COMP10001

WORKSHOP #8

Shevon Mendis
shevonm@unimelb.edu.au

Libraries

- Definition: A collection of resources (data structures, functions, modules, subroutines, packages, etc)

- Library functions are stable and often optimised, so you're guaranteed that they will work correctly in all situations. Using them can save you time as you don't have to implement those functions by yourself

- Importing an entire library/ module:

```
import < library/ module >
```

```
from < library/ module > import *
```

- Importing specific functions from a library/ module

```
from < library/ module > import < function name>
```

Libraries

- Popular Python libraries:
  - Python Standard Library
  - PIL (Python Imaging Library)
  - Pandas (for data manipulation)
  - TensorFlow (for machine learning)
  - NumPy (for scientific/ mathematical computing)

defaultdicts

- A **`defaultdict`** works the same way as a normal dict, except that the keys are initialised with a default value. Additionally, a **`defaultdict`** won't raise a **`KeyError`** if the key doesn't exist- instead, it creates a new key with a default value and returns that default value

- Example Usage:

```
from collections import defaultdict as dd

def get_char_frequencies(text):

    char_frequencies = dd(int)
    for char in text:
        char_frequencies[char] += 1

    return char_frequencies
```

`list` Comprehensions

- Essentially syntactic sugar for performing list operations (that would otherwise require a loop) using one expression

- Format:

```
new_list = [< expr > for var in < iterable > < optional condition >]
```

`list` Comprehensions

- Can be used for:
  - Mapping
    - *"square all the numbers in a list"*

      ```
      nums_squared = [num ** 2 for num in nums]
      ```

    - *"map all the numbers to whether they're even or odd"*

      ```
      odd_or_even = ["odd" if num % 2 else "even" for num in nums]
      ```

  - Filtering
    - *"extract all the even numbers from the list into a new list"*

      ```
      nums_even = [num for num in nums if num % 2 == 0]
      ```

Iterators

- An iterator is essentially an object which is used to iterate over other objects (iterables)
- Functions used for iterators:
  - `iter(< iterable >)` : Constructs an iterator out of an iterable
  - `next(< iterator >)` : Returns the next element in the iterator;
    Throws an error if you've already reached the end
- Iterators are useful as we don't need to store the entire container object in memory to iterate over it- we only need to store the iterator object.
  - Could be used to optimise a program by reducing its memory usage

itertools

- Useful functions:
  - `cycle(`*iterable*`)`
    - Produces an iterator to cycle through a container, looping from the end back to the beginning infinitely
  - `product(`*iterables*`[, `*repeat*`])`
    - Produces a sequence containing elements created through taking the cartesian product of the iterables specified
  - `combinations(`*iterable*`, `*group_size*`)`
    - Produces a sequence of every possible combination of elements from the iterable
  - `permutations(`*iterable*`, `*group_size*`)`
    - Produces a sequence of every possible permutation of elements from the iterable
  - `groupby(`*iterable*`[, `*keyfunc*`])`
    - Produces an iterator that returns consecutive keys and groups from the iterable

Sequences vs Iterators

- **Sequences:**
  1. Have *random access* (you can access any element in the sequence, as many times as you like)
  2. No position tracking within the sequence
  3. You can use `len()` to calculate the length
  4. Must be finite
  5. You can traverse it many times
- **Iterators:**
  1. No *random access*
  2. Remembers where you are up to
  3. Cannot use `len()`
  4. Can be infinite
  5. You can only traverse it once, forwards.

(Shamelessly copied from: https://groklearning.com/learn/unimelb-comp10001-2020-s1/w15/7/?ignore_lc=true)

Recursion

- Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the **same** problem.
  In other words, recursion is where a function calls itself (with a subset of the original argument that was passed into it)

- A recursive function is made up of two parts:

  - **Base case:** The simplest version of the problem (this will be the stopping condition for recursion)
  - **Recursive case:** Where the function makes a call to itself, breaking down the problem into a smaller version

Recursion

- Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the **same** problem.
  In other words, recursion is where a function calls itself (with a subset of the original argument that was passed into it)

- A recursive function is made up of two parts:

  - **Base case:** The simplest version of the problem (this will be the stopping condition for recursion)
  - **Recursive case:** Where the function makes a call to itself, breaking down the problem into a smaller version