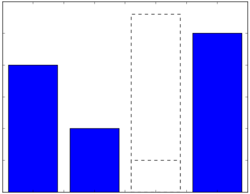
COMP1730/COMP6730
Programming for Scientists

Abstract data types and concrete data structures

Assignment Qs & As

- The assignment is the same for undergraduate (1730) and master (6730) students; hence no problem mixing 1730 and 6730 students in a group.
- The testing environment is the Anaconda python installation on the CSIT lab computers.

- What to do about missing data?
  - You can not assume it is zero.
  - If data is present but # of days missing, it is a single-day observation.
- What about the gaps?
  - For the quantile (method (b)) threshold, it doesn't matter.
  - Whichever way you treat a gap implies an assumption, a risk or a limitation – understand and document what it is.



Lecture outline

- Abstract data types
- Data structures

Abstract data types

- The type of a value determines what can be done with it (and what the result is).
- Conversely, we may define an *abstract data type* (ADT) by the set of operations that can be done on values of the type.
- Already seen examples:
  - “sequence type” (length, index, slice)
  - “iterable type” (`for` loop)
- No special syntax (or even a type name).

Interface

- An *interface* is a set of functions (or methods) that implement operations (create, inspect and modify) on the abstract data type.
- The interface creates an *abstraction*.
  - For example, “a date has a year, a month and a day” instead of “a date is a list with length 3”.
- The user of the ADT (that is, the programmer) must use only the interface functions to operate on values of the ADT – accessing/modifying the structure of the value directly *breaks the abstraction*.

```
def make_date(year, month, day):
    return [year, month, day]

def get_year(adate):
    return adate[0]

...

def is_before(date1, date2):
    return ((date1[0] < date2[0]) or
            (date1[0] == date2[0] and
             date1[1] < date2[1]) or
            (date1[0] == date2[0] and
             date1[1] == date2[1] and
             date1[2] < date2[2]))
```

Why data type abstraction?

- It makes code easier to read and understand.
  - For example,

    ```
    get_day(get_date(cal_entry))
    ```

    instead of

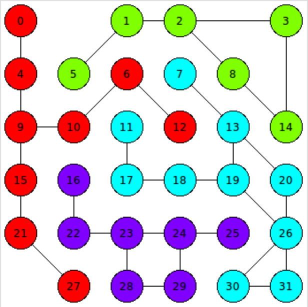
    ```
    cal_entry[2][2]
    ```

- It makes code *refactorable*.
  - The implementation behind the interface can be replaced without changing any code that uses it.

```
import datetime

def make_date(year, month, day):
    return datetime.date \
             (year, month, day)

def get_year(adate):
    return adate.year

...

def is_before(date1, date2):
    return date1 < date2
```

Example: Networks

- A *network* (or *undirected graph*) consists of *nodes*; some pairs of nodes are connected by *links*.
- Can represent physical structure (e.g., a power network), a social network, logical relationships (e.g., synonymy).



- Interface for the Network ADT:
  - Create a new network
    - An empty network, or with a given number/set of nodes.
  - Add or remove a node.
  - Add or remove a link between a pair of nodes.
    - Modifies the network (no return value).
  - Are a pair of nodes connected? (have a link)
  - Enumerate the nodes connected to a given node (it’s *neighbours*).

Data structures

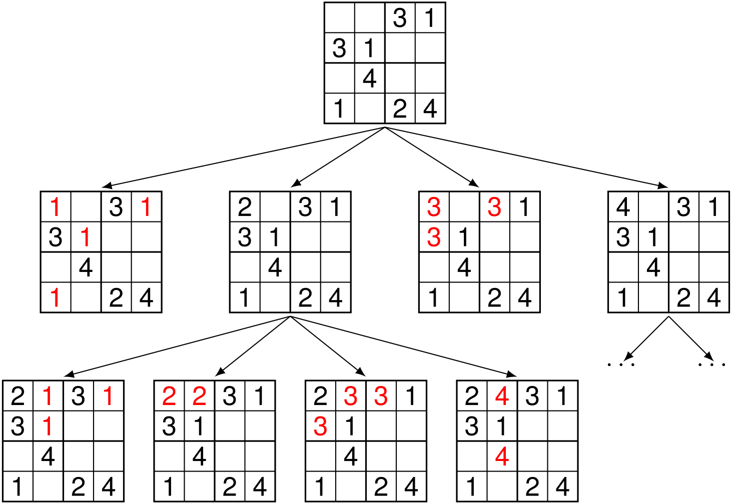
- A concrete implementation of an abstract data type must use some *data structure* – made up of built-in python types – to store values.
- Typically, several alternative data structures can implement an ADT.
- Consider:
  - Ease of implementation
  - Memory requirements
  - Computational complexity of operations

Example: Implementations of ADT network

- Store whether there is a link (`True`/`False`) for each pair of nodes in a list-of-lists or 2-d array.
  - Uses O(#nodes^2) memory.
  - Add/remove/check links in constant time.
  - Collecting neighbours takes linear time.
  - Insert or remove node?

- Store list or set of neighbours for each node.
  - Uses *O*(#links) memory.
    - #links is *at most* #nodes^2, can be much less.
  - Add/remove/check links:
    - (amortised) constant time using python's `set` type;
    - linear time using (unordered) lists.
  - Neighbour sets available in constant time (linear to copy).
  - Insert or remove node?

Extra example: Sudoku



		3	1
3	1		
	4		
1		2	4

1		3	1
3	1		
	4		
1		2	4

2		3	1
3	1		
	4		
1		2	4

3		3	1
3	1		
	4		
1		2	4

4		3	1
3	1		
	4		
1		2	4

2	1	3	1
3	1		
	4		
1		2	4

2	2	3	1
3	1		
	4		
1		2	4

2	3	3	1
3	1		
	4		
1		2	4

2	4	3	1
3	1		
	4		
1		2	4

Summary

- Creating and using abstract data types helps structure larger programs, making them easier to write, debug, read and maintain.
- Several ways to implement ADTs in python:
  - Function interface; and
  - data structures using built-in python types.
  - Defining classes (not covered in this course).