Midterm 2: Sample solutions

Bookmark this page

problem0 (Score: 10.0 / 10.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 2.0 / 2.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 1.0 / 1.0)
5. Test cell (Score: 4.0 / 4.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Problem 0: Graph search

This problem tests your familiarity with Pandas data frames. As such, you'll need this import:

In [1]:
```
import pandas as pd
```

This problem has four exercises worth a total of ten (10) points.

Dataset: (simplified) airport segments

The dataset for this problem is a simplified version of the airport segments dataset from Notebook 11. Start by getting and inspecting the data, so you know what you will be working with.

In [2]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/us-flights/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
```

```
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".forma
t(local_file,

body_checksum,

checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/us-flights/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/us-flights/"
DATA_PATH = ""

datasets = {'L_AIRPORT_ID.csv': 'e9f250e3c93d625cce92d08648c4bbf0',
            'segments.csv': 'b5e8ce736bc36a9dd89c3ae0f6eeb491',
            'two_away_solns.csv': '7421b3eead7b5107c7fbd565228e50c7'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'segments.csv' is ready!
'two_away_solns.csv' is ready!
'L_AIRPORT_ID.csv' is ready!

(All data appears to be ready.)
```

The first bit of data you'll need is a list of airports, each of which has a code and a string description.

In [3]:
```
airports = pd.read_csv('{}L_AIRPORT_ID.csv'.format(DATA_PATH))
airports.head()
```

Out[3]:

	Code	Description
0	10001	Afognak Lake, AK: Afognak Lake Airport
1	10003	Granite Mountain, AK: Bear Creek Mining Strip
2	10004	Lik, AK: Lik Mining Camp
3	10005	Little Squaw, AK: Little Squaw Airport
4	10006	Kizhuyak, AK: Kizhuyak Bay

The other bit of data you'll need is a list of available direct connections.

In [4]:
```
segments = pd.read_csv('{}segments.csv'.format(DATA_PATH))
print("There are {} direct flight segments.".format(len(segments)))
segments.head()
```

```
There are 4191 direct flight segments.
```

Out[4]:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID
0	10135	10397
1	10135	11433
2	10135	13930
3	10140	10397
4	10140	10423

Exercises

Complete the following exercises.

Exercise 0 (1 point). Given an airport code, implement the function, `get_description(code, airports)`, so that it returns the row of `airports` having that code.

For example,

```
get_description(10397, airports)
```

would return the dataframe,

	Code	Description
373	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...

In [5]:

Student's answer (Top)

```
def get_description(code, airports):
    ### BEGIN SOLUTION
    return airports[airports['Code'] == code]
    ### END SOLUTION

# Demo:
get_description(10397, airports)
```

Out[5]:

	Code	Description
373	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...

In [6]:

Grade cell: `get_description_test` Score: 1.0 / 1.0 (Top)

```
# Test cell: `get_description_test`

from numpy.random import choice
for offset in choice(len(airports), size=10):
    code = airports.iloc[offset]['Code']
    df = get_description(code, airports)
    assert type(df) is pd.DataFrame
    assert len(df) == 1
    assert (df['Code'] == code).all()

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 1 (2 points). Suppose that, instead of one code, you are given a Python set of codes. Implement the function, `get_all_descriptions(codes, airports)`, so that it returns a dataframe whose rows consist of all rows from `airports` that match one of the codes in `codes`.

For example,

```
get_all_descriptions({10397, 12892, 14057}, airports)
```

would return,

	Code	Description
373	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
2765	12892	Los Angeles, CA: Los Angeles International
3892	14057	Portland, OR: Portland International

In [7]:

In [7]: Student's answer (Top)

```
def get_all_descriptions(codes, airports):
    assert type(codes) is set
    ### BEGIN SOLUTION
    matches = airports['Code'].apply(lambda x: x in codes)
    return airports[matches]
    ### END SOLUTION

get_all_descriptions({10397, 12892, 14057}, airports)
```

Out[7]:

	Code	Description
373	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
2765	12892	Los Angeles, CA: Los Angeles International
3892	14057	Portland, OR: Portland International

In [8]: Grade cell: `get_all_descriptions_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `get_all_descriptions_test`

from numpy.random import choice
offsets = choice(len(airports), size=10)
codes = set(airports.iloc[offsets]['Code'])
df = get_all_descriptions(codes, airports)
assert type(df) is pd.DataFrame
assert len(df) == len(codes)
assert set(df['Code']) == codes

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 2 (2 points). Implement the function, `find_description(desc, airports)`, so that it returns the subset of rows of the dataframe `airports` whose `Description` string contains `desc`, where `desc` is a string.

For example,

```
find_description('Atlanta', airports)
```

should return a dataframe with these rows:

Code	Description
10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
11790	Atlanta, GA: Fulton County Airport-Brown Field
11838	Atlanta, GA: Newnan Coweta County
12445	Atlanta, GA: Perimeter Mall Helipad
12449	Atlanta, GA: Beaver Ruin
12485	Atlanta, GA: Galleria
14050	Atlanta, GA: Dekalb Peachtree
14430	Peachtree City, GA: Atlanta Regional Falcon Field

Notice that the last row of this dataframe has "Atlanta" in the middle of the description.

> *Hint*: The easiest way to do this problem is to apply a neat feature of Pandas, which is that there are functions that help do string searches within a column (i.e., within a Series): https://pandas.pydata.org/pandas-docs/stable/text.html (https://pandas.pydata.org/pandas-docs/stable/text.html)

In [9]: Student's answer (Top)

```
def find_description(desc, airports):
    ### BEGIN SOLUTION
    matches = airports['Description'].str.contains(desc)
    return airports[matches]
    ### END SOLUTION

find_description('Atlanta', airports)
```

Out[9]:

	Code	Description
373	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
1717	11790	Atlanta, GA: Fulton County Airport-Brown Field
1762	11838	Atlanta, GA: Newnan Coweta County
2350	12445	Atlanta, GA: Perimeter Mall Helipad
2354	12449	Atlanta, GA: Beaver Ruin
2387	12485	Atlanta, GA: Galleria
3885	14050	Atlanta, GA: Dekalb Peachtree
4222	14430	Peachtree City, GA: Atlanta Regional Falcon Field

In [10]: Grade cell: `find_description_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `lookup_description_test`

assert len(find_description('Los Angeles', airports)) == 4
assert len(find_description('Washington', airports)) == 12
assert len(find_description('Arizona', airports)) == 0
assert len(find_description('Warsaw', airports)) == 2

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 3 (4 points). Suppose you are given an airport code. Implement a function, `find_two_away(code, segments)`, so that it finds all airports that are **two hops** away. It should return this result as a Python set.

For example, the `segments` table happens to include these two rows:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID
...	...	...
178	10397	12892
...	...	...
2155	12892	14057
...	...	...

We say that 14057 is "two hops away" because there is one segment from 10397 to 12892, followed by a second segment from 12892 to 14057. Thus, the set returned by `find_two_away(code, segments)` should include 14057, i.e.,

```
assert 14057 in find_two_away(10397, segments)
```

Your function may assume that the given `code` is valid, that is, appears in the `segments` data frame and has at least one outgoing segment.

In [11]: Student's answer (Top)

```
def find_two_away(code, segments):
    ### BEGIN SOLUTION
    one_away = segments[segments['ORIGIN_AIRPORT_ID'] == code]
    two_away = one_away.merge(segments, left_on='DEST_AIRPORT_ID', right_on='ORIGIN_AIR
PORT_ID')
    return set(two_away['DEST_AIRPORT_ID_y'])
```

```
    return set(two_away['DEST_AIRPORT_ID_y'])
    ### END SOLUTION

atl_two_hops = find_two_away(10397, segments)
atl_desc = get_description(10397, airports)['Description'].iloc[0]
print("Your solution found {} airports that are two hops from '{}'.".format(len(atl_two
_hops), atl_desc))
```

```
Your solution found 277 airports that are two hops from 'Atlanta, GA: Hartsfield-Jackson
Atlanta International'.
```

In [12]:

Grade cell: `find_two_away_test1` Score: 1.0 / 1.0 (Top)

```
# Test cell: `find_two_away_test`

assert 14057 in find_two_away(10397, segments)
assert len(atl_two_hops) == 277

print("\n(Passed first test.)")
```

```
(Passed first test.)
```

In [13]:

Grade cell: `find_two_away_test2` Score: 4.0 / 4.0 (Top)

```
# Test cell: `find_two_away_test2`
print("Note: This test may take a minute...")
if False:
    solns = {}
    for code in airports['Code']:
        two_away = find_two_away(code, segments)
        if code not in solns:
            solns[code] = len(two_away)
    with open('{}two_away_solns.csv'.format(DATA_PATH), 'w') as fp:
        fp.write('Code,TwoAway\n')
        for code, num_two_away in solns.items():
            fp.write('{},{}\n'.format(code, num_two_away))

two_away_solns = pd.read_csv('{}two_away_solns.csv'.format(DATA_PATH))
for row in range(len(two_away_solns)):
    code = two_away_solns['Code'].iloc[row]
    count = two_away_solns['TwoAway'].iloc[row]
    your_count = len(find_two_away(code, segments))
    msg = "Expected {} airports two-away from {}, but your code found {} instead.".form
at(count, code, your_count)
    assert your_count == count, msg
print("\n(Passed!)")
```

```
Note: This test may take a minute...

(Passed!)
```

Fin! If you've reached this point and all tests above pass, you are ready to submit your solution to this problem. Don't forget to save you work prior to submitting.

---

problem1 (Score: 10.0 / 10.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 3.0 / 3.0)
5. Test cell (Score: 3.0 / 3.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Problem 1: Drug Trial

In this problem, you'll analyze some data from a medical drug trial. There are three exercises worth a total of ten points.

Two of the exercises allow you to use **either** Pandas **or** SQL to solve it. Choose the method that feels is more natural to you.

Setup

Run the following few code cells, which will load the modules and sample data you'll need for this problem.

In [1]:
```
import pandas as pd
import numpy as np
import sqlite3 as db

from IPython.display import display

def canonicalize_tibble (X):
    """Returns a tibble in _canonical order_."""
    # Enforce Property 1:
    var_names = sorted (X.columns)
    Y = X[var_names].copy ()

    # Enforce Property 2:
    Y.sort_values (by=var_names, inplace=True)

    # Enforce Property 3:
    Y.set_index ([list (range (0, len (Y)))], inplace=True)

    return Y

def tibbles_are_equivalent (A, B):
    """Given two tidy tables ('tibbles'), returns True iff they are
    equivalent.
    """
    A_canonical = canonicalize_tibble (A)
    B_canonical = canonicalize_tibble (B)
    cmp = A_canonical.eq (B_canonical)
    return cmp.all ().all ()
```

In [2]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)
```

```
    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".forma
t(local_file,

body_checksum,

checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/drug-trials/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/"
DATA_PATH = ""

datasets = {'Drugs_soln.csv': '6df060bde8dea48986dc12650a4fbef5',
            'avg_dose_soln.csv': 'f604e3da488d489792fec220ada738f8',
            'drugs.csv': '33bb1fa5068069a483a6e05fafde40d0',
            'nst_count_soln.csv': '7519ad4764eb238a9120fa7cd1f006de',
            'nst_count_soln--corrected.csv': '81f801cd20775a51f92a1b28593c0915',
            'swt_count_soln.csv': 'fbbb7368d31856665c3e5e1b19d93d65'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/Drugs_s
oln.csv ...

'Drugs_soln.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/nst_cou
nt_soln--corrected.csv ...

'nst_count_soln--corrected.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/drugs.c
sv ...

'drugs.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/avg_dos
e_soln.csv ...

'avg_dose_soln.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/swt_cou
nt_soln.csv ...

'swt_count_soln.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/drug-trials/nst_cou
nt_soln.csv ...

'nst_count_soln.csv' is ready!

(All data appears to be ready.)
```

The data

Company XYZ currently uses Medication A to treat all its patients and is considering a switch to Medication B. A critical part of the evaluation of Medication B is how much of it would be used among XYZ's patients.

The company did a trial of Medication B. The data in the accompanying CSV file, `Drugs.csv`, is data taken from roughly 130 patients at least 2 months before switching medications and up to 3 months while on the new medication.

A patient can be taking medication A or medication B, but cannot be taking both at the same time.

The following code cell will read this data and store it in a dataframe named `Drugs`.

In [3]:
```
Drugs = pd.read_csv("{}drugs.csv".format(DATA_PATH), header=0)
assert len(Drugs) == 2022
Drugs.head()
```

Out[3]:

In [3]:

	ID	Med	Admin Date	Units
0	1	Med A	7/2/12	1,500.00
1	1	Med A	7/6/12	1,500.00
2	1	Med A	7/9/12	1,500.00
3	1	Med A	7/11/12	1,500.00
4	1	Med A	7/13/12	1,500.00

Each row indicates that a patient (identified by his or her `'ID'`) took one **dose** of a particular drug on a particular day. The size of the dose was `'Units'`.

Exercises

Exercise 0 (1 points). All you have to do is read the code in the following code cell and run it. You should observe the following.

First, the `'Med'`, `'Admin Date'`, and `'Units'` columns are stored as strings initially.

Secondly, there are some handy functions in Pandas to change the `'Admin Date'` and `'Units'` columns into more "natural" native Python types, namely, a floating-point type and a Python `datetime` type, respectively. Indeed, once in this form, it is easy to use Pandas to, say, extract the month as its own column.

In [4]: Grade cell: `cell-d161fe1a08dde060` Score: 1.0 / 1.0 (Top)

```
# Observe types:
for col in ['Med', 'Admin Date', 'Units']:
    print("Column '{}' has type {}.".format(col, type(Drugs[col].iloc[0])))

# Convert strings to "natural" types:
Drugs = pd.read_csv("{}drugs.csv".format(DATA_PATH), header=0)
Drugs['Units'] = pd.to_numeric(Drugs['Units'].str.replace(',',''), errors='coerce')
Drugs['Admin Date'] = pd.to_datetime(Drugs['Admin Date'], infer_datetime_format=True)
Drugs['Month'] = Drugs['Admin Date'].dt.month

print ("\nFive random records from the `Drugs` table:")
display(Drugs.iloc[np.random.choice (len (Drugs), 5)])

assert Drugs['Units'].dtype == 'float64'
assert Drugs['Month'].dtype == 'int64'
```

```
Column 'Med' has type <class 'str'>.
Column 'Admin Date' has type <class 'str'>.
Column 'Units' has type <class 'str'>.

Five random records from the `Drugs` table:
```

	ID	Med	Admin Date	Units	Month
294	19	Med A	2012-07-04	1700	7
1659	110	Med A	2012-07-05	4900	7
1604	106	Med A	2012-07-09	5000	7
1034	69	Med A	2012-08-07	2000	8
545	36	Med A	2012-07-06	3700	7

Exercise 1 (1 point). Again, all you need to do is read and run the following code cell. It creates an SQLite database file named `drug_trial.db` and copies the Pandas dataframe from above into it as a table named `Drugs`.

The `conn` variable holds a live connection to this data.

In [5]: Grade cell: `cell-33c9cb8b19360894` Score: 1.0 / 1.0 (Top)

```
# Import Drugs_soln dataframe above to sqlite database
# Connect to a database (or create one if it doesn't exist)
conn = db.connect('drug_trial.db')
Drugs.to_sql('Drugs', conn, if_exists='replace', index=False)
pd.read_sql_query('SELECT * FROM Drugs LIMIT 5', conn)
```

```
/Users/richie/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:4: UserWarning:
The spaces in these column names will not be changed. In pandas versions < 0.14, spaces w
ere converted to underscores.
```

Out[5]:

	ID	Med	Admin Date	Units	Month
0	1	Med A	2012-07-02 00:00:00	1500	7
1	1	Med A	2012-07-06 00:00:00	1500	7
2	1	Med A	2012-07-09 00:00:00	1500	7
3	1	Med A	2012-07-11 00:00:00	1500	7
4	1	Med A	2012-07-13 00:00:00	1500	7

Exercise 2 (2 points). **Suppose you want to know the average dose, for each medication (A and B) and month ranging from July to November.**

For example, it will turn out that in July the average dose of drug A was 5,129.56 units (rounded to two decimal places), and in September the average dose of drug B was 7.04.

Write some code to perform this calculation. Store your results in a Pandas data frame named `avg_dose` having the following three columns:

- `'Month'`: The month;
- `'Med'`: The medication, either `'Med A'` and `'Med B'`;
- `'Units'`: The average dose, **rounded to 2 decimal digits**.

> You can write either Pandas code or SQL code. If using Pandas, the data exists in the `Drugs` dataframe; if using SQL, the `conn` database connection holds a table named `Drugs`.

In [6]:

Student's answer (Top)

```
### BEGIN SOLUTION
query = '''
    select Month, Med, round(avg(Units), 2) as Units
    from Drugs
    where Month >= 7 and Month <= 11
    group by Month, Med
'''
avg_dose = pd.read_sql_query (query, conn)
### END SOLUTION

# Show your solution:
display(avg_dose)
```

	Month	Med	Units
0	7	Med A	5129.56
1	8	Med A	5645.78
2	9	Med A	5311.88
3	9	Med B	7.04
4	10	Med B	5.78
5	11	Med A	10757.14
6	11	Med B	5.60

In [7]:

Grade cell: `avg_dose_test` Score: 2.0 / 2.0 (Top)

```
# Test code
# Read what we believe is the exact result (up to permutations)
avg_dose_soln = pd.read_csv('{}avg_dose_soln.csv'.format(DATA_PATH))

# Check that we got a data frame of the expected shape:
assert 'avg_dose' in globals(), "You need to store your results in a dataframe named `a
vg_dose`."
assert type(avg_dose) is type(pd.DataFrame()), "`avg_dose` does not appear to be a Pand
as dataframe."
assert len(avg_dose) == len(avg_dose_soln), "The number of rows of `avg_dose` does not
match our solution."
assert set(avg_dose.columns) == set(['Month', 'Med', 'Units']), "Your table does not ha
ve the right set of columns."

assert tibbles_are_equivalent(avg_dose, avg_dose_soln)
print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 3 (6 points). For each month, write some code to calculate the following:

- (3 points) How many patients switched from medication A to medication B? Store youre results in a Pandas dataframe named `swt_count`.
- (3 points) How many patients started on medication B, having never been on medication A before? Store your results in a Pandas dataframe named `nst_count`.

The two dataframes should have two columns: `Month` and `Count`. Again, you can choose to use SQL queries or Pandas directly to generate these dataframes.

> If it's helpful, recall that patients can only be switched from medication A to medication B, but not from B back to A.

In [8]:

Student's answer (Top)

```
# Write your solution to compute `swt_count` in this code cell.

### BEGIN SOLUTION

#============================================================
# Solution 0: A working solution using only elementary
# constructs from Python, Pandas, and SQL
#============================================================

def solve_swt_count_v0(Drugs, conn):
    # 1. Find patients who took A & B
    query = "SELECT DISTINCT ID FROM Drugs WHERE Med='{}'"
    IDs_A = set(pd.read_sql_query(query.format('Med A'), conn)['ID'])
    IDs_B = set(pd.read_sql_query(query.format('Med B'), conn)['ID'])
    IDs_switched = IDs_A & IDs_B

    # 2. Determine in which month each of these patients started taking B
    IDs_switched_str = ', '.join([str(i) for i in IDs_switched])
    query = "SELECT ID, MIN(Month) AS Month FROM Drugs WHERE ID IN ({}) AND Med='Med B'
GROUP BY ID"
    IDs_switched_first_month = pd.read_sql_query(query.format(IDs_switched_str), conn)

    # 3. Build output dataframe
    swt_count_months = []
    swt_count_counts = []
    for month in IDs_switched_first_month['Month'].unique():
        swt_count_months.append(month)
        swt_count_counts.append((IDs_switched_first_month['Month'] == month).sum())
    swt_count = pd.DataFrame({'Month': swt_count_months,
                              'Count': swt_count_counts})
    return swt_count

#============================================================
# Solution 1: "Advanced" Pandas-only solution
#============================================================
```

```
"""

# Patients can only switch from A to B or start on B, but not go from B to A.
# Therefore, we only need to count the unique values of medication for each
# patient across 3 months. There are 3 possible cases:
# 1. If the unique values are `['Med A', 'Med B']`, then the patient has
#    switched from Med A to Med B during the 3 months;
# 2. If the unique value is `['Med A']`, it means the patient hasn't
#    switched from Med A to Med B;
# 3. If the unique value is `['Med B']`, it means the patient newly
#    started Med B and had never used Med A before.

def calc_who_switched(Drugs):
    who_switched = Drugs.groupby(['ID'])['Med'].unique().to_frame(name='Unique Med')
    who_switched['Med A->B?'] = who_switched.apply(lambda x: 1 if len(x['Unique Med'])==2 else 0,axis=1)
    who_switched['Newly_started'] = who_switched.apply(lambda x: 1 if x['Unique Med'].tolist()==['Med B'] else 0,axis=1)
    Drugs_with_flags = Drugs.join(who_switched, on='ID', how='left')
    return Drugs_with_flags

def solve_swt_count_using_Pandas(Drugs):
    Drugs_with_flags = calc_who_switched(Drugs)
    month_switched = Drugs_with_flags[(Drugs_with_flags['Med A->B?']==1)&(Drugs_with_flags['Med']=='Med B')]\
                          .groupby(['ID']).first()
    swt_count = month_switched.groupby(['Month'])['Med'].count().to_frame(name='Count').reset_index()
    return swt_count

#=============================================================
# Solution 2: "Advanced" SQL-only solution
#=============================================================

def solve_swt_count_using_SQL(conn):
    query ="""
    select med_b.Month, count(distinct med_b.ID) as Count
    from
    (select ID, min(Month) as Month, Med
    from Drugs
    where Med = 'Med A'
    group by ID) med_a
    join
    (select ID, min(Month) as Month, Med
    from Drugs
    where Med = 'Med B'
    group by ID) med_b on med_a.ID = med_b.ID
    where med_a.Month <= med_b.Month
    group by med_b.Month
    """
    swt_count = pd.read_sql_query (query, conn)
    return swt_count

#=============================================================
# Pick any of the above solutions:
#=============================================================

swt_count_v0 = solve_swt_count_v0(Drugs, conn)
swt_count_Pandas = solve_swt_count_using_Pandas(Drugs)
swt_count_SQL = solve_swt_count_using_SQL(conn)

assert tibbles_are_equivalent(swt_count_v0, swt_count_Pandas)
assert tibbles_are_equivalent(swt_count_v0, swt_count_SQL)
swt_count = swt_count_v0

### END SOLUTION
```

In [9]:

Grade cell: `swt_count_test` Score: 3.0 / 3.0 (Top)

```
# Test code for exercise_a
# Read what we believe is the exact result
swt_count_soln = pd.read_csv('{}swt_count_soln.csv'.format(DATA_PATH))
```

```
# Check that we got a data frame of the expected shape:
assert 'swt_count' in globals ()
assert type (swt_count) is type (pd.DataFrame ())
assert len (swt_count) == len (swt_count_soln)
assert set (swt_count.columns) == set (['Month', 'Count'])

print ("Number of patients who switched from Med A to Med B each month:")
display (swt_count)

assert tibbles_are_equivalent (swt_count, swt_count_soln)
print ("\n(Passed!)")
```

```
Number of patients who switched from Med A to Med B each month:
```

	Count	Month
0	71	9
1	10	10

```
(Passed!)
```

In [10]:

Student's answer (Top)

```
# Write your solution to compute `nst_count` in this code cell.

### BEGIN SOLUTION

#============================================================
# Solution 0: A working solution using only elementary
# constructs from Python, Pandas, and SQL
#============================================================

def solve_nst_count_v0(Drugs, conn):
    # 1. Find patients who took A & B
    query = "SELECT DISTINCT ID FROM Drugs WHERE Med='{}'"
    IDs_A = set(pd.read_sql_query(query.format('Med A'), conn)['ID'])
    IDs_B = set(pd.read_sql_query(query.format('Med B'), conn)['ID'])
    IDs_B_only = IDs_B - IDs_A

    # 2. Determine in which month each of these patients started taking B
    IDs_B_only_str = ', '.join([str(i) for i in IDs_B_only])
    query = "SELECT ID, MIN(Month) AS Month FROM Drugs WHERE ID IN ({}) GROUP BY ID"
    IDs_B_only_first_month = pd.read_sql_query(query.format(IDs_B_only_str), conn)

    # 3. Build output dataframe
    nst_count_months = []
    nst_count_counts = []
    for month in IDs_B_only_first_month['Month'].unique():
        nst_count_months.append(month)
        nst_count_counts.append((IDs_B_only_first_month['Month'] == month).sum())
    nst_count = pd.DataFrame({'Month': nst_count_months,
                              'Count': nst_count_counts})
    return nst_count

#============================================================
# Solution 1: "Advanced" Pandas-only solution
#============================================================

def solve_nst_count_with_Pandas(Drugs):
    Drugs_with_flags = calc_who_switched(Drugs)
    patients_newly_started_medB = Drugs_with_flags[Drugs_with_flags['Newly_started']==1
]
    nst_count = patients_newly_started_medB.groupby(['ID']).first().groupby(['Month']).
count()\
                                           .reset_index()[['Month', 'Med']].rename(col
umns={'Med': 'Count'})
    return nst_count

#============================================================
# Solution 2: "Advanced" SQL-only solution
#============================================================

def solve_nst_count_with_SQL(conn):
```

```
    query ="""
        select Month, count(distinct ID) as Count
        from(
        select distinct ID, min(Month) as Month
        from Drugs
        where Med = 'Med B'
            and ID not in
                (select ID
                from Drugs
                where Med = 'Med A')
        group by ID)
        group by Month
    """
    return pd.read_sql_query(query, conn)


#============================================================
# Pick any of the above solutions:
#============================================================

nst_count_v0 = solve_nst_count_v0(Drugs, conn)
nst_count_Pandas = solve_nst_count_with_Pandas(Drugs)
nst_count_SQL = solve_nst_count_with_SQL(conn)

assert tibbles_are_equivalent(nst_count_v0, nst_count_Pandas)
assert tibbles_are_equivalent(nst_count_v0, nst_count_SQL)
nst_count = nst_count_v0
### END SOLUTION
```

In [11]:

Grade cell: `nst_count_test` Score: 3.0 / 3.0 (Top)

```
# Test code for exercise_b
# Read what we believe is the exact result
nst_count_soln_corrected = pd.read_csv ('{}nst_count_soln--corrected.csv'.format(DATA_P
ATH))
nst_count_soln_ok = pd.read_csv ('{}nst_count_soln.csv'.format(DATA_PATH))

# Check that we got a data frame of the expected shape:
assert 'nst_count' in globals ()
assert type (nst_count) is type (pd.DataFrame ())
assert (len (nst_count) == len (nst_count_soln_corrected)) or (len (nst_count) == len (
nst_count_soln_ok))
assert set (nst_count.columns) == set (['Month', 'Count'])

print ("Number of patients who newly start Med B each month:")
display (nst_count)

assert tibbles_are_equivalent(nst_count, nst_count_soln_ok) \
       or tibbles_are_equivalent(nst_count, nst_count_soln_corrected)
print ("\n(Passed!)")
```

```
Number of patients who newly start Med B each month:
```

	Count	Month
0	6	11
1	5	9
2	5	10

```
(Passed!)
```

In [12]:

```
# Some cleanup code
conn.close()
```

Fin! Well done! If you have successfully completed this problem, move on to the next one. Good luck!

problem2 (Score: 10.0 / 10.0)

1. Test cell (Score: 2.0 / 2.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 1.0 / 1.0)
5. Test cell (Score: 4.0 / 4.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Problem 2: Numpy

This problem consists of Numpy exercises. You are asked to implement a simulator for a system known as Conway's Game of Life (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life).

In [1]:
```
import numpy as np
import pandas as pd

from IPython.display import display

import matplotlib.pyplot as plt # Core plotting support
%matplotlib inline

def show_board(grid, title=None, **args):
    plt.matshow(grid, **args)
    if title is not None:
        plt.title(title)
    plt.show()
```

Important convention. To pass the test cells, your Numpy array should store *integer* values. The simplest way to convert a Numpy array of any type into one with integers is to use `astype()`. For example, consider the following boolean array, `B`:

In [2]:
```
B = np.array([[False, True,  False],
              [False, False, True],
              [True,  True,  False]])
B
```

Out[2]:
```
array([[False,  True, False],
       [False, False,  True],
       [ True,  True, False]], dtype=bool)
```

To convert it into an array of integers, use `astype()`:

In [3]:
```
A = B.astype(int)
print(A)
```

```
[[0 1 0]
```

```
[0 0 1]
[1 1 0]]
```

In this case, the conversion is done using Python's default for booleans to integers (`False` goes to 0, `True` goes to 1).

Data/test sets

Run the following code cell, which will download the missing data or solution files, if any.

In [4]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".forma
t(local_file,

body_checksum,

checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/gol/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/gol/"
DATA_PATH = ""

datasets = {'board_of_life_soln2.csv': '360fade983415eb884fa6354cfcfd56d',
            'life.csv': '93a9bc33328c46e226baabdac6a88321',
            'step.csv': 'b959690bbf59fb87ab27178eeccb15b8'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/gol/step.csv ...

'step.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/gol/life.csv ...

'life.csv' is ready!
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/gol/board_of_life_s
oln2.csv ...

'board_of_life_soln2.csv' is ready!

(All data appears to be ready.)
```

Background

In Conway's Game of Life, you have an $n \times n$ board (or grid). Let's call this board B. Each grid cell of the board, b_{ij}, exists in one of two states: alive or dead.

Starting from some initial configuration of living and dead cells, the cells evolve in discrete time steps according to the following rules:

- **Rule 0.** If the cell is alive at time t and has exactly two or three neighbors, it will remain alive at time $t+1$.
- **Rule 1.** If the cell is alive at time t and has only zero or one living neighbors, it will die from loneliness at time $t+1$.
- **Rule 2.** If the cell is alive at time t and has more than three living neighbors, it will die from overcrowding at time $t+1$.
- **Rule 3.** If the cell is dead at time t and has exactly three living neighbors, it will come alive at $t+1$.

Note that the cell changes happen *simultaneously*. That is, the board at time $t+1$ depends only on the board configuration at time t.

Example. Suppose the board is a 3×3 grid with the following initial configuration. ("1" is alive, "0" is dead.)

```
  0   1   2    <-- columns
+---+---+---+
| 0 | 1 | 1 |  row 0
+---+---+---+
| 0 | 1 | 0 |  row 1
+---+---+---+
| 1 | 0 | 0 |  row 2
+---+---+---+
```

At the next time step, the cell at positions (row 0, column 1) will be alive by Rule 0 because it has two living neighbors: at (0, 2) and (1, 1). Similarly, the cells at (1, 1) and (0, 2) will remain alive. However, the cell at (2, 0) will die from lonelieness by Rule 1. As for the currently dead cells, only (1, 0) and (1, 2) have exactly three neighbors, so by Rule 3 they will be resurrected at the next time step. The other dead cells will stay dead. Thus, the final configuration is as follows:

```
  0   1   2    <-- columns
+---+---+---+
| 0 | 1 | 1 |  row 0
+---+---+---+
| 1 | 1 | 1 |  row 1
+---+---+---+
| 0 | 0 | 0 |  row 2
+---+---+---+
```

If you were to evolve this new configuration, Rule 2 would come into play since the cell at (1, 1) has four living neighbors, and so it would have to die from overcrowding in the next time step.

Exercises

The initial configuration of live cells are stored in a comma-separated values (CSV) file. Only the coordinates of live cells are stored in this file. Here is what the file looks like, when read in and stored as a Pandas dataframe.

In [5]:
```
board_coords = pd.read_csv('{}life.csv'.format(DATA_PATH))
board_coords.head()
```

Out[5]:

	x	y
0	4	5
1	4	6
2	4	7
3	5	5
4	6	5

Exercise 0 (2 points). Implement a function to convert a coordinates dataframe, like the one shown above, into a dense 2-D array that represents the grid.

The function has the signature,

```
def make_board(coords_df, n):
        ...
```

where `coords_df` is a dataframe with `'x'` and `'y'` columns corresponding to the row and column coordinates of a living cell, and `n` is the dimension of the board (i.e., the board is of size `n`-by-`n`).

This function should return an `n`-by-`n` Numpy array of 0 and 1 values, where 1 means "alive" and 0 means dead. Per the note above, be sure its entries are of integer type.

For example, suppose you call your function on the above dataframe as follows.

```
board = make_board(board_coords)
```

Then `board` should have `board[4][5] == 1`, `board[4][6] == 1`, `board[4][7] == 1`, `board[5][5] == 1`, etc.

In [6]:

Student's answer (Top)

```
import numpy as np
import pandas as pd
from scipy.sparse import coo_matrix

def make_board(coords_df, n=50):
    ### BEGIN SOLUTION
    row = coords_df['x']
    col = coords_df['y']
    val = [1] * len(row)
    board = coo_matrix((val, (row, col)), shape=(n, n))
    board = board.toarray()
    return board
    ### END SOLUTION

board_of_life = make_board(board_coords)
print("Board dimensions:", board_of_life.shape)
print("\nUpper 10x20 corner:\n", board_of_life[:10, :20])
```

```
Board dimensions: (50, 50)

Upper 10x20 corner:
 [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1]
 [0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1]
 [0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1]
 [0 0 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1]
 [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1]
 [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1]]
```

In [7]:

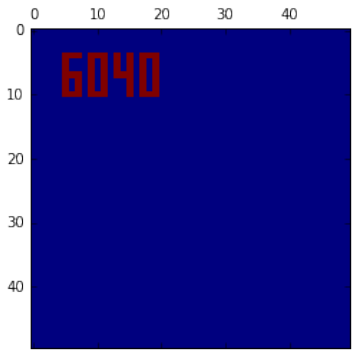
Grade cell: `make_board_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `make_board_test`

if False:
    np.savetxt("board_of_life_soln2.csv", board_of_life, fmt="%d", delimiter=",")

board_of_life_soln = np.loadtxt("board_of_life_soln2.csv", delimiter=",", dtype=int)
compare_boards = (board_of_life == board_of_life_soln)
mismatches_coords = np.where(compare_boards == False)
mismatches_df = pd.DataFrame(np.array([mismatches_coords[0], mismatches_coords[1]]).T,
columns=['x', 'y'])
if len(mismatches_df) > 0:
    display(mismatches_df)
    assert False, "Your solution does not match the instructor solution at these follow
ing positions."

print("\n(Passed!)")
```

```
(Passed!)
```

To aid your debugging, here is a convenience function for displaying the board as a graphic image. Depending on your system, purple or blue cells represent zeros; yellow or red cells represent ones.

In [8]:
```
show_board(board_of_life)
```



Exercise 1 (3 points). Implement a function that counts the number of living neighbors for each cell. The function should have the signature,

```
def count_living_neighbors(board):
    ...
```

It should return a new 2-D Numpy array of the same size as `board`. However, this array should contain a count of living neighbors at each position `(i, j)`. For example, suppose the board is the following:

```
  0   1   2    <-- columns
+---+---+---+
| 0 | 1 | 1 |  row 0
+---+---+---+
| 0 | 1 | 0 |  row 1
+---+---+---+
| 1 | 0 | 0 |  row 2
+---+---+---+
```

Then `count_living_neighbors()` should return the following Numpy array:

```
  0   1   2    <-- columns
+---+---+---+
| 2 | 2 | 2 |  row 0
+---+---+---+
| 3 | 3 | 3 |  row 1
+---+---+---+
| 1 | 2 | 1 |  row 2
+---+---+---+
```

To help you get started, the code below initializes an output board. Your task is to figure out how to update it to count the neighbors of every cell. You may assume the board is square and of size 3 x 3 or larger.

In [9]:
Student's answer (Top)

```
def count_living_neighbors(board):
    assert board.shape[0] == board.shape[1], "`board` must be square."
    assert board.shape[0] >= 3, "`board` must be at least 3 x 3."

    count = np.zeros(board.shape, dtype=int)
    ### BEGIN SOLUTION
    count[:, :-1]   += board[:, 1:]    # From the right
    count[:-1, :-1] += board[1:, 1:]   # From the lower-right
    count[:-1, :]   += board[1:, :]    # From below
    count[:-1, 1:]  += board[1:, :-1]  # From the lower-left
```

```
    count[:, 1:]    += board[:, :-1]   # From the left
    count[1:, 1:]   += board[:-1, :-1] # From the upper-left
    count[1:, :]    += board[:-1, :]   # From above
    count[1:, :-1]  += board[:-1, 1:]  # From the upper-right
    ### END SOLUTION
    return count

demo_board = np.array([[0, 1, 1],
                       [0, 1, 0],
                       [1, 0, 0]])
print("==> Demo board:\n{}".format(demo_board))
print("\n==> Counts:\n{}".format(count_living_neighbors(demo_board)))
```

```
==> Demo board:
[[0 1 1]
 [0 1 0]
 [1 0 0]]

==> Counts:
[[2 2 2]
 [3 3 3]
 [1 2 1]]
```

In [10]:

Grade cell: `count_living_neighbors_test1` Score: 1.0 / 1.0 (Top)

```
# Test cell: `count_living_neighbors_test1`

your_demo_count = count_living_neighbors(demo_board)
demo_count_soln = np.array([[2, 2, 2],
                            [3, 3, 3],
                            [1, 2, 1]])

assert type(your_demo_count) is np.ndarray, "Your function needs to return an Numpy arr
ay."
assert your_demo_count.shape == demo_count_soln.shape, \
       "Your counts have the wrong shape: it's {} instead of {}.".format(your_demo_coun
t.shape,
                                                                         demo_count_sol
n.shape)

assert your_demo_count.dtype == 'int64', \
       "Make sure your count array has integer elements (they appear to be {}
instead).".format(your_demo_count.dtype)

matches = (your_demo_count == demo_count_soln)
assert matches.all(), \
       "Counts for `demo_board` does not match expected counts, which are\n==>\n{}".for
mat(demo_count_soln)

print("\n(Passed, part 1.)")
```

```
(Passed, part 1.)
```

In [11]:

Grade cell: `count_living_neighbors_test2` Score: 2.0 / 2.0 (Top)

```
# Test cell: `count_living_neighbors_test2`

board_of_life_counts = count_living_neighbors(board_of_life)
assert board_of_life_counts.shape == board_of_life.shape, \
       "Counts shape, {}, does not match original board, {}.".format(board_of_life_coun
t.shape,
                                                                     board_of_life.shap
e)

from numpy.random import choice
for i in choice(board_of_life.shape[0], replace=False, size=7):
    ii_range = range(max(0, i-1), min(board_of_life.shape[0], i+2))
    for j in choice(board_of_life.shape[1], replace=False, size=7):
        jj_range = range(max(0, j-1), min(board_of_life.shape[1], j+2))
        your_count = board_of_life_counts[i][j]
        true_count = 0
```

```
        for ii in ii_range:
            for jj in jj_range:
                if not (ii == i and jj == j):
                    true_count += int(board_of_life[ii][jj])

        err_msg = "Your count at {} should be {} but is instead {}. Check the neighborh
ood:\n==> {}"
        ij_neighborhood = board_of_life[min(ii_range):max(ii_range)+1, min(jj_range):ma
x(jj_range)+1]
        assert your_count == true_count, \
               err_msg.format((i, j), true_count, your_count, ij_neighborhood)

print("\n(Passed, part 2.)")
```

```
(Passed, part 2.)
```

Recall the rules of the game:

- **Rule 0.** If the cell is alive at time t and has exactly two or three neighbors, it will remain alive at time $t+1$.
- **Rule 1.** If the cell is alive at time t and has only zero or one living neighbors, it will die from loneliness at time $t+1$.
- **Rule 2.** If the cell is alive at time t and has more than three living neighbors, it will die from overcrowding at time $t+1$.
- **Rule 3.** If the cell is dead at time t and has exactly three living neighbors, it will come alive at $t+1$.

Exercise 2 (4 point). Suppose you are given a board at time t. Compute the board at time $t+1$, according to the rules above.

You should specifically complete the function,

```
def step(board):
        ...
```

It should return the new board after applying the four rules. To help you out, we've implemented the first rule (Rule 0).

For example, given this `board`,

```
  0   1   2    <-- columns
+---+---+---+
| 0 | 1 | 1 |  row 0
+---+---+---+
| 0 | 1 | 0 |  row 1
+---+---+---+
| 1 | 0 | 0 |  row 2
+---+---+---+
```

your function should return this board:

```
  0   1   2    <-- columns
+---+---+---+
| 0 | 1 | 1 |  row 0
+---+---+---+
| 1 | 1 | 1 |  row 1
+---+---+---+
| 0 | 0 | 0 |  row 2
+---+---+---+
```

Hint: Boolean operations can help simplify the logic and checking for this problem:

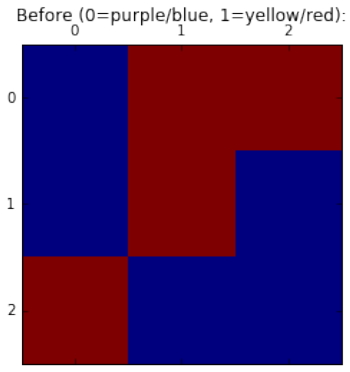
```
# Boolean "and"
    assert (0 & 0) == 0  # Also: (False and False) == False
    assert (0 & 1) == 0  #       (False and True) == False
    assert (1 & 0) == 0  #       (True and False) == False
    assert (1 & 1) == 1  #       (True and True) == True

    # Boolean "or"
    assert (0 | 0) == 0
```

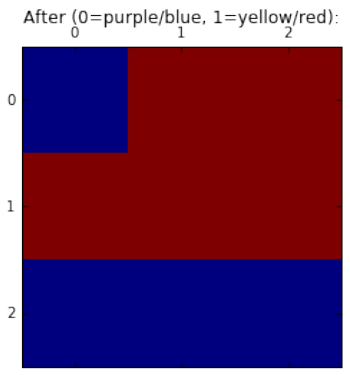
```
assert (0 | 1) == 1
assert (1 | 0) == 1
assert (1 | 1) == 1

# Boolean "exclusive-or." Same as "not equal"
assert (0 ^ 0) == 0
assert (0 ^ 1) == 1
assert (1 ^ 0) == 1
assert (1 ^ 1) == 0
```

In [12]:

Student's answer (Top)

```
def step(board):
    counts = count_living_neighbors(board)
    new_board = board.copy()

    # Rule 0. Alive with two or three neighbors ==> lives.
    new_board |= board & ((counts == 2) | (counts == 3))

    ### BEGIN SOLUTION
    # Rule 1. Alive with zero or one neighbors ==> dies. (lonely)
    new_board ^= (board & (counts <= 1))

    # Rule 2. Alive with more than three neighbors ==> dies. (overcrowded)
    new_board ^= (board & (counts > 3))

    # Rule 3. Dead with exactly three neighbors ==> lives. (resurrection)
    new_board |= (~board & (counts == 3))
    ### END SOLUTION

    return new_board

show_board(demo_board, title='Before (0=purple/blue, 1=yellow/red):')

new_demo_board = step(demo_board)
show_board(new_demo_board, title = 'After (0=purple/blue, 1=yellow/red):')
```





In [13]:

Grade cell: `step_test1` Score: 1.0 / 1.0 (Top)

```
# Test cell: `step_test1`

assert (new_demo_board == np.array([[0, 1, 1],
                                    [1, 1, 1],
                                    [0, 0, 0]])).all()

print("\n(Passed, part 1.)")
```

```
(Passed, part 1.)
```

In [14]:

Grade cell: `step_test2` Score: 4.0 / 4.0 (Top)

```
# Test cell: `step_test2`

step_soln = np.loadtxt('{}step.csv'.format(DATA_PATH), delimiter=',', dtype=int)
your_step = step(board_of_life)

matches = (your_step == step_soln)
if not matches.all():
    print("*** Detected mismatches. ***")
    mismatches = np.where(~matches)
    for i, j in zip(mismatches[0], mismatches[1]):
        print("({}, {}) was {} instead of {}.".format(i, j, your_step[i, j], step_soln[
i, j]))
    assert False

print("\n(Passed!)")
```

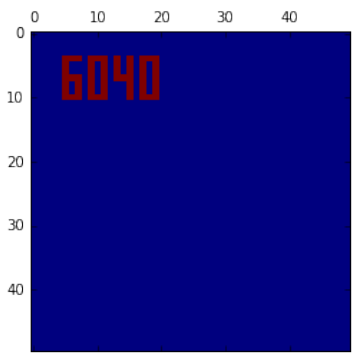
```
(Passed!)
```

Full simulation

The following code creates a widget that allows you to step through many iterations of the game. There is nothing to write here; it's just for your edification to see that you've completed a working implementation. The initial "6040" pattern from above will eventually converge to a repeating pattern.

In [15]:

```
MAX_STEPS = 75
N = board_of_life.shape[0]
all_boards = np.zeros((N, N, MAX_STEPS), dtype=int)

all_boards[:, :, 0] = board_of_life
for t in range(1, MAX_STEPS):
    all_boards[:, :, t] = step(all_boards[:, :, t-1])

def display_board(t=0):
    show_board(all_boards[:, :, t])

from ipywidgets import interact
interact(display_board, t=(0, MAX_STEPS), continuous_update=False)
```



Out[15]: <function __main__.display_board>

Fin! This problem is the last one for Midterm 2.

English