	When to use data structures	their methods + how to create them	
String	For sequence processing (e.g., manipulate integers: list(string) for swaps, reverse..). Check if a character is one of a string. Can index/slice into a string for access BUT cannot assign to a specific index/slice of a string (use list(string) to work with list of characters instead).	"delimiter".join(list of strings), string.split(), string.splitlines(), string.strip(), string[::-1]. Others in the documentation. Example: num='123', digits = list(num) → ['1', '2', '3'] → num = ''.join(digits).	
List	To store data to loop over, which can be nested (e.g., a list of tuples). Simplest sequence if standalone elements and expectation that it will get updated (not tuple) + we don't care about or want to keep if duplicate elements (not set) to count freqs for example.	s.append(), s.extend(), s.insert(idx, ele), s.pop(optional idx), s.remove(ele) for 1st occur., s.reverse(), reversed(s), sorted(s, reverse=True), s.clear(), del s[idx], s.copy(), s[::-1], s.count(ele). list.sort(), sorted(list) → doesn't make sense to sort unordered set	
Tuple	Store immutable collection of data, like key-value pairs.	t[idx], t[::-1] BUT no assign.	(ele,), (ele1, ele2), tuple(iterable)
Set + frozenset	Mutable collection of unique elements. Useful to check if duplicates. If need set of sets, use frozensets within set.	Same as list, s.add(ele), s.remove/discard(ele), s.pop(), s.clear(), s[idx]	{ele1, ele2}, set(), set(iterable]), set(frozenset(), frozenset()}, set comprehension
Dict	Store collection of pairs of data (need unique keys). Useful for storing counts/freqs; then, can do sort() on dict.items() with key=lambda item:item[0] or item[1] to sort by key or value for example.	d.keys(), d.values(), d.items(), d.copy(), d.clear(), d.get(k, def), d.pop(k), d.popitem(k), reversed(d), d.setdefault(k, def), key in d, d.update(d2)	{key1:val1, key2:val2}, dict(), dict([tuple1, tuple2]), dictionary comprehension

Can **sort by multiple conditions to avoid draws**: e.g., sorted(my_dict.items(), key=lambda item: (-item[1], len(item[0])) if count values and string keys, and sorting by descending order (reversed) for the 1st sorting condition, and by ascending (the default) order for the 2nd sorting condition.

List compr: [i for i in range(10) if X], **Set compr**: {i for i in list if X}, **Dict compr**: {key:val for (key,val) in dict2 if X} to get filtered dict from existing dict2

OOP + inheritance

```
class Enemy(Character):
    def __init__(self, name, health=50, strength=30, defence=20, evilness=50):
        super().__init__(name, health, strength, defence)
        self.evilness = 50
    def __str__(self):
        return f"{self.name} is an Enemy (health: {self.health},"\
            f"strength: {self.strength}, defence: {self.defence},"\
            f"evilness: {self.evilness})"
```

> instance methods (self + no fn decorator)
> class methods (cls + @classmethod)
> static methods (no implicit 1st arg + @staticmethod) Can be called on the class or its instance: Class.the_method() or Class().the_method())

Exception handling

Try/Except/Else/Finally: Lesson 7 Chap 7 [7.3]
Raising custom Exception: Lesson 10 Chapter 9 [9.3]
Built-in Exceptions: Lesson 10 Chapter 9 [9.2]

Reading/Writing files

```
with open("employees_detail.txt") as textfile:
    for line in textfile:
        stripped_line= line.strip()

assistants = ["Harry", "Joe", "Luca", "William"]
with open("assistants.txt", "w") as file:
    file.write(f"{len(assistants)} great 
assistants:\n")
    for assistant in assistants:
        file.write(f"{assistant}\n")
```

```
with open("input.json", "r") as jsonfile:
    data = json.load(jsonfile)
json_string = '[{'id':2, 'v':'A'}, {'id':6, 'v':'B'}]'
data = json.loads(json_string)

data = {"course": "Intro2ML", "term": 1}
with open("output.json", "w") as jsonfile:
    json.dump(data, jsonfile)
json_string = json.dumps(data)
```

```
courses = {1: {"lecturer": "JW", "title": "Python"},
2: {"lecturer": "RC", "title": "SymbAI"}}

with open("courses.pkl", "wb") as file:
    pickle.dump(courses, file)

with open("courses.pkl", "rb") as file:
    pickled_courses = pickle.load(file)
```

Functional Programming

- **Lambda fn**: used when specifying a key (i.e., what we consider when performing an action like sorting an iterable) on how to sort/min/max, or when using map() or filter(). → lambda x: x+2 or add_2 = lambda x: x+2 then you can use it as add2(3) → 5
- **map()**: alternative to list comprehension → map(ele-wise function, list). **See documentation in Built-in Functions for both map() and filter()**
- **filter()**: alternative to list comprehension with if statement when used like this → list(filter(lambda fn to specify truth/filter, the iterable)).

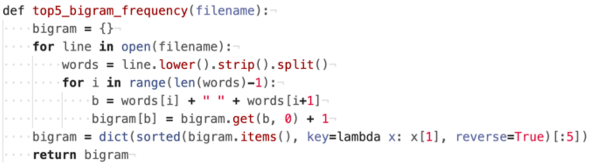
Useful features

- swap 2 elements: sequence[i], sequence[i+1] = sequence[i+1], sequence[i]. This sequence could be a list of characters from a string (e.g., of digits)
- reverse a sequence using [::-1], useful for swaps with left and right pointers (i.e., for loop over left to right and nested reverse for loop over right to left)
- enumerate() to retrieve/store coordinates/position/index of elements in the sequence(s)
- sequence.insert(index, element) to insert element before the index given in the sequence
- get list of numbers with range: numbers = list(range(0, 20))
- zip/unzip: listA = [1, 2, 3, 4], listB = ['a', 'b', 'c', 'd'], zl = zip(listA, listB), list(zl) → [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')], get original lists using listA, listB = zip(*zl)
- **dict methods**:

```
new = {}
for (key, value) in data:
    # key might exist already
    group = new.setdefault(key, [])
    group.append(value)
```

```
# instead of doing this way:
new = {}
for (key, value) in data:
    if key in new:
        new[key].append( value )
    else:
        new[key] = [value]
```

d[key] = d.get(key, 0)+1 ⇔ if key in d, d[key]+=1; else, d[key]=0



```
def top5_bigram_frequency(filename):
    bigram = {}
    for line in open(filename):
        words = line.lower().strip().split()
        for i in range(len(words)-1):
            b = words[i] + " " + words[i+1]
            bigram[b] = bigram.get(b, 0) + 1
    bigram = dict(sorted(bigram.items(), key=lambda x: x[1], reverse=True)[:5])
    return bigram
```

- use pprint.pp(datastructure) from pprint module for more readable nested list/dictionary
- use lists = [[] for i in range(3)] instead of lists = [[]] * 3 to create [[], [], []] with independent sublists (modifying one does NOT modify the others)
- any(), all(): on list of boolean values to check if any/all validity conditions are true (satisfied). Condition could be validity_var or statement like i%2==0
- map, filter, lambda functions. Lambda fn for specifying the key/how to sort/max/min/map/filter()
- in for-loops, continue (*thank u, next*) and break (*get out*) statements, or return (*get out and return smth*) if an if-condition is satisfied
- sum() with list comprehension of boolean expressions that may be if-conditioned on smth or just add a 2nd condition with "and" (see q10 in Exercises):
  sum([(sequence1[i] == sequence2[i] and sequence1[i] == 1) for i in range(len(dict1[7]))])
- copy a data structure to avoid changing it in-place (esp. useful for swappings) using y = x.copy() or y = x[:] or y = list(x)
- list/set/dict comprehensions (can do some filtering on existing data structures)
- difference between "x is y" and "x == y". Membership operator "in". Check membership to a built-in or custom class with isinstance(instance, class)
- can check membership in a string (not just with a list): e.g., if char in '123456789.'
- **Can't assign** to indices of a string, so initialise an empty string before looping and update it inside loop with string += char. No assig. to a tuple and a set
- **the interesting functions are in the Built-in Functions section of the documentation!!!**

Tips (also based on Exercises)

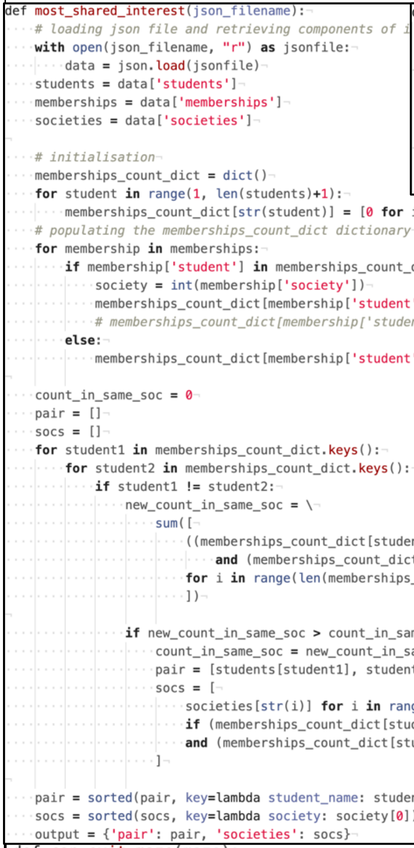
- default argument for a function = None, then in the function: if arg == None: arg = []
- and / or when **multiple conditions in the same if**: generally, I use "or" when really it's "and"
- don't forget to use range() with len() when for-looping
- if **checking for validity** based on satisfying multiple conditions at the same time, then do return False as soon as one condition is not satisfied. At the end, return True (because then, it means that every condition above has been True aka has not returned False)
- printing a string → output without " ". Returning a string → output with " "
- don't forget to check the **2 diagonals** in board/matrix-like questions! (e.g. q16 from Exercises):

```
first_diag_magic_num = sum( [ matrix[i][i] for i in range(len(matrix)) ] )
second_diag_magic_num = sum( [ matrix[j][i] for i, j in zip( range(len(matrix)), range(len(matrix)-1, -1, -1) ) ] )
```

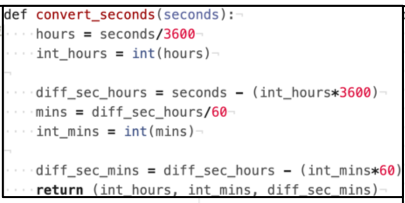
- options 1/2 easier than option 3 in board-like questions (be **careful with indexing**: print to see if correct subslices of board are being accessed)

for row in board: for cell in row:	for r, row in enumerate(board): for c, cell in enumerate(row):	for r in range(len(board)): for c in range(len(board[r])):

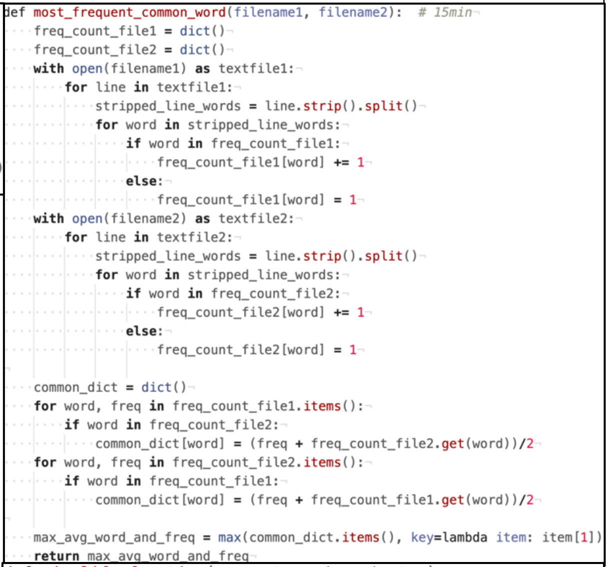
Check board assumptions!!!



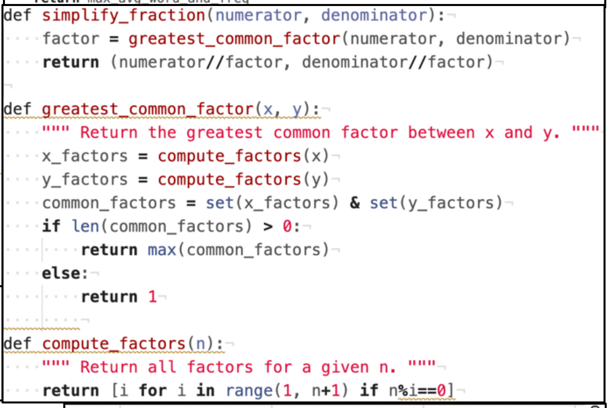
```
def most_shared_interest(json_filename):
    # loading json file and retrieving components of i
    with open(json_filename, "r") as jsonfile:
        data = json.load(jsonfile)
    students = data['students']
    memberships = data['memberships']
    societies = data['societies']

    # initialisation
    memberships_count_dict = dict()
    for student in range(1, len(students)+1):
        memberships_count_dict[str(student)] = [0 for i in range(len(societies))]
    # populating the memberships_count_dict dictionary
    for membership in memberships:
        if membership['student'] in memberships_count_dict:
            society = int(membership['society'])
            memberships_count_dict[membership['student']][society-1] = 1
            # memberships_count_dict[membership['student']].insert(society-1, 1)
        else:
            memberships_count_dict[membership['student']] = []

    count_in_same_soc = 0
    pair = []
    socs = []
    for student1 in memberships_count_dict.keys():
        for student2 in memberships_count_dict.keys():
            if student1 != student2:
                new_count_in_same_soc = \
                    sum([
                        ((memberships_count_dict[student1][i] == memberships_count_dict[student2][i])
                            and (memberships_count_dict[student1][i] == 1))
                        for i in range(len(memberships_count_dict[student1]))
                        ])

                if new_count_in_same_soc > count_in_same_soc:
                    count_in_same_soc = new_count_in_same_soc
                    pair = [students[student1], students[student2]]
                    socs = [
                        societies[str(i)] for i in range(1, len(memberships_count_dict[student1])+1)
                        if (memberships_count_dict[student2][i-1] == memberships_count_dict[student1][i-1])
                        and (memberships_count_dict[student2][i-1] == 1)
                    ]

    pair = sorted(pair, key=lambda student_name: student_name[0])
    socs = sorted(socs, key=lambda society: society[0])
    output = {'pair': pair, 'societies': socs}
```



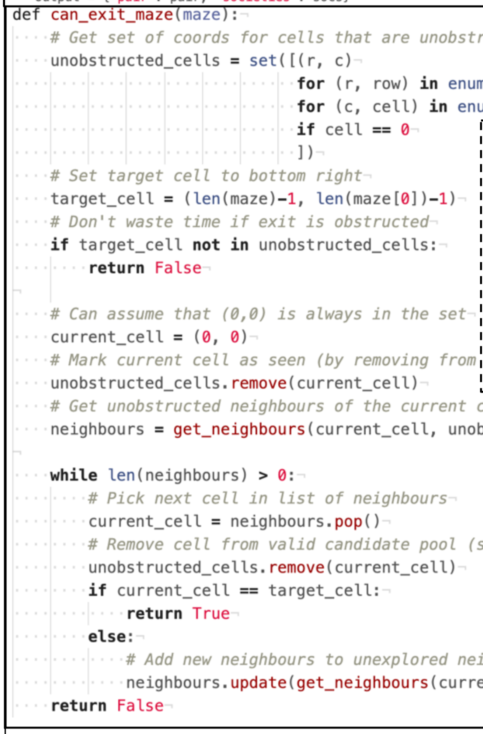
```
def convert_seconds(seconds):
    hours = seconds/3600
    int_hours = int(hours)

    diff_sec_hours = seconds - (int_hours*3600)
    mins = diff_sec_hours/60
    int_mins = int(mins)

    diff_sec_mins = diff_sec_hours - (int_mins*60)
    return (int_hours, int_mins, diff_sec_mins)
```



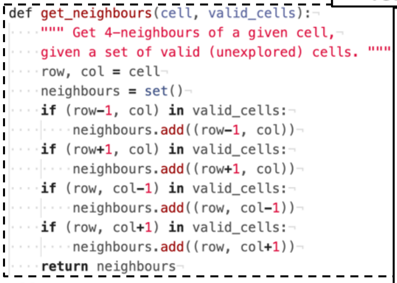
```
def most_frequent_common_word(filename1, filename2):  # 15min
    freq_count_file1 = dict()
    freq_count_file2 = dict()
    with open(filename1) as textfile1:
        for line in textfile1:
            stripped_line_words = line.strip().split()
            for word in stripped_line_words:
                if word in freq_count_file1:
                    freq_count_file1[word] += 1
                else:
                    freq_count_file1[word] = 1
    with open(filename2) as textfile2:
        for line in textfile2:
            stripped_line_words = line.strip().split()
            for word in stripped_line_words:
                if word in freq_count_file2:
                    freq_count_file2[word] += 1
                else:
                    freq_count_file2[word] = 1

    common_dict = dict()
    for word, freq in freq_count_file1.items():
        if word in freq_count_file2:
            common_dict[word] = (freq + freq_count_file2.get(word))/2
    for word, freq in freq_count_file2.items():
        if word in freq_count_file1:
            common_dict[word] = (freq + freq_count_file1.get(word))/2

    max_avg_word_and_freq = max(common_dict.items(), key=lambda item: item[1])
    return max_avg_word_and_freq
```



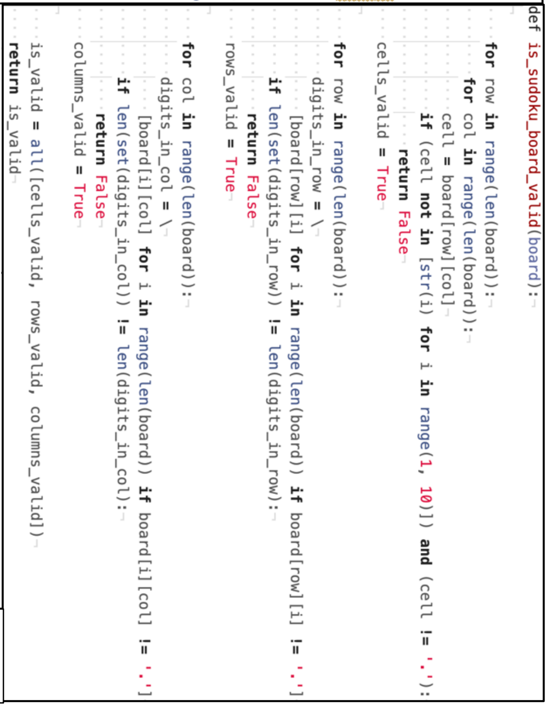
```
def simplify_fraction(numerator, denominator):
    factor = greatest_common_factor(numerator, denominator)
    return (numerator//factor, denominator//factor)

def greatest_common_factor(x, y):
    """ Return the greatest common factor between x and y. """
    x_factors = compute_factors(x)
    y_factors = compute_factors(y)
    common_factors = set(x_factors) & set(y_factors)
    if len(common_factors) > 0:
        return max(common_factors)
    else:
        return 1

def compute_factors(n):
    """ Return all factors for a given n. """
    return [i for i in range(1, n+1) if n%i==0]
```



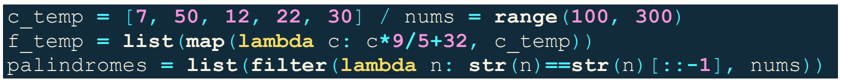
```
def can_exit_maze(maze):
    # Get set of coords for cells that are unobstructed
    unobstructed_cells = set([(r, c)
                              for (r, row) in enumerate(maze)
                              for (c, cell) in enumerate(row)
                              if cell == 0
                              ])
    # Set target cell to bottom right
    target_cell = (len(maze)-1, len(maze[0])-1)
    # Don't waste time if exit is obstructed
    if target_cell not in unobstructed_cells:
        return False

    # Can assume that (0,0) is always in the set
    current_cell = (0, 0)
    # Mark current cell as seen (by removing from
    unobstructed_cells.remove(current_cell)
    # Get unobstructed neighbours of the current cell
    neighbours = get_neighbours(current_cell, unobstructed_cells)

    while len(neighbours) > 0:
        # Pick next cell in list of neighbours
        current_cell = neighbours.pop()
        # Remove cell from valid candidate pool (since already seen)
        unobstructed_cells.remove(current_cell)
        if current_cell == target_cell:
            return True
        else:
            # Add new neighbours to unexplored neighbour set
            neighbours.update(get_neighbours(current_cell, unobstructed_cells))
    return False
```



```
def get_neighbours(cell, valid_cells):
    """ Get 4-neighbours of a given cell,
    given a set of valid (unexplored) cells. """
    row, col = cell
    neighbours = set()
    if (row-1, col) in valid_cells:
        neighbours.add((row-1, col))
    if (row+1, col) in valid_cells:
        neighbours.add((row+1, col))
    if (row, col-1) in valid_cells:
        neighbours.add((row, col-1))
    if (row, col+1) in valid_cells:
        neighbours.add((row, col+1))
    return neighbours
```



```
def is_sudoku_board_valid(board):
    for row in range(len(board)):
        for col in range(len(board[row])):
            cell = board[row][col]
            if (cell not in [str(i) for i in range(1, 10)]) and (cell != '.'):
                return False
    cells_valid = True

    for row in range(len(board)):
        digits_in_row = \
            [board[row][i] for i in range(len(board)) if board[row][i] != '.']
        if len(set(digits_in_row)) != len(digits_in_row):
            return False
    rows_valid = True

    for col in range(len(board)):
        digits_in_col = \
            [board[i][col] for i in range(len(board)) if board[i][col] != '.']
        if len(set(digits_in_col)) != len(digits_in_col):
            return False
    columns_valid = True

    is_valid = all([cells_valid, rows_valid, columns_valid])
    return is_valid
```



```
c_temp = [7, 50, 12, 22, 30] / nums = range(100, 300)
f_temp = list(map(lambda c: c*9/5+32, c_temp))
palindromes = list(filter(lambda n: str(n)==str(n)[::-1], nums))
```