CS1110 Spring 2016

Assignment 4: Due Friday Mar 11 at 6pm

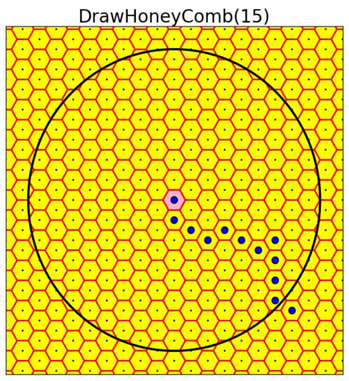
You must work either on your own or with one partner. If you work with a partner, you and your partner must first register as a group in CMS (this requires an invitation issued by one of you on CMS and the other of you accepting it on CMS) and then submit your work as a group.

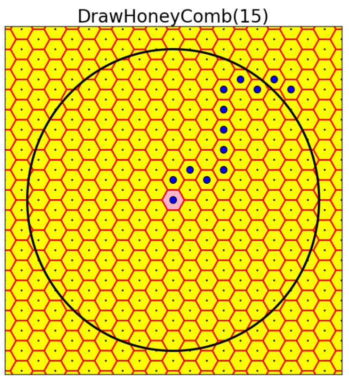
You may discuss background issues and general solution strategies with others, but the programs you submit must be the work of just you (and your partner). We assume that you are thoroughly familiar with the discussion of academic integrity that is on the course website. Any doubts that you have about "crossing the line" should be discussed with a member of the teaching staff before the deadline, *and you should also document such situation as a comment in the header of your submission file(s).*

Topics. Iteration with `for` and `while`. Using `randint`. Random walks. Estimating averages. The assignment is based on lectures through March 3 and Lab 5. The March 3 demo `ShowRandomWalk.py` is particularly relevant.

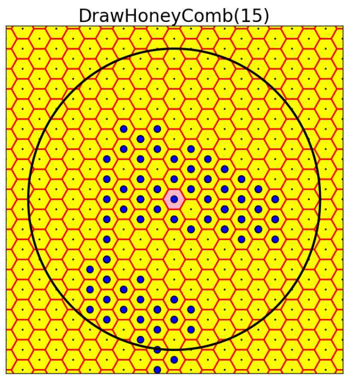
1 Background

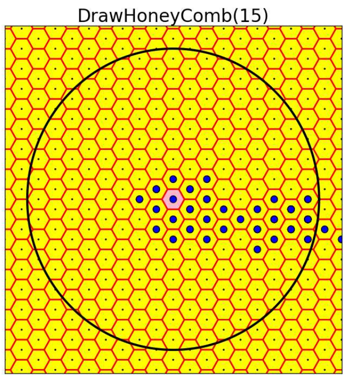
This assignment is about random walks on a "honeycomb" that is made up of hexagonal cells. You will write code that simulates a bee that starts at the center of the honeycomb and then randomly hops from cell to cell until he/she "escapes" by landing on a cell that is entirely outside a given circle. The Drones have smart phones with GPS, so their meanderings are somewhat directed:





(The blue dot means that the Bee visited that cell at least once on its journey.) On the other hand, the Worker bees are without GPS. Thus deprived, their meanderings are much more random and as a result, they take longer to escape:





By simulating hundreds of escapes, you will be able answer questions like these:

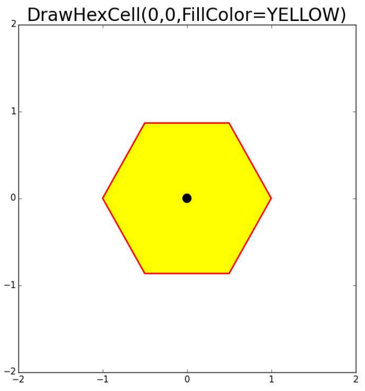
For a honeycomb of a given size, how long does it take (on average) for a Drone to escape?

For a honeycomb of a given size, how long does it take (on average) for a Worker to escape?

To get started we talk about "honeycomb geometry" and how to use what is in the module `HexTools.py` that we provide. This module and everything else you need is in `A4.zip`.

1.1 Hex-cells

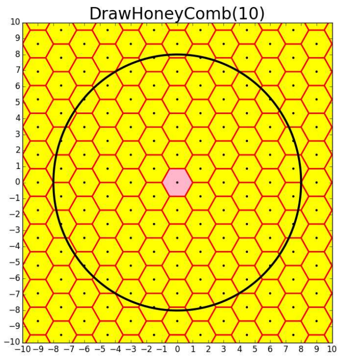
A *regular hexagon* is a polygon with six equal sides. We define a *hex-cell* to be a regular hexagon with sides that have unit length. The procedure `DrawHexCell` can be used to situate a hex-cell anywhere in the window:



The "dot" is located at the center of the displayed hex-cell. The Application Script in `HexTools.py` illustrates how to use `DrawHexCell`. Do not spend any time worrying about how this graphics procedure works. Just read its doc-string specification and know how to use it.

1.2 Honeycombs

We can tile a $2m$-by-$2m$ square with hex-cells like this:



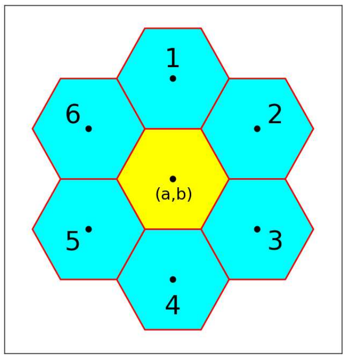
We refer to this as a *size-m honeycomb.* Here are some facts and definitions:

- The highlighted cell in the middle is centered at (0,0) and will be called the *center cell.*
- The *honeycomb circle* has radius $r = m - 2$ and is centered at (0,0).
- A cell that is entirely outside the honeycomb circle is called an *outside cell.*
- A cell with center (x, y) is an outside cell if $\sqrt{x^2 + y^2} > r + 1$.

The module HexTools.py has a procedure DrawHoneycomb that can be used to draw a size-m honeycomb together with the associated honeycomb circle. The Application Script in HexTools.py gives examples that use DrawHoneycomb. Do not spend any time worrying about how this graphics procedure works. Just read its doc-string specification and know how to use it.

1.3 The Neighbors of a Hex-cell

A given hex-cell with center (a, b) has six neighbors which we index 1 through 6:



For your information, if we set $\delta_1 = 3/2$ and $\delta_2 = \sqrt{3}/2$, then the centers of the six neighbors of the hex-cell at (a, b) are given in the following table:

Neighbor	Center
1	$(a, b + 2\delta_2)$
2	$(a + \delta_1, b + \delta_2)$
3	$(a + \delta_1, b - \delta_2)$
4	$(a, b - 2\delta_2)$
5	$(a - \delta_1, b - \delta_2)$
6	$(a - \delta_1, b + \delta_2)$

We supply a function in HexTools.py that makes it easy to compute these centers:

```
def NeighborCenter(a,b,i):
    """ Returns floats u and v with the property that
    (u,v) is the center of neighbor i.

    PreC: a and b are floats, (a,b) is the center of a hex-cell, and
    i is an int that satisfies 1<=i<=6.
```

This is our first example of a function that returns more than one value. Here is how one might use such a function:

```
from SimpleGraphics import *
from HexTools import *
MakeWindow(10,bgcolor=WHITE,labels=True)
x = 3
y = 4
j = 5
DrawHexCell(x,y,FillColor=YELLOW)   # Draw a yellow hex-cell centered at (3,4)
f,g = NeighborCenter(x,y,j)         # (f,g) is the center of neighbor 5.
DrawHexCell(f,g,FillColor=CYAN)     # Draw neighbor 5 and color it cyan
ShowWindow()
```

Notice that the two output destinations in the line calling NeighborCenter are separated with a comma.

Take a look at the procedure DrawNeighbors that is part of the given module HexTools.py. It draws all six neighbors of a given hex-cell. Check out the Application Script where it is used.

2 Getting Set Up

We provide you with a skeleton module `MyHive.py`, the essence of which we reproduce right here:

```
from SimpleGraphics import *
from random import randint as randi
from HexTools import *

def ShowBeeline(B):
    pass

def MakeBeeline(m,GPS):
    pass

def Dist(x,y):
    pass

def AveBeeline(m,GPS,Nmax):
    pass

if __name__ == '__main__':
   # some debugging help follows
```

The Application Script is set up to help you debug `ShowBeeline`, a graphics procedure that depicts the cells that were visited by an escaped bee. After that procedure is working, you will develop the function `MakeBeeline` that returns a string that encodes all the hop directions associated with the bee's journey to freedom. It will be handy to have a function `Dist` that computes the bee's distance to the origin (0,0). Finally, you will implement a function `AveBeeline` that can be used to estimate the average number of hops that it takes a bee to for escape a honeycomb of given size.

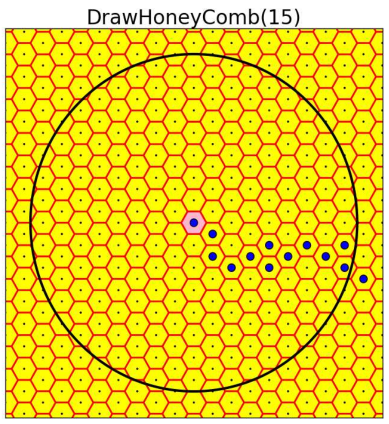
3 Beelines and Their Display

A *beeline string* is made up of the characters from `'123456z'`. Here is an example:

```
B = '3z432z2z42z2z3z24zz3'
```

A beeline string encodes what the bee does at every step in its journey. A digit means that the bee moved to a neighbor tile. Thus, because `B[4]` is `'2'`, we know that the bee's fourth hop was from its current cell to neighbor 2 of the current cell. A `'z'` means that the bee stayed put. Thus, because `B[5]` is `'z'` we know that the bee's fifth hop was basically a "hop in place"– the bee did not move. Here is a complete "decoding" of the above beeline string together with a marked up honeycomb showing the cells that were visited:

```
(Start at center cell)
Move to neighbor 3
Stay put.
Move to neighbor 4
Move to neighbor 3
Move to neighbor 2
Stay put
Move to neighbor 2
Stay put
Move to neighbor 4
Move to neighbor 2
Stay put
Move to neighbor 2
Stay put
Move to neighbor 3
Stay put
Move to neighbor 2
Move to neighbor 4
Stay put
Stay put
Move to neighbor 3
(Escaped)
```



Note: you cannot tell from the honeycomb graphic where the "stay puts" are located. Nor can you deduce if a marked cell was visited more than once.

Implement the function `ShowBeeline(B)` so that it puts a blue dot on every cell that is visited on the escape journey defined by the beeline string `B`. `ShowBeeline` has two preconditions:

1. A size-m honeycomb has been added to the current plot window for some integer m.
2. The escape route encoded in `B` is a escape from a size-m honeycomb.

Here are some requirements and hints:

- Use `DrawDisk` for the blue dots. Set the radius to 0.3.
- Define a pair of variables that will keep track of the x and y location of the bee as the simulation progresses.
- Use a `for` loop that "walks down" the input string `B`. As it does this, use `NeighborCenter` to determine the coordinates of the "next stop" and update the xy location variables accordingly.

The given `MyHive.py` module has been set up with an Application Script that checks to see what your implementation of `ShowBeeline` does with a pair of beeline string examples. Both those examples should result in the hopping pattern shown above.

4 Making a Beeline String

We now turn our attention to the simulation of the honeycomb random walk. We need rules that determine just how the bee hops from cell to cell. In particular, how does the bee figure out whether to stay put or whether to hop to a neighbor cell? And in the latter case, how is the neighbor cell selected? The rules are different for Workers and Drones.

4.1 The Workers Use a "No-GPS" Rule

Assume that a Worker bee is currently situated at (a, b), the center of some hex-cell in the honeycomb. The Worker does this:

- It rolls a (tiny, bee-sized) die: `i = randi(1,6)`. (Assume the line `from random import randint as randi` occurred previously)
- It moves to the center of neighbor i. (Hint: Update a and b so that (a, b) specifies the Worker's new location. Make use of `NeighborCenter`.)

We will call this the "No-GPS" rule because the hops are totally random. Without a GPS the Worker has no idea that some hop directions are better than others.

4.2 The Drones Use a "GPS" Rule

Assume that a Drone is currently situated at (a, b), the center of some hex-cell in the honeycomb. The Drone does this:

- It rolls a (tiny, bee-sized) die: `i = randi(1,6)`.
- It moves to the center of neighbor i if that increases its distance from (0,0). Otherwise, it stays put. (Hint: Update a and b so that (a, b) specifies the Drone's new location. Make use of `NeighborCenter`.)

We will call this the "GPS" rule because it pays attention to destination. The Drone is trying to reach an outside hex-cell and never makes a move unless there is progress in that regard. That means that the Drone doesn't make a move to neighbor i unless the distance from that hex-cell to (0,0) is greater than the Drone's current distance to (0,0).

4.3 The Random Walk

We are now in a position to describe informally the random walk:

```
The bee starts at (0,0) and decides on a hopping rule (GPS or No-GPS)
While the bee is not on an outside cell it repeats this:
    It rolls a die.
    Following the chosen hopping rule, it either moves to a neighbor cell or it stays put.
```

The definition of an outside cell is given in §1.2. Implement a function `MakeBeeline(m,GPS)` that carries out this simulation. Details:

- `m` is a positive `int` that specifies the size of the honeycomb. We don't want ridiculously small honeycombs so let's insist that `m` is 3 or larger.
- `GPS` should be `True` if the GPS hopping rule is used and `False` if the No-GPS hopping rule is used.
- The function should return a beeline string that encodes the bee's escape journey. Beeline strings are defined in §3.

You will need variables that keep track of the bee's x and y coordinates and a variable where the beeline string is built up through repeated concatenation.

A good debugging strategy would be to modify the Application Script so that it prints out the beeline string produced by `MakeBeeline` and then uses `ShowBeeline` to display the resulting escape route. Use a suitably small value for `m`. Don't forget to test both hopping rules.

Things are simplified if you implement (and use) a function `Dist(x,y)` that returns $\sqrt{x^2+y^2}$, the distance from (x,y) to (0,0). To force issues we require that you implement this function and make effective use of it in your implementation of `MakeBeeline`.

5 Computing Average Time to Escape

We are interested in how long it takes (on average) for a bee to escape a size-m honey comb. Towards that end implement the following function:

```
def AveBeeline(m,GPS,Nmax):
    """ Returns a float that estimates the average length of a Beeline string.
    The average is based on Nmax trials. If GPS is True then the GPS hopping rule
    is used. Otherwise the non-GPS hopping rule is used.

    Pre: m is a positive integer that satisfies m>=3.
    """
```

Finally, *change* the Application Script so that *all* it does is it prints out a table like this:

```
      Averages with    Averages with
  m    GPS Hopping    No-GPS Hopping
 --------------------------------------
 10      12.5             29.4
 20      27.9            110.0
 30      45.2            274.1
 40      64.0            514.8
 50      80.5            791.3
 60      97.2           1160.4
 70     113.8           1415.4
 80     133.1           2257.0
 90     151.4           2748.4
100     167.4           3357.5

(Averages based on 100 trials)
```

To say that the number of trials is 100 is to say that each average was computed by running 100 random walks and then averaging the number of hops. Do not forget that with GPS hopping an in-place hop (signaled by a `'z'`) counts as a hop.

6 Submission to CMS

You have one file to upload: `MyHive.py`. It should have implementations of the functions `ShowBeeline`, `MakeBeeline`, `Dist`, and `AveBeeline`. Set up the Application Script so that when we run `MyHive.py` it produces a table like the one above. In particular, it should report approximate averages that are based on 100 trials.

1. Make sure your submitted `.py` files begin with the header comments listing:
   (a) The name of the file
   (b) The name(s) and netid(s) of the person or group submitting the file
   (c) The date the file was finished

   If there were other people who contributed to your thought processes in developing your code, it is a courtesy to mention them by name in a header comment as well.
2. Make sure all your functions have appropriate docstrings. These should include explanations of what the parameter variables "mean" and what preconditions (constraints) are assumed on their values, and what the user can expect as a result of calling your functions.
3. Do not hit "reload" on the CMS assignment submission page after the submission deadline passes: this will trigger another upload of your files, which can result in your submission being counted as late.
4. If you submit earlier than two hours before the submission deadline, CMS allows you to download your files to check that they are what you think you submitted. We thus strongly suggest that you submit a version early and do this check. (If you want to make changes, no problem: CMS allows you to overwrite an older version by uploading a new version.)