

## LECTURE 15 — SETS

### 27.1 Overview

- Example: finding all individuals listed in the Internet Movie Database (IMDB)
- A solution based on lists
- Sets and set operations
- A solution based on sets.
- Efficiency and set representation

Reading is Section 11.1 of *Practical Programming*.

### 27.2 Finding All Persons in the IMDB file

- We are given a file extracted from the Internet Movie Database (IMDB) called `imdb_data.txt` containing, on each line, a person's name, a movie name, and a year. For example,

Kishiro, Yukito		Battle Angel		2016
-----------------	--	--------------	--	------

- Goal:
  - Find all persons named in the file
  - Count the number of different persons named.
  - Ask if a particular person is named in the file
- The challenge in doing this is that many names appear multiple times.
- First solution: store names in a list. We'll start from the following code, posted on the Piazza in `lec15_find_names_start.py`, which is part of a Lecture 15 zip file.

```
imdb_file = input("Enter the name of the IMDB file ==> ").strip()
name_list = []
for line in open(imdb_file, encoding = "ISO-8859-1"):
    words = line.strip().split('|')
    name = words[0].strip()
```

and complete the code in class.

- The challenge is that we need to check that a name is not already in the list before adding it.
- You may access the data files and the starting code .py file from the Resources page of the Piazza site.

## 27.3 How To Test?

- The file `imdb_data.txt` has about 260K entries. How will we know our results are correct?
- Even if we restrict it to movies released in 2010-2012 (the file `imdb_2010-12.txt`), we still have 25K entries!
- We need to generate a smaller file with results we can test by hand
  - I have generated `hanks.txt` for you and will use it to test our program before testing on the larger files.

## 27.4 What Happens?

- Very slow on the large files because we need to scan through the list to see if a name is already there.
- We'll write a faster implementation based on Python *sets*.
- We'll start with the basics of sets.

## 27.5 Sets

- A Python set is an implementation of the mathematical notion of a set:
  - No order to the values (and therefore no indexing)
  - Contains no duplicates
  - Contains whatever type of values we wish; including values of different types.
- Python set methods are exactly what you would expect.
  - Each has a function call syntax and many have operator syntax in addition.

## 27.6 Set Methods

- Initialization comes from a list, a range, or from just `set()`:

```
>>> s1 = set()
>>> s1
set()
>>> s2 = set(range(0,11,2))
>>> s2
{0, 2, 4, 6, 8, 10}
>>> v = [4, 8, 4, 'hello', 32, 64, 'spam', 32, 256]
>>> s3 = set(v)
>>> s3
{32, 64, 4, 'spam', 8, 256, 'hello'}
```

- The actual methods are
  - `s.add(x)` — add an element if it is not already there
  - `s.clear()` — clear out the set, making it empty
  - `s1.difference(s2)` — create a new set with the values from `s1` that are not in `s2`.
    - \* Python also has an “operator syntax” for this:

```
s1 - s2
```

- `s1.intersection(s2)` — create a new set that contains only the values that are in **both** sets. Operator syntax:

```
s1 & s2
```

- `s1.union(s2)` — create a new set that contains values that are in either set. Operator syntax:

```
s1 | s2
```

- `s1.issubset(s2)` — are all elements of `s1` also in `s2`? Operator syntax:

```
s1 <= s2
```

- `s1.issuperset(s2)` — are all elements of `s2` also in `s1`? Operator syntax:

```
s1 >= s2
```

- `s1.symmetric_difference(s2)` — create a new set that contains values that are in `s1` or `s2` but **not in both**.

```
s1 ^ s2
```

- `x in s` - evaluates to `True` if the value associated with `x` is in set `s`.

- We will explore the intuitions behind these set operations by considering

- `s1` to be the set of actors in *comedies*,
- `s2` to be the set of actors in *action movies*

and then consider who is in the sets

```
s1 - s2
```

```
s1 & s2
```

```
s1 | s2
```

```
s1 ^ s2
```

## 27.7 Exercises

1. Sets should be relatively intuitive, so rather than demo them in class, we'll work through these as an exercise:

```
>>> s1 = set(range(0,10))
>>> s1

>>> s1.add(6)
>>> s1.add(10)

>>> s2 = set(range(4,20,2))
>>> s2
```

```
>>> s1 - s2

>>> s1 & s2

>>> s1 | s2

>>> s1 <= s2

>>> s3 = set(range(4,20,4))
>>> s3 <= s2
```

## 27.8 Back to Our Problem

- We'll modify our code to find the actors in the IMDB. The code is actually very simple and only requires a few set operations.

## 27.9 Side-by-Side Comparison of the Two Solutions

- Neither the set nor the list is ordered. We can fix this at the end by sorting.
  - The list can be sorted directly.
  - The set must be converted to a list first. The function `sorted` does this for us.
- What about speed? The set version is **MUCH FASTER** — to the point that the list version is essentially useless on a large data set.
  - We'll use some timings to demonstrate this quantitatively
  - We'll then explore why in the rest of this lecture.

## 27.10 Comparison of Running Times for Our Two Solutions

- List-based solution:
  - Each time before a name is added, the code — through the method `in` — scans through the entire list to decide if it is there.
  - Thus, the work done is proportional to the size of the list.
  - The overall running time is therefore roughly proportional to the square of the number of entries in the list (and the file).
  - Letting the mathematical variable  $N$  represent the length of the list, we write this more formally as  $O(N^2)$ , or “the order of  $N$  squared”
- Set-based code

- For sets, Python uses a technique called *hashing* to restrict the running time of the add method so that it is *independent of size of the set*.
  - \* The details of hashing are covered in CSCI 1200, Data Structures.
- The overall running time is therefore roughly proportional to the length of the set (and number of entries in the file).
- We write this as  $O(N)$ .
- We will discuss this type of analysis more later in the semester.
  - It is covered in much greater detail in Data Structures and again in Intro. to Algorithms.

## 27.11 Discussion

- Python largely hides the details of the containers — set and list in this case — and therefore it is hard to know which is more efficient and why.
- For programs applied to small problems involving small data sets, efficiency rarely matters.
- For longer programs and programs that work on larger data sets, efficiency does matter, sometimes tremendously. What do we do?
  - In some cases, we still use Python and choose the containers and operations that make the code most efficient.
  - In others, we must switch to programming languages, such as C++, that generate and use compiled code.

## 27.12 Summary

- Sets in Python realize the notion of a mathematical set, with all the associated operations.
- Operations can be used as method calls or, in many cases, operators.
- The combined core operations of finding if a value is in a set and adding it to the set are **much faster when using a set** than the corresponding operations using a list.
- We will continue to see examples of programming with sets when we work with dictionaries.

## 27.13 Extra Practice Problems

1. Write Python code that implements the following set functions using a combination of loops, the `in` operator, and the `add` function. In each case, `s1` and `s2` are sets and the function call should return a set.
  - (a) `union(s1,s2)`
  - (b) `intersection(s1,s2)`
  - (c) `symmetric_difference(s1,s2)`

