Data Structures and Algorithms

Lecture 4: Data Structures and OOP

Heikki Peura

h.peura@imperial.ac.uk

Last time

Algorithm complexity

- Asymptotic analysis and Big-O notation
- Searching and sorting

Plan for today:

- Introduction to data structures
- Object-oriented programming (OOP)

Data structures

How to organize data for quick access?

- Like with algorithms: recipe $\rightarrow$ translate to Python

Examples: lists, stacks, queues, dictionaries (hash tables), graphs, trees, etc

Different data structures are suitable for different tasks

- Support different sets of operations (`list` vs `dict`)
- How to choose?

We have already been using data structures

```
'Hello World'
3.14159
9
L = [1, 1999, 0, -2, 9]
```

We have already been using data structures

```
'Hello World'
3.14159
9
L = [1, 1999, 0, -2, 9]
```

These are all **objects**. An object has:

- A type: `int`, `str`, `list` (`L` is an **instance** of a `list`)
- An internal representation of data
- A set of functions that operate on that data (methods)

We have already been using data structures

```
'Hello World'
3.14159
9
L = [1, 1999, 0, -2, 9]
```

These are all **objects**. An object has:

- A type: `int`, `str`, `list` (`L` is an **instance** of a `list`)
- An internal representation of data
- A set of functions that operate on that data (methods)

An object bundles data and relevant actions

A Python list has many operations

Some of the operations:

- `len(L),max(L),min(L),...`
- `L[start:stop:step]`: returns elements of `L` from `start` to `stop` with step size `step`
- `L[i]= e`: sets the value at index `i` to `e`
- `L.append(e)`: adds `e` to the end of `L`
- `L.count(e)`: returns how many times `e` occurs in `L`
- `L.insert(i,e)`: inserts `e` at index `i` of `L`
- `L.extend(L1)`: appends the items of `L1` to the end of `L`
- `L.remove(e)`: deletes the first occurrence of `e` from `L`
- `L.index(e)`: returns the index of first occurrence of `e` in `L`
- `L.pop(i)`: removes and returns the item at index `i`, default `i = -1`
- `L.sort()`: sorts elements of `L`
- `L.reverse()`: reverses the order of elements of `L`

A list is an `object`

An object bundles data and actions

A list is an `object`

An object bundles data and actions

```
L = [1,1999,0,-2,9]
L.append(8)
L.insert(2,1000)
t = L.pop()
L.remove(1)
help(L)
```

A list is an `object`

An object bundles data and actions

```
L = [1,1999,0,-2,9]
L.append(8)
L.insert(2,1000)
t = L.pop()
L.remove(1)
help(L)
```

The point:

- Interface: the user knows what she can do with a list
- Abstraction: the user does not need to know the details of what goes on under the hood (similarly to functions)
- Invaluable in managing complexity of programs

List operations complexity?

```
L = [1,1999,0,-2,9]
L.append(9)
t = L[2]
t = L.pop(0)
```

We have **assumed** that list operations like retrieving or adding an item are $O(1)$

- A list has internal functions with algorithms to perform these operations

List operations complexity?

```
L = [1,1999,0,-2,9]
L.append(9)
t = L[2]
t = L.pop(0)
```

We have **assumed** that list operations like retrieving or adding an item are $O(1)$

- A list has internal functions with algorithms to perform these operations

But we have seen that **how we write algorithms matters** a lot...

- Does it matter how you implement a list?
- Yes!
- What is the internal data representation of a list?

How can we implement a list?

For a list, we would like to:

- Add and remove elements, look up values, change values, ...

How can we implement a list?

For a list, we would like to:

- Add and remove elements, look up values, change values, ...

An indexed **array**?

56	24	99	32	9	61	57	79

How can we implement a list?

For a list, we would like to:

- Add and remove elements, look up values, change values, ...

An indexed **array**?

56	24	99	32	9	61	57	79

- Reserve n slots of memory for list from computer memory
- Easy to access an item at index `i`: $O(1)$

How can we implement a list?

For a list, we would like to:

- Add and remove elements, look up values, change values, ...

An indexed **array**?

56	24	99	32	9	61	57	79

- Reserve n slots of memory for list from computer memory
- Easy to access an item at index `i`: $O(1)$
- Easy to add an item to end: $O(1)$ (but details are advanced...)

How can we implement a list?

For a list, we would like to:

- Add and remove elements, look up values, change values, ...

An indexed **array**?

56	24	99	32	9	61	57	79

- Reserve n slots of memory for list from computer memory
- Easy to access an item at index `i`: $O(1)$
- Easy to add an item to end: $O(1)$ (but details are advanced...)
- **Difficult to add item to beginning**: $O(n)$ (need to move all other elements)

A linked list of “nodes”?

Linked list?

$$\boxed{56} \to \boxed{24} \to \boxed{99} \to \boxed{32} \to \boxed{9} \to \boxed{61} \to \boxed{57} \to \boxed{79}$$

A linked list of “nodes”?

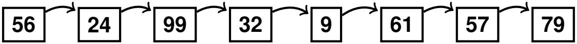
Linked list?

$\boxed{56} \to \boxed{24} \to \boxed{99} \to \boxed{32} \to \boxed{9} \to \boxed{61} \to \boxed{57} \to \boxed{79}$

- **Each node contains information on the next node** – the list itself just knows the first and last one

A linked list of “nodes”?

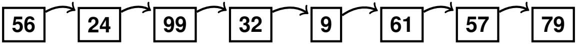
Linked list?



- **Each node contains information on the next node** – the list itself just knows the first and last one
- Easy to add items to either end: $O(1)$

A linked list of “nodes”?

Linked list?



- **Each node contains information on the next node** – the list itself just knows the first and last one
- Easy to add items to either end: $O(1)$
- **Difficult to look up item by index**... $O(n)$ (need to walk through the nodes)

A Python list has many operations

Some of the operations:

- `len(L)`,`max(L)`,`min(L)`,...
- `L[start:stop:step]`: returns elements of `L` from `start` to `stop` with step size `step`
- `L[i]= e`: sets the value at index `i` to `e`
- `L.append(e)`: adds `e` to the end of `L`
- `L.count(e)`: returns how many times `e` occurs in `L`
- `L.insert(i,e)`: inserts `e` at index `i` of `L`
- `L.extend(L1)`: appends the items of `L1` to the end of `L`
- `L.remove(e)`: deletes the first occurrence of `e` from `L`
- `L.index(e)`: returns the index of first occurrence of `e` in `L`
- `L.pop(i)`: removes and returns the item at index `i`, default `i = -1`
- `L.sort()`: sorts elements of `L`
- `L.reverse()`: reverses the order of elements of `L`

Object-oriented programming (OOP)

Everything is an object with a type: `L=[1,2,3,4]` is an **instance** of a `list` object

Abstraction — creating an object type:

- Define internal representation and interface for interacting with object — user only needs interface

Object-oriented programming (OOP)

Everything is an object with a type: `L=[1,2,3,4]` is an **instance** of a `list` object

Abstraction — creating an object type:

- Define internal representation and interface for interacting with object — user only needs interface
- Then we can create new **instances** of objects and delete them

Object-oriented programming (OOP)

Everything is an object with a type: `L=[1,2,3,4]` is an **instance** of a `list` object

Abstraction — creating an object type:

- Define internal representation and interface for interacting with object — user only needs interface
- Then we can create new **instances** of objects and delete them

This is **“divide-and-conquer” development**

- Modularity — treat a complex thing like a list as primitive
- Easier to reuse code — keep code clean: eg ‘+’ method for integers and strings

Why define objects?

Suppose you're designing a game where players catch **pocket monsters** and make them fight each other

Why define objects?

Suppose you're designing a game where players catch **pocket monsters** and make them fight each other

```
# Using lists?
monsters = ['Pikachu','Squirtle','Mew']
combat_points = [20,82,194]
hit_points = [53,90,289]
```

Why define objects?

Suppose you're designing a game where players catch **pocket monsters** and make them fight each other

```
# Using lists?
monsters = ['Pikachu','Squirtle','Mew']
combat_points = [20,82,194]
hit_points = [53,90,289]
```

```
# Using a dictionary?
monsters = {'Pikachu':[20,53],'Squirtle':[82,90],'Mew':[194,289]}
```

Defining an object type

An object contains

- **Data**: attributes (of a monster)
- **Functions**: methods that operate on that data

Defining an object type

An object contains

- **Data**: attributes (of a monster)
- **Functions**: methods that operate on that data

Suppose you're designing a game where players catch **pocket monsters** and make them fight each other

```
class Monster(object):
    """
    Attributes and methods
    """
```

`class` statement defines new object type

We've created a Monster

Attributes of a monster?

We’ve created a Monster

Attributes of a monster?

```python
class Monster(object):
    """
    Pocket monster
    """
    def __init__(self,combat_points):
        self.combat_points = combat_points

Pikachu = Monster(65)
Squirtle = Monster(278)
print(Pikachu.combat_points)
```

`self`: Python passes the object itself as the first argument — convention to use word “self”

- But you omit this when calling the function

We’ve created a Monster

Attributes of a monster?

```python
class Monster(object):
    """
    Pocket monster
    """
    def __init__(self,combat_points):
        self.combat_points = combat_points

Pikachu = Monster(65)
Squirtle = Monster(278)
print(Pikachu.combat_points)
```

`self`: Python passes the object itself as the first argument — convention to use word “self”

- But you omit this when calling the function

- Notice the “`.`” operator (like with a `list`)
- The `__init__` method is called when you call `Monster()`

Growing our monsters

```python
class Monster():
    def __init__(self, name, combat_points, hit_points):
        self.name = name
        self.combat_points = combat_points
        self.hit_points = hit_points
        self.health = hit_points

    def hurt(self, damage):
        self.health = self.health - damage
        if self.health <= 0:
            print(self.name + ' is dead!')
```

More in the workshop!

Why OOP is useful

Easy to handle many “things” with **common attributes**

Abstraction isolates the use of objects from implementation details

Build **layers of abstractions** — our own on top of Python’s classes

- Keeping track of different monsters and their attributes

Accessing data

```python
class Monster(object):
    def __init__(self, name, combat_points, hit_points):
        self.name = name
        self.combat_points = combat_points
        self.hit_points = hit_points
        self.health = hit_points

    def get_combat_points(self): # access data through method
        return self.combat_points

    # more Monster code...

Pikachu = Monster('Pikachu', 100, 30)
cp = Pikachu.combat_points # risky - what if you make a mistake?
cp = Pikachu.get_combat_points() # safer - cannot mess stuff up
```

(If you go deeper into OOP, there are more advanced ways of making data “private”)

Review

Data structures and OOP

- OOP is a way of designing programs to bundle data and actions
- Data structures are ways to organize data efficiently
- Python has excellent data structures for common tasks

Workshop after the break

- More Monsters
- A data structure we'll need later: queue

Workshop

Workshop zip file on the Hub

- HTML instructions
- At some point, you'll need the `.py`-file with skeleton code (open in Spyder)