## Question 1. [7 marks]

### Part (a) [3 marks]

Consider this program:

```
L = [4, 10, 8]
x = L.sort()
L.append(20)
L2 = L[1:]
```

Fill in the Python Shell output **after** this program has executed.

```
>>> x
```

None

```
>>> L
```

[4, 8, 10, 20]

```
>>> id(L) == id(L2)
```

False

### Part (b) [4 marks]

Consider this program in file `words.py`:

```
def repeat_word(word, num_times):
    """ (str, int) -> str
    """

    print(__name__)
    word = word * num_times
    print('Repeated word is:', word)
    return word

if __name__ == '__main__':
    word = 'Yes'
    print('Original word is:', word)
    repeat_word(word, 3)
    print('New word is:', word)
    word = repeat_word(word, 2) + '!'
    print('New word is:', word)
```

Write what this program prints when you run it. Write one line per box. There are more boxes than you need; leave unused ones blank.

**Solution:**

```
Original word is: Yes
__main__
Repeated word is: YesYesYes
New word is: Yes
__main__
Repeated word is: YesYes
New word is: YesYes!
```

## Question 2. [5 marks]

Your instructor was in a hurry when typing the following expressions, so unfortunately they will all result in an error and will not be evaluated. Explain the errors in the table below.

| **Expression** | **Briefly explain why an error occurs** |
|---|---|
| `["Hello", "there!"].split()` | split is a string method, it cannot be used with lists |
| `"April" + 16` | cannot concatenate a string with an int |
| `["Temperature", " is "].extend(10)` | list method extend expects an iterable |
| `degrees_to_season = {[25] : "summer", [-15] : "winter"}` | dictionary keys must be immutable (cannot use a list) |
| `word = "computer"` `word[8]` | index error (max is 7) |

## Question 3. [5 marks]

```
def return_mix(num, word):
    """(int, str) -> object

    Precondition: word contains at least 3 characters.
    """

    if word[2] == 'r':
        return 2 * num
    elif num > 10:
        return (word + word)
    elif word[1:3] == "ab":
        return (2 == 3)
```

Write the return value and type for each of the following function calls:

| **Function Call** | **Return Value** | **Return Type** |
|---|---|---|
| `return_mix(5, "Canada")` | None | NoneType |
| `return_mix(12, "Toronto")` | 24 | int |
| `return_mix(16, "Canada")` | "CanadaCanada" | str |
| `return_mix(-2, "cabinet")` | False | bool |
| `return_mix(11, return_mix(11, "Can"))` | "CanCanCanCan" | str |

## Question 4. [4 marks]

You decide to save all your passwords in files, one password per file. You then decide you want to update your 6-character UTORid password, but you don't remember which of your password files you saved it in. Luckily, this is the only 6-character password you have so you decide to write a function to find the password and change it.

Your code runs, but the function's body contains **four** bugs which will result in the function execution not matching its docstring description. In the space provided, rewrite the buggy lines with the bugs fixed. Do not add or remove any lines.

```
def change_password(files_list, new_password):
    """ (list of str, str) -> str

    Preconditions:
    (1) files_list is not empty.
    (2) Each file in files_list contains one single-line password.
    (3) All files in files_list contain passwords of different lengths.
    (4) Your 6-character password is guaranteed to be in one of these files.

    In the appropriate file from files_list, replace your 6-character password
    with new_password, and return your old password.
    """
```

| #Code | If you think a given line contains a bug, rewrite it. Otherwise leave it blank. |
|---|---|
| `found = True` | `found = False` |
| `for filename in files_list[:-1]:` | `for filename in files_list:` |
| `    passwd_file = open(filename, 'r')` | |
| `    first_line = passwd_file.read()` | |
| `    password = first_line.strip("2")` | `password = first_line.strip()` |
| `    if password == 6:` | `if len(password) == 6` |
| `        passwd_file.close()` | |
| `        passwd_file = open(filename, 'w')` | |
| `        passwd_file.write(new_password)` | |
| `        found = True` | |
| `    passwd_file.close()` | |
| `    if found:  # This is correct. Do not modify.` | |
| `        return password` | |

## Question 5. [7 MARKS]

**Part (a)** [6 MARKS] Write the body of the following function according to its docstring description.

```
def numeric_phone(alphanumeric_phone, mapping):
    """ (str, list of str) -> int

    Preconditions:
    - len(mapping) <= 10
    - alphanumeric_phone contains only digits and uppercase letters
    - each letter in alphanumeric_phone is guaranteed to appear in mapping once
    - mapping may contain letters not in alphanumeric_phone

    Return a numeric phone number that corresponds to alphanumeric_phone.
    Each letter from alphanumeric_phone is replaced with the index of the item from
    mapping that contains the letter. Digits are not replaced.

    >>> numeric_phone('416310BELL', ['ABC', 'DEF', 'GHI', 'JKL', 'NO', 'PQRS', 'TUV', 'WXYZ'])
    4163100133
    """

    phone_number = ''
    for ch in alphanumeric_phone:
        if ch.isdigit():
            phone_number += ch
        else:
           for i in range(len(mapping)):
                if ch in mapping[i]:
                    phone_number += str(i)

    return int(phone_number)
```

**Part (b)** [1 MARK] Instead of representing the mapping as a `list of str`, representing it as which of the following types would make implementing this function easier? (circle one)

(a) `list of list of str` (b) `dict of {int: str}` [✓ (c) `dict of {str: int}`]

## Question 6. [6 marks]

Two friends like to send messages to each other, but they don't want anyone else to read the messages. To keep the messages private, they change each letter in the original message to a different letter of the alphabet. For every letter, they both know which letter will be substituted for it, which is called an *encoding*. The message will contain only lowercase letters. It will not contain whitespace, digits, or punctuation.

The receiver of a secret message doesn't want to convert it back to the original message by hand. Instead, that person asks you to write a function to do it.

Following the function design recipe, define a function that given a secret message and the encoding used, returns the original message. Choose a meaningful function name, and add a docstring that includes the type contract, the description, and two examples that return different values. You must choose the types to use to represent the data, and you will be marked not only on the correctness of the function, but also on your choice of data structures.

```
def decode(secret_message, encoding):
    """ (str, dict of {str: str}) -> str

    Return a decoded version of secret_message using the mapping of secret character to
    actual character in encoding.

    >>> encoding = {'x': 'a', 't': 'n', 'n': 'e', 'e': 's', 'r': 'w', 'i': 'r'}
    >>> decode('xternie', encoding)
    'answers'
    >>> decode('abba', {'a': 'o', 'b': 't'})
    'otto'
    """

    decoded = ''
    for ch in secret_message:
     decoded += encoding[ch]

    return decoded
```

## Question 7. [11 marks]

This question has you write the bodies of two functions. Complete each function according to its docstring.

### Part (a) [6 marks]

Note: you will most likely not need all of the space on this page.

```
def count_letter_case(L):
    """ (list of list of str) -> list of tuple of int

    Precondition: each str in L is non-empty and contains only alphabetic characters

    Count the number of words in each sublist of L that start with lowercase and uppercase
    letters. Return a new list where each element is a two-item tuple in which
    the first item is the number of words in the list at the corresponding index of L
    that start with a lowercase letter and the second item is the number of words in the
    list at the corresponding index of L that start with an uppercase letter.

    >>> count_letter_case([['apple', 'Banana'], ['PEAR'], [], ['PEACH', 'apRICot', 'plum']])
    [(1, 1), (0, 1), (0, 0), (2, 1)]
    """

    result = []

    for lst in L:
        num_lower = 0
        num_upper = 0
        for word in lst:
            if word[0].isupper():
                num_upper += 1
            else:
                num_lower +=1
        result.append((num_lower, num_upper))

    return result
```

**Part (b)** [3 MARKS] Complete the function body according to its docstring description. You will be marked only on the parts that need to be different from (a).

```
def count_letter_case_mutate(L):
    """ (list of list of str) -> NoneType

    Precondition: each str in L is non-empty and contains only alphabetic characters

    Replace each item in L with a two-item tuple in which the first item is
    the number of words in the list at the corresponding index of L that start with
    a lowercase letter and the second item is the number of words in the list at the
    corresponding index of L that start with an uppercase letter

    >>> data = [['apple', 'Banana'], ['PeAr'], [], ['PEACH', 'apRICot', 'plum']]
    >>> count_letter_case_mutate(data)
    >>> data
    [(1, 1), (0, 1), (0, 0), (2, 1)]
    """

    for i in range(len(L)):
        num_lower = 0
        num_upper = 0
        for word in L[i]:
            if word[0].isupper():
                num_upper += 1
            else:
                num_lower +=1
        L[i] = (num_lower, num_upper)
```

**Part (c)** [2 MARKS]

You almost certainly have duplicate code between your two functions. Write a helper function that you *could* have used to eliminate that duplicate code. Do not rewrite your functions for parts (a) and (b). It is **not** necessary to write a docstring.

```
def counter(L):
    num_lower = 0
    num_upper = 0
    for word in L:
        if word[0].isupper():
            num_upper += 1
        else:
            num_lower +=1
    return (num_lower, num_upper)
```

## Question 8. [6 MARKS]

Consider this code:

```
def sort_guesses(guess_list, answer):
    """ (list of float, float) -> dict of {str: list of float}

    Return a new dictionary where each item in guess_list appears as an item
    in one of the dictionary's values lists:
    - each item less than answer is in a list associated with key 'low',
    - each item equal to answer is in a list associated with key 'correct', and
    - each item greater than answer is in a list associated with key 'high'.
    If there are no items associated with one of 'low', 'correct', or 'high',
    then that string should not appear as a key in the new dictionary.
    """
```

In the table below, we have outlined two test cases for `sort_guesses`. Add six more test cases chosen to test the function thoroughly.

| **Test Case Description** | `guess_list` | `answer` | **Return Value** |
|---|---|---|---|
| no guesses | `[]` | `5.5` | `{}` |
| one low guess | `[4.7]` | `6.0` | `{'low': [4.7]}` |
| one correct guess | `[6.0]` | `6.0` | `{'correct': [6.0]}` |
| one high guess | `[7.5]` | `6.0` | `{'high': [7.5]}` |
| multiple low guesses | `[1.3, 1.4, 1.5]` | `2.0` | `{'low': [1.3, 1.4, 1.5]}` |
| multiple correct guesses | `[1.5, 1.5, 1.5]` | `1.5` | `{'correct': [1.5, 1.5, 1.5]}` |
| multiple high guesses | `[1.5, 1.6, 1.5]` | `1.3` | `{'correct': [1.5, 1.6, 1.5]}` |
| mix of multiple low, correct, high guesses | `[1.3, 1.2, 1.4, 1.4, 1.5, 1.6 1.5, 1.6]` | `1.4` | `{'low': [1.3, 1.2], 'correct': [1.4, 1.4], 'high': [1.5, 1.6]}` |

## Question 9. [9 marks]

Write the body of the following function according to its docstring description. Do not use methods `read` or `readlines`, and do not use `for` loops. Instead, use `while` and method `readline`.

```
def populate_dictionary(definition_file):
    """ (file open for reading) -> dict of {str: list of str}

    Preconditions:
      - definition_file contains 0 or more entries with the following format:
        - 1 line containing a word
        - 0 or more lines with a definition of that word (one definition per line)
      - there will be one blank line between entries

    Return a dictionary where each key is a word and each value is the list of
    definitions of that word from definition_file.  Even if a word has 0 definitions,
    it should appear as a key in the dictionary.
    """

    word_to_definitions = {}
    word = definition_file.readline().strip()

    while(word != ''):
        word_to_definitions[word] = []
        definition = definition_file.readline().strip()
        while(definition != ''):
            word_to_definitions[word].append(definition)
            definition = definition_file.readline().strip()
        word = definition_file.readline().strip()

    return word_to_definitions
```

## Question 10. [5 MARKS]

**Part (a)** [1 MARK] The list below is shown after each pass of a sorting algorithm.

```
['M', 'A', 'D', 'E', 'B', 'F', 'C'] # initial list

['A', 'D', 'E', 'B', 'F', 'C', 'M'] # after one pass
['A', 'D', 'B', 'E', 'C', 'F', 'M'] # after two
['A', 'B', 'D', 'C', 'E', 'F', 'M'] # after three
['A', 'B', 'C', 'D', 'E', 'F', 'M'] # after four
```

Which sorting algorithm is being executed? (circle one)

(a) bubble sort

(b) selection sort

(c) insertion sort

**Solution: (b) bubble sort**

**Part (b)** [1 MARK] The list below is shown after each pass of a sorting algorithm.

```
['M', 'A', 'D', 'E', 'B', 'F', 'C'] # initial list

['A', 'M', 'D', 'E', 'B', 'F', 'C'] # after one pass
['A', 'B', 'D', 'E', 'M', 'F', 'C'] # after two
['A', 'B', 'C', 'E', 'M', 'F', 'D'] # after three
['A', 'B', 'C', 'D', 'M', 'F', 'E'] # after four
['A', 'B', 'C', 'D', 'E', 'F', 'M'] # after five
```

Which sorting algorithm is being executed? (circle one)

(a) bubble sort

(b) selection sort

(c) insertion sort

**Solution: (c) selection sort**

**Part (c)** [1 MARK]

List `[6, 5, 2, 3, 7, 1, 4]` is being sorted using **insertion sort**. Fill in the blanks to show the list after the next two passes.

| | |
|---|---|
| After one pass: | `[6, 5, 2, 3, 7, 1, 4]` |
| After two passes: | `[5, 6, 2, 3, 7, 1, 4]` |
| After three passes: | `[2, 5, 6, 3, 7, 1, 4]` |
| After four passes: | `[2, 3, 5, 6, 7, 1, 4]` |

**Part (d)** [1 MARK]

Some number of iterations of **selection sort** have been performed on a list, resulting in this list:
`[1, 2, 4, 5, 3, 8, 7, 6, 9]`

What is the maximum number of passes that could have been performed so far?

`2`

**Part (e)** [1 MARK]

What additional piece of information do you know about the sorted section during **selection sort** that you don't know during **insertion sort**? `Solution: that the items are in their correct final location.`

## Question 11. [5 MARKS]

Each code fragment in the table below operates on list `L`, which has length `k` where `k` is very large — at least in the tens of thousands. For each fragment, give an expression in terms of `k` for how many times `happy!` is printed, and circle whether the behaviour is constant, linear, quadratic or something else.

| Code | How many times is happy! printed? | Complexity (circle one) |
|---|---|---|
| `for i in range(len(L)):`<br>`    print('happy!')`<br>`for item in L:`<br>`    print('happy!')` | 2k | constant<br>[✓ linear]<br>quadratic<br>something else |
| `for i in range(len(L)):`<br>`    for item in L[:i]:`<br>`        print('happy!')` | k (k-1) / 2 | constant<br>linear<br>[✓ quadratic]<br>something else |
| `# Precondition: len(L) % 10 == 0`<br>`i = 0`<br>`while i < len(L):`<br>`    print('happy!')`<br>`    i = i + len(L) // 10` | 10 | [✓ constant]<br>linear<br>quadratic<br>something else |
| `for item in L[1000:2000]:`<br>`    print('happy!')` | 1000 | [✓ constant]<br>linear<br>quadratic<br>something else |
| `for item in L[10:]:`<br>`    print('happy!')` | `k - 10` | constant<br>[✓ linear]<br>quadratic<br>something else |

## Question 12. [15 marks]

In this question, you will develop two classes to keep track of airplanes and flights.

Here is the header and docstring for class `Airplane`.

```
class Airplane:
    """
    Information about a particular airplane including the model, the serial
    number, the number of seats, and the number of miles travelled.
    """
```

### Part (a) [2 marks]

Complete method `__init__` for class `Airplane`.

Note: you will most likely not need all of the space on this page.

```
    def __init__(self, plane_model, serial_num, num_seats, miles_travelled):
        """ (Airplane, str, str, int, float)

        Record the airplane's model plane_model, serial number serial_num, the
        number of seats, and the distance travelled miles_travelled.

        >>> airplane = Airplane('Boeing 747', '19643', 366, 45267.7)
        >>> airplane.model
        'Boeing 747'
        >>> airplane.serial
        '19643'
        >>> airplane.seats
        366
        >>> airplane.miles
        45267.7
        """

        self.model = plane_model
        self.serial = serial_num
        self.seats = num_seats
        self.miles = miles_travelled
```

**Part (b)** [2 MARKS] Here is the header, type contract, and description for method `log_trip` in class `Airplane`. Add an example that creates an `Airplane` object, logs a trip of `1000.0` miles, and shows that those miles have been logged. Also write the body of the method.

```
    def log_trip(self, num_miles):
        """ (Airplane, float) -> NoneType

        Precondition: num_miles > 0.0

        Record that the airplane travelled num_miles additional miles.

        >>> airplane = Airplane('Boeing 747', '19643', 366, 45267.7)
        >>> airplane.log_trip(1000.0)
        >>> airplane.miles
        46267.7

        """

        self.miles = self.miles + num_miles
```

**Part (c)** [3 MARKS]

Write an `__eq__` method in class `Airplane` that compares two `Airplane` objects to see if they are equal. Consider two `Airplane`s equal if they have the same serial number. Follow the function design recipe, which includes writing a docstring.

```
    def __eq__(self, other):
        """ (Airplane, Airplane) -> bool

        Return whether this Airplane and other are the same.

        >>> a1 = Airplane('Boeing 747', '19643', 366, 45267.7)
        >>> a2 = Airplane('Boeing 747', '19643', 366, 45267.7)
        >>> a3 = Airplane('Boeing 747', '20536', 366, 45267.7)
        >>> a1 == a2
        True
        >>> a1 == a3
        False
        """

        return self.serial == other.serial
```

**Note:** For the rest of this question, you should assume that there is a `__str__` method in class `Airplane` that returns strings of this form: `'Airplane(Boeing 747, 19643, 366, 45267.7)'`

```
class Flight:
    """ Information about an airplane flight. """
```

**Part (d)** [2 MARKS] Complete method `__init__` in class `Flight`:

```
    def __init__(self, plane):
        """ (Flight, Airplane) -> NoneType

        Create a Flight with an empty passenger list on airplane plane.

        >>> a = Airplane('Boeing 747', '19643', 366, 45267.7)
        >>> f = Flight(a)
        >>> str(f.airplane)
        'Airplane(Boeing 747, 19643, 366, 45267.7)'
        >>> f.passengers
        []
        """

        self.airplane = plane
        self.passengers = []
```

**Part (e)** [3 MARKS]

Complete method `add` in class `Flight`.

```
    def add(self, passenger):
        """ (Flight, str) -> bool

        If there are still seats available on this flight, add passenger to the
        passenger list. Return True iff passenger is added to this flight.

        >>> a = Airplane('Cessna 150E', '9378', 1, 824.8)
        >>> f = Flight(a)
        >>> f.add('Myrto')
        True
        >>> f.add('Jen')
        False
        """

        if len(self.passengers) < self.airplane.seats:
            self.passengers.append(passenger)
            return True
        return False
```

**Part (f)** [3 MARKS] Complete method `change_planes` in class `Flight`:

```
def change_planes(self, other_airplane):
    """ (Flight, Airplane) -> bool

    If other_airplane has enough seats to hold the passengers on this flight,
    use other_airplane for this flight. Whether or not we change to other_airplane,
    return the number of available seats on this flight (seats not currently occupied
    by passengers).

    >>> a1 = Airplane('Boeing 747', '19643', 366, 45267.7)
    >>> f = Flight(a1)
    >>> f.add('Myrto')
    True
    >>> f.add('Jen')
    True
    >>> a2 = Airplane('Bombardier Dash 8', '11234', 39, 6444.6)
    >>> f.change_planes(a2)
    37
    >>> a3 = Airplane('Cessna 150E', '9378', 1, 824.8)
    >>> f.change_planes(a3)
    37
    """

    if len(self.passengers) <= other_airplane.seats:
        self.airplane = other_airplane
    return self.airplane.seats - len(self.passengers)
```