# OOP and Inheritance

## Natalie Agus

10.020 Data Driven World

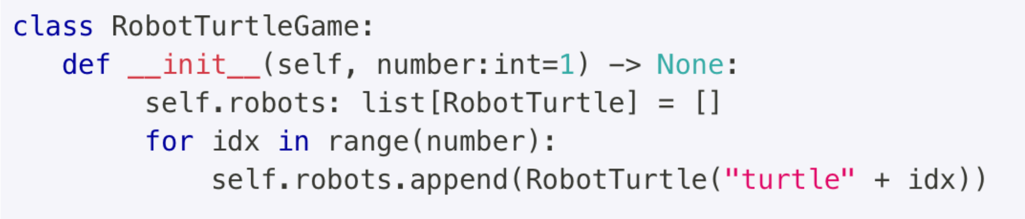
# Learning Objectives

- Use OOP to implement both **data** and **computation**.
- Apply **Stack** and **Queue** for some applications.
- Explain is-a relationship for **inheritance**.
  - Draw UML class diagram for is-a relationship.
- **Inherit** a class to create a child class from a base class.
- **Override** operators to **extend** parent's methods.
- Implement **Deque** data structure as a subclass of Queue
- State the purpose of **Abstract Base Class** & define it
- Implement **Array** and **Linked List** data structure from the same base class.

# OOP Paradigm

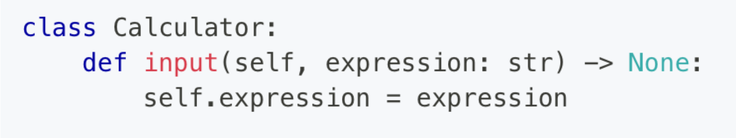
- **Definition**: programming paradigm that organizes code into objects, which **encapsulate** *data* and *behavior*, fostering **modularity**, **reusability**, and easier **maintenance** of software systems.

- Initialization (starting values), then computation (do something to these values)



```python
class RobotTurtleGame:
   def __init__(self, number:int=1) -> None:
        self.robots: list[RobotTurtle] = []
        for idx in range(number):
            self.robots.append(RobotTurtle("turtle" + idx))
```

- We can initialize values during:
    - Instantiation (creation of the object), or
    - Later on (when used) via methods



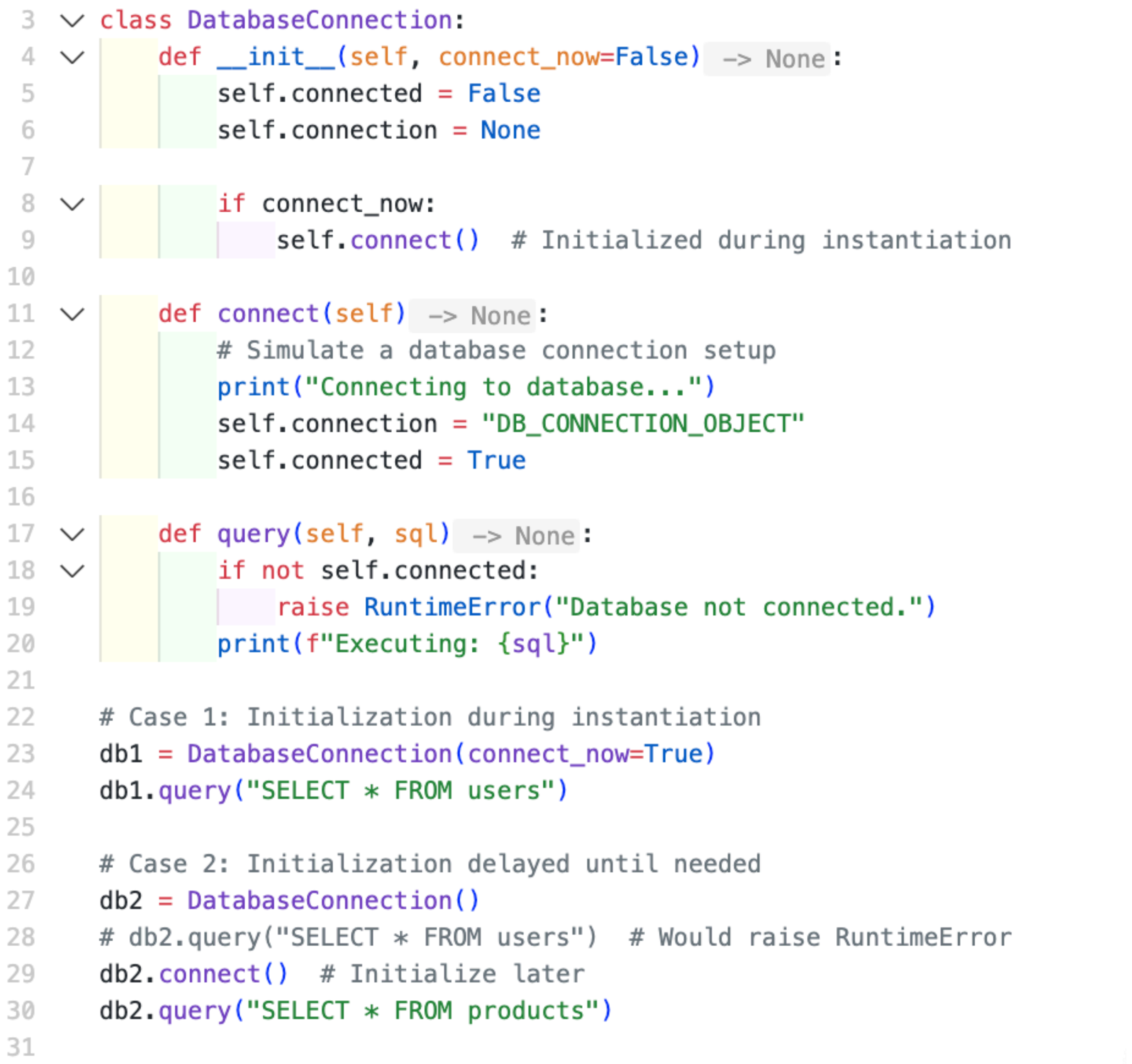
```python
class Calculator:
    def input(self, expression: str) -> None:
        self.expression = expression
```

# Usage

- How to use:
  - **Initialization** (starting values), then
  - **Computation** (do something to these values)
- We can initialize values during:

  1. Instantiation (creation of the object), or

  2. Later on (when used) via methods

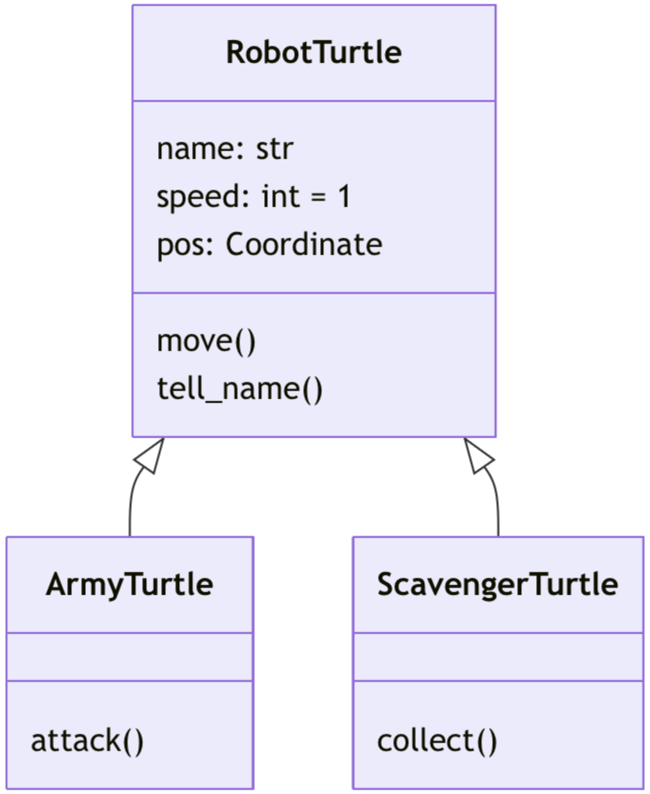


```python
class DatabaseConnection:
    def __init__(self, connect_now=False) -> None:
        self.connected = False
        self.connection = None

        if connect_now:
            self.connect()  # Initialized during instantiation

    def connect(self) -> None:
        # Simulate a database connection setup
        print("Connecting to database...")
        self.connection = "DB_CONNECTION_OBJECT"
        self.connected = True

    def query(self, sql) -> None:
        if not self.connected:
            raise RuntimeError("Database not connected.")
        print(f"Executing: {sql}")

# Case 1: Initialization during instantiation
db1 = DatabaseConnection(connect_now=True)
db1.query("SELECT * FROM users")

# Case 2: Initialization delayed until needed
db2 = DatabaseConnection()
# db2.query("SELECT * FROM users")  # Would raise RuntimeError
db2.connect()  # Initialize later
db2.query("SELECT * FROM products")
```

# Inheritance

## The open-closed principle

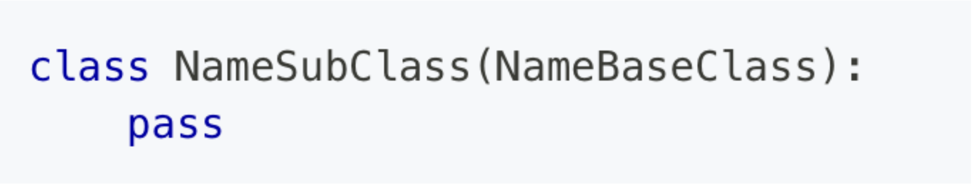
- A class should be **open** for extension but **closed** for modification
- UML diagram:
  - Unified Modelling Language
  - A **standardized** visual language used to describe and design the *structure* and *behavior* of software systems.



# Inheritance

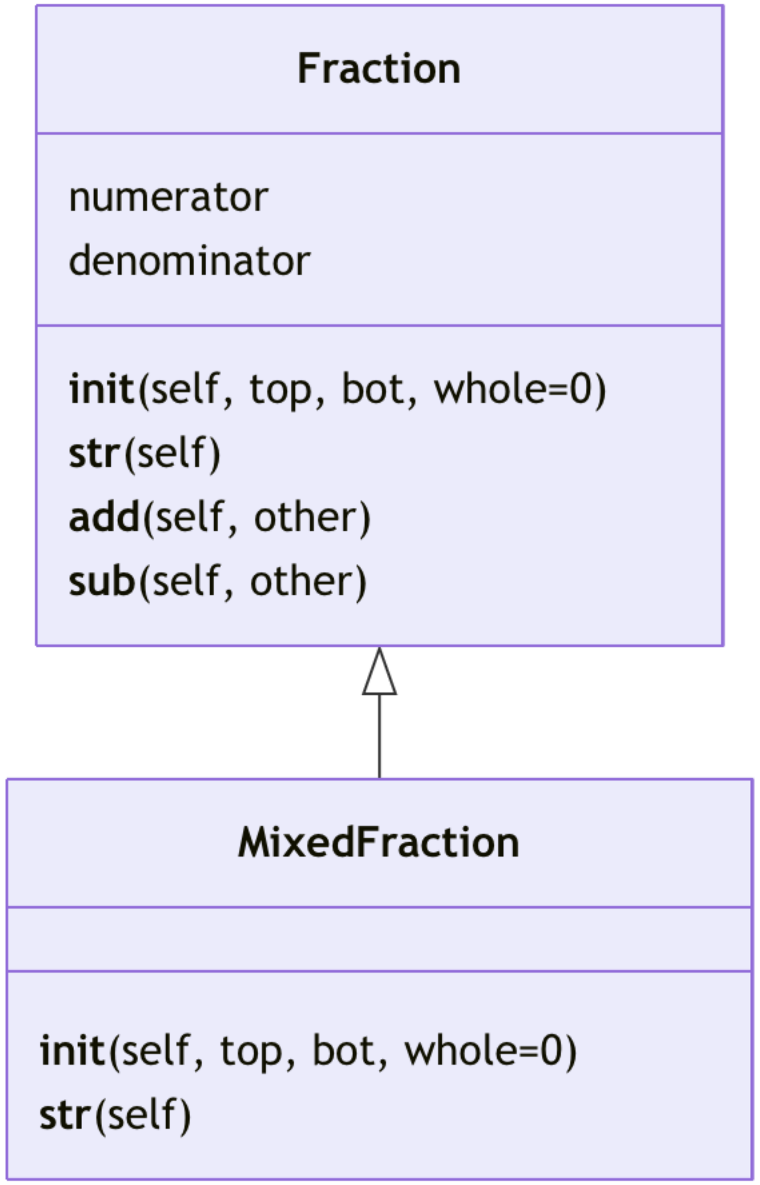
## is-a relationship

- Syntax:



```
class NameSubClass(NameBaseClass):
    pass
```

- You can only inherit **one** parent class at a time
- Inheritance can be **chained**
- **Defines is-a relationship:**
  - isinstance(mixedFraction, Fraction)



Fraction

numerator
denominator

**init**(self, top, bot, whole=0)
**str**(self)
**add**(self, other)
**sub**(self, other)

MixedFraction

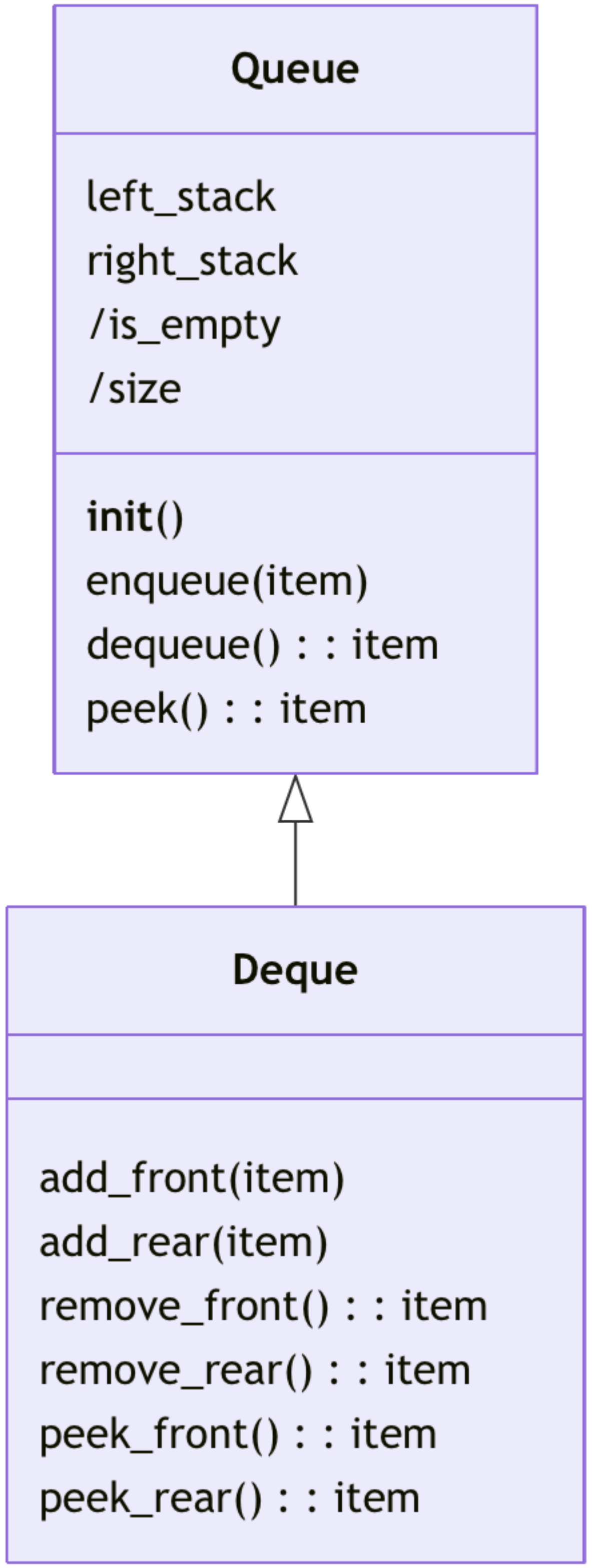
**init**(self, top, bot, whole=0)
**str**(self)

# Data Structures

## Queue & Deque

- **Definition**: a way to **organize** and **store** data so it can be used efficiently

- Common data structures: arrays, linked lists, **stacks**, **queues**, hash tables, sets, trees, **heaps**, graphs, tries, **deques**, priority queues, and matrices.



Queue

left_stack
right_stack
/is_empty
/size

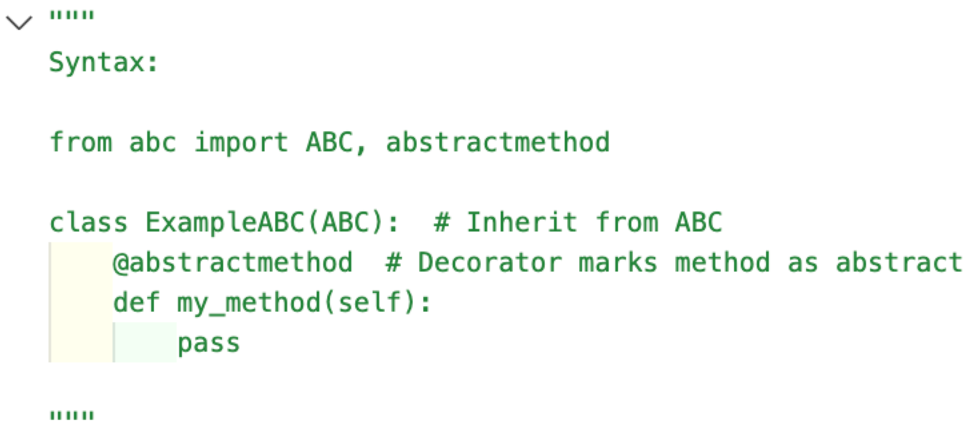
**init**()
enqueue(item)
dequeue() : : item
peek() : : item

Deque

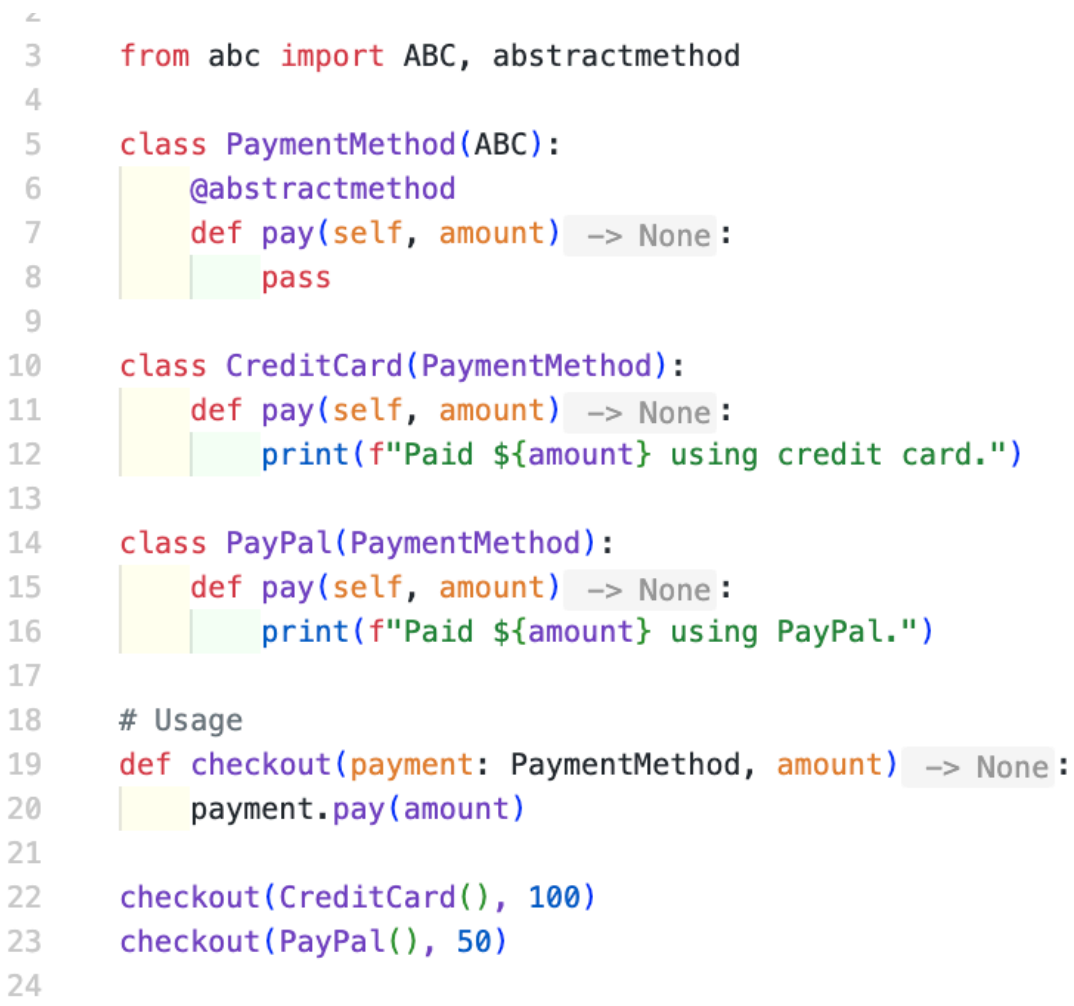
add_front(item)
add_rear(item)
remove_front() : : item
remove_rear() : : item
peek_front() : : item
peek_rear() : : item

# Abstract Base Class

- We want to create a base class (parent) and **enforce** implementation of **certain methods** in the subclasses (child class)
  - Methods are **declared** in parent class (base class)
  - Then **implemented** in child class
  - Should error out if child class did not implement
- Purpose: ***sets rules for what methods subclasses must have.***



```
"""
Syntax:

from abc import ABC, abstractmethod

class ExampleABC(ABC):  # Inherit from ABC
    @abstractmethod  # Decorator marks method as abstract
    def my_method(self):
        pass

"""
```

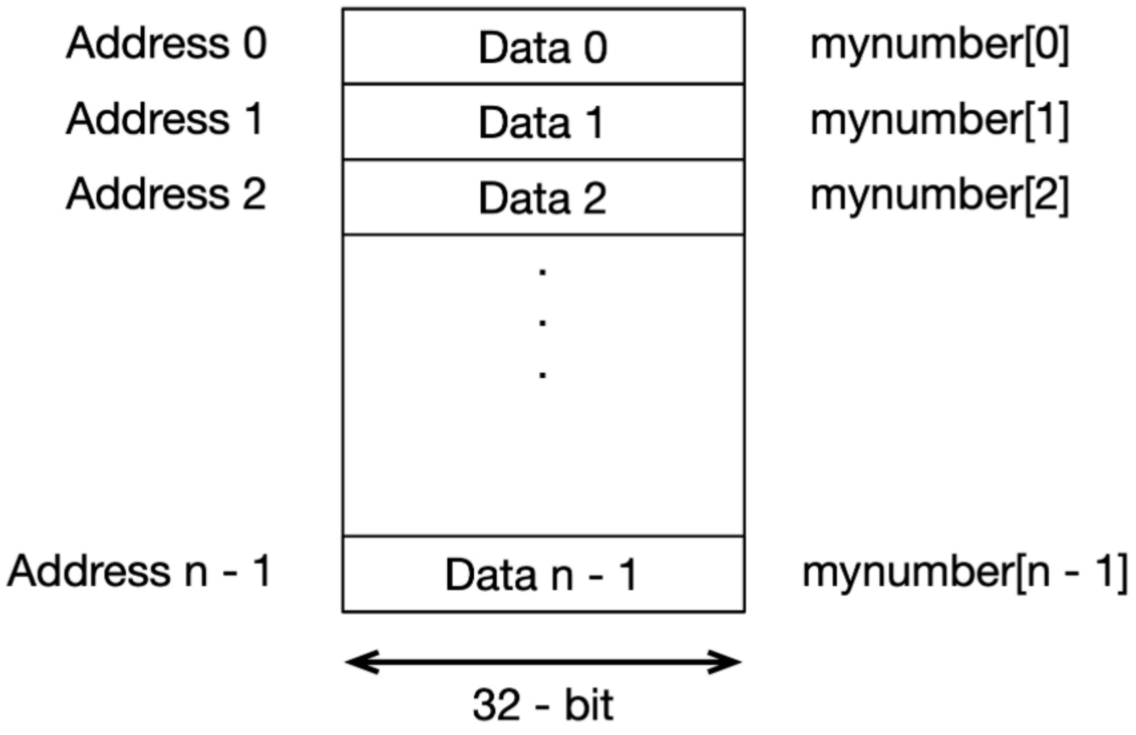


```
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def pay(self, amount) -> None:
        pass

class CreditCard(PaymentMethod):
    def pay(self, amount) -> None:
        print(f"Paid ${amount} using credit card.")

class PayPal(PaymentMethod):
    def pay(self, amount) -> None:
        print(f"Paid ${amount} using PayPal.")

# Usage
def checkout(payment: PaymentMethod, amount) -> None:
    payment.pay(amount)

checkout(CreditCard(), 100)
checkout(PayPal(), 50)
```

# Fixed-Sized Array

## Data Structure

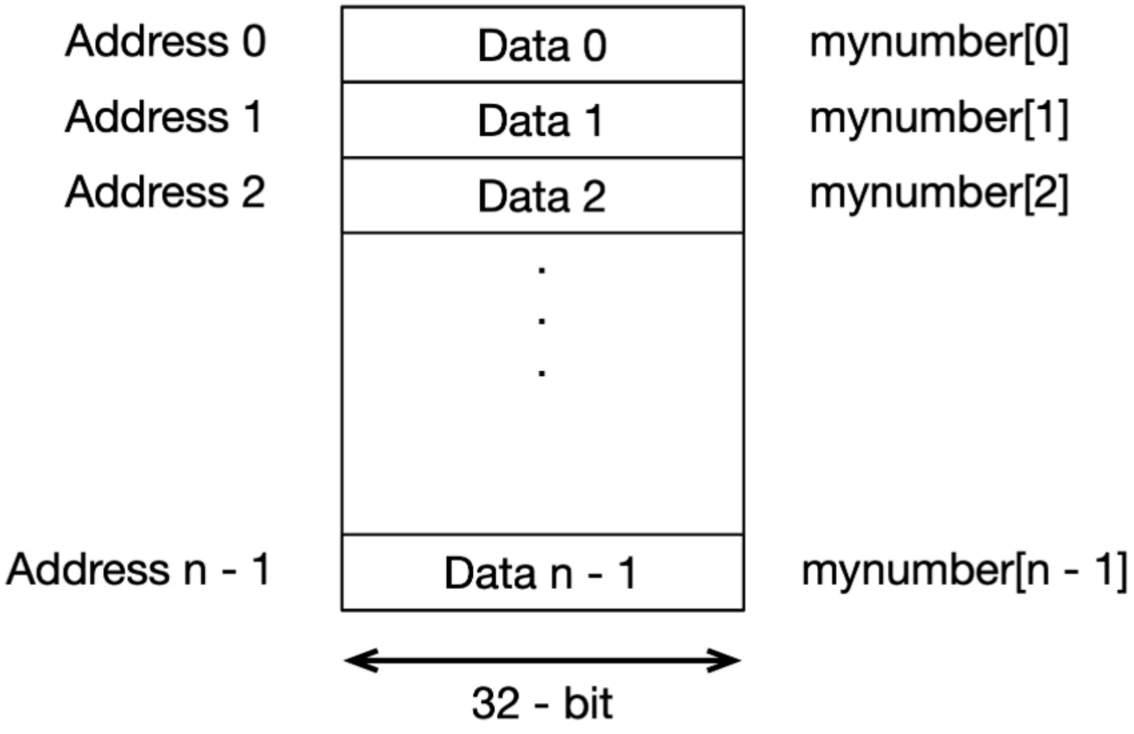
- **Properties:**
  - Declare size in the beginning, gets fixed addresses
  - Elements are in contiguous block of addresses
  - Cannot be appended dynamically in place
  - **Any change in size requires O(N)**
- Python does **NOT** have fixed-size array (C/C++ have). Hence, we are only *simulating* its behavior



# Fixed-Sized Array

## Accessing Element

- Time Complexity: O(1)



Address 0 | Data 0 | mynumber[0]

Address 1 | Data 1 | mynumber[1]

Address 2 | Data 2 | mynumber[2]

.
.
.

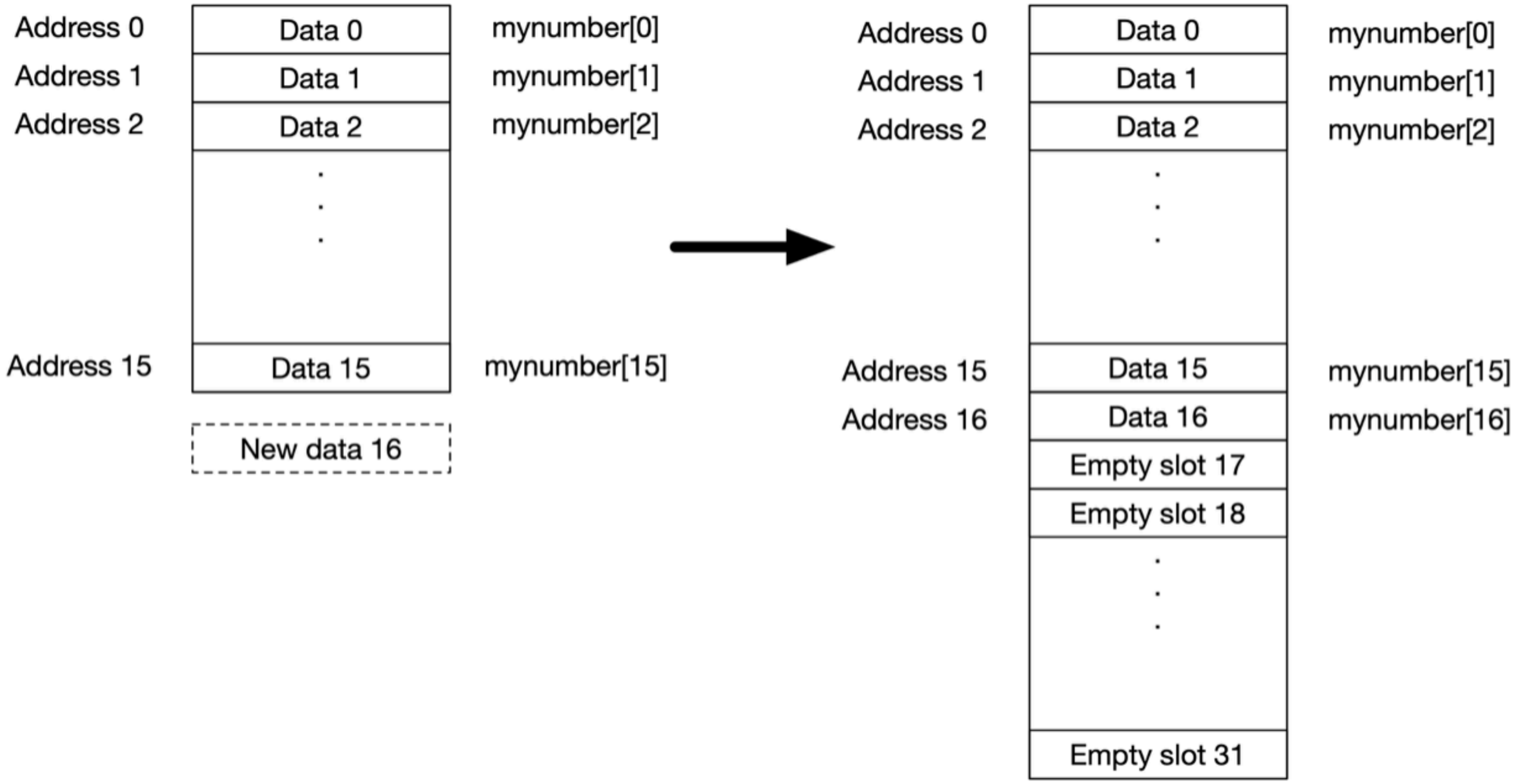
Address n - 1 | Data n - 1 | mynumber[n - 1]

32 - bit

# Fixed-Sized Array

## Increasing Array Size

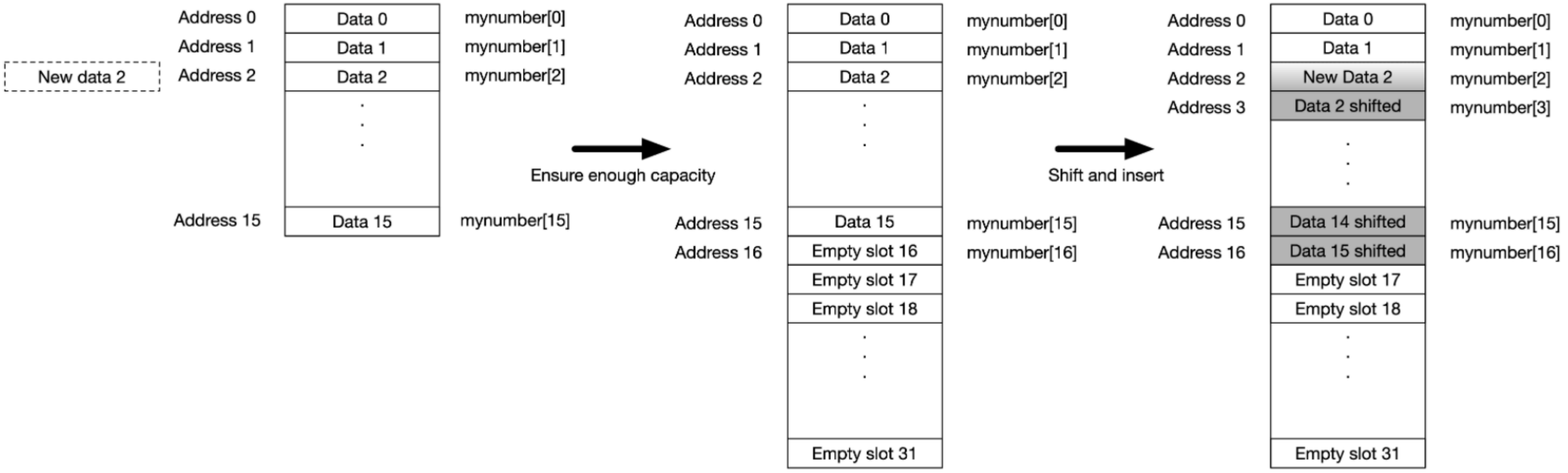
- Time Complexity: O(N)



# Fixed-Sized Array

## Insertion

- Time Complexity: O(N)



New data 2

| | | |
|---|---|---|
| Address 0 | Data 0 | mynumber[0] |
| Address 1 | Data 1 | mynumber[1] |
| Address 2 | Data 2 | mynumber[2] |
| | . . . | |
| Address 15 | Data 15 | mynumber[15] |

Ensure enough capacity

| | | |
|---|---|---|
| Address 0 | Data 0 | mynumber[0] |
| Address 1 | Data 1 | mynumber[1] |
| Address 2 | Data 2 | mynumber[2] |
| | . . . | |
| Address 15 | Data 15 | mynumber[15] |
| Address 16 | Empty slot 16 | mynumber[16] |
| | Empty slot 17 | |
| | Empty slot 18 | |
| | . . . | |
| | Empty slot 31 | |

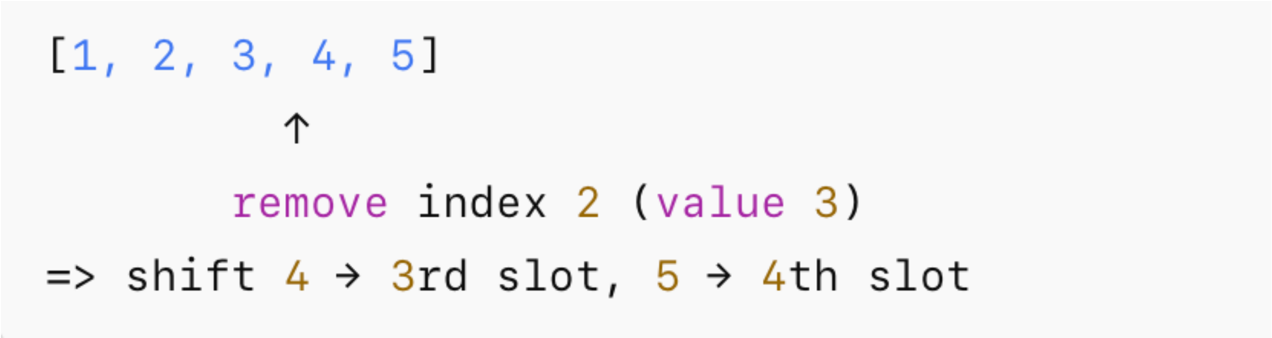
Shift and insert

| | | |
|---|---|---|
| Address 0 | Data 0 | mynumber[0] |
| Address 1 | Data 1 | mynumber[1] |
| Address 2 | New Data 2 | mynumber[2] |
| Address 3 | Data 2 shifted | mynumber[3] |
| | . . . | |
| Address 15 | Data 14 shifted | mynumber[15] |
| Address 16 | Data 15 shifted | mynumber[16] |
| | Empty slot 17 | |
| | Empty slot 18 | |
| | . . . | |
| | Empty slot 31 | |

# Fixed-Sized Array

## Deletion

- Time Complexity: O(N)

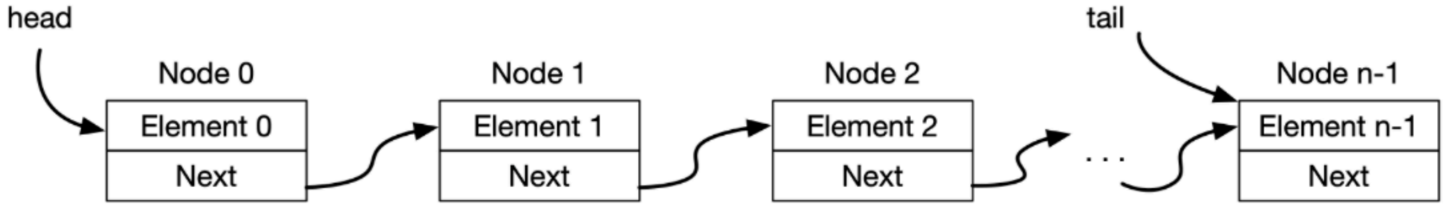


```
[1, 2, 3, 4, 5]
         ↑
       remove index 2 (value 3)
=> shift 4 → 3rd slot, 5 → 4th slot
```

# Linked List

## Data Structure

- Properties:
  - Each "element" has a separate address
  - Each "element" points to the *next* element only
  - We know the address of the first (head) and the last (tail) element only

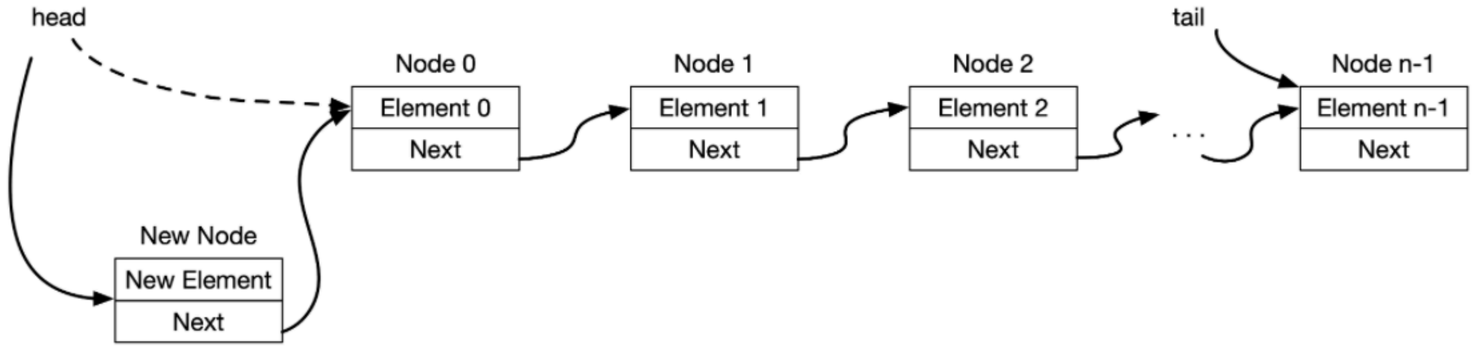


# Linked List

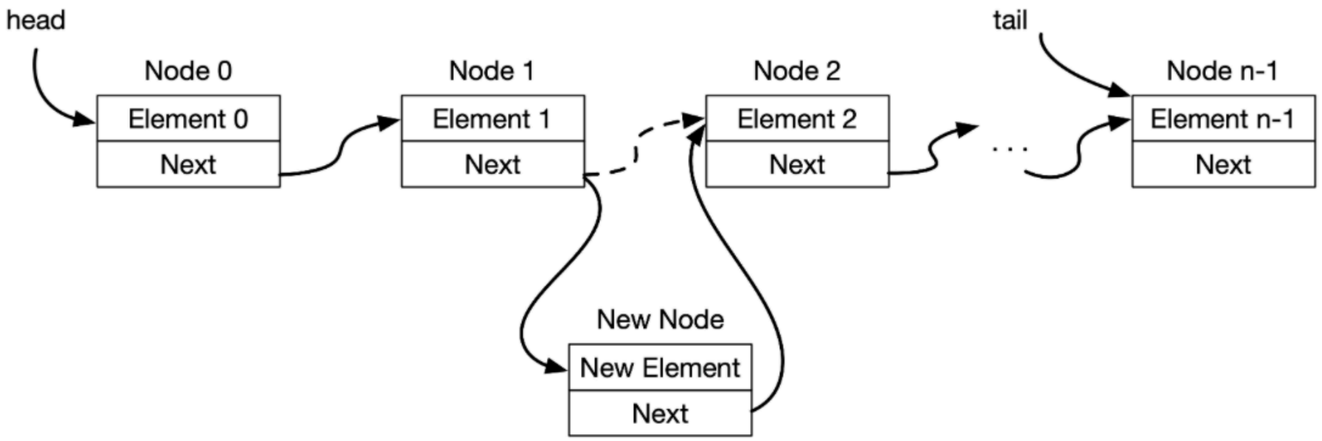
## Insertion

- Time Complexity: O(N)

Best case



Worst case

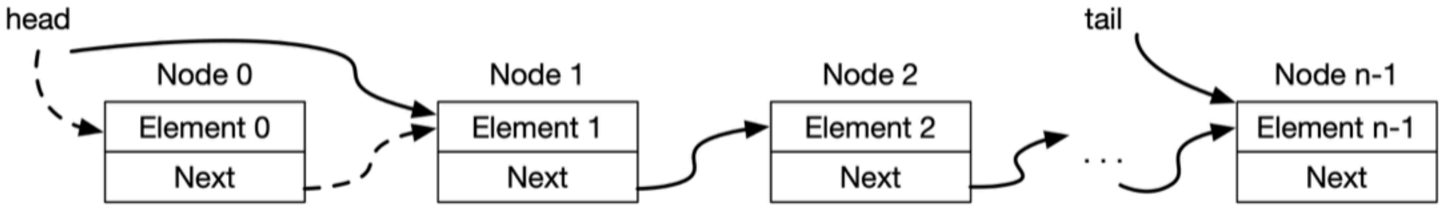


# Linked List

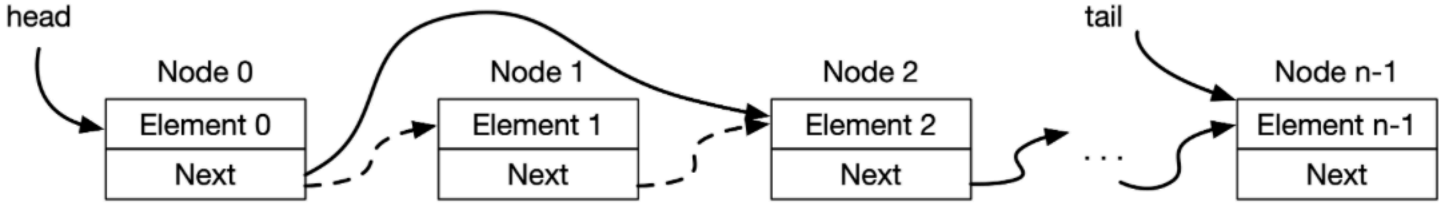
## Deletion

- Time Complexity: O(N)

Best case

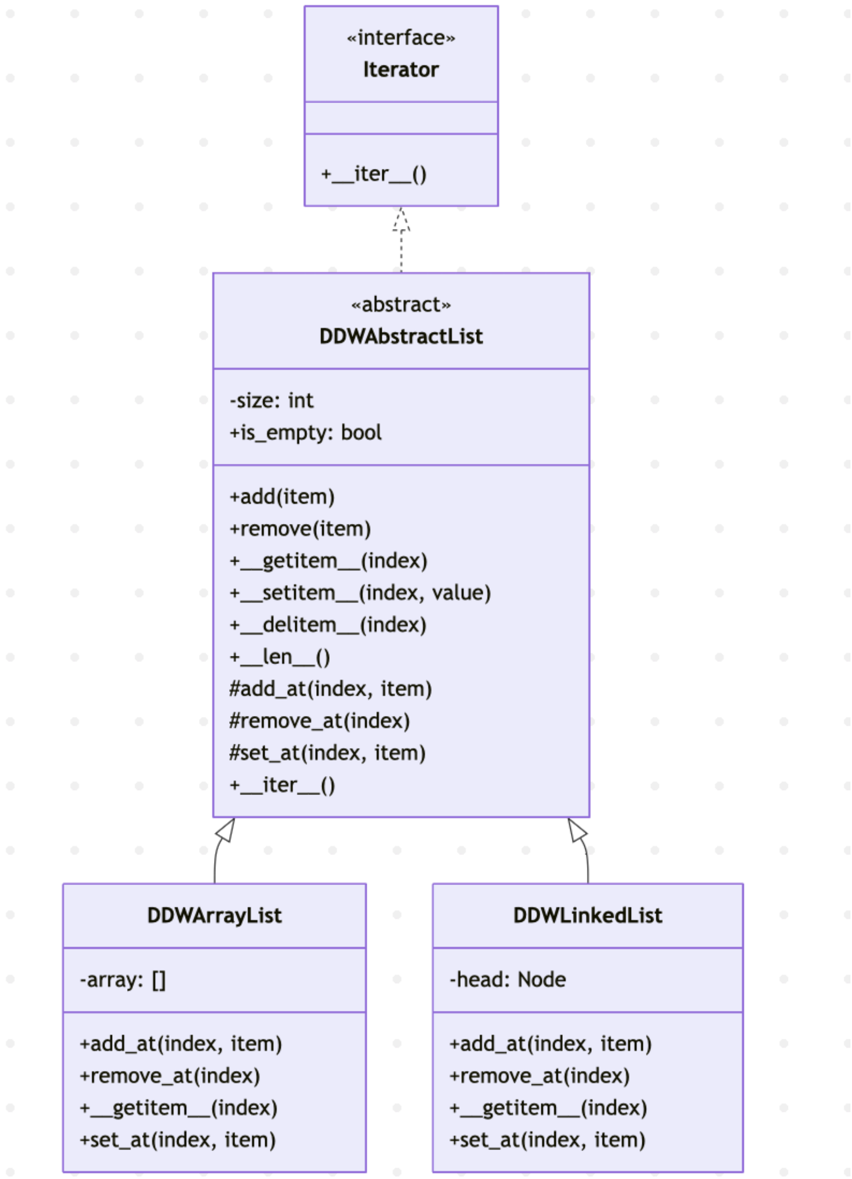


Worst case



# Abstract Base Class: List

## LL and FSA are List, List is an iterator



«interface»
Iterator

+__iter__()

«abstract»
DDWAbstractList

-size: int
+is_empty: bool

+add(item)
+remove(item)
+__getitem__(index)
+__setitem__(index, value)
+__delitem__(index)
+__len__()
#add_at(index, item)
#remove_at(index)
#set_at(index, item)
+__iter__()

DDWArrayList

-array: []

+add_at(index, item)
+remove_at(index)
+__getitem__(index)
+set_at(index, item)

DDWLinkedList

-head: Node

+add_at(index, item)
+remove_at(index)
+__getitem__(index)
+set_at(index, item)

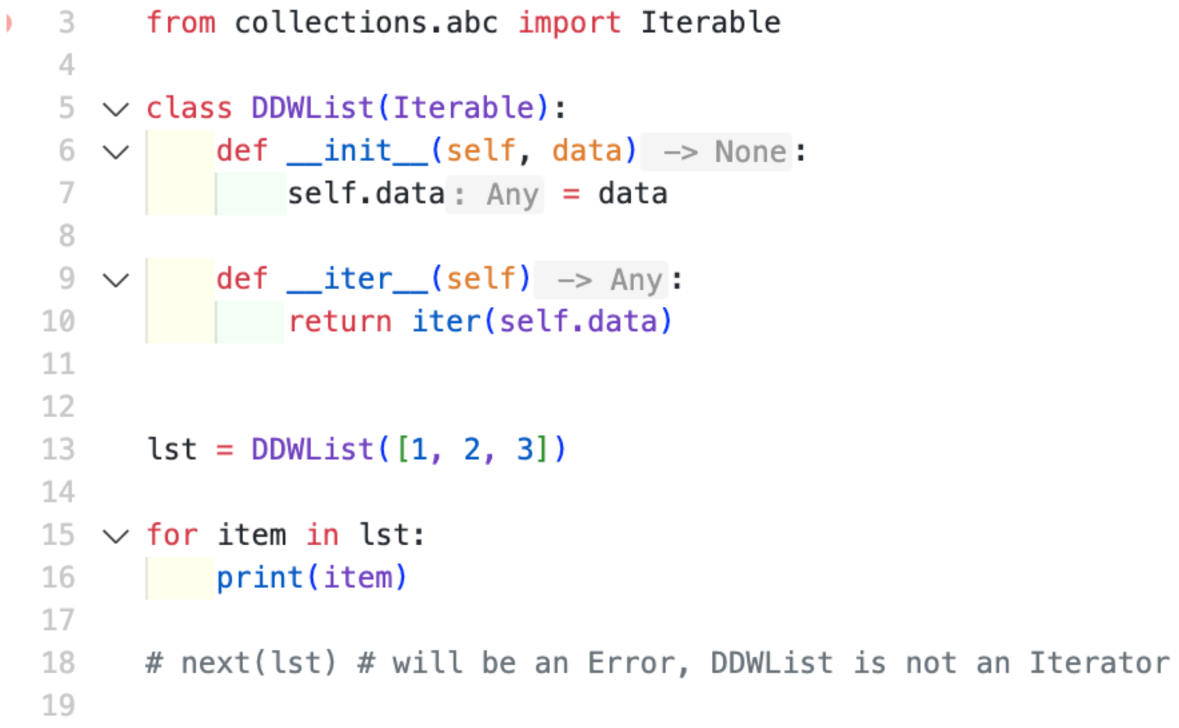
- Symbols:
  - + (public)
  - - (private)
  - # (protected)

# Iterable & Iterator Class

- **Iterable**: any class that lets you loop over its items one by one, like how you loop through a list with a for loop

- `__iter__()` must return an iterator — an object with a `__next__()` method.

- If you return self, then your class must implement `__next__()`



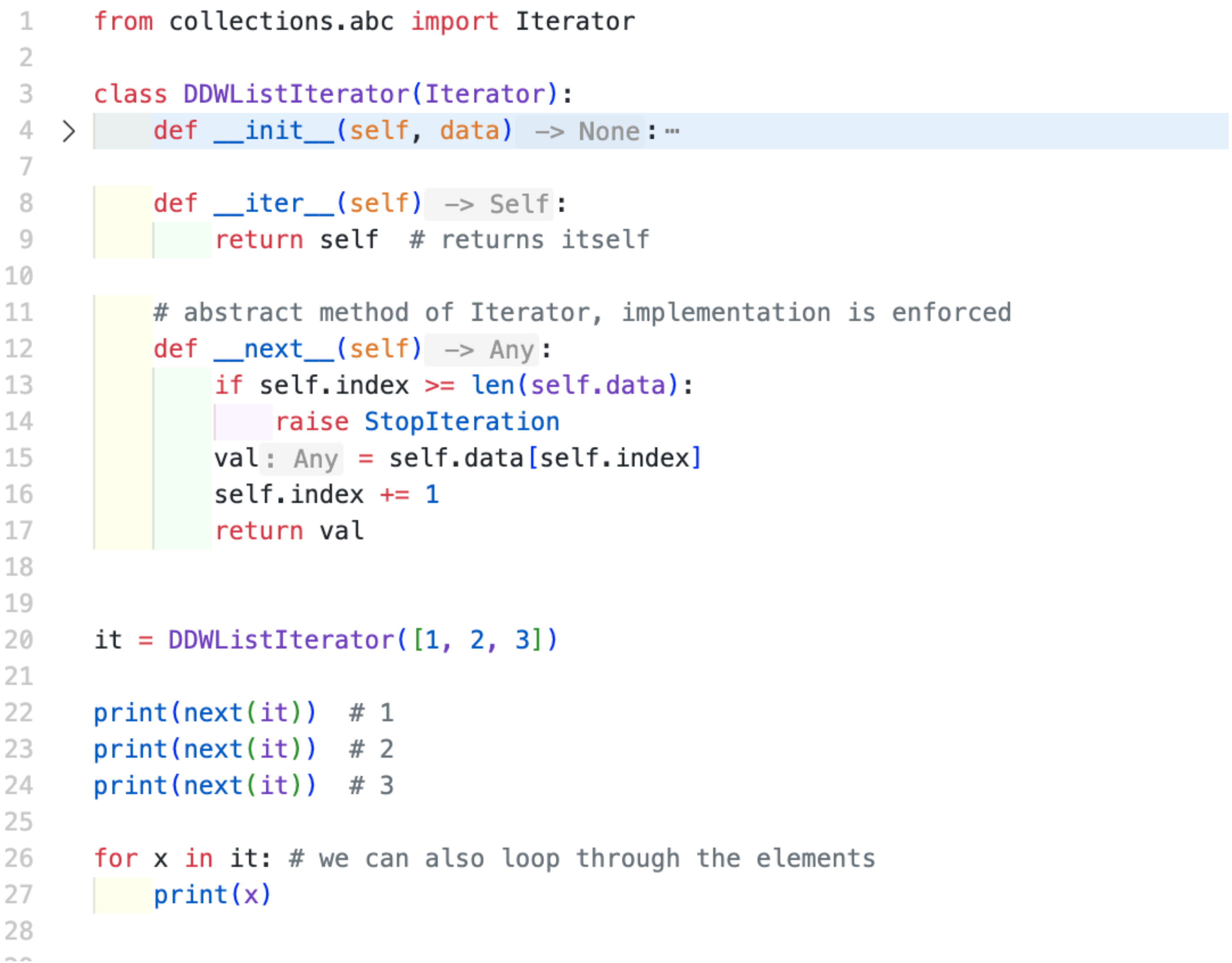
```python
from collections.abc import Iterable

class DDWList(Iterable):
    def __init__(self, data) -> None:
        self.data: Any = data

    def __iter__(self) -> Any:
        return iter(self.data)


lst = DDWList([1, 2, 3])

for item in lst:
    print(item)

# next(lst) # will be an Error, DDWList is not an Iterator

```

# Iterable & Iterator Class

- You can inherit an Iterable or an Iterator class, depending on your use case to make your object exhibit **these** behaviors:
  - **An Iterable is like a book**: you can open it and start reading from the beginning, *again and again*
  - **An Iterator is like a bookmark**: it remembers where you are right now, *but only once*
- Use cases:
  - *Make for-loop work for your custom class (iterable)*
  - *Support next() and don't need to reuse (iterator)*

# Iterable & Iterator Class

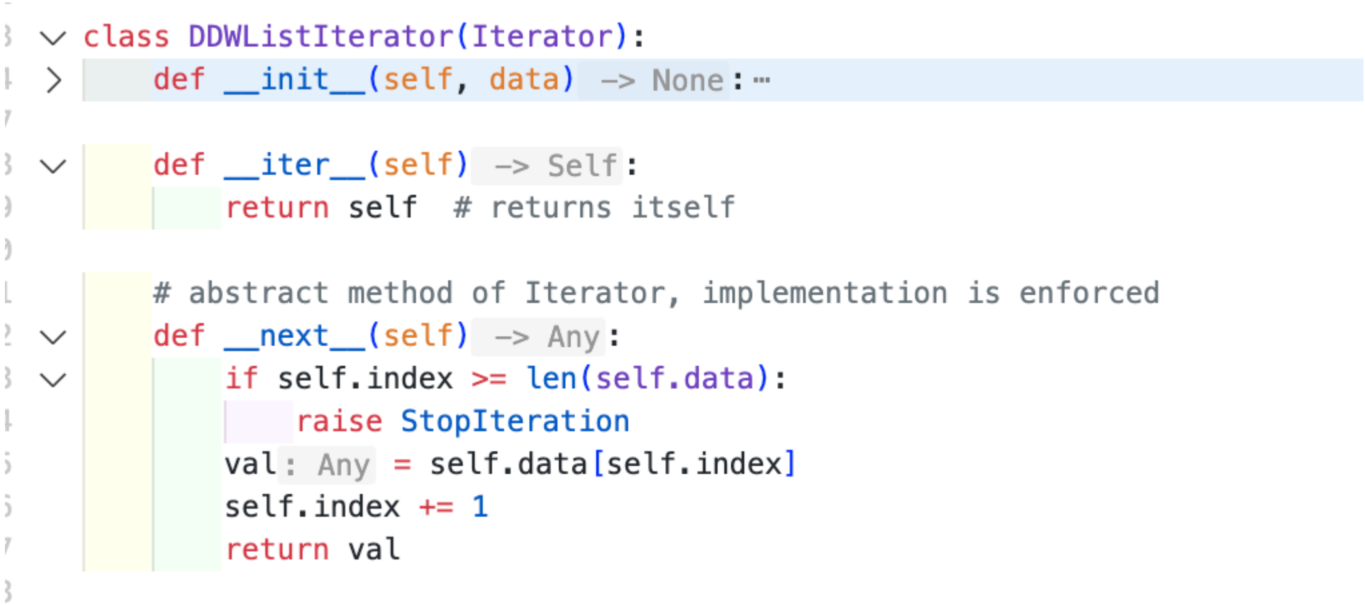
- **Iterator**: Both an iterable and also knows how to return the *next* element in question and *remembers* where you are in a loop



```python
from collections.abc import Iterator

class DDWListIterator(Iterator):
    def __init__(self, data) -> None:⋯

    def __iter__(self) -> Self:
        return self  # returns itself

    # abstract method of Iterator, implementation is enforced
    def __next__(self) -> Any:
        if self.index >= len(self.data):
            raise StopIteration
        val: Any = self.data[self.index]
        self.index += 1
        return val


it = DDWListIterator([1, 2, 3])

print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # 3

for x in it: # we can also loop through the elements
    print(x)
```

# Iterable & Iterator Class

- In Python, for loops need iterables: that is, objects with `__iter__()`.
- But, if an object is an iterator (i.e., it has both `__iter__()` and `__next__()`),
  - and its `__iter__()` returns `self` (which is standard),
  - **Then it's also iterable, because it satisfies the iterable protocol**



```python
class DDWListIterator(Iterator):
    def __init__(self, data) -> None: ···

    def __iter__(self) -> Self:
        return self  # returns itself

    # abstract method of Iterator, implementation is enforced
    def __next__(self) -> Any:
        if self.index >= len(self.data):
            raise StopIteration
        val: Any = self.data[self.index]
        self.index += 1
        return val
```

# Summary

- Use OOP to implement both **data** and **computation**.
- Apply **Stack** and **Queue** for some applications.
- Explain is-a relationship for **inheritance**.
    - Draw UML class diagram for is-a relationship.
- **Inherit** a class to create a child class from a base class.
- **Override** operators to **extend** parent's methods.
- Implement **Deque** data structure as a subclass of Queue
- State the purpose of **Abstract Base Class** & define it
- Implement **Array** and **Linked List** data structure from the same base class.