Data Classes and Linked Lists

1DV501/1DT901: Introduction to programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

October 9, 2022

The Mini-project

The Python Mini-Project is a small project exercise where you in a team (1-2 students) handle a given task which will be presented by the end of the course.

Deadline Assignment 3: October 16
Project Start: Project lecture on Monday October 17
Project Deadline 1: Demonstrate project at tutoring session before Deadline 2
Project Deadline 2: Submit code and report using GitLab, November 6

Mini-project Rules

- You work in teams of two. Register your team in sheet in Moodle.
- You choose your companion yourself – use, for example, Slack or the tutoring sessions to find someone to work with. If you are unable to find a companion, please contact your tutoring supervisor. You might be allowed to work alone.
- Help will be given at the tutoring sessions
- More details/rules are available in Moodle

Today ...

- Data classes
- Implementing linked lists

Reading instructions: Only these slides!

Data Classes – A First Example

```python
from dataclasses import dataclass

@dataclass
class Rectangle:
    height: float
    width: float

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

# Program starts
rec = Rectangle(4.5, 2)    # Create and initialize a new rectangle
print(rec)

area = rec.get_area()
print("Area: ", area)
```

Output

```
Rectangle(height=4.5, width=2)
Area:  9.0
```

Data Classes – Basics

```
from dataclasses import dataclass

@dataclass
class Rectangle:
    height: float
    width: float

    def get_area(self):
        return self.height * self.width
```

- A data class contains data (`height`, `width`) and operations (`get_area(self)`)
- The data are called **fields**, the operations are called **methods**
- Data classes was introduced in Python 3.7 ⇒ a rather new Python feature
- They are a simplified version of ordinary classes
- `@dataclass` is an **annotation**. It used to inform the compiler that this is not an ordinary class, and that it follows different syntax rules.

Fields (or Attributes)

```
from dataclasses import dataclass

@dataclass
class Rectangle:
    height: float = 0.0        # Field with default value 0.0
    width: float = 0.0

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

# Program starts
rec = Rectangle()
print(rec)                      # Rectangle(height=0.0, width=0.0)

rec.height = 6.2                # Update field "height"
rec.width = 5                   # Update field "width"
print(rec)                      # Rectangle(height=6.2, width=5)
print("Height:", rec.height)    # Height: 6.2
```

- `height: float = 0.0` ⇒ the default value for field `height` is `0.0`
- We can access (write to, read from) the fields directly. Example: `rec.width = 5`

Type hints

```python
from dataclasses import dataclass

@dataclass
class Rectangle:
    height: float = 0.0        # Field with hinted type 'float'
    width: float = 0.0

    ...

# Program starts
rec = Rectangle()
rec.height = True       # Not following type hint
rec.width = "hello"     # Not following type hint
print(rec)              # Rectangle(height=True, width='hello')
```

- `height: float = 0.0` ⇒ `float` is a **type hint** ⇒ shows expected value type
- Violations of the type hints are accepted (but likely to cause errors later on)
- Type hints can be dropped when we provide a default value. Don't do it, they provide valuable information
- Python is a dynamically typed language ⇒ variable/field types are decided at runtime and can change. Enforcing type checks would break that

The self-reference `self`

Rectangle data class code

```
...     # Annotations and imports dropped
class Rectangle:
    height: float = 0.0
    width: float = 0.0

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

    def update(self, h, w):
        self.height = h
        self.width = w
```

Program that uses Rectangle

```
# Program starts
rec = Rectangle()
rec.update(9.8, 5.6)
print("Area: ", rec.get_area())
print(rec)
```

Output

```
Area:  54.88
Rectangle(height=9.8, width=5.6)
```

- `self` is the first parameter in any method declaration
- We use `self` to reference fields (and other methods) in the data class
- `self` is a self-reference ⇒ self is always referencing the current object
- After a call `rec.get_area()`, inside method `get_area(self)`, `self` takes the value of the Rectangle object referenced by call target `ref`

Data Classes – Summary

- Data classes are a simplified version of ordinary classes
- A data class contains data (fields) and operations (methods)
- Instances of a data class are called objects
- `rec = Rectangle(4.5, 2)` ⇒ create a new Rectangle object
- `area = rec.get_area()` ⇒ call method `get_area(self)` in object `rec`
- Fields are not protected ⇒ we can access (write to, read from) the fields directly. Example: `rec.width = 5`
- Data class definitions are typically in a separate file named as the class they define (e.g. Rectangle.py)
- Data class definitions requires `from dataclasses import dataclass` and an annotation (`@dataclass`)
- Data classes was introduced in Python 3.7 ⇒ they do not work on older versions

Programming Example – Loan with interest

Exercise: Create a class `Loan` representing a loan with interest. The fields are: `amount` (an integer), `interest` (float), and `years` (an integer). The class should have the following methods:

- `get_total_payment(self)`: Total amount to pay
- `get_total_interest(self)`: Total interest to pay
- `get_monthly_payment(self)`: Monthly payment

Create also (in a separate file) a demo program that shows how to use the `Loan` class using the following two scenarios:

- Borrowing from the bank: amount = 10000, rate = 3.5%, years = 5
- Borrowing from the Mafia: amount = 10000, rate = 25%, years = 5

Example run

```
Borrowing from the bank
Loan(years=5, interest=3.5, amount=10000)
Total payment: 11877
Total interest: 1877
Monthly payments: 198
```

The Loan class (in Loan.py)

```
from dataclasses import dataclass

@dataclass
class Loan:
    years: int = 0
    interest: float = 0
    amount: int = 0

    def get_total_payment(self):
        rate = 1 + self.interest/100
        total = self.amount * rate**self.years
        return round(total)

    def get_total_interest(self):
        total_amount = self.get_total_payment()
        return total_amount - self.amount

    def get_montly_payment():
        ...
```

The entire Loan implementation is available in Moodle.

The demo class (in LoanMain.py)

```
import Loan as loan

# Scenario 1: amount = 10000, rate = 3.5%, years = 5
print("\nBorrowing from the bank")
loan1 = loan.Loan(5, 3.5, 10000)
print(loan1)

total_amount = loan1.get_total_payment()
print("Total payment:", total_amount)

total_interest = loan1.get_total_interest()
print("Total interest:", total_interest)

montly = loan1.get_monthly_payment()
print("Monthly payments:", montly)
```

The entire LoanMain implementation is available in Moodle.

Example: A Deck of Cards

Using class Deck

```python
# Program starts
deck = Deck()
deck.init()      # Initialize deck
print("Size:", deck.size())

for i in range(5):
    print("Deal one:", deck.deal_one())
print("Size:", deck.size())

deck.shuffle()
for i in range(5):
    print("Deal one:", deck.deal_one())
print("Size:", deck.size())
```

Output

```
Size: 52
Deal one: Card(suit='Spades', rank='Ace')
Deal one: Card(suit='Spades', rank='2')
Deal one: Card(suit='Spades', rank='3')
Deal one: Card(suit='Spades', rank='4')
Deal one: Card(suit='Spades', rank='5')
Size: 47
Deal one: Card(suit='Clubs', rank='4')
Deal one: Card(suit='Diamonds', rank='4')
Deal one: Card(suit='Clubs', rank='9')
Deal one: Card(suit='Diamonds', rank='7')
Deal one: Card(suit='Hearts', rank='Ace')
Size: 42
```

- We create a Deck object (`deck = Deck()`) and initialize it with 52 cards (`deck.init()`)
- We use `deck.size()` to get the current deck size (number of cards)
- We pull five cards from the sorted deck using `deck.pull_one()`
- We shuffle the deck and pull five more cards

The entire `Deck` implementation is available in Moodle.

The Deck Data Class

```
from dataclasses import dataclass
from typing import List
import random as rnd

@dataclass
class Card:
    suit: str = None
    rank: str = None

@dataclass
class Deck:
    cards: List[Card] = None    # A deck is a list of cards

# Deck methods on next slide
```

- Two data classes: `Card` and `Deck`
- A card is a **suit** (e.g. Spades, Hearts, ...) and a **rank** (Ace, 2,3,4, ..., King)
- A deck is a list of cards
- `import random` needed to shuffle the deck
- We import type List to assign cards a type hint `List[Card]`

Initializing the Deck

```
@dataclass
class Deck:
    cards: List[Card] = None   # A deck is a list of cards

    def init(self):
        ranks = "Ace 2 3 4 5 6 7 8 9 10 Jack Queen King".split()
        suits = ["Spades", "Hearts", "Diamonds", "Clubs"]
        lst = []
        for s in suits:
            for r in ranks:
                lst.append(Card(s, r))
        self.cards = lst

# More Deck methods on next slide
```

- Card suits (4) and ranks (13) are represented as strings
- We iterate over all possible suit-rank combinations ...
- ... and append to the list a new card for each combination

Initializing the Deck

```
@dataclass
class Deck:
    cards: List[Card] = None    # A deck is a list of cards

    def init(self):
        # See previous slide

    def size(self):
        return len(self.cards)    # Size of list cards

    def shuffle(self):
        rnd.shuffle(self.cards)  # We use shuffle from the random library

    def deal_one(self):
        return self.cards.pop(0) # pop(0) ==> remove and return first card
```

- `size`, `shuffle`, and `deal_one` are short methods manipulating the list cards
- `deal_one(self)` removes and returns first card using list method `pop(n)`

10 minute break

ZZZZ

Sequential Data Structures – Introduction

- We often need to handle large sets of data
- A *data structure* is a model for storing/handling such data sets

Sequential Data Structures

List
: A *sequential* collection where each *element* has a position. In principal: a growing/flexible sequence of data

Queue
: A sequential collection with add and remove at different sides $\Rightarrow$ a *FiFo* (First in, First out)

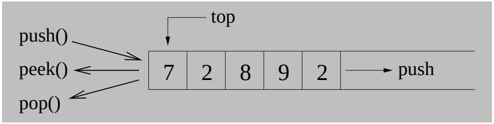
Stack
: A sequential collection with add and remove at the same side $\Rightarrow$ a *LiFo* (Last in, First out)

Deque
: A sequential collection with add and remove at both sides (Deque = Double-Ended Queue)

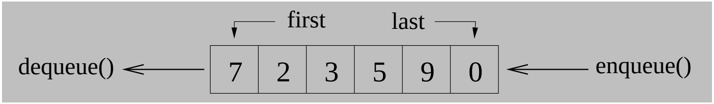
Sequential $\Rightarrow$ a sequence where each element has a position.

Sequential Data Structures

Stack (Last in, first out) Add and remove at one side only.



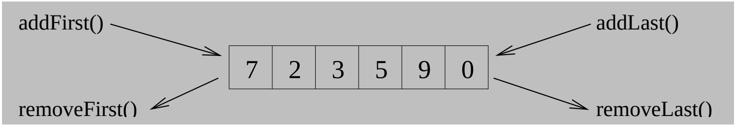
Queue (First in, first out) Add at one side, remove at the other.



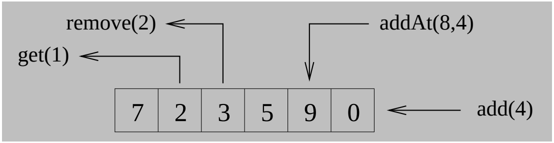
Note: Stack and Queue has a very limited set of operations
$\Rightarrow$ they are the most simple data structures

Deque and List

Deque (Double-ended Queue) Add and remove at both sides.



List (Add and remove everywhere)

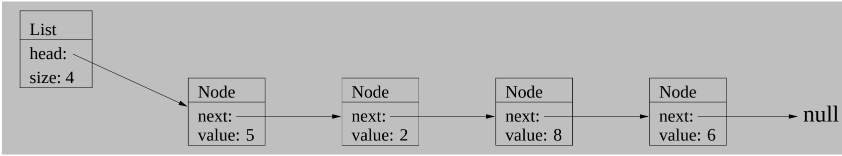


- Note: List is the most general sequential data structure
- Most general ⇒ has most features ⇒ somewhat harder to implement

Linked Lists

List can be implemented using a technique called **Linked Lists**

- Usually made up of two entities (list + node)
- The list holds a reference to the first node (the list field `head`)
- Each element is stored in a node (the node field `value`)
- Each node knows its predecessor node (the node field `next`)
- A user interacts with the list, nodes are encapsulated within the list class
- Example: A list with four elements `[5, 2, 8, 6]`



My linked list implementation

The following slides will show fragments of code from my linked list implementation `LinkedList.py`. The implementation (`.py`-files) is also available in Moodle.

Notice

- An implementation using two data classes: `Node` and `LinkedList`
- LinkedList knows about the first node (`head`) and the current list size (`size`).
- Example of using class `LinkedList`

```
import LinkedList as LL

lst = LL.LinkedList()      # Create new empty list

for i in range(21, 41):  # Add 20 integers
    lst.add(i)
print("to_string()", list.to_string())  # print list content

# Use list methods
print("Size:", lst.size)           # 20
print("get(7):", lst.get(7))       # 28
print("contains(7): ", lst.contains(33)) # True
print("remove(7): ", lst.remove(7)) # 28
print("Size:", lst.size)            # 19
```

A Linked Implementation

Start of file `LinkedList.py`

```
from dataclasses import dataclass
from typing import Any

@dataclass
class Node:
    val: int = None   # Value stored in node
    nxt: Any = None   # Points to next node in the list

@dataclass
class LinkedList:
    head: Node = None     # First node in list
    size: int = 0         # Keep track of current list size

    # More LinkedList methods on the following slides
```

- Class `Node` represents a node in the linked list
- Class `LinkedList` keeps track of first node (`head`) and current list size
- Field type `Any` since `Node` not properly defined at this point

The method `add(self, n)`

```
class LinkedList:
    head: Node = None
    size: int = 0

    # Append element n to the list
    def add(self, n):
        new = Node(n, None)       # Node to be added
        if self.head is None:     # An empty list
            self.head = new
        else:                     # Non-empty ==> Find last node
            node = self.head      # and attach new node as last
            while node.nxt is not None:
                node = node.nxt   # Move to next node
            node.nxt = new
        self.size += 1
```

Usage:

```
for i in range(21, 41):# Add 20 integers
   lst.add(i)
```

Hence, we move along the node chain and add `new` as last element

The function to_string(self)

```
# Returns string representation of list content
    def to_string(self):
        s = "{ "
        node = self.head
        while node is not None:
            s += str(node.val) + " "
            node = node.nxt
        s += "}"
        return s
```

Usage:

```
print(lst.to_string())   # { 1 2 3 ... 18 19 20 }
```

Hence, we move along the node chain and add `str(node.val) + " "` to the result string for each node.

contains(self, n)

```
# Returns True if n is in list, otherwise False
def contains(self, n):
        node = self.head
        while node is not None:
            if node.val == n:
                return True
            node = node.nxt
        return False
```

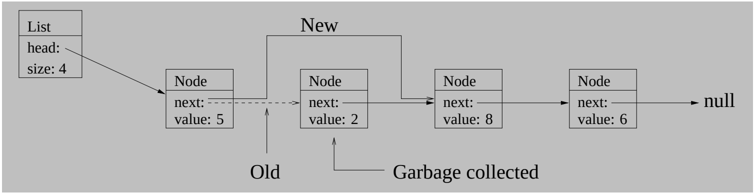
We start at the head node (`node = self.head`) and move along node chain as long as we have more nodes (`while node is not None`).

Function get(self, pos)

```
# Get element at position pos, show error message
# if position pos is out of range
def get(self, pos):
        if self.head is None:
            print("get() can't be applied on an empty list")
            return None
        elif pos < 0 or pos >= self.size:
            print(f"Access outside list index range [0,{self.size-1}]")
            return None
        else:
            node = self.head
            for i in range(pos):     # Take pos steps along the node ch
                node = node.nxt      # the node chain
            return node.val
```

Error message if a) index out of range, b) we apply get on an empty list.
Default: Take pos steps along the node chain and return the value of the node at position pos.

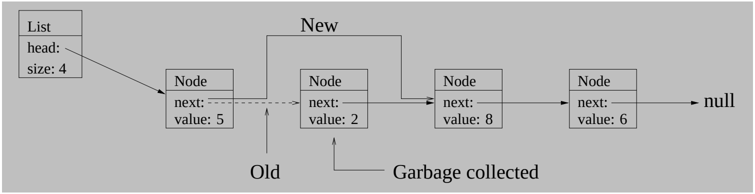
The `remove(self, pos)` function (Part 1)



Note: Error message if a) index out of range, b) we apply `remove` on an empty list. Remove first node: We by-pass first node by making `head` point to `head.next`.

```
# Remove element at position pos
    def remove(self, pos):
        if self.head is None:
            print("remove() can't be applied on an empty list")
        elif pos < 0 or pos >= self.size:
            print(f"Access outside list index range [0,{self.size-1}]")
        elif pos == 0:
            self.head = self.head.nxt  # By-pass head node
            self.size -= 1
        else:
            # See next slide for the case of removing a non-first node
```

The `remove(self, pos)` function (Part 2)



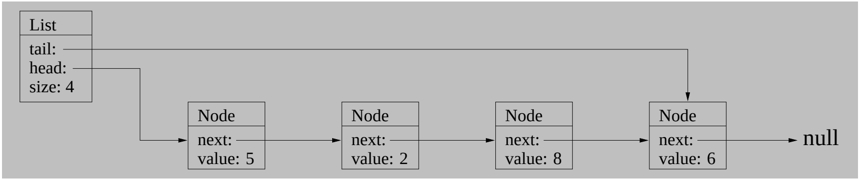
Note: We can't move backwards ⇒ we must operate from the node **before** the node we want to remove

```
def remove(self, pos):
    # ... See previous slide
    else:
        # Find node before node to be deleted
        before = self.head
        for i in range(pos-1):
            before = before.nxt
        # By-pass node at position pos
        delete = before.nxt          # Node to be deleted
        before.nxt = delete.nxt      # By-pass node to be deleted
        self.size -= 1
```

Variant 1: Head and tail

Problem: `add(self, n)` ⇒ step through the whole list ⇒ very slow
(Serious since `add(self, n)` is a frequently used operation.)

Solution: Keep also track of the tail node ⇒ we can jump there directly

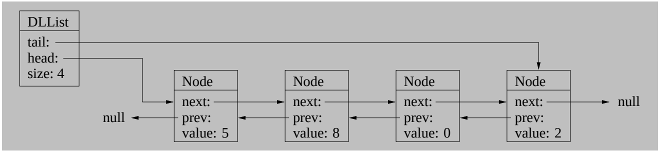


This is the approach to use in queues and deques (Assignment 3, Exercise 16) where we do all operations on both ends of the node chain.

Variant 2: Double-linked List

Problem: We can only traverse list in one direction
⇒ (for example) printing the list content backwards is very slow

Solution: Each node has a field `prev` that references the previous node.



Our linked list vs Pythons list

- Python lists are not linked lists
- Python lists are implemented as *dynamic arrays*
  (They start using a mutable C array of size (say) 1000 to store the data. They then double the size every time we need more size $\Rightarrow$ sizes are 1000, 2000, 4000, 8000, ...)
- Faster than linked lists in operations like `append` and `get`
  (since we don't need to move along the chain of nodes).
- Slower than linked list in operations like `remove(0)` since it must shuffle all elements one step left to cover for the hole at position 0.
- **Python lists are much (much!) faster since implemented in C**
- Many time critical parts in Python are implemented in highly optimized C code and called from the main program by the virtual machine.

However, linked data structures (one node referencing others) is an important concept that are used in trees and graphs.

Assignment 3 – Exercise 16

Exercise: Implement a deque data structure using the **head-and-tail** approach.

- We provide a file `deque_skeleton.zip` containing three files:
  1. A data class `Deque.py` containing the skeleton of a deque implementation. It provides a complete `Node` data class and a non-complete `Deque` data class.
  2. A program `deque_main.py` that shows how to use the `Deque` class.
  3. A file `output.txt` that shows the expected output from `deque_main.py` for a correct complete `Deque` implementation.

Your task is to complete the class Deque by providing code for the missing methods. We strongly suggest that you implement the methods one at the time, starting from the top with method `add_last(self, n)` followed by `to_string(self)`. Use the demo program `deque_main.py` to verify that the implemented methods works correctly.