Fundamentals of Computer Programming

Lecture slides - Sets and Dictionaries

Intro

- This lesson covers
  - Introduction to Sets
  - Working with Sets
  - Introduction to Dictionaries
  - Working with Dictionaries
  - Comparing Lists, Tuples, Sets, and Dictionaries

Sets

A compound type

- A **set** is similar to a list and tuple, but with very important differences
- A value can appear at most once within the set, i.e. no *duplicates* exist
  like a mathmatical set
- A set contains an *unordered* collection of *immutable* values, hence they cannot contain lists etc. (since lists are mutable)
- Since sets do not support the concept of element positioning, they do not support *indexing*, *slicing* or any method based on an element's position
  Position of elements can change so no point doing this
- Sets support membership testing, along with traditional set type operations such as *union*, *intersection* and *difference*
  Very quick + easy to find out if a set contains a specific element.
- Sets are *iterable*, so can be used within `for...in` type statements
  Not typically done but can use a for loop to extract all of the elements

Creating a Set

- A set is written as a comma separated list of values, appearing between braces, e.g.

```
vowels = {"a", "e", "i", "o", "u"}
```

Sets use curly brackets

- Do not get these mixed up with lists, that use square brackets […] or Tuples that use parentheses (…)
- If during initialisation the same value does appear more than once, it will only actually exist once in the resulting set
- Since sets are unordered, the order of initialisation values will not usually be maintained in the resulting set, e.g.

```
print(vowels)
{'e', 'u', 'i', 'o', 'a'}
```

→ Can change at any point.

Used to store a selection of things where an order does not matter

Duplicates are ignored in a set

Creating a Set

- An empty set can exist, but has to be created using the `set()` *constructor* function, rather than using `{}` e.g.

  ↳ Creates an empty dictionary

  ```
  empty = set()
  ```

- This can also be used to create a set from an existing *iterable* type, e.g.

  ```
  vowels = set("aeiou")
  ```

- Notice how the *constructor* takes a single value (which is used to create the set) rather than several values, e.g. the following is NOT allowed -

  ```
  vowels = set("a", "e", "i", "o", "u")
  ```

- This is an important distinction, as we will see when examining the various types of operations that we can perform on sets

Set Comprehensions

- A set can also be constructed using a **set comprehension**, which allows construction of a set from an expression
- These are very similar to *list comprehensions*, but use braces instead
- The set contents are created by evaluating the expression for each value iterated over within one or more for loops, e.g.

```
letters = {chr(x) for x in range(ord("a"), ord("z")+1)}
```

Code generates the contents of the list when run

- As with lists comprehensions, an optional `if` statement can be used to restrict inclusion based on a condition, e.g.

```
consonants = {n for n in letters if n not in vowels}
```

Mutable and immutable sets

- Like a list, a Set is **mutable** - that is, the elements can be changed after it has been created
- There is also an **immutable** version of a set known as a `frozenset`, created using the `frozenset()` constructor function, e.g. ↳ Set can never change

```
suits = frozenset({"heart", "club", "spade", "diamond"})
```

- Notice that a single value is passed, which is a set created using `{ … }`
- A `frozenset` supports only the *accessor* type operations, since it is immutable, whereas a regular `set` supports both *mutators* and *accessors*
- Sets can be manipulated using either expression *operators*, or equivalent *methods*

Set accessor operators

Operators for Sets

- Accessor type operations are available using operators `|`, `&`, `-`, and `^`
- Return a new set by combining all elements of other sets

```
letters = vowels | consonants      # union of the sets
```

- Return a new set containing only common elements

```
graduates = students & passed      # intersection of the sets
```

- Return a new set containing elements in one, but not the other

```
employees = staff - resigned       # difference of the sets
```

- Return a new set containing elements in one or the other, but not both

```
finished = passed ^ failed         # symmetric difference of the sets
```

- Note: since these are *accessors* they return a new set, rather than changing the existing set. The *operands* and return value is always a *set*

Done this way instead of calling methods.

Set comparison operators

- Comparison type operations are available using operators <, <=, >=, >

can be used to represent typical mathmatical operations.

- Test whether all elements of a set are in another set

```
word <= vowels   # sub-set, True if all word elements are vowels
```

- Test whether all elements of a set are in another set, but sets are not equal

```
consonants < letters # proper (strict) sub-set, would return True
```

- Test whether all elements of another set are in a set

```
primary >= colours   # super-set, True if all colours are primary
```

- Test whether all elements of another set are in a set, but sets are not equal

```
    letters > vowels              # proper (strict)super-set, would return
True
```

- Note: since these are *comparison* operations they return `True` or `False`, and the *operands* are always a *set*

Set accessor methods

Mathematical terms.

- Rather than use *operators*, the same type of functionality is also available using *methods*, e.g.

```
letters = vowels.union(consonants)     # union of the sets
employees = staff.difference(resigned)     # difference of the sets
word.issubset(vowels)                  # is a subset test
```

- The methods, like the `set()` *constructor* function take a single value, e.g.

```
word.issubset("aeiou")                 # this is legal
word.issubset("a", "e", "i", "o", "u")     # this is not legal
```

- The operators however take a *set* type operand, e.g.

```
word <= "aeiou"                        # this is not legal
word <= {"a", "e", "i", "o", "u"}      # this is legal
```

Set mutator operations

- As with accessors, *mutators* are available using *operators* or *methods*
- The *operators* match those we have seen, but are applied in the *Augmented* assignment style, e.g.

```
graduated |= passed_students  # update the set with the union of both
current -= graduated          # update the set by removing elements
```

- Equivalent *methods* also exist, as before they take a single value, e.g.

```
graduated.update(passed_students)
current.difference_update(graduated)
```

- Methods such as `add()`, `remove()`, and `pop()` allow adding and removal of single elements

Dictionaries

A dictionary has a key (word looking up) and a value (a definition)

↳ A way of associating a unique value with another value.

- A **dictionary** stores values like the other *collection* types
- A dictionary stores elements as pairs, often called a ***key: value*** pair
- The *key* is a *unique immutable* value that can appear at most once (i.e. the keys are basically a set)
- Each *key* has an associated *value*, the values in the dictionary do not need to be unique, i.e. different keys can be mapped to the same value
- Dictionaries are *mutable* and can have *key:value* pairs added and removed
- Unlike a set a dictionary is *ordered*, with the ordering been based on the order of insertion (as of Python 3.7) - wasn't in previous versions of Python

Often not important that the order is maintained

Creating a Dictionary

- A *dictionary* is created in a similar way to a *set*, but with *key:value* pairs, e.g.

```
stock = {"apple":10, "banana":15, "orange":11 }
```

Key value

- An empty *dictionary* is created using empty braces, e.g.

```
grades = {}
```

- The `dict()` constructor function may also be used in various ways, e.g.

Alternative way to create

```
# create by passing a dictionary
stock = dict({"apple":10, "banana":15, "orange":11})

# create by passing keywords (only possible if keys are strings)
stock = dict(apple=10, banana=15, orange=11)

# create by passing list of tuples
stock = dict([("apple",10), ("banana",15), ("orange",11)])
```

Dictionary Comprehensions

Final way to create them. Calculating the contents of the dictionary.

- A dictionary can also be constructed using a **dictionary comprehension**, which allows construction of a dictionary from an expression
- These are very similar to *set comprehensions*, but generate both a *key* and a *value*
- The *key:value* pairs are created by evaluating the expression for each value iterated over within one or more for loops, e.g.

```
powers = {x: x ** x for x in range(2,8)}
```

- Would produce a dictionary such as -

```
{2: 4, 3: 27, 4: 256, 5: 3125, 6: 46656, 7: 823543}
```

Working with Dictionaries

- Dictionary manipulation can be done using *indexing* type notation, but the correct *key* should be provided (not always an integer), e.g.

Changes values that are already stored →

```
stock["pear"] = 50            # add new key:value pair
stock["apple"] += 1           # increase apple stock level
del stock["orange"]           # remove ‘orange’ key and value
if "apple" in stock:          # test if ‘apple’ is a key in ‘stock’
    print("Apple stock level is", stock["apple"])
```

- Many *functions* and *methods* also exist to support the use of dictionaries
- These include `clear()`, `copy()`, `get()`, `pop()` and `update()`
- The most useful methods however allow access to a dictionary’s `keys()`, `values()` and `items()`

Iterating over Dictionaries

How to unpack and Show (print) data from dictionaries

- To iterate over each *key* within a dictionary -

```
for item in stock:
    print(item)
```

- To iterate over each *value* within a dictionary -

```
for level in stock.values():
    print(level)
```

- To iterate over each *key:value* pair within a dictionary (either of these)-

```
for item,level in stock.items():
    print(item, "has a stock level of", level)
```

```
for item in stock:
    print(item, "has a stock level of", stock[item])
```

Dictionaries as arguments

- The lesson describing functions mentioned the concept of specifying an *arbitrary keyword argument* (by prefixing the parameter with ‘**’), e.g.

```
def show_details(title="Details", **info):
    print(title)
    for name in info:
        print(name, ":", info[name])
```

Shows to pass in as a dictionary

- When a call to the function is made, any *keyword argument* not known to the function is packed into a dictionary, e.g.

```
show_details(title="Info", msg="file created", err="no issues")
```

- Would results in the following dictionary being implicitly created and assigned to ‘info’ -

```
info = { "msg" : "file created", "err" : "no issues" }
```

Dictionaries as arguments

- It is also possible to *unpack* a Dictionary prior to a function being called
- This is useful if values are within a dictionary, but a function requires separate *keyword arguments*, e.g. given the following dictionary -

```
details = {"module":"Databases", "course":"Comp.Science"}
```

- We could unpack this and call a function with separate keyword arguments using -

Dictionaries can be unpacked when called.

```
print_module(**details)
```

- The '**' prefix causes the dictionary to be unpacked prior to the function call, and actually results in the call be made as follows -

```
print_module( module="Databases", course="Comp.Science")
```

Seeing a double ** shows a dictionary is in use in some way.

Lists vs Tuples vs Sets vs Dictionaries

- Since Python has several built-in *collection* types, it can be sometimes difficult to know which one to pick for a particular situation
- Don’t try to always use the same type simply because you understand how it works, pick the right type for the right circumstances
- All of these collections and operations may seem confusing at first, but just look them up as required until you become more accustomed
- Also remember that Python makes it easy to create one collection from a different type of collection, e.g. it is easy to create a *set* from a *list*

  ```
  unique_names = set(names)
  ```
- or a *list* from the values within a *dictionary*,

  ```
  stock_levels = list(stock.values())
  ```

Lists vs Tuples vs Sets vs Dictionaries

- Lists tend to be used when -
  - The stored values are based on the same type, e.g. all strings
  - The content is dynamic, i.e. elements often need adding or removing
  - The order of the elements is important and indexing is required
- Tuples tend to be used when -
  - The stored values are based on different types
  - The content is static, i.e. elements rarely need adding or removing
  - The order of the elements is important
  - The total number of elements is typically small

Lists vs Tuples vs Sets vs Dictionaries

- Sets tend to be used when -
  - The stored values are based on the same immutable type, e.g. all integers
  - The content is dynamic, i.e. elements often need adding or removing
  - The order of the elements is irrelevant and indexing is not required
  - Duplicates are not allowed and testing for membership is often required
- Dictionaries tend to be used when -
  - A number of values each need to be associated with a unique key
  - The stored keys are based on the same immutable type
  - Finding elements quickly (given the key) is important and often performed
  - A large number of *key:value* pairs need to be stored

Summary

- A *set* stores multiple values with no **duplicates** allowed
- Special **set operators** can be used to manipulate *sets*
- An immutable **frozen set** can be created
- A *dictionary* stores values in ***key, value*** pairs
- Keys must be **unique** and should be based on the same **immutable type**
- Both *sets* and *dictionaries* can be constructed using **comprehensions**
- Knowing how to best select between *Lists*, *Tuples*, *Sets* and *Dictionaries* is important.