Australian National University

COMP1730/COMP6730
Programming for Scientists

Abstract data types and concrete data structures

Lecture outline

* Abstract data types
* Data structures
* Dictionaries & sets

Reminder: Code quality

* Good code organisation:
  - raises the level of abstraction; and
  - isolates subproblems and their solutions.
* The name of a function *or type* should suggest what it does, or is used for.
* Use the function docstring to elaborate.

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)
  - "iterable type" (`for` loop)
- No special syntax.

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*.

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.

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: mapping

* A *mapping* (a.k.a. *dictionary*) stores key–value pairs; each key stored in the mapping has exactly one value. Keys do not have to be consecutive integers.
* Examples of use:
  - Storing a look-up index (e.g., a contact list).
  - Organising data with “complex” labels (like a multi-dimensional table).
  - Storing solutions to subproblems in a dynamic programming algorithm.

- Interface – what you can do with a mapping:
  - Create new, empty mapping.
  - Store a (new) value with a key.
  - Is a given key stored in the mapping?
  - Look up the value stored for a given key.
  - Remove key.
  - Enumerate keys, values, or key–value pairs.
- What can be used as a key?
  - What can happen if keys are mutable?
  - Do keys have to be comparable?

* Implementations of mapping:
  - Store key–value pairs in a list.
    - All operations are linear time.
  - Store key–value pairs in a list, sorted by keys.
    - Key look-up is $O(\log n)$ time, but adding a new key takes linear time.
  - Hashtable (built-in python type `dict`).
    - Insertion and lookup can be done in amortised constant time.

python's `dict` type

- Create a new dictionary:

```
>>> adict = {}
>>> adict = dict()
>>> adict = { (2015,12) : 33.4,
              (2016,6) : 148.3 }
>>> adict = { "be" : 2, "can" : 3 }
```

  - Dictionary (and set!) literals are written with curly brackets ({ and }).
  - The literal can contain *key* : *value* pairs, which become the initial contents.

- Key exists in dictionary:

```
>>> key in adict
```

- Look-up and storing values:

```
>>> adict = { "be" : 2, "can" : 1 }
>>> adict["can"]
1
>>> adict["now"] = 2
>>> adict[3] = "yet"
```

  - To index a value, write the key in square brackets after the dictionary expression.
  - Assigning to a dictionary index expression adds or updates the key.

- `dict` is a mutable type.
  - Like lists, arrays.
- Keys must be *immutable* $^{(\star)}$.

```
>>> alist = [1,0]
>>> adict = { alist :  2 }
TypeError: unhashable type: 'list'
```

- A dictionary can contain a mix of key types.
- Stored values can be of any type.

- Removing keys:
  - `del` *`adict`*`[`*`key`*`]`
    Removes *`key`* from *`adict`*.
  - *`adict`*`.pop(`*`key`*`)`
    Removes *`key`* from *`adict`* and returns the associated value.
  - *`adict`*`.popitem()`
    Removes an arbitrary `(`*`key`*`, `*`value`*`)` pair and returns it.
- `del` and `pop` cause a runtime error if *`key`* is not in dictionary; `popitem` if it is empty.

Iteration over dictionaries

* *adict*.keys(), *adict*.values(), and *adict*.items() return *views* of the keys, values and key–value pairs.
* Views are iterable, but *not* sequences.

```
for item in adict.items():
    the_key = item[0]
    the_value = item[1]
    print(the_key, ':', the_value)
```

Programming problem(s)

* Counting frequency of items:
  - words in a file (or web page);
  - (combinations of) values in a data table.
* Building a Markov model (over text, for example).
* Cross-referencing data tables with common keys.

Sets

* A *set* is an unordered collection of (immutable) values without duplicates.
* Like a dictionary with only keys (no values).
* What you can do with a set:
  - Create a new set (empty or from an iterable).
  - Add or remove values.
  - Is a given element in the set? (membership).
  - Mathematical operators: union, intersection, difference (note: not complement!).
  - Enumerate values.

python's `set` type

- Set literals are written with `{ .. }`, but with elements only, not key–value pairs:

```
>>> aset = { 1, 'c', (2.5, 'b') }
```

- `{ }` creates an empty dictionary, not a set!
- A set can be created from any iterable:

```
>>> aset = set("AGATGATT")
>>> aset
{'T', 'A', 'G'}
```

  – No duplicate elements in the set.
  – No order of elements in the set.

Set operators

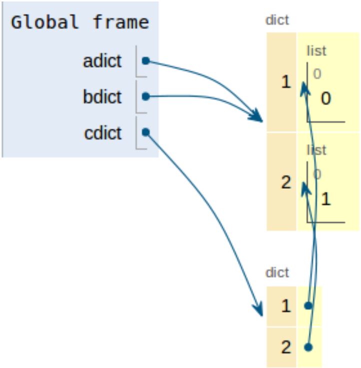
`elem in aset`	membership ($e \in A$)
`aset.issubset(bset)`	subset ($A \subseteq B$)
`aset | bset`	union ($A \cup B$)
`aset & bset`	intersection ($A \cap B$)
`aset - bset`	difference ($A \setminus B$, $A - B$)
`aset ^ bset`	symmetric difference

- Set operators return a new result set, and do not modify the operands.
- Also exist as methods (`aset.union(bset)`, `aset.intersection(bset)`, etc).

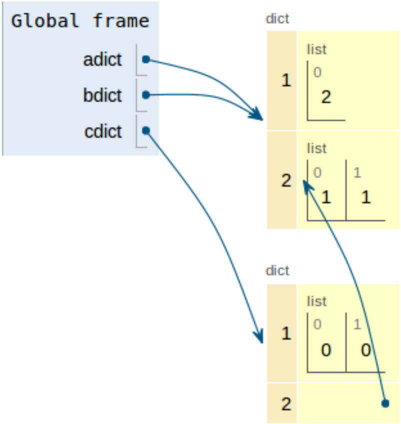
Copying

- Dictionaries and sets are mutable objects.
- Like lists, dictionaries and sets store *references* to values.
- `dict.copy()` and `set.copy()` create a *shallow* copy of the dictionary or set.
  - New dictionary / set, but containing references to the same values.
  - Dictionary keys and set elements are immutable, so shared references do not matter.
  - Values stored in a dictionary can be mutable.

```
adict = {1:[0],2:[1]}
bdict = adict
cdict = adict.copy()
bdict[1] = [2]
cdict[1] = [0, 0]
adict[2].append(1)
```



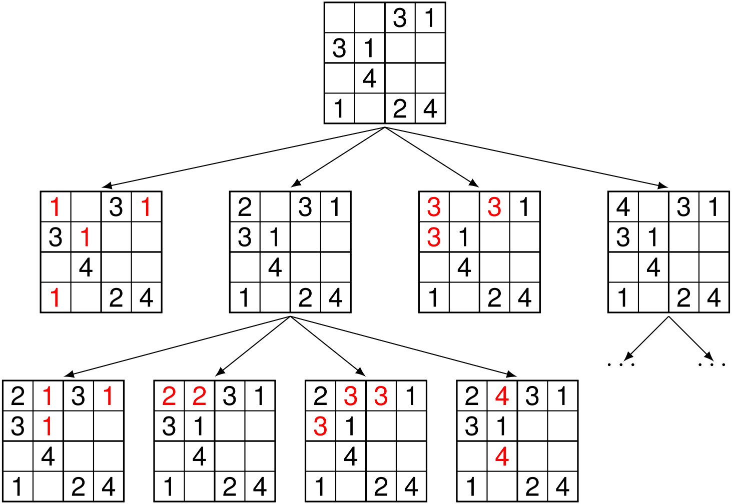
```
adict = {1:[0],2:[1]}
bdict = adict
cdict = adict.copy()
bdict[1] = [2]
cdict[1] = [0, 0]
adict[2].append(1)
```



Summary

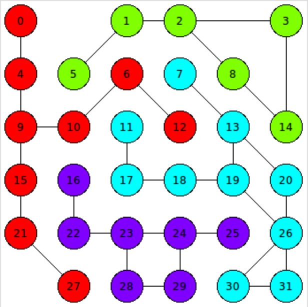
* Creating and using abstract data types helps structure larger programs, making them easier to write, debug, read and maintain.
* Several ways to implement ADTs in python:
  – Function interface; and
  – data structures using built-in python types.
  – Defining classes (not covered in this course).

Extra example: Sudoku



Extra 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*).

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 $O(\#\text{nodes}^2)$ 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* $\#\text{nodes}^2$, 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?