1811/2807/7001ICT
Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

24 Dictionaries

Class `dict` is Python's powerful implementation of a dictionary or map.

24.1 Dictionaries

Definition A *dictionary* or *map* is like a set, in that it contains an unordered collection of values, but for each of the unique values there is another associated value.

The unique value is called the *key*.

The associated value is then just referred to as the *value*.

Note:

- The keys are unique (no duplicates).
- The values don't have to be unique.

They are named after ordinary dictionaries for words, where each word is a key and its meaning is its associated value.

If a word has multiple meanings, the associated value can be a list of meanings.

Two words may have the same meaning.

There are plenty of real-life examples of this kind of data structure: a telephone book; the contacts list in your phone; and the index of a book.

The name *map* is used for the same type of data structure in languages such as Java and Haskell.

24.2 Class dict

Python implements dictionaries with its class `dict`.

A `dict` object is itself mutable, but:

- its keys must be immutable (like set elements); and
- the values may be mutable.

24.2.1 Empty dicts

There are two ways to create an empty `dict`:

- a pair of empty braces, `{}`, which is why we can't use this for an empty set; and
- calling the constructor, `dict()`.

```
>>> {} == dict()
True
>>>
```

24.2.2 Non-empty dicts

Non-empty sets may be formed:

- with `dict` literals, using braces and colons separating the keys from the values;

```
>>>  {'Andrew' : 'N44_1.37', 'Jun' : 'N44_1.38'}
{'Andrew': 'N44_1.37', 'Jun': 'N44_1.38'}
>>>
```

- using the `dict` constructor with argument(s) that are:
  - keys and values as keyworded arguments;
  - keys and values in a list of tuples of length 2; and
  - keys and values in a list of lists of length 2,; and

```
>>>  dict(Andrew = 'N44_3.37', Jun = 'N44_1.38')
{'Andrew': 'N44_1.37', 'Jun': 'N44_1.38'}
>>>  dict([('Andrew', 'x55016'), \
... ('Jun', 'x55017')])
{'Andrew': 'x55016', 'Jun': 'x55017'}
>>>  dict([['Andrew', ('N44_1.37', 'x55016')], \
...  ['Jun', ('N44_1.38', 'x55017')]])
{'Andrew': ('N44_1.37', 'x55016'),
'Jun': ('N44_1.38', 'x55017')}
>>>
```

- by adding elements to existing `dict`s, using square brackets as if the key were an index.

```
>>> directory = \
... dict([['Andrew', ('N44_1.37', 'x55016')], \
... ['Jun', ('N44_1.38', 'x55017')]])
>>> directory['John'] = ('G09_1.54', 'x28630')
>>> directory
{'Andrew': ('N44_1.37', 'x55016'),
'Jun': ('N44_1.38', 'x55017'),
'John': ('G09_1.54', 'x28630')}
>>>
```

24.2.3 Dictionary operations

Looking up values by key:

`d[key]`	Return the value associated with `key` in `d`.
`d.get(key, [default])`	Return the value associated with `key` in `d`, optionally returning `default` if the `key` is specified is not in `d`.

Looking up a key that doesn't exist will raise an exception, unless you give the `get` method its optional `default` argument.

You can check if the key exists first with:

`key in d`	Does the `key` exist in `d`?

Updating or adding values:

`d[key] = value`	Add or replace the *value* associated with the *key* in *d*.
`d.update(d2)`	Update *d* with the data in dictionary *d2*.

Getting all of the data out of a `dict`:

`d.keys()`	Return a sequence of all of the keys in *d*.
`d.values()`	Return a sequence of all of the values in *d*.
`d.items()`	Return a sequence of tuples containing all of the keys and values in *d*.

A `dict` does not store values in any particular order. To sort the sequences returned above, use the built-in function `sorted(sequence)`.

Removing values from a `dict`:

`del d[key]`	Delete the *key* from *d*.
`d.clear()`	Delete all keys from *d*.

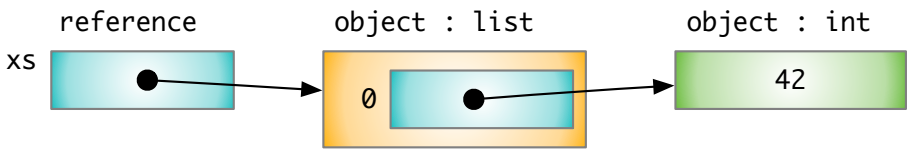
24.3 Collections only store references

Recall that a python variable is only a reference to an object, and that the value is really stored in the object.

What really happens when we say we put values in a tuple, list, set, or dict?

This example shows a `list` with an `int` "in it".

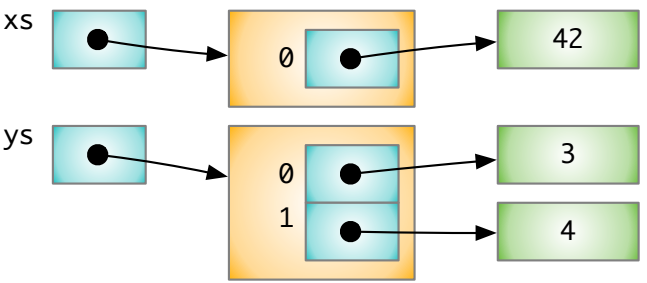
```
>>> xs = [42]
>>> xs
>>> [42]
>>> xs[0]
42
>>>
```



The `int` value isn't really in the `list`, only a reference to it.

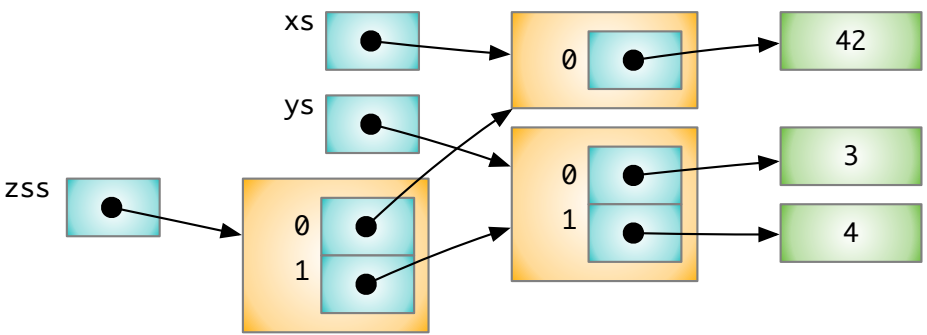
Let's make another list (and make the diagram a bit more compact).

```
>>> ys = [3, 4]
>>> ys
[3, 4]
>>>
```



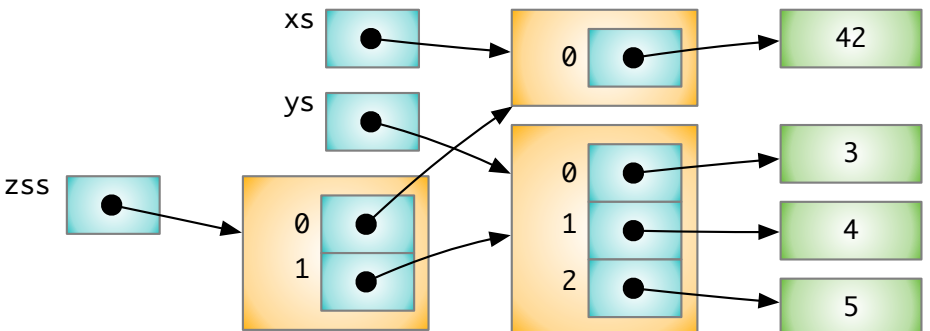
Make `zss` a list of lists.

```
>>> zss = [xs, ys]
>>> zss
[[42], [3, 4]]
>>>
```



Lets change `ys`, and see what happens to `zss`.

```
>>> ys.append(5)
>>> zss
[[42], [3, 4, 5]]
>>>
```



The elements of a list are allowed to be mutable, that is change.

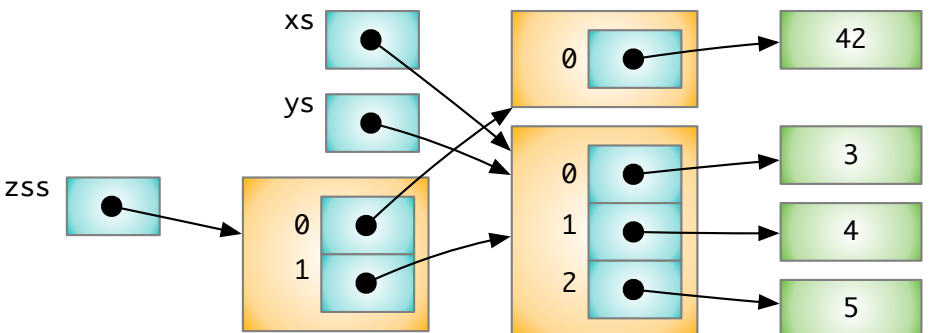
It is not OK for the values in `sets` or the keys in `dicts` to be changed like this as their uniqueness may be broken.

In the last example, we modified list `ys` with its `append` method.

What happens if we change the *variable* `xs` by reassigning it?

Let's assign `ys` to `xs`.

```
>>> xs = ys
>>> xs
[3, 4, 5]
>>> zss
[[42], [3, 4, 5]]
>>>
```



Changing `xs` changed what it was referencing.

`xs` became an *alias* of `ys`.

But it did not change what `xs` used to reference, so `zss` was not changed.

24.4 Dict example

This is a classic problem that needs a `dict`. If follows on from the sample problem for sets.

> *Problem:* Write a program that prompts for and reads a file name, then reads the file, and prints all the distinct words in the file and how often they occurred (their frequency) in decreasing order of frequency.

Section summary

This section covered:

- the `dict` class, `dict` literals, and operations on `dicts`; and
- demonstrates that the collection clases really only store references.