ methods
- contextlib module
- @contextmanager decorator
- Automatic cleanup of resources
- Custom context managers

```
from contextlib import contextmanager

# Custom context manager class
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

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

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

# Using decorator
@contextmanager
def timer_context():
    start = time.time()
    yield
    print(f"Elapsed: {time.time()-start:.2f}s")

with timer_context():
    # Code to time
    time.sleep(1)
```

Project: CSV Data Processor

```python
import csv
import logging

def log_operation(func):
    """Decorator to log operations"""
    def wrapper(*args, **kwargs):
        logging.info(f"Starting {func.__name__}")
        try:
            result = func(*args, **kwargs)
            logging.info(f"Completed {func.__name__}")
            return result
        except Exception as e:
            logging.error(f"Error in {func.__name__}: {e}")
            raise
    return wrapper

class CSVProcessor:
    def __init__(self, input_file):
        self.input_file = input_file
    @log_operation
    def read_data(self): # TODO: Read CSV with error handling
        pass
    @log_operation
    def filter_data(self, condition): # TODO: Filter rows based on
        condition
        pass
    @log_operation
    def transform_data(self, transformation): # TODO: Apply
        transformation to columns
        pass
    @log_operation
    def export_results(self, output_file): # TODO: Write processed data
        to new CSV
        pass
```

Project: Config File Manager

```python
import json
import yaml
from contextlib import contextmanager
from datetime import datetime

class ConfigManager:
    def __init__(self, config_file):
        self.config_file = config_file
        self.config = {}

    @contextmanager
    def config_context(self): # TODO: Backup current config
        """Context manager for safe config operations"""
        try:
            yield self
        except Exception as e: # TODO: Restore from backup
            raise
        finally: # TODO: Cleanup
            pass

    def load_config(self): # TODO: Load JSON/YAML config
        pass
    def validate_config(self, schema): # TODO: Validate against schema
        pass
    def save_config(self): # TODO: Save config with backup
        pass
    def get(self, key, default=None): # TODO: Get config value with
        dot notation
        pass
```

Comprehensions

- List comprehensions for concise lists
- Dict comprehensions for dictionaries
- Set comprehensions for unique values
- Conditional logic in comprehensions
- Nested comprehensions
- Performance benefits

```python
# List comprehension
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]

# Dict comprehension
word_lengths = {word: len(word)
               for word in ['apple', 'banana']}

# Set comprehension
unique_squares = {x**2 for x in range(-5, 6)}

# Nested comprehension
matrix = [[i*j for j in range(3)]
          for i in range(3)]

# Conditional expression
results = [x if x > 0 else 0
           for x in range(-5, 5)]
```

Generators & Generator Expressions

- Memory-efficient iteration
- yield keyword for generators
- Generator expressions (lazy evaluation)
- next() function
- StopIteration exception
- Infinite sequences

```python
# Generator function
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Using generator
for num in fibonacci(10):
    print(num)

# Generator expression
squares_gen = (x**2 for x in range(1000000))
print(next(squares_gen))  # 0
print(next(squares_gen))  # 1

# Infinite generator
def infinite_counter():
    n = 0
    while True:
        yield n
        n += 1
```

Iterators & Advanced Iteration

- Iterator protocol: __iter__ and __next__
- itertools module
- Chain, cycle, repeat functions
- Combinations and permutations
- zip() and zip_longest()
- Creating custom iterators

```python
from itertools import islice, chain, cycle

# Custom iterator
class CountDown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
        return self.current + 1

# Using itertools
combined = chain([1, 2], [3, 4])
first_five = islice(range(100), 5)

# Zip multiple iterables
names = ['Alice', 'Bob']
ages = [30, 25]
for name, age in zip(names, ages):
    print(f"{name}: {age}")
```

Concurrent Operations - Threading

- threading module for concurrency
- Thread creation and management
- Global Interpreter Lock (GIL)
- Thread-safe operations
- concurrent.futures.ThreadPoolExecutor
- Use for I/O-bound tasks

```python
import threading
from concurrent.futures import ThreadPoolExecutor
import time

# Basic threading
def task(name):
    print(f"Task {name} starting")
    time.sleep(1)
    print(f"Task {name} complete")

threads = []
for i in range(3):
    t = threading.Thread(target=task, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

# ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(task, i) for i in range(3)]
```

Async/Await Basics

- Asynchronous programming concepts
- async def for coroutines
- await keyword for awaiting
- Event loop with asyncio
- Concurrent execution of coroutines
- Non-blocking I/O operations

```python
import asyncio

# Basic async function
async def fetch_data(id):
    print(f"Fetching {id}...")
    await asyncio.sleep(1)  # Simulates I/O
    print(f"Fetched {id}")
    return f"Data {id}"

# Running coroutines
async def main():
    # Sequential
    result1 = await fetch_data(1)

    # Concurrent
    results = await asyncio.gather(
        fetch_data(2),
        fetch_data(3),
        fetch_data(4)
    )
    print(results)

# Run event loop
asyncio.run(main())
```

Project: Async Web Scraper

```python
import asyncio
import aiohttp
from bs4 import BeautifulSoup

class AsyncWebScraper:
    def __init__(self, urls):
        self.urls = urls
        self.results = []

    async def fetch_url(self, session, url):
        """Fetch single URL asynchronously"""
        try: # TODO: Get HTML content, Parse with BeautifulSoup,
         Extract required data
            async with session.get(url) as response:
                pass
        except Exception as e:
            print(f"Error fetching {url}: {e}")
            return None

    async def scrape_all(self): # TODO: Gather all results concurrently
        async with aiohttp.ClientSession() as session:
            tasks = [self.fetch_url(session, url) for url in self.urls]
            pass

    async def save_results(self, filename): # TODO: Write results to file
        pass

# asyncio.run(scraper.scrape_all())
```

Project: Concurrent File Processor

```python
import asyncio
import aiofiles
from pathlib import Path
```

```python
class ConcurrentFileProcessor:
    def __init__(self, file_paths):
        self.file_paths = file_paths
        self.progress = {}

    async def process_file(self, file_path):
        try: # TODO: Read file in chunks, Process, Update progress
            async with aiofiles.open(file_path, 'r') as f:
                pass
        except Exception as e:
            print(f"Error processing {file_path}: {e}")

    async def track_progress(self):
        while True: # TODO: Display progress for all files
            await asyncio.sleep(0.5) # TODO: Break when all complete
            pass

    async def process_all(self): # TODO: Run all tasks concurrently
        tasks = [self.process_file(fp) for fp in self.file_paths]
        progress_task = asyncio.create_task(self.track_progress())
        pass

# processor = ConcurrentFileProcessor(file_list)
# asyncio.run(processor.process_all())
```

Conclusion & Best Practices

- Follow PEP 8 style guide
- Write readable, self-documenting code
- Use type hints for clarity
- Test your code thoroughly
- Handle exceptions appropriately
- Document with docstrings
- Use virtual environments
- Version control with Git

Further Learning Resources

- Home page – http://www.python.org
- Wiki – http://wiki.python.org/
- Packages – https://pypi.python.org/pypi
- Projects at http://sourceforge.net and github.org
- Real Python tutorials
- Python Package Index (PyPI)
- Stack Overflow community
- GitHub open source projects
- Python Enhancement Proposals (PEPs)
- Online courses and certifications

Data Structures and Algorithms

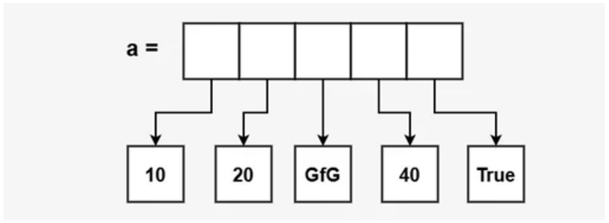
Data Structures and Algorithms

Course Overview

- Fundamental Data Structures
- Core Algorithms & Techniques
- Time & Space Complexity Analysis
- 8 Essential LeetCode Patterns
- Real-world Problem Solving
- Best Practices & Optimization

Arrays & Lists - Fundamentals

- Dynamic arrays in Python (lists)
- O(1) access by index
- O(n) insertion/deletion (worst case)
- Contiguous memory allocation
- Common operations: append, insert, remove



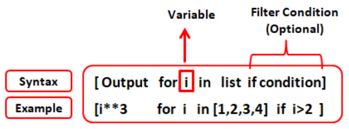
(Ref: https://www.geeksforgeeks.org/python/python-lists/)

```python
# List creation
arr = [1, 2, 3, 4, 5]

# Access - O(1)
print(arr[0])  # 1

# Append - O(1) amortized
arr.append(6)

# Insert - O(n)
arr.insert(0, 0)

# Delete - O(n)
arr.remove(3)

# Slicing
sub = arr[1:4]
```

List Comprehensions & Advanced Operations

- Pythonic way to create lists
- More readable and faster
- Supports filtering and mapping
- Can handle nested structures
- Essential for interview coding

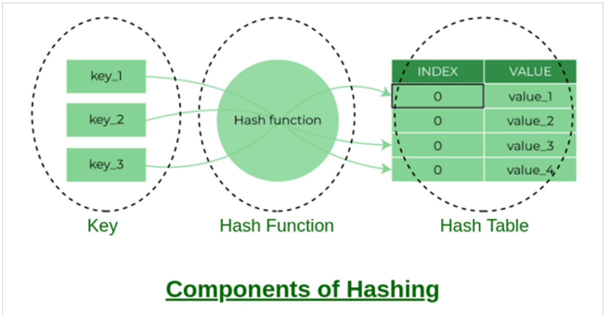
```
# Basic comprehension
squares = [x**2 for x in range(10)]

# With condition
evens = [x for x in range(20) if x % 2 == 0]

# Nested comprehension
matrix = [[i*j for j in range(3)]
          for i in range(3)]

# Map-like operation
names = ['alice', 'bob']
upper = [n.upper() for n in names]

# Filter and transform
nums = [1, -2, 3, -4]
positive_sq = [x**2 for x in nums if x > 0]
```


Variable
Filter Condition (Optional)
Syntax
Example
[Output for i in list if condition]
[i**3 for i in [1,2,3,4] if i>2]

(Ref: https://www.listendata.com/2019/07/python-list-comprehension-with-examples.html)

Hash Tables & Dictionaries

- O(1) average lookup, insert, delete
- Key-value pairs storage
- Hash function maps keys to indices
- Handles collisions internally
- Most versatile data structure



(Ref: https://www.listendata.com/2019/07/python-list-comprehension-with-examples.html)

```
# Dictionary creation
d = {'a': 1, 'b': 2}

# Access - O(1)
val = d.get('a', 0)  # Safe access

# Insert/Update - O(1)
d['c'] = 3

# Delete - O(1)
del d['b']

# Check existence
if 'a' in d:
    print(d['a'])

# Iterate
for key, val in d.items():
    print(f"{key}: {val}")

# defaultdict for cleaner code
from collections import defaultdict
count = defaultdict(int)
count['apple'] += 1
```

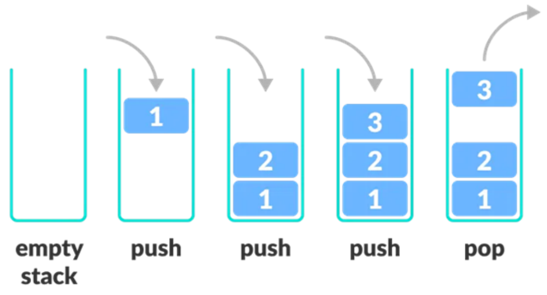
Sets - Unique Collections

- Unordered collection of unique items
- O(1) membership testing
- Efficient set operations (union, intersection)
- No duplicate elements
- Useful for deduplication

```
# Set creation
s = {1, 2, 3, 4}
s = set([1, 2, 2, 3])  # {1, 2, 3}

# Add - O(1)
s.add(5)

# Remove - O(1)
s.discard(2)  # Safe remove
s.remove(3)   # Raises error if not found

# Membership - O(1)
if 4 in s:
    print("Found")

# 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}
```

Stacks - LIFO Structure

- Last In First Out (LIFO) principle
- O(1) push and pop operations
- Used in recursion, parsing, backtracking
- Python list works perfectly as stack
- Common uses: expression evaluation, DFS

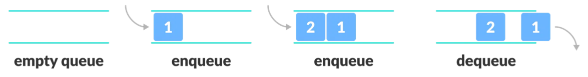


(Ref: https://www.programiz.com/dsa/stack)

```
# Stack using list
stack = []

# Push - O(1)
stack.append(1)
stack.append(2)
stack.append(3)

# Pop - O(1)
top = stack.pop()  # 3

# Peek
if stack:
    top = stack[-1]  # Don't remove

# Check if empty
is_empty = len(stack) == 0

# Valid Parentheses example
def isValid(s):
    stack = []
    pairs = {'(':')', '[':']', '{':'}'}
    for char in s:
        if char in pairs:
            stack.append(char)
        elif not stack or pairs[stack.pop()] != char:
            return False
    return len(stack) == 0
```

Queues - FIFO Structure

- First In First Out (FIFO) principle
- O(1) enqueue and dequeue with deque
- Used in BFS, scheduling, buffering
- collections.deque is optimal
- Supports operations at both ends



(Ref: https://www.programiz.com/dsa/queue)

```
from collections import deque

# Queue using deque
queue = deque()

# Enqueue - O(1)
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue - O(1)
first = queue.popleft()  # 1

# Peek front
if queue:
    front = queue[0]

# Peek back
if queue:
    back = queue[-1]

# BFS example
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
    return visited
```

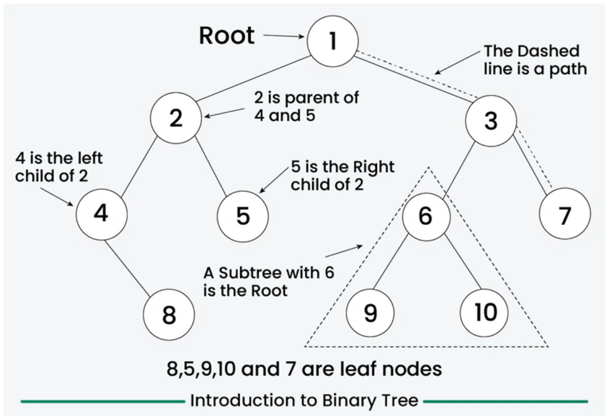
Linked Lists - Dynamic Structure

- Nodes connected via pointers
- O(1) insertion/deletion at known position
- O(n) search and access
- No contiguous memory required
- Types: singly, doubly, circular

```
# Node definition
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# Create linked list
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)

# Traverse
curr = head
while curr:
    print(curr.val)
    curr = curr.next

# Reverse linked list
def reverseList(head):
    prev = None
    curr = head
    while curr:
        next_temp = curr.next
        curr.next = prev
        prev = curr
        curr = next_temp
    return prev
```

Binary Trees - Hierarchical Structure

- Each node has at most 2 children
- Root, internal nodes, and leaves
- Traversals: inorder, preorder, postorder
- Height: O(log n) balanced, O(n) worst
- Foundation for BST, heaps, etc.



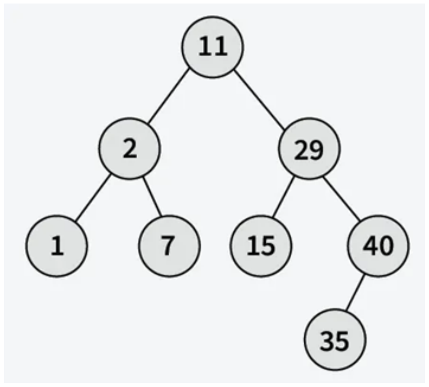
Root
The Dashed line is a path
2 is parent of 4 and 5
4 is the left child of 2
5 is the Right child of 2
A Subtree with 6 is the Root
1
2
3
4
5
6
7
8
9
10
8,5,9,10 and 7 are leaf nodes
Introduction to Binary Tree

(Ref: https://www.geeksforgeeks.org/python/binary-tree-in-python/)

```
# Tree node definition
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Inorder traversal (Left-Root-Right)
def inorder(root):
    if not root:
        return []
    return (inorder(root.left) +
            [root.val] +
            inorder(root.right))

# Level order traversal (BFS)
def levelOrder(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

Binary Search Trees (BST)

- Left subtree ¡ root ¡ right subtree
- O(log n) search, insert, delete (balanced)
- O(n) worst case (skewed tree)
- Inorder traversal gives sorted order
- Self-balancing variants: AVL, Red-Black

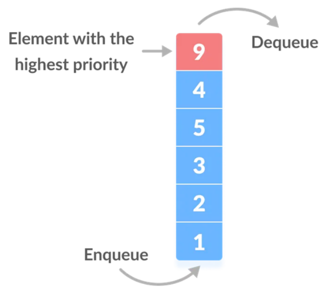


(Ref: https://www.geeksforgeeks.org/dsa/introduction-to-binary-search-tree/)

```
# Search in BST
def searchBST(root, val):
    if not root or root.val == val:
        return root
    if val < root.val:
        return searchBST(root.left, val)
    return searchBST(root.right, val)

# Insert into BST
def insertBST(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insertBST(root.left, val)
    else:
        root.right = insertBST(root.right, val)
    return root

# Validate BST
def isValidBST(root, min_val=float('-inf'),
               max_val=float('inf')):
    if not root:
        return True
    if not (min_val < root.val < max_val):
        return False
    return (isValidBST(root.left, min_val, root.val) and
            isValidBST(root.right, root.val, max_val))
```

Heaps & Priority Queues

- Complete binary tree structure
- Min-heap: parent $\leq$ children
- Max-heap: parent $\geq$ children
- O(log n) insert and extract-min/max
- O(1) peek at min/max element
- Used in: Dijkstra, heap sort, scheduling

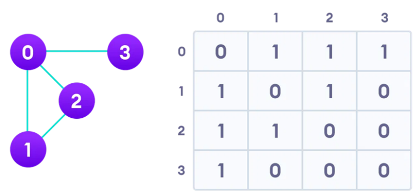


(Ref: https://www.geeksforgeeks.org/dsa/introduction-to-binary-search-tree/)

```
import heapq

# Min heap (default in Python)
heap = []
heapq.heappush(heap, 5)
heapq.heappush(heap, 2)
heapq.heappush(heap, 8)

# Extract min - O(log n)
min_val = heapq.heappop(heap)  # 2

# Peek min - O(1)
if heap:
    min_val = heap[0]

# Heapify array - O(n)
arr = [5, 2, 8, 1, 9]
heapq.heapify(arr)

# Max heap (negate values)
max_heap = []
heapq.heappush(max_heap, -5)
max_val = -heapq.heappop(max_heap)

# K largest elements
def kLargest(nums, k):
    return heapq.nlargest(k, nums)
```

Graphs - Representation

- Vertices (nodes) and edges (connections)
- Directed vs undirected
- Weighted vs unweighted
- Adjacency list: O(V+E) space
- Adjacency matrix: O(V^2) space
- Most problems use adjacency list



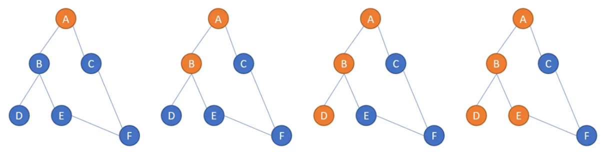
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

(Ref: https://www.programiz.com/dsa/graph)

```
from collections import defaultdict

# Adjacency list representation
graph = defaultdict(list)

# Add edge (undirected)
def add_edge(u, v):
    graph[u].append(v)
    graph[v].append(u)

# Add weighted edge
weighted_graph = defaultdict(list)
def add_weighted_edge(u, v, w):
    weighted_graph[u].append((v, w))
    weighted_graph[v].append((u, w))

# Example graph
add_edge(0, 1)
add_edge(0, 2)
add_edge(1, 2)
add_edge(2, 3)

# Adjacency matrix (for dense graphs)
n = 4
matrix = [[0]*n for _ in range(n)]
matrix[0][1] = matrix[1][0] = 1
```

Graph Traversals - DFS

- DFS: Depth First Search (stack/recursion)
- BFS: Breadth First Search (queue)
- $O(V + E)$ time complexity
- DFS for: cycles, topological sort, paths

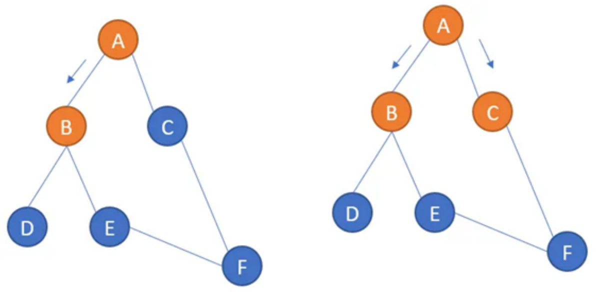


(Ref: DFS https://medium.com/nerd-for-tech/graph-traversal-in-python-depth-first-search-dfs-ce791f48af5b)

```
# DFS - Recursive
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# DFS - Iterative
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node)
            stack.extend(graph[node])
    return visited
```

Graph Traversals - BFS

- BFS: Breadth First Search (queue)
- $O(V + E)$ time complexity
- BFS for: shortest path, level-order

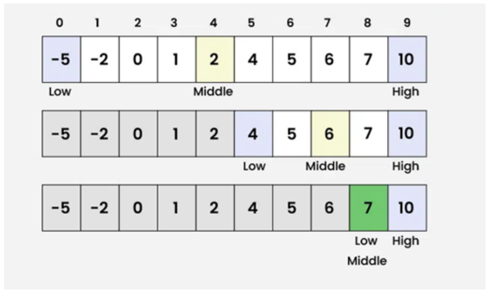


(Ref: BFS https://medium.com/nerd-for-tech/graph-traversal-in-python-breadth-first-search-bfs-b6cff138d516)

```
# BFS
def bfs(graph, start):
    visited = set([start])
    queue = deque([start])
    while queue:
        node = queue.popleft()
        print(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

Binary Search Algorithm

- O(log n) search in sorted array
- Divide and conquer approach
- Eliminates half of elements each step
- Must have sorted input
- Template applicable to many problems
- Find exact match or insertion point

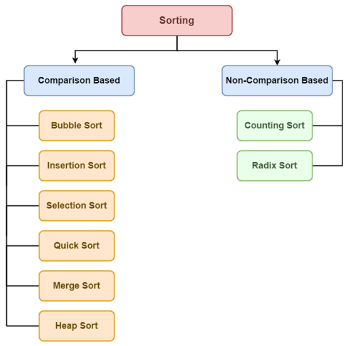


(Ref: BFS https://medium.com/nerd-for-tech/graph-traversal-in-python-breadth-first-search-bfs-b6cff138d516)

```
# Standard binary search
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target: return mid
        elif arr[mid] < target:
            left = mid + 1
        else: right = mid - 1
    return -1

# Find insertion position
def searchInsert(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < target: left = mid + 1
        else: right = mid
    return left

# Find first occurrence
def findFirst(arr, target):
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1
        elif arr[mid] < target: left = mid + 1
        else: right = mid - 1
    return result
```

Sorting Algorithms - Comparison Based

- Quick Sort: O(n log n) avg, O(n²) worst
- Merge Sort: O(n log n) guaranteed
- Heap Sort: O(n log n), in-place
- Bubble/Insertion: O(n²), simple
- Python's sorted() uses Timsort

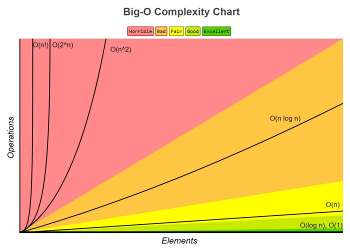


(Ref: https://www.geeksforgeeks.org/dsa/introduction-to-sorting-algorithm/)

```
# Quick Sort
def quicksort(arr):
    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 quicksort(left) + middle + quicksort(right)

# Merge Sort
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(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
```

Big O Notation - Time & Space Complexity

- Measures algorithm efficiency
- $O(1)$: Constant time
- $O(logn)$: Logarithmic (binary search)
- $O(n)$: Linear (single loop)
- $O(nlogn)$: Linearithmic (merge sort)
- $O(n^2)$: Quadratic (nested loops)
- $O(2^n)$: Exponential (recursion)
- Drop constants and lower order terms

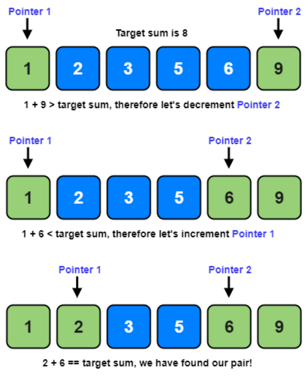


(Ref: https://medium.com/data-science/understanding-time-complexity-with-python-examples-2bda6e8158a7)

```
def get_first(arr): # O(1) - Constant
    return arr[0]

def find_max(arr): # O(n) - Linear
    max_val = arr[0]
    for num in arr:
        if num > max_val: max_val = num
    return max_val

def bubble_sort(arr): # O(n^2) - Quadratic
    n = len(arr)
    for i in range(n):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# O(log n) - Logarithmic
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

def fibonacci(n): # O(2^n) - Exponential
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Pattern 1: Two Pointers Technique

- Use two pointers to iterate array
- Opposite direction or same direction
- O(n) time, O(1) space typically
- Common in: sorted arrays, palindromes
- Example: Two Sum II, Container With Most Water

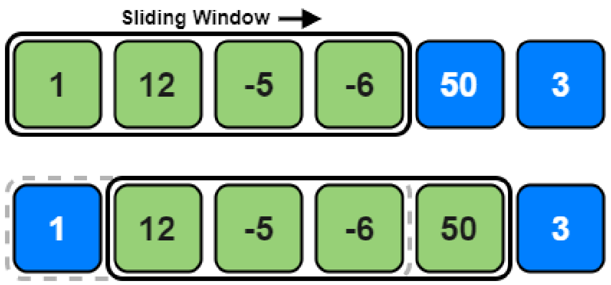

Pointer 1
Pointer 2
Target sum is 8
1 + 9 > target sum, therefore let's decrement Pointer 2
1 + 6 < target sum, therefore let's increment Pointer 1
2 + 6 == target sum, we have found our pair!

(Ref: https://emre.me/coding-patterns/two-pointers/)

```
# Problem: Two Sum II (sorted array)
# Given sorted array, find two numbers
# that add up to target
def twoSum(numbers, target):
    left, right = 0, len(numbers) − 1
    while left < right:
        curr_sum = numbers[left] + numbers[right]
        if curr_sum == target:
            return [left + 1, right + 1]
        elif curr_sum < target:
            left += 1
        else:
            right −= 1
    return []

# Problem: Valid Palindrome
def isPalindrome(s):
    left, right = 0, len(s) − 1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right −= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right −= 1
    return True
```

Pattern 2: Sliding Window

- Fixed or variable size window
- Maintains window state efficiently
- O(n) time for array/string problems
- Common in: subarray/substring problems
- Example: Max Sum Subarray, Longest Substring

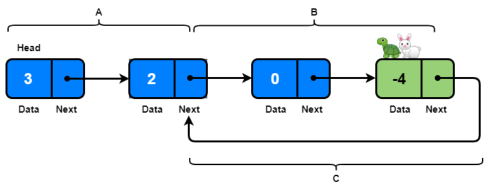


(Ref: https://emre.me/coding-patterns/sliding-window/)

```
# Problem: Max Sum of Subarray of Size K
def maxSumSubarray(arr, k):
    if len(arr) < k:
        return 0

    # Calculate sum of first window
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide window
    for i in range(k, len(arr)):
        window_sum = window_sum − arr[i−k] + arr[i]
        max_sum = max(max_sum, window_sum)

    return max_sum

# Problem: Longest Substring Without Repeating
def lengthOfLongestSubstring(s):
    char_set = set()
    left = 0
    max_len = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_len = max(max_len, right − left + 1)

    return max_len
```

Pattern 3: Fast & Slow Pointers (Floyd's Cycle)

- Two pointers moving at different speeds
- Detects cycles in linked lists
- Finds middle element efficiently
- O(n) time, O(1) space
- Example: Linked List Cycle, Happy Number

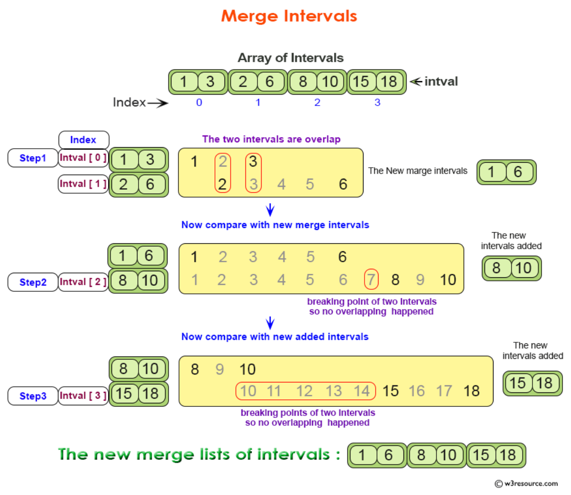


(Ref: https://emre.me/coding-patterns/fast-slow-pointers/)

```
def hasCycle(head): # Problem: Detect Cycle in Linked List
    if not head or not head.next: return False
    slow = head
    fast = head.next
    while slow != fast:
        if not fast or not fast.next: return False
        slow = slow.next
        fast = fast.next.next
    return True

def middleNode(head): # Problem: Find Middle of Linked List
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

def isHappy(n): # Problem: Happy Number
    def get_next(num):
        total = 0
        while num > 0:
            digit = num % 10
            total += digit ** 2
            num //= 10
        return total
    slow = n
    fast = get_next(n)
    while fast != 1 and slow != fast:
        slow = get_next(slow)
        fast = get_next(get_next(fast))
    return fast == 1
```

Pattern 4: Merge Intervals

- Sort intervals by start time
- Merge overlapping intervals
- O(n log n) time due to sorting
- Common in: scheduling, ranges
- Example: Merge Intervals, Insert Interval


Merge Intervals
Array of Intervals
intval
Index
Step1
Intval [0]
Intval [1]
The two intervals are overlap
The New marge intervals
Now compare with new merge intervals
Step2
Intval [2]
The new intervals added
breaking point of two intervals so no overlapping happened
Now compare with new added intervals
Step3
Intval [3]
breaking points of two intervals so no overlapping happened
The new merge lists of intervals :
© w3resource.com

(Ref: https://www.w3resource.com/data-structures-and-algorithms/array/dsa-merge-intervals.php)

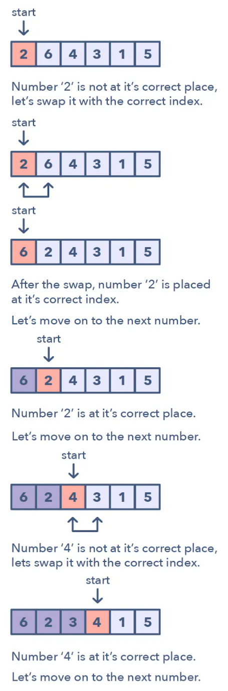
Pattern 4: Merge Intervals

```python
# Problem: Merge Overlapping Intervals
def merge(intervals):
    if not intervals:
        return []

    # Sort by start time
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for current in intervals[1:]:
        last = merged[−1]
        if current[0] <= last[1]:
            # Overlapping, merge
            last[1] = max(last[1], current[1])
        else:
            # Non−overlapping, add
            merged.append(current)

    return merged
```

```python
# Problem: Insert Interval
def insert(intervals, newInterval):
    result = []
    i = 0
    n = len(intervals)

    # Add all intervals before newInterval
    while i < n and intervals[i][1] < newInterval[0]:
        result.append(intervals[i])
        i += 1

    # Merge overlapping intervals
    while i < n and intervals[i][0] <= newInterval[1]:
        newInterval[0] = min(newInterval[0], intervals[i][0])
        newInterval[1] = max(newInterval[1], intervals[i][1])
        i += 1
    result.append(newInterval)

    # Add remaining intervals
    result.extend(intervals[i:])
    return result
```

Pattern 5: Cyclic Sort

- For arrays with numbers in range [1, n]
- Place each number at its correct index
- O(n) time, O(1) space
- Finds missing/duplicate numbers
- Example: Find Missing Number, First Missing Positive


start
2 6 4 3 1 5
Number '2' is not at it's correct place, let's swap it with the correct index.
start
2 6 4 3 1 5
start
6 2 4 3 1 5
After the swap, number '2' is placed at it's correct index.
Let's move on to the next number.
start
6 2 4 3 1 5
Number '2' is at it's correct place.
Let's move on to the next number.
start
6 2 4 3 1 5
Number '4' is not at it's correct place, lets swap it with the correct index.
start
6 2 3 4 1 5
Number '4' is at it's correct place.
Let's move on to the next number.

(Ref: paulonteri/data-structures-and-algorithms)

Pattern 5: Cyclic Sort

```python
# Problem: Find Missing Number (0 to n)
def missingNumber(nums):
    i = 0
    n = len(nums)

    while i < n:
        correct_idx = nums[i]
        if nums[i] < n and nums[i] != nums[correct_idx]:
            # Swap to correct position
            nums[i], nums[correct_idx] = nums[correct_idx], nums[i]
        else:
            i += 1

    # Find first missing
    for i in range(n):
        if nums[i] != i:
            return i
    return n
```

```python
# Problem: Find All Duplicates (1 to n)
def findDuplicates(nums):
    i = 0
    while i < len(nums):
        correct_idx = nums[i] − 1
        if nums[i] != nums[correct_idx]:
            nums[i], nums[correct_idx] = nums[correct_idx], nums[i]
        else:
            i += 1

    duplicates = []
    for i in range(len(nums)):
```

```python
        if nums[i] != i + 1:
            duplicates.append(nums[i])

    return duplicates
```

Pattern 6: In-place Reversal of Linked List

- Reverse links without extra space
- Three pointers: prev, curr, next
- O(n) time, O(1) space
- Fundamental for many LL problems
- Example: Reverse Linked List, Reverse in K-Group

```python
# Problem: Reverse Linked List
def reverseList(head):
    prev = None
    curr = head

    while curr:
        next_temp = curr.next
        curr.next = prev
        prev = curr
        curr = next_temp

    return prev
```

```python
# Problem: Reverse Linked List II
# (from position left to right)
def reverseBetween(head, left, right):
    if not head or left == right:
        return head

    dummy = ListNode(0)
    dummy.next = head
    prev = dummy

    # Move to position before left
    for _ in range(left − 1):
        prev = prev.next

    # Reverse from left to right
    curr = prev.next
    for _ in range(right − left):
        next_temp = curr.next
        curr.next = next_temp.next
        next_temp.next = prev.next
        prev.next = next_temp

    return dummy.next
```

Pattern 7: Tree BFS (Level Order Traversal)

- Use queue for level-by-level traversal
- Process nodes at same level together
- O(n) time, O(w) space (w = max width)
- Example: Level Order, Zigzag Traversal

```python
from collections import deque

# Problem: Binary Tree Level Order Traversal
def levelOrder(root):
    if not root: return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)

        result.append(current_level)

    return result
```

```
# Problem: Zigzag Level Order Traversal
def zigzagLevelOrder(root):
    if not root: return []

    result = []
    queue = deque([root])
    left_to_right = True

    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)

        if not left_to_right: level.reverse()
        result.append(level)
        left_to_right = not left_to_right

    return result
```

Pattern 8: Tree DFS (Depth First Search)

- Recursion or stack for deep traversal
- Preorder, Inorder, Postorder variants
- O(n) time, O(h) space (h = height)
- Common in: path problems, subtree checks
- Example: Max Depth, Path Sum, Diameter

```
# Problem: Maximum Depth of Binary Tree
def maxDepth(root):
    if not root:
        return 0
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)
    return 1 + max(left_depth, right_depth)
```

```
# Problem: Path Sum
def hasPathSum(root, targetSum):
    if not root:
        return False

    if not root.left and not root.right:
        return root.val == targetSum

    targetSum -= root.val
    return (hasPathSum(root.left, targetSum) or
            hasPathSum(root.right, targetSum))
```

```
# Problem: Diameter of Binary Tree
def diameterOfBinaryTree(root):
    diameter = 0

    def height(node):
        nonlocal diameter
        if not node:
            return 0

        left = height(node.left)
        right = height(node.right)
        diameter = max(diameter, left + right)

        return 1 + max(left, right)

    height(root)
    return diameter
```

Dynamic Programming - Foundations

- Break problem into overlapping subproblems
- Store solutions to avoid recomputation
- Top-down (memoization) or bottom-up (tabulation)
- Identify: optimal substructure + overlapping subproblems
- Example: Fibonacci, Climbing Stairs

```
# Fibonacci - Naive (exponential)
def fib_naive(n):
    if n <= 1:
        return n
    return fib_naive(n-1) + fib_naive(n-2)
```

```
# Fibonacci - Memoization (top-down)
def fib_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

# Fibonacci - Tabulation (bottom-up)
def fib_tab(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Fibonacci - Space Optimized
def fib_optimal(n):
    if n <= 1:
        return n
    prev, curr = 0, 1
    for _ in range(2, n + 1):
        prev, curr = curr, prev + curr
    return curr
```

Dynamic Programming - Classic Problems

- Coin Change: min coins for amount
- Longest Common Subsequence (LCS)
- 0/1 Knapsack Problem
- House Robber variations
- Edit Distance

```
# Problem: Coin Change
def coinChange(coins, amount):
    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
```

```
# Problem: Longest Common Subsequence
def longestCommonSubsequence(text1, text2):
    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]

# Problem: House Robber
def rob(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    prev2, prev1 = 0, 0
    for num in nums:
        temp = prev1
        prev1 = max(prev2 + num, prev1)
        prev2 = temp

    return prev1
```

Backtracking - Exhaustive Search

- Try all possibilities recursively
- Prune invalid paths early
- Build solution incrementally
- Undo choices (backtrack) when needed
- Example: Permutations, Subsets, N-Queens

```
# Problem: Generate All Subsets
def subsets(nums):
    result = []

    def backtrack(start, path):
        result.append(path[:])

        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()  # Backtrack

    backtrack(0, [])
    return result
```

```
# Problem: Permutations
def permute(nums):
    result = []

    def backtrack(path, remaining):
        if not remaining:
            result.append(path[:])
            return

        for i in range(len(remaining)):
            backtrack(path + [remaining[i]],
                     remaining[:i] + remaining[i+1:])

    backtrack([], nums)
    return result

# Problem: Combination Sum
def combinationSum(candidates, target):
    result = []
    def backtrack(start, path, total):
        if total == target:
            result.append(path[:])
            return
        if total > target: return

        for i in range(start, len(candidates)):
            path.append(candidates[i])
            backtrack(i, path, total + candidates[i])
            path.pop()

    backtrack(0, [], 0)
    return result
```

Advanced Topics & Techniques

- Trie (Prefix Tree) for string problems
- Union-Find for connected components
- Topological Sort for DAGs
- Dijkstra's & Bellman-Ford for shortest paths
- Bit Manipulation tricks
- Greedy algorithms
- Monotonic Stack/Queue

Conclusion & Best Practices

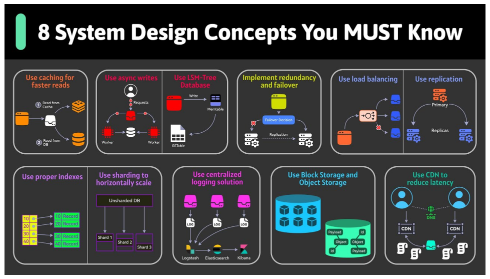
- Start with brute force, then optimize
- Identify pattern before coding
- Draw diagrams for complex problems
- Test with edge cases
- Analyze time & space complexity
- Practice consistently on LeetCode
- Master the 8 common patterns
- Review and learn from solutions

System Design

System Design

Why System Design?

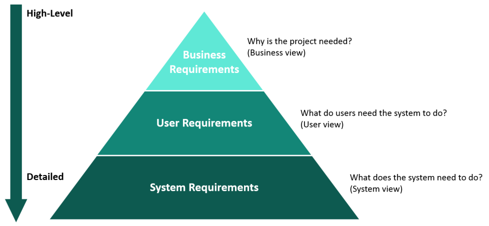
- Comprehensive guide to system design concepts
- From fundamentals to Implementations
- Real-world scalability challenges


8 System Design Concepts You MUST Know
Use caching for faster reads
Use async writes
Use LSM-Tree Database
Implement redundancy and failover
Use load balancing
Use replication
Use proper indexes
Use sharding to horizontally scale
Use centralized logging solution
Use Block Storage and Object Storage
Use CDN to reduce latency

(Ref: https://www.youtube.com/watch?v=BTjxUS_PylA)

Requirements Analysis: Foundation of Design

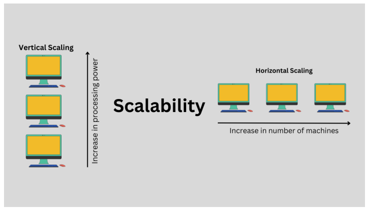
- **Functional Requirements**: What system should do (features, APIs, flows)
- **Non-Functional Requirements**: How system should perform (latency, availability, consistency)
- **Capacity Estimation**: Users, requests/sec, storage needs, bandwidth
- **Constraints**: Budget, timeline, existing infrastructure
- Always clarify ambiguities before diving into design


High-Level
Detailed
Business Requirements
Why is the project needed? (Business view)
User Requirements
What do users need the system to do? (User view)
System Requirements
What does the system need to do? (System view)

(Ref: http://www.crvs-dgb.org/en/activities/analysis-and-design/8-define-system-requirements/)

Scalability: Growing Your System

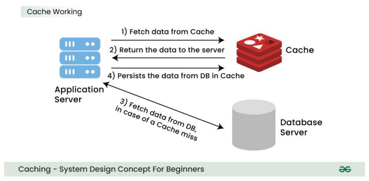
- **Vertical Scaling**: Add more resources (CPU, RAM) to single machine - simple but limited
- **Horizontal Scaling**: Add more machines - unlimited growth, requires coordination
- **Stateless Services**: Enable easy horizontal scaling by removing server-side session state
- **Sharding/Partitioning**: Split data across multiple databases
- **Microservices**: Decompose monolith into independent, scalable services



(Ref: https://www.linkedin.com/pulse/system-design-horizontal-scaling-vs-vertical-harsh-kumar-sharma-jadpf/)

Caching: Speed Through Memory

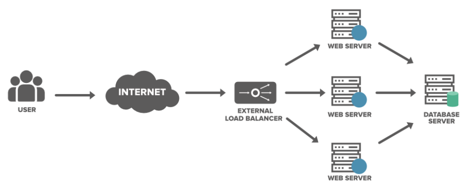
- **Cache-Aside**: App checks cache first, loads from DB on miss, updates cache
- **Write-Through**: Write to cache and DB simultaneously - consistent but slower writes
- **Write-Back**: Write to cache only, async persist to DB - fast but risk data loss
- **Eviction Policies**: LRU, LFU, FIFO for managing limited cache space
- **CDN**: Cache static content geographically close to users
- Tools: Redis, Memcached, Varnish


Cache Working
1) Fetch data from Cache
2) Return the data to the server
4) Persists the data from DB in Cache
Cache
Application Server
3) Fetch data from DB, in case of a Cache miss
Database Server
Caching - System Design Concept For Beginners

(Ref: https://www.geeksforgeeks.org/system-design/caching-system-design-concept-for-beginners/)

Load Balancing: Distributing Traffic

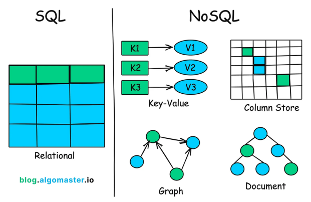
- **Round Robin**: Distribute requests equally across servers
- **Least Connections**: Route to server with fewest active connections
- **IP Hash**: Consistent routing based on client IP for session affinity
- **Weighted**: Assign more traffic to powerful servers
- **Layer 4 vs Layer 7**: Network layer (TCP/UDP) vs Application layer (HTTP)
- Tools: Nginx, HAProxy, AWS ELB, Cloud Load Balancers



(Ref: https://www.geeksforgeeks.org/computer-networks/load-balancing-approach-in-distributed-system/)

Database Design: Choosing the Right Store

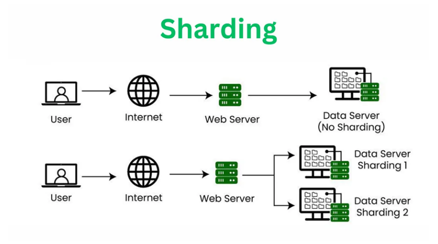
- **SQL (RDBMS)**: ACID, complex queries, relationships - MySQL
- **NoSQL Key-Value**: High performance, simple lookups - Redis
- **NoSQL Document**: Flexible schema, JSON - MongoDB, Couchbase
- **NoSQL Column-Family**: Time-series, analytics - Cassandra, HBase
- **NoSQL Graph**: Relationships, social networks - Neo4j
- Choose based on: consistency needs, query patterns, scale requirements



(Ref: https://www.geeksforgeeks.org/computer-networks/load-balancing-approach-in-distributed-system/)

Database Scalability: Handling Growth

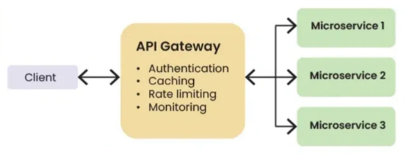
- **Replication**: Master-slave for read scaling, Multi-master for write scaling
- **Sharding/Partitioning**: Horizontal split by key range, hash, or geography
- **Indexing**: B-trees for fast lookups, trade-off with write speed
- **Denormalization**: Duplicate data to avoid expensive joins
- **Connection Pooling**: Reuse database connections to reduce overhead
- CAP Theorem: Choose 2 of Consistency, Availability, Partition Tolerance



(Ref: https://medium.com/@hksrise/understanding-sharding-in-system-design-a-key-to-scalability-214ad71784c4)

API Design: Building Interfaces

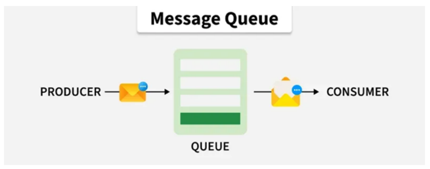
- **REST**: Resource-based, HTTP methods (GET, POST, PUT, DELETE), stateless
- **GraphQL**: Flexible queries, single endpoint, client defines response structure
- **gRPC**: High performance, Protocol Buffers, bidirectional streaming
- **Versioning**: URL path (/v1/), header, or query parameter
- **Pagination**: Limit/offset or cursor-based for large datasets
- **Error Handling**: Consistent HTTP status codes and error messages



(Ref: https://www.geeksforgeeks.org/system-design/what-is-api-gateway-system-design/)

Message Queues: Asynchronous Processing

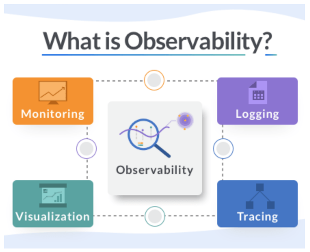
- **Purpose**: Decouple services, handle traffic spikes, reliable delivery
- **Patterns**: Point-to-point (Queue), Publish-Subscribe (Topic)
- **Guarantees**: At-least-once, at-most-once, exactly-once delivery
- **Use Cases**: Order processing, email notifications, video encoding
- **Tools**: Kafka (high throughput), RabbitMQ (flexible), AWS SQS/SNS
- Consider: message ordering, idempotency, dead letter queues



(Ref: https://www.geeksforgeeks.org/system-design/message-queues-system-design/)

Monitoring: Know Your System Health

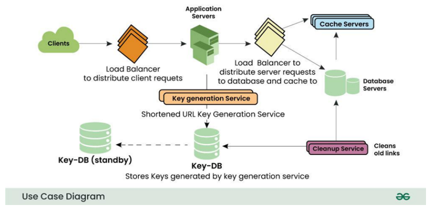
- **Metrics**: CPU, memory, disk, request rate, latency (p50, p99), error rate
- **Logging**: Structured logs, centralized aggregation, log levels
- **Tracing**: Distributed tracing for microservices (Jaeger, Zipkin)
- **Alerting**: Threshold-based, anomaly detection, on-call rotation
- **Dashboards**: Real-time visualization (Grafana, Datadog)
- SLIs, SLOs, SLAs: Define reliability targets



(Ref: https://distributedcomputing.dev/SystemDesign/ObservabilityEngineering)

Project: URL Shortener Design

- **Requirements**: Shorten URLs, redirect, analytics, 100M URLs, low latency
- **Hashing**: Base62 encoding (a-zA-Z0-9) of counter or MD5/SHA hash
- **Database**: Key-value store (shortURL $\rightarrow$ longURL), with index on shortURL
- **Caching**: Redis for hot URLs (80/20 rule), cache-aside pattern
- **API**: POST /shorten {longURL}, GET /:shortURL (302 redirect)
- **Scale**: DB sharding by hash, read replicas, CDN for static assets


Application Servers
Cache Servers
Clients
Load Balancer to distribute client requets
Load Balancer to distribute server requests to database and cache to
Database Servers
Key generation Service
Shortened URL Key Generation Service
Key-DB (standby)
Key-DB
Stores Keys generated by key generation service
Cleanup Service
Cleans old links
Use Case Diagram

(Ref: https://www.geeksforgeeks.org/system-design/system-design-url-shortening-service/)

System Design: Structured Approach

- **Step 1**: Clarify requirements (functional, non-functional, scale)
- **Step 2**: Capacity estimation (storage, bandwidth, QPS)
- **Step 3**: High-level design (draw boxes and arrows)
- **Step 4**: Deep dive into components (database choice, caching)
- **Step 5**: Identify bottlenecks and optimize
- **Step 6**: Discuss trade-offs (consistency vs availability)
- Think aloud, communicate clearly, consider multiple solutions

Design Patterns: Quick Reference

- **Circuit Breaker**: Stop calling failing service, prevent cascading failures
- **CQRS**: Separate read and write models for different optimization
- **Event Sourcing**: Store state changes as events, rebuild state by replay
- **Saga Pattern**: Manage distributed transactions across microservices
- **Bulkhead**: Isolate resources to prevent total system failure
- **Strangler Fig**: Gradually replace legacy system

Conclusion: Mastering System Design

- Practice with real systems: Twitter, Netflix, Uber designs
- Understand trade-offs, not just solutions
- Keep learning: new technologies, patterns emerge constantly
- Resources: "Designing Data-Intensive Applications", System Design Primer (GitHub)
- Mock interviews: Practice communication and thinking process
- Remember: No perfect design, only appropriate design for requirements

References

References

Many publicly available resources are referred for this presentation.

Some of the notable ones are:

- An introduction to Python programming with NumPy, SciPy and Matplotlib/Pylab - Antoine Lefebvre
- Automate the Boring Stuff with Python - Al Sweigart (https://automatetheboringstuff.com/)
- Python Evangelism 101 - Peter Wang (https://conference.scipy.org/scipy2010/slides/lightning/peter_wa

Suggested Learning Resources:

- Book: van Rossum, G. (2011). The Python Tutorial (http://openbookproject.net/)
- PythonTurtle: Logo-like environment for kids and beginners. (http://pythonturtle.org/)
- Book: Learning Python by Mark Lutz & David Ascher. O'Reilly and Associates
- Python Cookbook http://aspn.activestate.com/ASPN/Cookbook/