Lecture 18a: Errors!

Breaking our code on purpose!

(And otherwise handling errors)

Dealing with problems in code

```python
x = 10
y = 0

z = x / y
```

```
In [52]: z = x / y
Traceback (most recent call last):

  File "<ipython-input-52-ecc64115d3c8>", line 1, in <module>
    z = x / y

ZeroDivisionError: division by zero
```

What are some ways we can deal with this?

1. We can leave it as-is; the traceback gives a clear error message

Dealing with problems in code

```python
x = 10
y = 0
assert(y != 0), 'Error: y cannot be 0'
z = x / y
```

```
In [54]: assert(y != 0), 'Error: y cannot be 0'
Traceback (most recent call last):

  File "<ipython-input-54-4bb5e71fe6f1>", line 1, in <module>
    assert(y != 0), 'Error: y cannot be 0'

AssertionError: Error: y cannot be 0
```

2. We can force the program to halt *based on a conditional statement*. This lets us control the flow, raise custom warnings, and prevent errors that might otherwise pass unnoticed.

Dealing with problems in code

```python
x = 10
y = 0

try:
    z = x / y
except ZeroDivisionError:
    print('Error: y cannot be 0')
```

```
In [55]: try:
    ...:     z = x / y
    ...: except ZeroDivisionError:
    ...:     print('Error: y cannot be 0')
Error: y cannot be 0
```

3. We can *capture the error* using a try/except block, and then proceed with our program without halting.

Dealing with problems in code

```
x = 10
y = [0]
try:
    z = x / y[0]
except ZeroDivisionError:
    print('Error: y cannot be 0')
except IndexError:
    print('Error: the length of list y is not correct')
except:
    print('Something else went wrong...')
```

```
x = 10
y = [0]
```

```
Error: y cannot be 0
```

```
x = 10
y = []
```

```
Error: the length of list y is not correct
```

```
x = 10
y = [5]
```

```
In [59]: z
Out[59]: 2.0
```

We can chain except statements together, just like elif, and conclude with a catch-all except at the end. Note that the last except block will capture ALL tracebacks, and is not generally good practice to do.

Combining assert with isinstance

```python
def string_fixer(s):
    s = s.lower().strip()
    if s.startswith('a'):
        return "it's an a!"
    else:
        return "it's not an a."
```

Leaving it like this is called “duck typing”

But how can we have more control if we need it?

```
In [61]: string_fixer(42)
Traceback (most recent call last):

  File "<ipython-input-61-03a78e8eb66f>", line 1, in <module>
    string_fixer(42)

  File "<ipython-input-60-c0604abdd377>", line 2, in
string_fixer
    s = s.lower().strip()

AttributeError: 'int' object has no attribute 'lower'
```

Combining assert with isinstance

```
In [80]: isinstance('Hello world!', str)
Out[80]: True

In [81]: isinstance('Hello world!', int)
Out[81]: False
```

isinstance returns True if the first argument is an instance of the second argument, else False.

Combining assert with isinstance

```
In [80]: isinstance('Hello world!', str)
Out[80]: True

In [81]: isinstance('Hello world!', int)
Out[81]: False
```

isinstance returns True if the first argument is an instance of the second argument, else False.

```
class MyClass():
    pass

my_instance = MyClass()
```

```
In [68]: isinstance(my_instance, MyClass)
Out[68]: True
```

It works with classes we write ourselves also.

Combining assert with isinstance

```python
def string_fixer(s):
    assert(isinstance(s, str)), 'string_fixer requires string arg'
    s = s.lower().strip()
    if s.startswith('a'):
        return "it's an a!"
    else:
        return "it's not an a."
```

```
In [64]: string_fixer(42)
Traceback (most recent call last):

  File "<ipython-input-64-03a78e8eb66f>", line 1, in <module>
    string_fixer(42)

  File "<ipython-input-63-ae51ef981c60>", line 2, in string_fixer
    assert(isinstance(s, str)), 'string_fixer requires string arg'

AssertionError: string_fixer requires string arg
```

Combining assert with isinstance

```python
def math_some_numbers(a, b):
    val = (a + b) * 2
    return val
```

```
In [70]: math_some_numbers(10, 20)
Out[70]: 60

In [71]: math_some_numbers('Hello', 'World')
Out[71]: 'HelloWorldHelloWorld'
```

This code runs incorrectly, but without errors.

This is the worst result!

Combining assert with isinstance

```python
import numbers
#https://docs.python.org/3/library/numbers.html

def math_some_numbers(a, b):
    assert(isinstance(a, numbers.Number) and
           isinstance(b, numbers.Number)), 'Must pass in numeric arguments!'
    val = (a + b) * 2
    return val
```

Note that “numbers” is a standard Python library that gives us access to the base class of all numeric data types. Alternatively, we could test isinstance against the int and float types separately for both a and b:

```python
assert((isinstance(a, int) or isinstance(a, float)) and
       (isinstance(b, int) or isinstance(b, float)))
```

Combining assert with isinstance

```python
import numbers
#https://docs.python.org/3/library/numbers.html

def math_some_numbers(a, b):
    assert(isinstance(a, numbers.Number) and
           isinstance(b, numbers.Number)), 'Must pass in numeric arguments!'
    val = (a + b) * 2
    return val
```

```
In [77]: math_some_numbers(10, 20)
Out[77]: 60

In [78]: math_some_numbers('Hello', 'World')
Traceback (most recent call last):

  File "<ipython-input-78-6aeb3b9d2778>", line 1, in <module>
    math_some_numbers('Hello', 'World')

  File "<ipython-input-76-f89ab47dfdb6>", line 6, in math_some_numbers
    isinstance(b, numbers.Number)), 'Must pass in numeric arguments!'

AssertionError: Must pass in numeric arguments!
```

Lecture 18b: Simulations

A sensible way to apply classes and methods.

The Schelling Model

- In 1971, used checkers and checkers boards to simulate neighborhood segregation.
- Showed that a relatively small preference for similar neighbors (less than 50%) can result in segregated neighborhoods, all else equal.

The Schelling Model

1. Agents are a member of one of at least two categories (originally red and black for checkers)
2. Agents are randomly assigned a spot on the board
3. On its turn, an agent looks at its 8 neighbors, and:
   a. Counts the proportion that are the same color as them
   b. Decides they’re “happy” if the proportion meets or exceeds the threshold
   c. Moves to a new random location if they are not happy
4. Simulation ends when no one moves, or improves their position

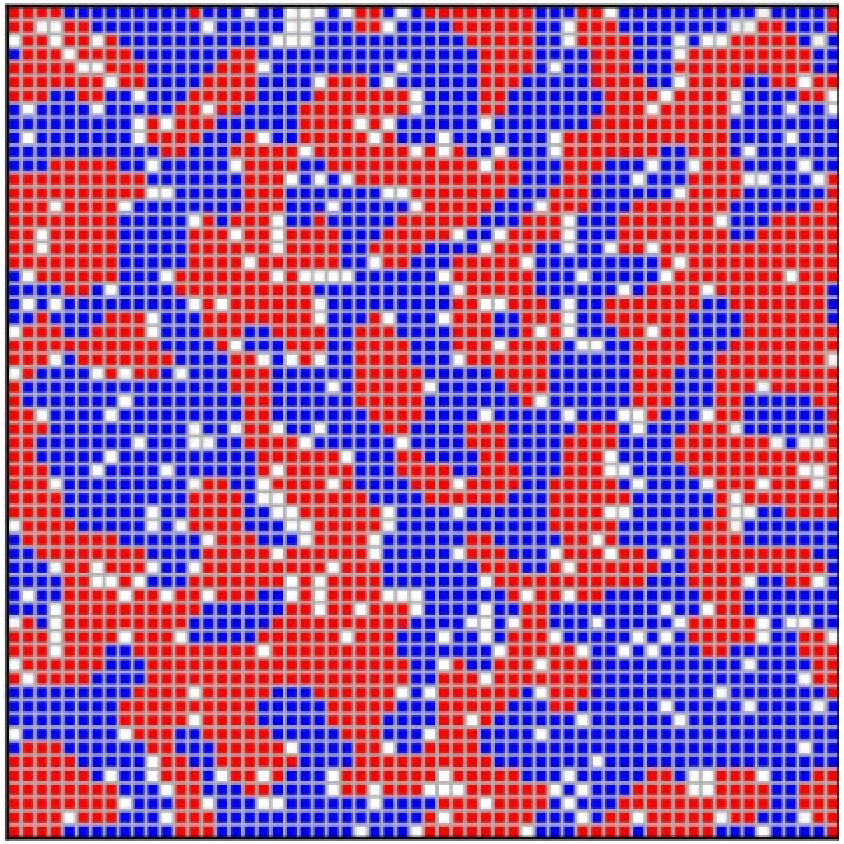
The Schelling Model

1. Agents are a member of one of at least two categories (originally red and black for checkers)
2. Agents are randomly assigned a spot on the board
3. On its turn, an agent looks at its 8 neighbors, and:
   a. Counts the proportion that are the same color as them
   b. Decides they're "happy" if the proportion meets or exceeds the threshold
   c. Moves to a new random location if they are not happy
4. Simulation ends when no one moves, or improves their position

The Schelling Model

1. Agents are a member of one of at least two categories (originally red and black for checkers)
2. Agents are randomly assigned a spot on the board
3. On its turn, an agent looks at its 8 neighbors, and:
   a. Counts the proportion that are the same color as them
   b. Decides they're "happy" if the proportion meets or exceeds the threshold
   c. Moves to a new random location if they are not happy
4. Simulation ends when no one moves, or improves their position

The Schelling Model



Example of a final equilibrium with same-preference of 0.4

The Schelling Model

- An <u>Agent</u> class to represent each individual actor in the game
  - How to move
  - How to decide if they're happy
  - How to look at their neighbors
- A <u>World</u> class that holds knowledge of the grid and agents
  - How to set everything up
  - How to provide information to Agents
  - How to report on the state of the simulation
  - How to run the simulation

The Schelling Model

- An <u>Agent</u> class to represent each individual actor in the game
  - How to move
  - How to decide if they're happy
  - How to look at their neighbors
- A <u>World</u> class that holds knowledge of the grid and agents
  - How to set everything up
  - How to provide information to Agents
  - How to report on the state of the simulation
  - How to run the simulation

The Schelling Model

- An Agent class to represent each individual actor in the game
  - How to move
  - How to decide if they're happy
  - How to look at their neighbors
- A World class that holds knowledge of the grid and agents
  - How to set everything up
  - How to provide information to Agents
  - How to report on the state of the simulation
  - How to run the simulation

The Agent Class

```
class Agent():
    def __init__(self):
        #needs to know its color and its same preference
        pass

    def move(self):
        #decide if it wants to move
        #move to new position if it does
        #uses the self.am_i_happy method
        pass
```

The Agent Class

```
    def am_i_happy(self):
        #return a boolean for whether an agent is happy in its
        #current location
        # uses the self.locate_neighbors method
        pass

    def locate_neighbors(self):
        #given a location, return a list of all patches that count
        #as neighbors
        pass
```

The World Class

```python
class World():
    def __init__(self):
        #stores the grid as a container of some sort
        #calculates how many agents there should be
        #initializes agents in starting locations
        #uses the build_grid and, build_agents, and init_world methods
        pass

    def build_grid(self):
        #sets up the world agents can move in, returning a dict
        pass
```

The World Class

```
def build_agents(self):
    #generates the list of agents that can be iterated over
    pass

def init_world(self):
    #sets up the starting conditions of the world
    pass

def find_vacant(self):
    #find a list of empty patches and returns a random one
    pass
```

The World Class

```python
    def report_integration(self):
        #generates a report at the end of the current round
        pass

    def report(self):
        #generate the final report at model end
        pass

    def run(self):
        #executes the model as set up
        pass
```

Execution

```
world = World()
world.run()
```

Details: Imagining the grid

- x-y coordinates, e.g. (0,0), (0,1), (12,34), etc.
- Is it a torus?

(0,0)	(1,0)	(2,0)	⋯	(n,0)
(0,1)	(1,1)	(2,1)	⋯	(n,1)
(0,2)	(1,2)	(2,2)	⋯	(n,2)
⋮	⋮	⋮		⋮
(0,n)	(1,n)	(2,n)	⋯	(n,n)

Details: Imagining the grid

- x-y coordinates, e.g. (0,0), (0,1), (12,34), etc.
- Is it a torus?
- A dictionary with keys equal to tuples, and values equal to agent instances:

```
{(13,34):<Agent1>,
 (13,35):None,
 (13,36):<Agent47>,
 …}
```

(0,0)	(1,0)	(2,0)	⋯	(n,0)
(0,1)	(1,1)	(2,1)	⋯	(n,1)
(0,2)	(1,2)	(2,2)	⋯	(n,2)
⋮	⋮	⋮		⋮
(0,n)	(1,n)	(2,n)	⋯	(n,n)

Details: Imagining the grid

- x-y coordinates, e.g. (0,0), (0,1), (12,34), etc.
- Is it a torus?
- A dictionary with keys equal to tuples, and values equal to agent instances:

```
{(13,34):<Agent1>,
 (13,35):None,
 (13,36):<Agent47>,
 …}
```

- But what if the “patches” need to know more than who is there?
  - Resources for agents to take?
  - Environmental factors that affect agents?
  - Conditions that spread to neighbor patches?

(0,0)	(1,0)	(2,0)	⋯	(n,0)
(0,1)	(1,1)	(2,1)	⋯	(n,1)
(0,2)	(1,2)	(2,2)	⋯	(n,2)
⋮	⋮	⋮		⋮
(0,n)	(1,n)	(2,n)	⋯	(n,n)

Details: Imagining the grid

- x-y coordinates, e.g. (0,0), (0,1), (12,34), etc.
- Is it a torus?
- A dictionary with keys equal to tuples, and values equal to agent instances:

```
{(13,34):<Agent1>,
 (13,35):None,
 (13,36):<Agent47>,
 …}
```

- But what if the "patches" need to know more than who is there?
  - Resources for agents to take?
  - Environmental factors that affect agents?
  - Conditions that spread to neighbor patches?
- We could make each location a class instance also!

```
class Patch():
    def __init__(self, location):
        self.location = location
```

(0,0)	(1,0)	(2,0)	···	(n,0)
(0,1)	(1,1)	(2,1)	···	(n,1)
(0,2)	(1,2)	(2,2)	···	(n,2)
⋮	⋮	⋮		⋮
(0,n)	(1,n)	(2,n)	···	(n,n)

Full model

https://github.com/levyjeff/simple_abm