Question 1. [16 MARKS]

Part (a) [3 MARKS]

Consider this Python code:

```
L = [8, 12, 3]
X = L.sort()
Y = L[:]
L.extend([1])

print(X)
print(id(Y) == id(L))
print(L)
```

Write what this code prints when run, with one line per box. There may be more boxes than you need; leave unused boxes blank.

None
False
[3, 8, 12, 1]

Part (b) [2 MARKS]

Consider this Python code:

```
for i in range(2):
  for j in range(2):
    print('({0}, {1})'.format(i, j))
```

Write what this code prints when run, with one line per box. There may be more boxes than you need; leave unused boxes blank.

(0, 0)
(0, 1)
(1, 0)
(1, 1)

Part (c) [2 MARKS]

Consider this Python code:

```
def find_bias(lst):
  """ (list of int) -> int
  """

  bias = 0
  for num in lst:
    if num % 2 == 0:
      return bias + 1
    else:
      return bias - 1
  return bias

my_list = [2, 4, 5, 6]
print('The even/odd bias is:', find_bias(my_list))
```

Write what this code prints when run, with one line per box. There may be more boxes than you need; leave unused boxes blank.

The even/odd bias is: 1

Part (d) [3 MARKS]

Consider this Python code that has been saved in the file `words.py`:

```
def func(word):
    print(__name__)
    word = word + 'Na'
    print ('0:', word)
    return word

if __name__ == '__main__':
    word = 'Hey'
    print('1:', word)
    func(word)
    print('2:', word)
    word = func(word) + '!'
    print('3:', word)
```

Write what this code prints when `words.py` is run, with one line per box. There may be more boxes than you need; leave unused boxes blank.

1: Hey
__main__
0: HeyNa
2: Hey
__main__
0: HeyNa
3: HeyNa!

Part (e) [2 MARKS]

Consider this Python code:

```
def f1(x, y):
    print('f1:', x, y)
    return x + y

def f2(x, y):
    print('f2:', x, y)
    return x * y

print(f1(f2(6, 5), f1(2, 4)))
```

Write what this code prints when run, with one line per box. There may be more boxes than you need; leave unused boxes blank.

`f2: 6 5`
`f1: 2 4`
`f1: 30 6`
`36`

Part (f) [4 MARKS]

Consider this Python function:

```
def f(x):
    if x % 2 != 0:
        if x ** 2 <= 36:
            return 'Pow'
        else:
            return x // 3
    else:
        if x < 0 and abs(x) > 5:
            return False
        elif not x + 2 > 8:
            return x / 2
    return 'Zonk'
```

Four different calls to the function `f()` are given in the table below. Beside each call, write the value returned by `f()` and that value's type.

Call	**Return Value**	**Return Type**
`f(2)`	1.0	float
`f(13)`	4	int
`f(-8)`	False	bool
`f(10)`	'Zonk'	str

Question 2. [4 MARKS]

Each of the following sets of Python statements will result in an error message being displayed when the code is run. Explain briefly the cause of each error in the table below.

Python statements	**Explain briefly why an error message is displayed**

`month = 'April'`
`date = 23`
`today =  month + ' ' + date`
	cannot concatenate a str and an int

`seasons = [' Winter ', ' Spring ',`
`' Summer ', ' Fall ']`
`terms = seasons[:]`
`terms.upper()`
	upper is a method of str, cannot be applied to a list

`['2015', 'September'].extend(4)`	extend requres an iterable (list) as an argument

`department = 'Computer Science'`
`faculty = department[9:]`
`faculty = faculty + [' 2015']`
	cannot concatenate a string and a list

Question 3. [7 MARKS]

Part (a) [3 MARKS] Consider this function header and docstring:

```
def cut_in_half(message):
    """ (str) -> list of str

    Return a two-item list in which the first item is the first half of message
    and the second item is the second half of message.  If the two halves of
    message are not the same length, the longer half should appear as the first
    item in the list.
    """
```

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

Test Case Description	`message`	Return Value
empty string	`''`	`['','']`
one character string	`'p'`	`['p', '']`
two character strings	`'ab'`	`['a','b']`
multi-characters, even length	`'abcdef'`	`['abc', 'def']`
multi-characters, odd length	`'abcdefg'`	`['abcd', 'efg']`

Part (b) [4 MARKS] Consider this function header and docstring:

```
def convocation_status(gpa):
    """ (float) -> str

    Precondition: 0.0 <= gpa <= 4.0

    Return 'with high distinction' if gpa is at least 3.5, 'with distiction' if
    gpa is at least 3.2 and less than 3.5, and 'regular' otherwise.
    """
```

In the table below, we have outlined one test case for `convocation_status`. Add four more test cases chosen to test the function thoroughly.

Test Case Description	`gpa`	Return Value
under 3.2	`2.5`	`'regular'`
3.2	`3.2`	`'with distinction'`
between 3.2 and 3.5	`3.4`	`'with distinction'`
3.5	`3.5`	`'with high distinction'`
over 3.5	`3.9`	`'with high distinction'`

Question 4. [5 MARKS]

Before completing your answer to this question, you may find it helpful to be reminded that the `input()` function returns a `str` containing the characters typed in by the program user. The trailing newline/return character (`'\n'`) is stripped from the `str` before `input()` returns.

You may also find it helpful to consult the short Python function/method descriptions for `S.isdigit()` and `int(x)` that appear at the end of this exam paper.

Now consider the following function header, docstring and partial function body:

```
def get_valid_month():
    """ () -> int

    Return a valid month number input by user after (possibly repeated) prompting.

    A valid month number is an int between 1 and 12 inclusive.
    """

    prompt = 'Enter a valid month number: '
    error_message = 'Invalid input! Read the instructions and try again!'

    # Use this statement as many times as needed for input:
    #     month = input(prompt)

    # Use this statement as many times as needed for output:
    #     print(error_message)
```

Complete the body of the function according to its docstring description.

SAMPLE SOLUTION:

```
    month = input(prompt)
    while not month.isdigit() or int(month) < 1 or int(month) > 12:
        print(error_message)
        month = input(prompt)
    return int(month)
```

Question 5. [7 MARKS]

Part (a) [3 MARKS]

Consider this function header and docstring:

```
def convert_time_to_seconds(time_as_str):
    """ (str) -> int

    Precondition: time_as_str is a str in the format 'h:m:s', with
                  0 <= int(h) <= 23 and 0 <= int(m) <= 59 and 0 <= int(s) <= 59

    Return the number of seconds in time_as_str.

    >>> convert_time_to_seconds('1:10:25')
    4225
    """
```

Write the body of the function according to its docstring description.

```
    time_as_list = time_as_str.split(':')
    h = int(time_as_list[0])
    m = int(time_as_list[1])
    s = int(time_as_list[2])
    time_in_seconds = h * 60 * 60 + m * 60 + s

    return time_in_seconds
```

Part (b) [4 MARKS]

You have a **very large** file named `twm_times.txt` that contains the completion times for runners in the 2014 Toronto Waterfront Marathon. Each line in the file contains a single time in the format of `h:m:s`. The first three lines in `twm_times.txt` are:

```
2:8:15
2:8:36
2:8:41
```

Suppose that all runners with a completion time under `3:20:14` were to receive a special prize. To determine the number of runners who qualified for the prize, you ran the following statements in a Python shell:

```
>>> twm = open('twm_times.txt', 'r')
>>> print('Number of prize winners:', number_of_winners('3:20:14', twm))
>>> twm.close()
```

Write the body of the `number_of_winners` function according to its docstring description so that you will be able to easily determine the number of prize winners.

```
def number_of_winners(qualifying_time, race_results):
    """ (str, file open for reading) -> int

    A valid time is a str with format 'h:m:s', with 0 <= int(h) <= 23,
    0 <= int(m) <= 59 and 0 <= int(s) <= 59.

    Precondition: qualifying_time is a valid time,
                  Each line in race_results contains a single valid time.

    Return the number of lines in race_results that contain a time that is
    below qualifying_time.
    """

    q_t = convert_time_to_seconds(qualifying_time)

    num_winners = 0
    for line in race_results:
        l_t = convert_time_to_seconds(line.strip())
        if l_t < q_t:
            num_winners = num_winners + 1

    return num_winners
```

Question 6. [7 MARKS]

On Assignment Two, you wrote code to implement the game Battleship. You may recall that in your code, you worked with a `view_board` and a `symbol_board`. Each board was a `list of list of str`, where each inner list represented one row of the board. We called each item in an inner list a cell. Cells were used to keep track of ship symbols, hits, misses, and so on.

In this question, you are to write a function that counts the number of hits and misses in each row of a board. Recall that we defined the named constants `HIT` and `MISS` using the statements:

```
HIT = 'X'
MISS = 'M'
```

Complete the following function according to its docstring description.

```
def count_hits_and_misses(board):
    """ (list of list of str) -> list of int

    Precondition: board != [] and each list in board has len(board)

    Return a list that contains the number of occurrences of the
    HIT or MISS symbol in each row of board.

    >>> board = [['-','M','-'], ['X','M','-'], ['-','-','-']]
    >>> count_hits_and_misses(board)
    [1, 2, 0]
    """

    h_m_list = []
    for row in board:
        h_m_counter = 0
        for cell in row:
            if cell == HIT or cell == MISS:
                h_m_counter = h_m_counter + 1
        h_m_list.append(h_m_counter)

    return h_m_list
```

Question 7. [5 MARKS]

Throughout this question, lists are to be sorted into ascending (increasing) order.

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

```
[4, 9, 2, 1, 6, 5, 8] #initial list

[1, 9, 2, 4, 6, 5, 8] # after one pass
[1, 2, 9, 4, 6, 5, 8] # after two
[1, 2, 4, 9, 6, 5, 8] # after three
[1, 2, 4, 5, 6, 9, 8] # after four
[1, 2, 4, 5, 6, 9, 8] # after five
```

Which sorting algorithm is being executed? (circle one)

(a) bubble sort

(b) [selection sort]

(c) insertion sort

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

```
[4, 9, 2, 1, 6, 5, 8] #initial list

[4, 9, 2, 1, 6, 5, 8] # after one pass
[4, 9, 2, 1, 6, 5, 8] # after two
[2, 4, 9, 1, 6, 5, 8] # after three
[1, 2, 4, 9, 6, 5, 8] # after four
[1, 2, 4, 6, 9, 5, 8] # after five
```

Which sorting algorithm is being executed? (circle one)

(a) bubble sort

(b) selection sort

(c) [insertion sort]

Part (c) [2 MARKS] List `[7, 6, 3, 8, 1, 2, 0, 5]` is being sorted using selection sort. Fill in the blanks to show the list after the next two passes.

Solution:

After one pass: `[0, 6, 3, 8, 1, 2, 7, 5]`

After two passes: `[0, 1, 3, 8, 6, 2, 7, 5]`

After three passes: `[0, 1, 2, 8, 6, 3, 7, 5]`

After four passes: `[0, 1, 2, 3, 6, 8, 7, 5]`

Part (d) [1 MARK] Consider the following list: `L = [1, 2, 3, 4, 5, 6]`

Which algorithm would you expect to have a faster runtime? Circle one.

[`insertion sort`] `selection sort` they would require the same time

Question 8. [9 MARKS]

On Assignment Three, you wrote code to detect whether or not an input poem followed a particular rhyming scheme. You did this by first reading a Pronouncing Dictionary that contained lines like:

```
EXAMINATION  IH0 G Z AE2 M AH0 N EY1 SH AH0 N
```

and storing the lines in a Python `dict of {str: list of str}` that had key: value pairs like:

```
'EXAMINATION': ['IH0', 'G', 'Z', 'AE2', 'M', 'AH0', 'N', 'EY1', 'SH', 'AH0', 'N']
```

The helper function `read_pronunciation` was called, and a Python `dict` named `words_to_phonemes` was used to store all of the words and associated phonemes that were in the Pronouncing Dictionary file `dictionary.txt`. After reading the Pronouncing Dictionary, you could get shell output like the following:

```
>>> words_to_phonemes['WILLIAM'] == ['W', 'IH1', 'L', 'Y', 'AH0', 'M']
True
```

To determine whether or not two words rhyme, you could make use of the helper function:

```
def last_phonemes(phoneme_list):
    """ (list of str) -> list of str

    Return the last vowel phoneme and subsequent consonant phoneme(s) in phoneme_list.

    >>> last_phonemes(['AE1', 'B', 'S', 'IH0', 'N', 'TH'])
    ['IH0', 'N', 'TH']
    >>> last_phonemes(['IH0', 'N'])
    ['IH0', 'N']
    >>> last_phonemes(['B', 'S'])
    []
    """
```

On the assignment, we said that two different words rhyme if and only if their last vowel phonemes and all subsequent consonant phoneme(s) after the last vowel phonemes matched. Using this definition, you could get the shell results:

```
>>> last_phonemes(words_to_phonemes['COW']) == last_phonemes(words_to_phonemes['HOW'])
True
>>> last_phonemes(words_to_phonemes['COW']) == last_phonemes(words_to_phonemes['PIG'])
False
```

and conclude that `'COW'` and `'HOW'` rhyme, while `'COW'` and `'PIG'` do not rhyme.

Part (a) [7 MARKS]

When writing poetry, it would be helpful to have a list of words that rhyme. Complete the following function according to its docstring description. Use the `last_phonemes` function as a helper function.

```
def build_rhyming_dict(words_to_phonemes):
    """ (dict of {str: list of str}) -> dict of {str: list of str}

    Return a dict where the keys are the same as the keys in word_to_phonemes
    and the value for each key is a list of all words that rhyme with the key.

    Two words rhyme if and only if they are different and their last
    vowel phonemes and all subsequent consonant phoneme(s) after the
    last vowel phonemes match.

    >>> words_to_phonemes = read_pronunciation(open('dictionary.txt'))
    >>> words_to_rhyming_words = build_rhyming_dict(words_to_phonemes)
    >>> words_to_rhyming_words['CRAIG']
    ['BAIG', 'BEGUE', 'FLAIG', 'HAGUE', 'HAIG', 'LAPHROAIG', 'MACIAG',
     'MCCAGUE', 'MCCAIG', 'MCKAIG', 'MCQUAIG', 'MCTAGUE',
     'NEST-EGG', "O'LAGUE", 'PLAGUE', 'RAGUE', 'SPRAGUE', 'VAGUE']
    >>> # Notice that 'CRAIG' is not in the list of words that rhyme with 'CRAIG'
    """

    words_to_rhyming_words = {}
    for word in words_to_phonemes:
        rhyming_words = []
        for potential_rhyme in words_to_phonemes:
            if ( last_phonemes(words_to_phonemes[word]) ==
                     last_phonemes(words_to_phonemes[potential_rhyme])):
                rhyming_words.append(potential_rhyme)
        rhyming_words.remove(word)
        # alternate: add word != potential_rhyme to if condition, then
        # removal not required
        words_to_rhyming_words[word] = rhyming_words

    return words_to_rhyming_words
```

Part (b) [2 MARKS] One instructor's solution to Part (a) had a runtime that is best described as being quadratic in the length of the `words_to_phonemes` Python `dict`. If it took 1 second for the `build_rhyming_dict` function to run for a `words_to_phonemes dict` containing 1000 words, roughly how long would you expect the `build_rhyming_dict` function to take when the `words_to_phonemes dict` was doubled in length to a size of 2000 words? Justify your response.

```
1000 -> 2000 is a factor of 2 increase
quadratic algorithm means 1 sec -> (2)^2 times 1 sec = 4 seconds
```

Question 9. [8 MARKS]

Consider the following Python function. The docstring has been shortend to save space.

```
def bark_like_a_dog(L):
    """ (list of object) -> NoneType
    """

    for item in L:
        print('Woof!')
```

Each of the following sets of Python code operate on a list named `L`. You may assume that `L` refers to a list of objects and that `len(L)` is `n`. For each set of Python code, write a formula that expresses approximately how many times the word `Woof!` is printed. The formula may depend on `n`. In addition, circle whether the dependence on `n` is constant, linear, quadratic or something else.

Python code	**How many times is Woof! printed? (approximately)**	**Dependence on `n` (circle one)**
`bark_like_a_dog(L)`	`n`	constant **[linear]** quadratic something else

`i = 0`
`while i < len(L):`
`  bark_like_a_dog(L[i:i+1])`
`  i = i + 1`
	`n`	constant
		[linear]
		quadratic
		something else

`i = 0`
`while i < len(L):`
`  bark_like_a_dog(L[i:])`
`  i = i + 1`
	$1 + 2 + 3 + ... + n$
	$= \frac{1}{2}n(n+1)$
	or approx. $n^2/2$
		constant
		linear
		[quadratic]
		something else

`i = 0`
`while not(i < len(L)):`
`  bark_like_a_dog(L)`
`  i = i + 1`
	0	**[constant]**
		linear
		quadratic
		something else

Question 10. [12 MARKS]

In this question you will develop two classes that are part of an alarm notification system. The class `Timestamp` creates an object that holds a time (hour, minute and second) and a message. The class `AlarmSchedule` holds a list of `Timestamp` objects.

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

```
class Timestamp:
    """ Time and message for a timestamp. """
```

Part (a) [2 MARKS]

Complete method `__init__` for class `Timestamp`.

```
    def __init__(self, h, m, s, msg):
        """ (Timestamp, int, int, int, str) -> NoneType

        Precondition: 0 <= h <= 23 and 0 <= m <= 59 and 0 <= s <= 59

        Initialize the hour h, minute m, second s, and message msg associated
        with this Timestamp.

        >>> ts1 = Timestamp(14, 10, 42, 'Relax')
        >>> ts1.hour
        14
        >>> ts1.min
        10
        >>> ts1.sec
        42
        >>> ts1.msg
        'Relax'
        """

        self.hour = h
        self.min = m
        self.sec = s
        self.msg = msg
```

Part (b) [2 MARKS]

In order to compare objects of type `Timestamp`, it is useful to be able to work with just the time (hour, minute and second) and not the message. Here is the header, type contract and description for a method `time` in class `Timestamp`. Write the body of the method.

```
    def time(self):
        """ (Timestamp) -> str

        Return a string representation of the time associated with this Timestamp.

        >>> ts1 = Timestamp(14, 9, 1, 'Relax')
        >>> ts1.time()
        '14:9:1'
        """

        return str(self.hour) + ":" + str(self.min) + ":" + str(self.sec)
        # Alternate:
        # return '{0}:{1}:{2}'.format(self.hour, self.min, self.sec)
```

Part (c) [4 MARKS]

Follow the function design recipe to write an `__eq__` method for class `Timestamp`. This method will give us a way to determine whether or not two `Timestamp` objects contain the same information. Consider two timestamps to be equal if and only if they have the same time (hour, minute and second) and the same message. (Your code may call helper functions/methods, but this is not required.)

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

        Return True if and only if this Timestamp and the other Timestamp
        refer to the same time and have the same message.

        >>> ts1 = Timestamp(14, 10, 0, 'Relax')
        >>> ts2 = Timestamp(14, 10, 0, 'Panic')
        >>> ts3 = Timestamp(14, 10, 0, 'Relax')
        >>> ts4 = Timestamp(14, 14, 14, 'Write')
        >>> ts1 == ts2
        False
        >>> ts1 == ts3
        True
        >>> ts1 == ts4
        False
        """

        return ( self.hour == other.hour and self.min == other.min and
                 self.sec == other.sec and self.msg == other.msg )
        # Alternate:
        # return ( self.time() == other.time() and self.msg == other.msg )
```

Here is the header and docstring for class `AlarmSchedule`. In parts (d) and (e), you will write methods for class `AlarmSchedule`.

```
class AlarmSchedule:
    """ Contains information about Timestamp objects in an alarm schedule. """
```

Part (d) [1 MARK]

Complete method `__init__` for class `AlarmSchedule`.

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

```
    def __init__(self):
        """ (AlarmSchedule) -> NoneType

        Initialize an AlarmSchedule with an empty list named schedule.

        >>> alarms = AlarmSchedule()
        >>> alarms.schedule
        []
        """

        self.schedule = []
```

Part (e) [3 MARKS]

Complete method `add` in class `AlarmSchedule`.

```
    def add(self, tstamp):
        """ (AlarmSchedule, Timestamp) -> NoneType

        Modify schedule to add Timestamp tstamp, provided there is not an
        existing Timestamp with the same time.

        >>> alarms = AlarmSchedule()
        >>> alarms.add(Timestamp(14, 10, 42, 'Relax'))
        >>> alarms.add(Timestamp(14, 23, 39, 'Sigh'))
        >>> alarms.add(Timestamp(14, 10, 42, 'Burp'))
        >>> alarms.schedule[0].msg
        'Relax'
        >>> alarms.schedule[1].msg
        'Sigh'
        >>> len(alarms.schedule)
        2
        """

        tstamp_exists = False
        for ts in self.schedule:
            if ts.time() == tstamp.time():
                tstamp_exists = True

        if not tstamp_exists:
            self.schedule.append(tstamp)

        # alternate - always put tstamp in, and then take out if already there
        #
        #self.schedule.append(tstamp)
        #for ts in self.schedule:
        #    if ts.time() == tstamp.time():
        #        self.schedule.pop
        #        return
```

Short Python function/method descriptions:

```
__builtins__:
  abs(x) -> number
    Return the absolute value of x.
  input([prompt]) -> str
    Read a string from standard input. The trailing newline is stripped. The prompt string,
    if given, is printed without a trailing newline before reading.
  int(x) -> int
    Convert x to an integer, if possible.  A floating point argument will be truncated
    towards zero.
  len(x) -> int
    Return the length of the list, tuple, dict, or string x.
  max(iterable) -> object
  max(a, b, c, ...) -> object
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
  min(iterable) -> object
  min(a, b, c, ...) -> object
      With a single iterable argument, return its smallest item.
      With two or more arguments, return the smallest argument.
  open(name[, mode]) -> file open for reading, writing, or appending
    Open a file.  Legal modes are "r" (read), "w" (write), and "a" (append).
  print(value, ..., sep=' ', end='\n') -> NoneType
    Prints the values. Optional keyword arguments:
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
  range([start], stop, [step]) -> list-like-object of int
    Return the integers starting with start and ending with stop - 1 with step specifying
    the amount to increment (or decrement).
    If start is not specified, the list starts at 0.  If step is not specified,
    the values are incremented by 1.

dict:
  D[k] --> object
    Produce the value associated with the key k in D.
  del D[k]
    Remove D[k] from D.
  k in d --> bool
    Produce True if k is a key in D and False otherwise.
  D.get(k) -> object
    Return D[k] if k in D, otherwise return None.
  D.keys() -> list-like-object of object
    Return the keys of D.
  D.values() -> list-like-object of object
    Return the values associated with the keys of D.
  D.items() -> list-like-object of tuple of (object, object)
    Return the (key, value) pairs of D, as 2-tuples.
```

```
file open for reading:
  F.close() -> NoneType
    Close the file.
  F.read() -> str
    Read until EOF (End Of File) is reached, and return as a string.
  F.readline() -> str
    Read and return the next line from the file, as a string. Retain newline.
    Return an empty string at EOF (End Of File).
  F.readlines() -> list of str
    Return a list of the lines from the file.  Each string ends in a newline.

file open for writing:
  F.close() -> NoneType
    Close the file.
  F.write(x) -> int
    Write the string x to file F and return the number of characters written.

list:
  x in L --> bool
    Produce True if x is in L and False otherwise.
  L.append(x) -> NoneType
    Append x to the end of the list L.
  L.extend(iterable) -> NoneType
    Extend list L by appending elements from the iterable. Strings and lists are
    iterables whose elements are characters and list items respectively.
  L.index(value) -> int
    Return the lowest index of value in L.
  L.insert(index, x) -> NoneType
    Insert x at position index.
  L.pop() -> object
    Remove and return the last item from L.
  L.remove(value) -> NoneType
    Remove the first occurrence of value from L.
  L.reverse() -> NoneType
    Reverse *IN PLACE*.
  L.sort() -> NoneType
    Sort the list in ascending order *IN PLACE*.

str:
  x in s --> bool
    Produce True if and only if x is in s.
  str(x) -> str
    Convert an object into its string representation, if possible.
  S.count(sub[, start[, end]]) -> int
    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are interpreted
    as in slice notation.
  S.find(sub[, i]) -> int
    Return the lowest index in S (starting at S[i], if i is given) where the
    string sub is found or -1 if sub does not occur in S.
  S.index(sub) -> int
    Like find but raises an exception if sub does not occur in S.
```

```
S.isalpha() -> bool
  Return True if and only if all characters in S are alphabetic
  and there is at least one character in S.
S.isdigit() -> bool
  Return True if all characters in S are digits
  and there is at least one character in S, and False otherwise.
S.islower() -> bool
  Return True if and only if all cased characters in S are lowercase
  and there is at least one cased character in S.
S.isupper() -> bool
  Return True if and only if all cased characters in S are uppercase
  and there is at least one cased character in S.
S.lower() -> str
  Return a copy of the string S converted to lowercase.
S.lstrip([chars]) -> str
  Return a copy of the string S with leading whitespace removed.
  If chars is given and not None, remove characters in chars instead.
S.replace(old, new) -> str
  Return a copy of string S with all occurrences of the string old replaced
  with the string new.
S.rstrip([chars]) -> str
  Return a copy of the string S with trailing whitespace removed.
  If chars is given and not None, remove characters in chars instead.
S.split([sep]) -> list of str
  Return a list of the words in S, using string sep as the separator and
  any whitespace string if sep is not specified.
S.strip([chars]) -> str
  Return a copy of S with leading and trailing whitespace removed.
  If chars is given and not None, remove characters in chars instead.
S.upper() -> str
  Return a copy of the string S converted to uppercase.
```