- If you have any scratch work, please circle your final answer.
- Any code you write should run in a Jupyter cell; every character counts!
- For all questions on this exam, assume that all necessary packages have been imported.

(1) For the following 4 problems, write down what each code block would display if executed in a Jupyter cell. If the code generates an error or infinite loop, write `Error`.

(a)
```
name = ’matt’
name_dict = {x : name.count(x) for x in name}
name_dict[name[1]], name_dict[name[2]]
```

(b)
```
(lambda x: x + 2 * x)(np.ones(2))
```

(c)
```
arr = np.arange(9).reshape(3, 3)
arr[::2, 1:] = -1
arr
```

(d)
```
def func(string):
    while len(string) > 1:
        new = str((int(string[0]) + int(string[-1])) % 10)
        print(new)
        return func(new + string[1:-1])
    return string

func(’987’)
```

Solution:

(a)
```
(1, 2)
```

(b)
```
array([3., 3.])
```

(c)
```
array([[0, -1. -1],
       [3, 4, 5],
       [6, -1 -1]])
```

(d)
```
6
4
’4’
```

(2) Euler's number $e \approx 2.71828$ can be approximated by the sequence a_n given by

$$a_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

The values of the sequence a_n get closer to e as n gets larger.

(a) Write a function `approx(n)` which returns the value of the approximation a_n. For example, `approx(1)` would return 2.0, and `approx(2)` would return 2.5 because

$$a_1 = 1 + \frac{1}{1!} = 2, \quad a_2 = 1 + \frac{1}{1!} + \frac{1}{2!} = 2.5.$$

(b) Write a function `first_n(tol)` which returns the first value of n such $e - a_n <$ `tol`. For example, `first_n(0.5)` would return 2, because

$$e - a_2 \approx 2.71828 - 2.5 = 0.21828 < 0.5.$$

Solution:

(a)
```
def approx(n):
    reciprocals = [1] + [1 / math.factorial(a) for a in range(1, n + 1)]
    return sum(reciprocals)
```

(b)
```
def first_n(tol):
    n = 1
    while math.e - approx(n) >= tol:
        n += 1
    return n
```

(3) Create a class called **Circle**. Each instance of this class should have two attributes:

- `center`, which is a list of the coordinates of the circle's center, and has a default value of $[0, 0]$,
- `radius`, which is a nonnegative number and has a default value of 1,

and methods

- `stretch(factor)`, which returns nothing but updates the radius by multiplying by the given factor,
- `move(tup)`, returns nothing but which shifts the circle according to the values in `tup`. For example, if `tup = (-1, 2)`, this method will shift the circle one unit left, and two units up,
- `plot_top()`, which returns nothing but plots the top half of the circle.

Solution:

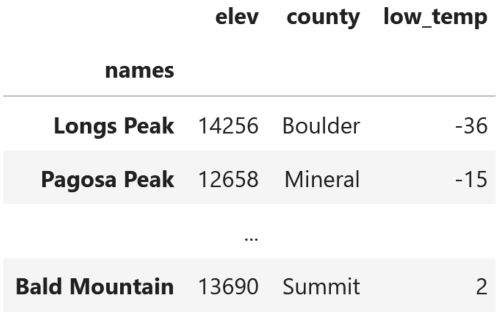
```
class Circle:
    def __init__(self, center=[0, 0], radius=1):
        self.center = center
        self.radius = radius

    def stretch(self, factor):
        self.radius *= factor

    def move(self, tup):
        self.center = [self.center[0] + tup[0], self.center[1] + tup[1]]

    def plot_top(self):
        xvals = np.linspace(self.center[0] - self.radius, self.center[0] + self.radius, 100
        yvals = np.sqrt((self.radius ** 2) - (xvals - self.center[0]) ** 2) + self.center[1
        plt.plot(xvals, yvals)
        plt.show()
```

(4) The dataframe **dfpeaks** contains information about mountains in Colorado. The index column is `names`, and the other columns are `elev` for the elevation in feet at the summit, `county` specifies which county in Colorado the mountain is in, and `low_temp` is the lowest temperature (in °C) recorded at the summit:



	elev	county	low_temp
names			
Longs Peak	14256	Boulder	-36
Pagosa Peak	12658	Mineral	-15
	...		
Bald Mountain	13690	Summit	2

Write code to do the following:

(a) Create a new column `mt13` which is `True` if the mountain has elevation from 13,000 and 13,999 feet, and `False` otherwise.

(b) Create a list of all the names of mountains located in Boulder County.

(c) Create a list of counties which have a mountain whose record low temperature is less than −20°C. County names in your list should only appear once.

Solution:

(a) `dfpeaks[’mt13’] = (13000 <= dfpeaks[’elev’]) & (dfpeaks[’elev’] <= 13999)`

(b) `dfpeaks[dfpeaks[’county’] == ’Boulder’].index.tolist()`

(c) `dfpeaks[dfpeaks[’low_temp’] < -20].county.unique().tolist()`

(5) (a) The *determinant* of the 2×2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, where a, b, c and d are numbers, is given by

$$\det(A) = ad - bc.$$

Write a function `det2(arr)` which returns the determinant of a 2×2 Numpy array `arr`. If `arr` is not 2×2, then `det2()` should return nothing and print '`arr is not 2x2.`'

(b) Write a function `swap(arr, tup)` which returns a copy of the array `arr`, but with the rows specified in `tup` swapped. You may assume that the values in `tup` are valid row indices.

For example, if `arr` represents the array $\begin{bmatrix} 2 & 1 \\ 5 & 0 \\ 0 & -1 \end{bmatrix}$, then `swap(arr, (0, 2))` should return an array which represents $\begin{bmatrix} 0 & -1 \\ 5 & 0 \\ 2 & 1 \end{bmatrix}$.

Solution:

(a)
```
def det2(arr):
    if arr.shape == (2, 2):
        return arr[0, 0] * arr[1, 1] - arr[0, 1] * arr[1, 0]
    else:
        print('arr is not 2x2')
```

(b)
```
def swap(arr, tup):
    row0 = np.copy(arr[tup[0], :])
    row1 = np.copy(arr[tup[1], :])
    arr[tup[0], :] = row1
    arr[tup[1], :] = row0
    return arr
```