Lecture 9: Files. Recursive Functions.

Dr Simon D'Alfonso

School of Computing and Information Systems
Melbourne School of Engineering

Lecture 8 Challenges

- Write a function swap_dict(), which takes one dictionary as input and returns another dictionary resulting from swapping the keys and values around. For example, if the input is {'a':1, 'b':2}, the output would be {1:'a', 2:'b'}.
- Write a function cartesian_product(), which takes two sets, A and B, as inputs. The function then returns the Cartesian product (x) of A and B. For example, if A = {1,2} and B = {3,4}, then A x B = {(1,3), (1,4), (2,3), (2,4)}

Second Smallest - Sets

```
my_set = {6, 3, 7, 5, 9}
smallest = min(my_set)
my_set.remove(smallest)
second_smallest = min(my_set)
```

Remainder of Semester

- A2 will be released later today. Due at the end of Week 12.
- Final exam later in June. Plan to hold consultation sessions throughout June until the exam.
- After this week, the final two weeks of semester will be about going back over previous lecture material and looking at some exam questions.
- Nominate any lecture material/slides/topics that you would like to go over again: send me an email <dalfonso@unimelb.edu.au>, post in LMS Discussion Forum or make suggestions in the Zoom chat.

Today

1. Files – reading, writing, appending
2. Recursive functions

Why Files?

- Computer memory is volatile
  - RAM, cache, registers
  - Power OFF, it is all gone
  - Python program finishes, it is all gone
- Computers use disks (hard disks, floppy disks, CD-ROMs) for longer term storage
- Disks have filesystems
- Python can read from and write to files

Opening Files

A file is a linear sequence of data that is stored on persistent storage. The first step in using a file is to open it.

Default (read only):

```
file_object = open("stuff.txt")
```

To read from:

```
file_object = open("stuff.txt", "r")
```

To write to:

```
file_object = open("stuff.txt", "w")
```

To add data to (append):

```
file_object = open("stuff.txt", "a")
```

These are the core operations. A few further details and methods beyond the scope of this lecture can be found here: https://www.programiz.com/python-programming/file-operation

Reading from files

Reading data in from a (local) file:

- open() creates a pointer to file
- *for* iterates over lines via file pointer
- read() reads the entire file
- readlines() reads into a list of lines

```
filename = 'example_file.txt' #filepath
fp = open(filename) #a file pointer
for line in fp:
    print(line, end="")

text = open(filename).read()
lines = open(filename).readlines()
```

Closing a file

To process the data in a file, the following steps are performed:

1. Open the file
2. Read data from a file or write data to a file
3. Close the file (things can still work if you don't explicitly close the file, but in some cases, due to buffering, changes made to a file may not show until you close the file)

To close a file:

```
file_pointer.close()
```

Exercise 1

Given a file of tram stops and their routes, print a list of routes for each stop. Each line of the file has two comma-separated values, the stop and then the route. For example:

```
Tennis Centre,3
Tennis Centre,38
Aquarium,38
Docklands,3
```

With this sample, Route 38 stops at both the Tennis Centre and Aquarium. The Tennis Centre has two routes (3 and 38) passing through it.

Hint: split() method and rstrip() methods may be useful

Writing to Files

Writing to a file:

```
filename = 'my_file.txt'
text = open(filename ,'w')
text.write('This is a line')
text.close()
```

Appending to a file:

```
filename = 'my_file.txt'
#can't just use 'w' again because it creates a new
file again
text = open(filename ,'a')
text.write('This is another line')
text.close()
```

What is a CSV File?

- A common data format is CSV ("comma-separated values")
- CSV files store tabulated data (think spreadsheets and databases) as plain-text, separated by commas.
- So common, in fact, that there is a Python csv module that helps you read them.

```
#sample CSV with cars
year,make,model
1997,Ford,Falcon
2006,Honda,Odyssey
```

Parsing CSV files: Lists

We can parse CSV files using csv.reader from the csv library, which imports the data into a collection of lists:

```
import csv
filename = 'rainfall.csv'
for line in csv.reader(open(filename)):
    print(line)

#The first two lines of rainfall.csv
['Station', 'Year', 'Month', 'Day', 'Rain_mm']
['86035', '2019', '1', '3', '0']
…
```

Parsing CSV Files: Dictionaries

Use csv.DictReader from the csv library, which imports the data into a collection of dictionaries:

```
import csv
filename = 'rainfall.csv'
for line in csv.DictReader(open(filename)):
    print(line)
```

```
{'Station': '86035', 'Year': '2019', 'Month': '1', 'Day': '3', 'Rain_mm': '0'}
{'Station': '86035', 'Year': '2019', 'Month': '1', 'Day': '4', 'Rain_mm': '0'}
…
```

As the DictReader method returns each row as a dictionary you can access data from a row by the column headings, for example row['Day'].

Exercise 2

Write code that opens rainfall.csv and prints out the maximum rainfall record value

Writing to CSV Files

To write to CSV files you will need to format your data into a 2D way (such as list of lists):

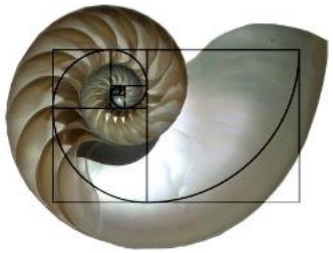
```
import csv


data_2d = [['header1', 'header2', 'header3'], [1, 2,
3], [4, 5, 6]]
csv_file = open("2d-data.csv", "w", newline='')
writer = csv.writer(csv_file)
writer.writerows(data_2d)
csv_file.close()
```

Exercise 3

Write some code to return the 3 least used alphabetic characters from a string variable named *text*. Ties should be broken alphabetically; for example, if ‘a’ and ‘b’ appeared the same number of times, ‘a’ would be returned before ‘b’.

Recursive Functions

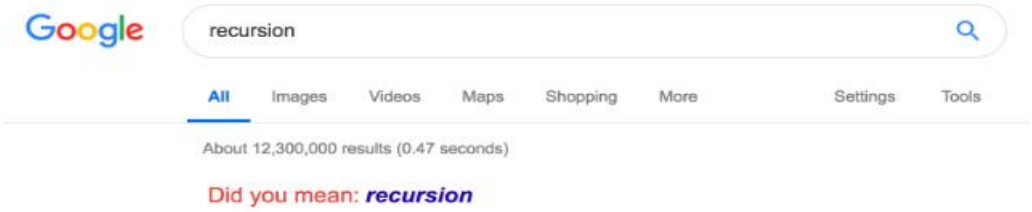
Recursion is the concept of things being made up of smaller versions of themselves.


Examples of recursion in nature[1]

[1] https://medium.com/code-zen/recursion-demystified-24867f045c62

When a function calls itself, we call this function a *recursive function*.



Factorials

The factorial of integer *x* (*x*!) is the product of *x* and all positive numbers below it:

- 3! = 3 x 2 x 1 = 6
- 5! = 5 x 4 x 3 x 2 x 1 = 120

Factorials

- Can you calculate 5! if you only know 4! ?

  A: Yes, 5! = 5 x 4!
- Can you calculate 4! if you only know 3! ?

  A: Yes, 4! = 4 x 3!
- Can you calculate n! if you only know (n - 1)! ?

  A: Yes, n! = n x (n - 1)!

This means that we can define a factorial **recursively**: n! = n x (n-1)!

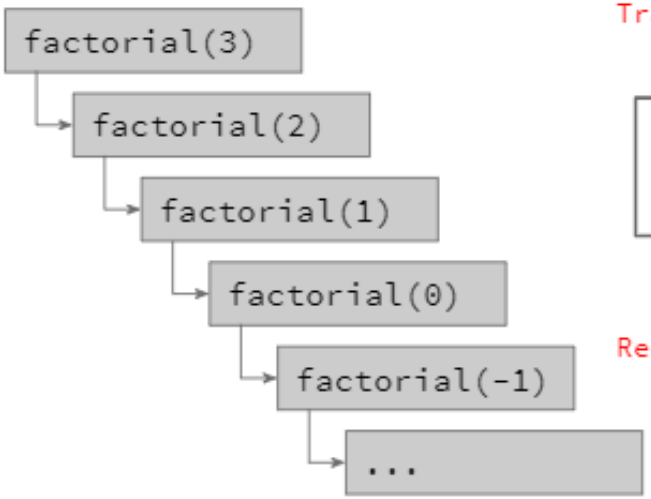
Writing Recursive Functions

```
def factorial(n):
    return n * factorial(n - 1)
```

What happens if we call factorial(3)?

Writing Recursive Functions

```
1  >>> def factorial(n):
2  ...     return n * factorial(n - 1)
```




```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    factorial(3)
  File "<stdin>", line 2, in factorial
    return n * factorial(n - 1)
  [Previous line repeated 991 more times]
RecursionError: maximum recursion depth exceeded
```

- How do we prevent **infinite recursion**?

  A: We need a **base case**, 1! = 1

Base case(s): the simplest form of the problem that has a trivial solution.

Recursive Factorial Function

```
def factorial(n):
    if n == 1:
        return 1 #base case

    #recursive case
    return n * factorial(n - 1)
```

Recursion versus Iteration

Note that recursive solutions, such as the one we have just looked at, have equivalent solutions which use iteration, and the loop constructs we are already familiar with:

```
def factorial_iteration(n):
    #base case
    factorial = 1

    #loop down from the input number until 1
    for number in range(n, 1, -1):
        #multiply curr num with product of prev numbers
        factorial = factorial * number

    return factorial
```

Recursion versus Iteration

- Iteration is repeated execution of a set of statements while Recursion is a way of programming in which a function calls itself until it reaches some satisfactory condition.
- Recursion version of a solution generally more compact and ‘elegant’ in terms of code size and structure.
- Iteration version though perhaps less tricky and more straightforward.
- Since Recursion involves repeated function calls, it invokes any extra time/memory resource costs associated with these repeated calls.

Exercise 4

- The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Starting with 0 and 1, the next number is found by adding up the two numbers before it.
- Write a *recursive* function that accepts one input argument, an integer *n*, and returns the n^{th} Fibonacci number. If n = 0 then return 0, the 0^{th} number, if n = 1 then return 1, the 1^{st} number, and if n = 2, then return 1, which is also the 2^{nd} number.

Exercise 4 Solution (Iteration)

```
def fibonacci(n):

    fib_lst = [0, 1]
    if n in fib_lst:
        return n

    for i in range(2, n+1):
        next_element = fib_lst[i-1] + fib_lst[i-2]
        fib_lst.append(next_element)

    return fib_lst[-1]
```

Exercise 4 Solution (Recursion)

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Summary

Today we covered:

- Files
- Recursive functions

Lecture 9 Challenges

- Write a function file_compare(file1, file2), which receives the path names of two files and checks whether they have identical content. If they do not have identical content, then append the content of file2 to file1.
- Write a *recursive* function sum_nums(n) that accepts an integer *n* and returns the sum of integers from 1 to n.
- Write a *recursive* function reverse(string), that accepts a string input and returns that string in reverse order.

Lecture Identification and Acknowledgement

Coordinator / Lecturer: Simon D'Alfonso

Semester: Semester 1, 2022

These slides include materials from 2020 - 2021 instances of COMP90059 run by Kylie McColl or Wally Smith