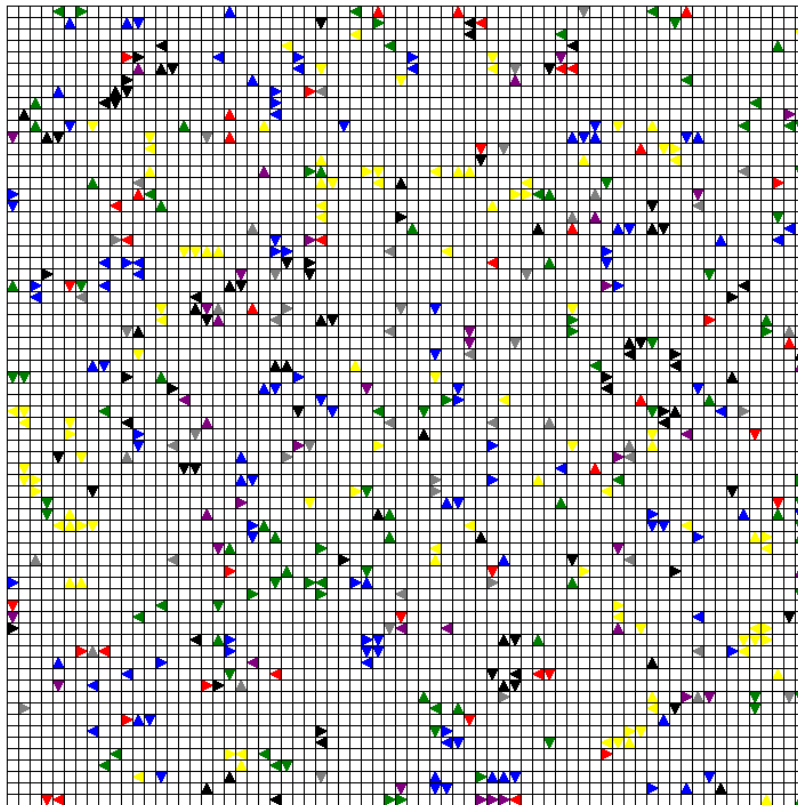




# COMPSCI 130 – Semester 1 – 2019

## Project Two



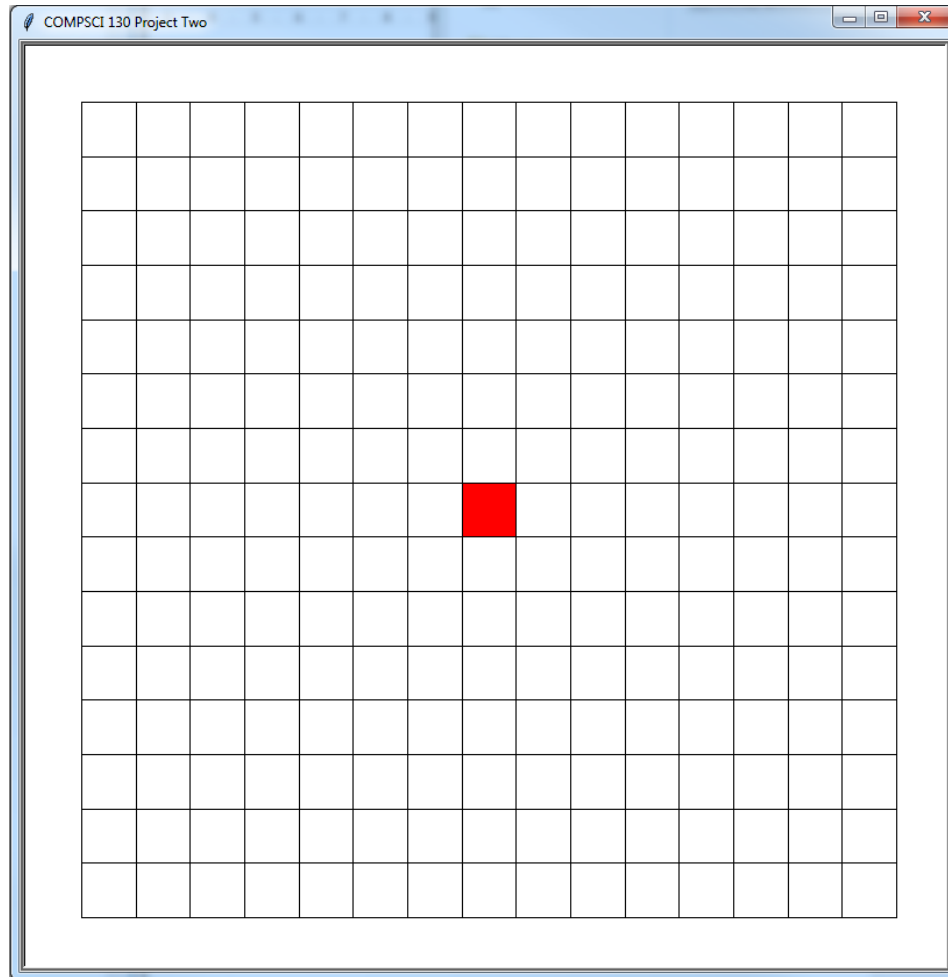
## Creatures

**Deadline:** 11:59pm, Friday 7<sup>th</sup> June  
**Worth:** 10% of your final grade

## Getting started

To get started, download the resource file (“ProjectTwoCreatures.zip”) from Canvas.

If you run the provided program “Creatures.py” you will see the following 15x15 grid with a small red square drawn in the middle cell:



Now, if you push a key, two things will happen:

1. the red square (a small “creature”) will move up five positions on the grid
2. a message will be printed to the screen: “A world full of creatures”

Take a look at the files that are provided along with the program. There is one input file called “world\_input.txt” and a folder called “Creatures” inside of which is a file called “Hopper.txt”.

### The world input file

Here is the initial contents of the world\_input.txt file:

```
15
5
Hopper 8 8 North
```

The “15” indicates the size of the world. That’s why the grid appeared with size 15x15. The “5” indicates that the simulation will run for 5 generations. That’s why the red square moved 5 places. The “Hopper 8 8 North” indicates that the creature was of type “Hopper” with a starting location of (8, 8) and initially facing North. Note that the first row of the grid is **row 1**, and the first column is **column 1**. As the creature was facing North, it moved *up* the grid.

You should try experimenting and changing a few of these values and observe the effects on the program.

### The Hopper “DNA” file

Here is the contents of the file “Hopper.txt” (which is inside the “Creatures” folder):

```
Hopper:red
hop
go 1

This creature just hops forward
```

The first line of this file defines the name of the species (“Hopper”) and the colour that creatures of this species appear (which is why the square was red). The last line of this file is just a comment.

The other lines are important, and have been shown with numbers below (although these numbers do not appear in the actual data file, it is useful to write them down - the first line, after the name and colour of the species, is line 1):

```
1: hop
2: go 1
```

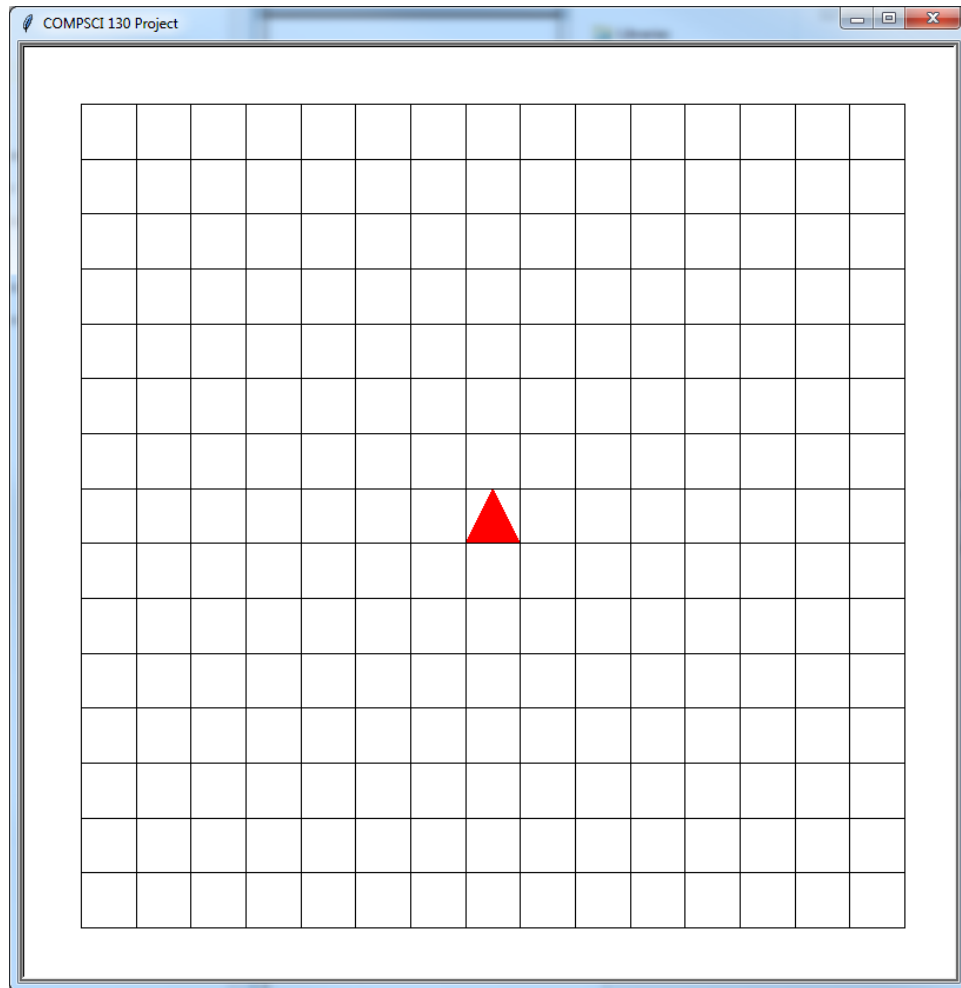
This is the DNA of the creature. You can think of this like a program that begins executing at line 1. The very first thing the creature does is “hop”. The “hop” instruction moves the creature forward by one position. Then we move to the next line, line 2, in the program. The next thing the creature does is “go 1” which means, go back to instruction 1. Basically, this creature just hops over and over again.

## Task 1

*Where am I going?*

Currently, the creature is drawn as a square. The problem with this is that you can't easily see the direction that the creature is facing. Modify the `draw()` function in the `Creature` class so that instead of being displayed as a square, the creature is displayed as a triangle. The point of the triangle must depict the direction that the creature is facing.

When you have completed this task, the creature will look more like this:



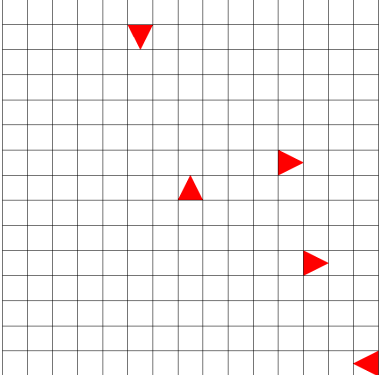
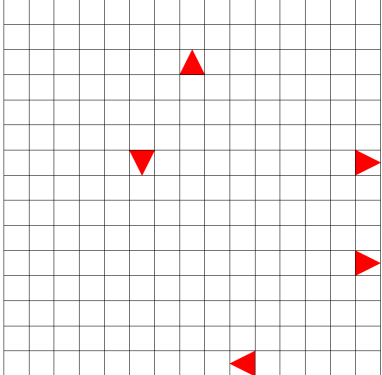
Test this by changing the direction the creature is facing (see the "world\_input.txt" file).

## Task 2

*Lonely, I'm so lonely...*

It would be a pretty lonely world for a creature on its own. Modify the program so that the “world\_input.txt” file can list an arbitrary number of creatures. For now, you can assume that none of the creature positions in this file will overlap. For example, given this file:

```
15
5
Hopper 8 8 North
Hopper 7 12 East
Hopper 2 6 South
Hopper 15 15 West
Hopper 11 13 East
```

	
At the start of the simulation, the creatures will be positioned exactly according to the “world_input.txt” file	At the end of this simulation, after 5 generations (or steps), the positions of the creatures will be as follows.

Note that the two creatures facing to the East stopped when they reached the right hand edge. You should inspect the code carefully and make sure you understand why this is. Pay particular attention to the `make_move()` function in the Creature class. Notice how the “go” and “hop” operations are implemented for the creature. When the `make_move()` function is called on a Creature object, the creature can execute as many “go” instructions as it wants to, but its turn will be over as soon as it executes a “hop” instruction (and then another creature gets a turn).

Each location is specified as: **row column**, starting from the top left cell as (1, 1). In other words: “Hopper 2 6 South” places the Hopper at row 2 (down from the top) and column 6 (across from the left).

The simulation proceeds with each creature getting one turn. The creatures are given turns in the order they appear in the “world\_input.txt” file. In this example, a creature’s turn ends when it executes the “hop” instruction.

### Task 3

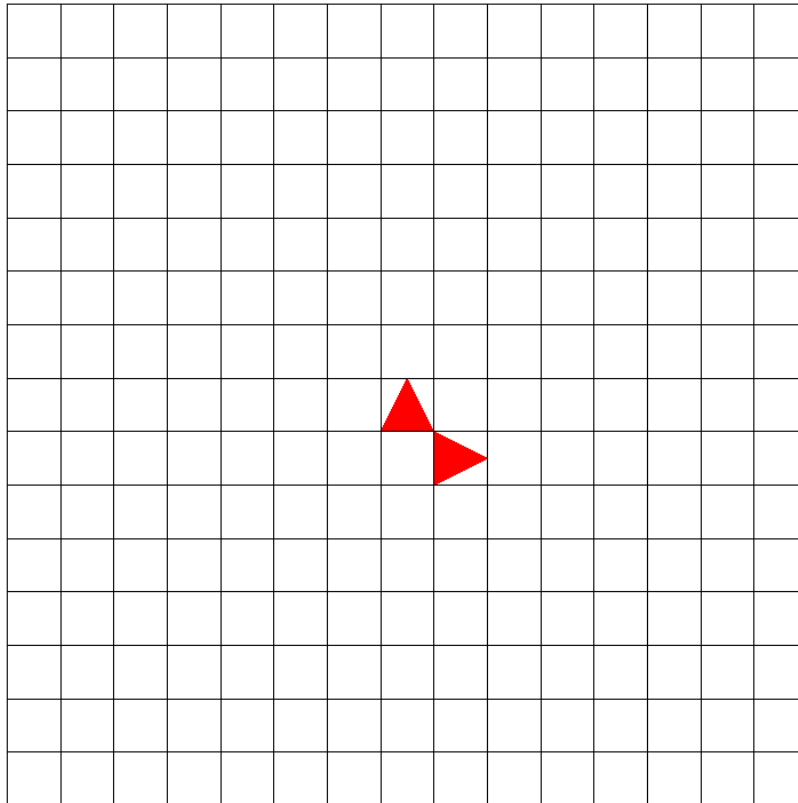
*I need some space.*

If the “world\_input.txt” file lists more than one creature at the same location there would be a problem. Creatures should be added to the world in the order they appear in the file - and if a creature on the list has the same location as a creature previously added to the world, then it should be ignored (and not added to the world).

For example, given this file:

```
15
5
Hopper 8 8 North
Hopper 8 8 South
Hopper 9 9 East
Hopper 8 8 West
Hopper 9 9 North
```

Only two creatures should actually appear on the world (one at 8,8 facing North, and one at 9,9 facing East):



## Task 4

*Let there be life.*

We will now add two new types of creatures - one called a “Parry” and one called a “Rook”.

A **Parry** doesn’t move anywhere, it just sits in one place and reverses directions. A **Rook** moves in a straight line ahead until it hits a wall, and then it reverses its direction and moves in a straight line ahead until it hits a wall, and then it reverses directions and moves in a straight line ahead until it hits a wall, and then it reverses directions and moves in a straight line ahead until it hits a wall, and then it reverses directions and, well, you get the picture.

Let’s start with the **Parry**. Here is the very simple DNA for the Parry creature:

```
reverse  
go 1
```

We have just replaced the “hop” instruction from the Hopper with a “reverse” instruction (which makes the creature rotate 180 degrees)! That was easy. To add the Parry to the simulation, you will need to do the following things:

1. Create a new file in the Creatures folder called “Parry.txt”
2. The first line of this file should be “Parry:grey” (this creature will be grey in colour)
3. There **must** be **one or more blank lines** after the last instruction (after “go 1”). A blank line signals the end of the list of DNA instructions.
4. You may like to add a comment after the blank lines to describe the Parry.

Also, in the CreatureWorld class (in the setup\_simulation() function) there is a line of code:

```
all_creatures = ['Hopper']
```

You will need to add ‘Parry’ to this list. Each time you add a new creature type, you will have to append it to this list.

### Implement the “reverse” instruction

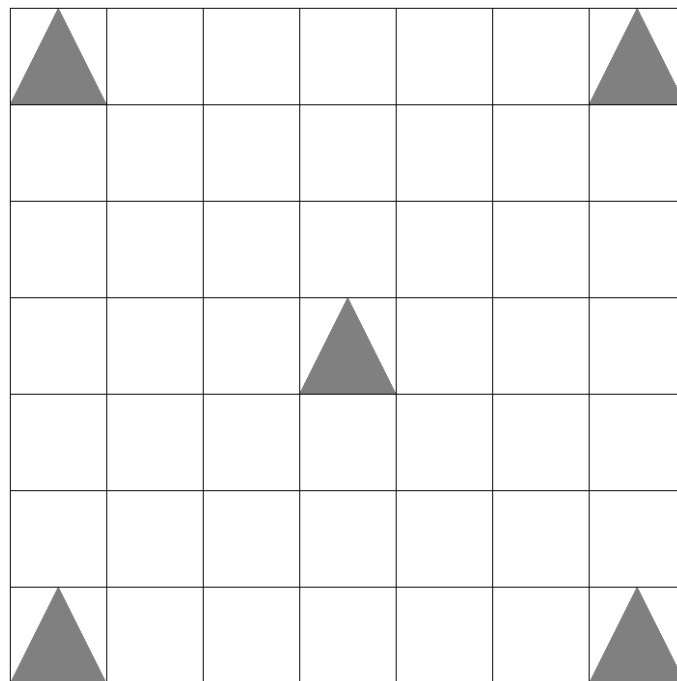
Oh yeah, one last thing - the make\_move() function in the Creature class has never heard of the “reverse” instruction. It only knows about the “hop” and “go” functions. So you are going to have to add some code to the make\_move() function to process “reverse” instructions. You will have to do the following:

1. update the direction that the creature is facing appropriately.
2. creatures execute their DNA instructions sequentially (unless it is a “go” instruction), so you will need to update the self.next\_instruction variable to refer to the very next instruction
3. finally, once a creature executes a “reverse” instruction, that is the end of their turn (just like with the “hop” instruction. So make sure you finish the creature’s turn!

To test this new Parry creature, try this set up in “world\_input.txt” (note we have a bigger grid this time, and a longer simulation consisting of 1000 steps):

```
7
1000
Parry 1 1 North
Parry 1 7 North
Parry 4 4 North
Parry 7 1 North
Parry 7 7 North
```

You should see something like:



and they should all reverse directions quickly when you press a key!

Now for the **Rook**. This creature can detect walls. It moves forward in a straight line, but when it reaches a wall, it reverses direction. We will need a new instruction for this wall detection. The new instruction is called “ifnotwall”, and it is a little bit like the “go” instruction. The format of “ifnotwall” is as follows:

```
ifnotwall X
```

In this case, X will be a line number in the DNA. The instruction works as follows: if there is not a wall immediately in front of the creature, then , it jumps to line X in the DNA. Otherwise we



continue with the very next instruction in the DNA list. Here is the complete Rook.txt file describing this new creature (which appears in orange):

```
Rook:orange  
ifnotwall 4  
reverse  
go 1  
hop  
go 1
```

**This creature bounces between the walls**

#### Implement the “ifnotwall” instruction

You will also need to provide code in the `make_move()` function to execute “ifnotwall” instructions. Basically, if there is a not wall immediately ahead of the creature, then you should jump to the instruction specified after “ifnotwall”. In this sense, it is similar to the “go” instruction. If there is a wall ahead of the creature, then simply advance to the next instruction in the DNA list. Like the “go” instruction, the creature can continue to execute other instructions after “ifnotwall”, so their turn shouldn’t finish after the “ifnotwall” instruction (like it does with “reverse”).

OK, time to test the Rook creature:

Let’s say we have this “world\_input.txt” file:

```
7  
1000  
Rook 4 4 North
```

The Rook will just keep bouncing back and forth between the North and South walls. If its initial direction had been West, then it would keep bouncing between the West and East walls.

You won’t use the twist instruction in this task, but to get full marks for this task you need to implement it for this task. You will actually start using this instruction in task 6.

#### Implement the “twist” instruction

You will also need to provide code in the `make_move()` function to execute “twist” instructions. Basically, twisting is turning 90 degrees clockwise. You will have to do the following:

1. update the direction that the creature is facing appropriately.
2. creatures execute their DNA instructions sequentially (unless

it is a “go” instruction), so you will need to update the `self.next_instruction` variable to refer to the very next instruction

3. finally, once a creature executes a “twist” instruction, that is the end of their turn (just like with the “hop” instruction). So make sure you finish the creature’s turn!

OK, let’s review all of the instructions we can now use to control our creatures:

- **hop**: moves forward one place (*and ends the creature’s turn*)
- **reverse**: turns 180 degrees (*and ends the creature’s turn*)
- **twist**: turns 90 degrees clockwise (*and ends the creature’s turn*)
- **go X**: this indicates we should jump to line X of the DNA to continue executing the instructions from there (*does not end the creature’s turn*)
- **ifnotwall X**: if there is not a wall in front of the creature, then jump to instruction X in the DNA, otherwise just continue from the next instruction (i.e. the one that follows the “ifnotwall” instruction (*does not end the creature’s turn*))

## Task 5

### *What in the world?*

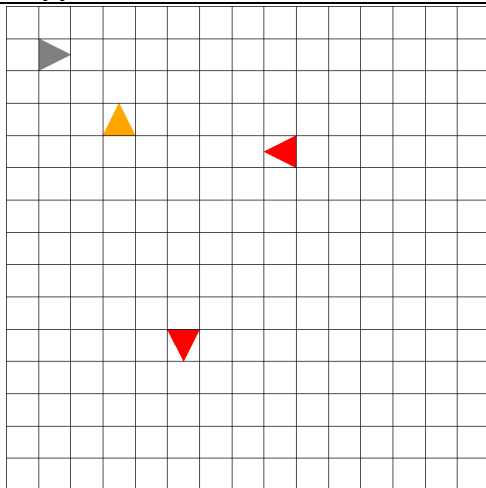
The World object is printed once at the very end of the simulation (after the steps have all been executed). This happens in the “else” branch of the simulate() function in the World class:

```
print(self)
```

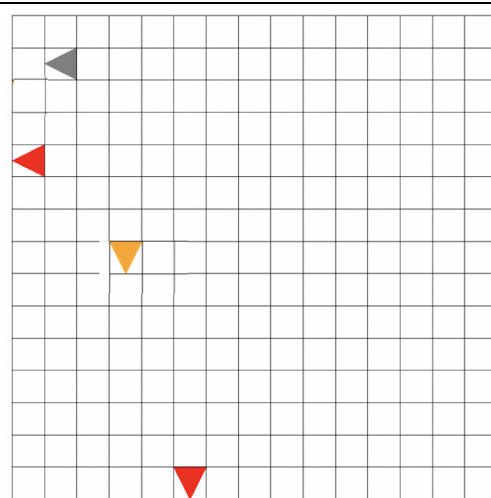
The World class currently has a `__str__()` function, but it doesn’t print anything useful. It just prints: “A world full of creatures “. Update this function so that it prints all of the creatures and their positions. It should also display a summary showing how many creatures of each type exist in the world at the end of the simulation.

The format of what is printed must match exactly what is described here. Let’s start with an example “world\_input.txt” file:

```
15
11
Rook 4 4 North
Hopper 11 6 South
Parry 2 2 East
Hopper 5 9 West
```



The image above shows the initial configuration of the world, the image on the right above shows the final configuration after the 11 steps of the simulation. The text to the right shows what should be printed.



```
15
[('Hopper', 2), ('Parry', 1), ('Rook', 1)]
Rook 8 4 South
Hopper 15 6 South
Parry 2 2 West
Hopper 5 1 West
```

As shown in the previous example, instead of the program printing “A world full of creatures “, it should now print:

```
15  
[('Hopper', 2), ('Parry', 1), ('Rook', 1)]  
Rook 8 4 South  
Hopper 15 6 South  
Parry 2 2 West  
Hopper 5 1 West
```

The first line is the size of the grid, the next line is a list containing tuples showing a count of the number of creatures of each type. This list must be sorted in decreasing order of counts (which is why the Hopper is listed first) and the secondary key is alphabetic (which is why Parry is listed before Rook).

The next lines of this output show the final positions and directions of the creatures. The order in which these appear must be the same as the order in which the creatures were listed in the “world\_input.txt” file.

## Task 6

### *Introducing the Roomber.*

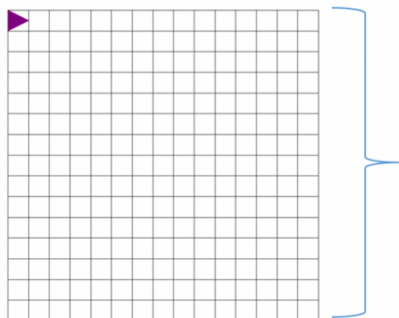
None of the creatures we have seen so far are capable of covering the entire area of the grid. The Parry doesn't even move, the Hopper stops as soon as it hits a wall, and the Rook only bounces back and forth between the walls.

Design a new creature called a **Roomber**. Assuming that this creature is the only creature on the grid (so that it won't get blocked by another one), it should systematically be able to cover every single cell of the grid.

One approach would be to move back and forwards along the rows (or columns) of the grid, but moving one row (or column) up (or down) each cycle. Let's say the creature was positioned in the middle of the grid... what would be a good strategy to start with?

You might want to make use of all of the instructions we have seen so far:

- ifnotwall X
- hop
- go X
- twist
- reverse



The Roomber needs to move across all of the cells in the grid

The Roomber should be purple in colour.

Once you have completed your Roomber, try it with the following "world\_input.txt" files:

<b>15</b> <b>1000</b> <b>Roomber 1 1 North</b>	<b>15</b> <b>1000</b> <b>Roomber 1 1 South</b>	<b>15</b> <b>1000</b> <b>Roomber 1 1 East</b>	<b>15</b> <b>1000</b> <b>Roomber 1 1 West</b>
--	--	---	---

and ensure that in each case it does cover the entire grid.

## Task 7

*Free will is a myth.*

For this task, you must add three more instructions - all of them are “if” instructions (like “ifnotwall”) which test some condition and then either jump to a new instruction in the DNA list or just continue with the next instruction.

- **ifsame X**  
This instruction looks in the cell immediately in front of the creature - and if that cell contains another creature *of the same species*, then we jump to the instruction at line X. Otherwise, we continue with the next instruction in the sequence.
- **ifenemy X**  
This instruction looks in the cell immediately in front of the creature - and if that cell contains another creature *of a different species*, then we jump to the instruction at line X. Otherwise, we continue with the next instruction in the sequence.
- **ifrandom X**  
This instruction allows some creatures to behave (kind of) randomly. With (approximately) 50% probability, we will jump to instruction X, otherwise we will continue with the next instruction. **DO NOT USE THE random() function. Read below!**

### An important note about random numbers

Most computers cannot generate *truly* random numbers - instead, they use an algorithm called a pseudo-random number generator which generates numbers which look fairly random. For example, see:

<https://docs.python.org/3/library/random.html>

For the purposes of this project, we would like our program’s behavior to be *deterministic* - that is, we can accurately predict the outcome for any given starting conditions. If we use the default Python random number generator, we would get different sequences of random numbers (and therefore unpredictable behavior) every time we run the program. There are two solutions to this:

- Set the seed for the pseudo-random number generator (there is a special function we can call to set a value, called a seed, which means we will get the same sequence of random numbers every time we run the program)
- Create our own (kind of) random numbers!

We will take the latter approach - we will define our own function in the World class that returns numbers which look pretty random, but actually, are completely determined by the positions of the creatures in the world.

Define a function called `pseudo_random()` in the `World` class:

```
def pseudo_random(self):  
    < YOUR CODE GOES HERE - YOU MUST CALCULATE string_total >  
    return int(hashlib.sha256(string_total.encode()).hexdigest(), 16) % 2
```

That last line looks a bit intimidating - but don't worry about it. It uses a hash function (called SHA256) to generate a hash value. And we then look to see if this hash value is an even or odd number. Just use this line exactly as it appears here in your code.

All you need to do is define some code to initialise a string variable called: `string_total`

This `string_total` value should be computed as follows:

- For every creature in the world, compute the sum of that creature's row and column positions
- Add all of these creatures' sums together to get a total sum
- Multiply this total sum by the current generation (see `self.generation`)
- `string_total` is simply this product converted to a string

For example, if the world contained two creatures at positions (5, 7) and (10, 2) on generation 6, then `string_total` should be:

$$(5 + 7 + 10 + 2) * 6 = \text{'144'}$$

So how should the "ifrandom X" instruction work?

The `pseudo_random()` function returns 0 or 1.

To implement the "ifrandom X" behavior, you should call `world.pseudo_random()` and **if** the value it returns is **1**, then **jump** to instruction X, otherwise (if it returns **0**) move on to the **next** instruction. Essentially:

```
if the instruction is "ifrandom X":  
    if world.pseudo_random() == 1:  
        jump to instruction X  
    else:  
        go to the very next instruction
```

#### Testing Task 7

To test the "ifrandom" instruction and the `pseudo_random()` function, define one more creature type. This creature is called Randy and appears in pink:

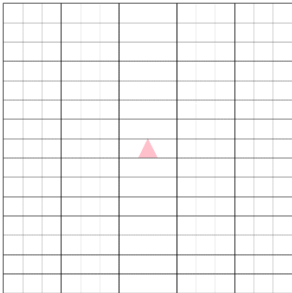
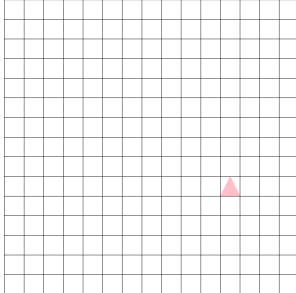
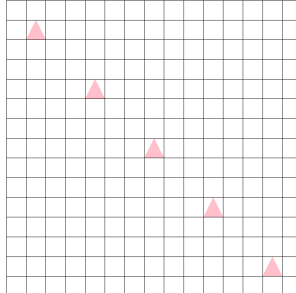
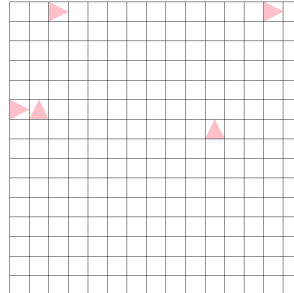
```

Randy: pink
if random 4
  hop
go 1
if random 7
  twist
go 1
reverse
go 1

```

This creature randomly either moves forward one position, or stays in the same place, in which case it either turns clockwise or reverses at random

This creature either randomly hops forward or stays in the same place. If it stays in the same, it randomly either turns clockwise or reverses. You can test the behavior with the following simulations:

<b>15</b> <b>200</b> <b>Randy 8 8 North</b>		<b>15</b> <b>200</b> <b>Randy 2 2 North</b> <b>Randy 5 5 North</b> <b>Randy 8 8 North</b> <b>Randy 11 11 North</b> <b>Randy 14 14 North</b>	
Starting position:	After 200 generations:	Starting position:	After 200 generations:
			
<i>Output:</i>	<b>15</b> [('Randy', 1)] Randy 10 12 North	<i>Output:</i>	<b>15</b> [('Randy', 5)] Randy 1 14 East Randy 6 1 East Randy 1 3 East Randy 6 2 North Randy 7 11 North



## Task 8

### *Survival of the fittest.*

The final instruction to define is called **“infect”**. This instruction allows one creature to infect another creature with its DNA.

#### **infect**

If the cell immediately in front of this creature contains another creature *of a different species* then this instruction will infect that other creature. When a creature is infected, it keeps its position and direction, but its DNA (i.e. its list of instructions) are replaced with those of the infecting creature. Essentially, it changes type to that of the infecting creature, and begins executing its new DNA starting from the instruction at line 1. If a creature issues an “infect” instruction, that is the end of its turn.

Recall that the creatures take turns to move - the creatures are given their turns in the same order they appear in the “world\_input.txt” file. The “hop”, “reverse”, “twist” and “infect” instructions (see `make_move()`) should all be terminated by the statement:

```
finished = True
```

That is because if a creature executes one of these instructions, then its turn is over and it must wait for all of the other creatures to have a turn before it can execute more instructions. Basically, a creature can execute as many “if” instructions, and as many “go” instructions, as it wants to on its turn. A creature’s turn is only finished when it executes a **hop**, **reverse**, **twist** or **infect** instruction. To illustrate how the “infect” instruction should work, consider this new species called Flytrap:

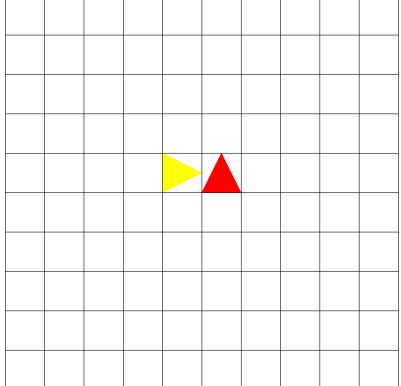
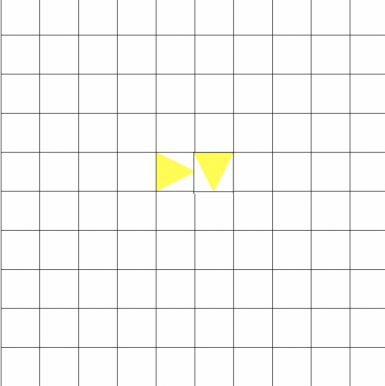
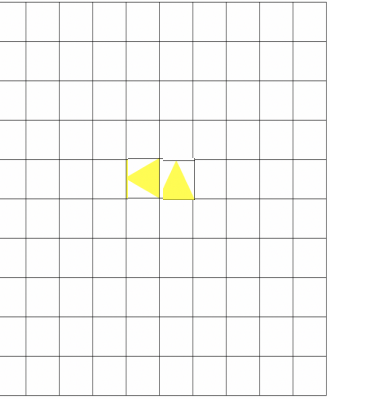
```
Flytrap:yellow
ifenemy 4
reverse
go 1
infect
go 1
```

**This creature sits in one place and reverses direction, and will infect anything which comes in front of it.**

If a Flytrap is placed next to a Hopper in a simulation that runs for just 2 steps:

```
10
2
Flytrap 5 5 East
Hopper 5 6 North
```

The diagrams below show the two generations of this simulation:

		
<p>The initial configuration, as defined in the "world_input.txt" file</p>	<p>After 1 generation. The Flytrap moved first and infected the Hopper. The new Flytrap (which just used to be a Hopper) is still facing North in its previous direction when it has its first step. In this case, it follows its new set of instructions - and because there was no enemy in front of it, it simply reversed direction.</p>	<p>After 2 generations - and the end of the simulation, both Flytraps reverse directions once more.</p> <p>The output of the simulation is:</p> <p><b>10</b>  <b>[('Flytrap', 2)]</b>  <b>Flytrap 5 5 West</b>  <b>Flytrap 5 6 North</b></p>

## Task 9

### *Experiment and report.*

Design your own creatures and experiment with them to see which ones perform the best (for example, when competing against each other). For this task, you should write a short report which describes your creatures and summarizes their performance (the expectation for this part of the report would be around two pages of text, although you may like to include diagrams, listings of the creatures' instructions, etc.)

At the very start of this report there should be a brief overall summary of your reflections on this project. This should include a short statement to the marker to inform them which of the tasks you completed (this will help them when grading your project). You should also include some commentary on what you have learned, what you found most challenging and what you would like to have done if you had more time.

The format for this report should be:

- A title
- Your name and ID number
- A short summary - what tasks you completed and your commentary about the project
- A report describing any creatures you have defined (you will be submitting these in the Creatures folder along with your code) and how well they perform (expectation: approximately two pages, not including diagrams or code listings)

## Marking

Each task is worth 1 mark except for task 9 which is worth 2 marks. The code that you submit should be clearly organized and well commented. You should ensure that your code does not contain any syntax errors.

Please submit your code in a single source file (rather than splitting each class into a separate file). You will only need to submit one source file: `Creatures.py` (along with your Creatures folder and a `"world_input.txt"` file).

The `"world_input.txt"` file that you submit should create some kind of interesting world which helps demonstrate any creatures that you have created. You can describe this in the opening summary statement of your report.

## What to submit for Project Two

You should submit a single .zip archive file (please use either WinZip or 7-Zip to create a .zip file). The extension must be .zip. Please don't submit other formats. The name of this file should be your UPI. For example:

**mbar098.zip**

This archive should contain:

- Your **Creatures folder** with any creatures you have created, including the Roomber from Task 6, and the Rook and Parry from Task 4
- A "**world\_input.txt**" file that illustrates your creatures interacting in some interesting way
- Your final **Creatures.py** file
- A report which must be in .pdf format (DO NOT submit Word documents) which is named with your UPI (e.g. **mbar098.pdf**) and which contains:
  - A short summary of which of the Tasks of the project you have completed, along with any other notes for the markers - and your commentary/reflections on the project
  - The summary should be followed by any documentation you have written for Task 9 (if you tackled that task) - this should describe any interesting creatures you have created along with a summary of their performance. As a guide, around two pages of text would be expected, along with any charts/figures or code listings you want to include.

## Academic honesty

This is an individual project, so you should write all of the code that you submit by yourself. Discussing *ideas* with others is perfectly fine. For this project, you may share Creature files that you have created, but if you use someone else's creatures you **MUST** acknowledge the author of those creature files in your report.

-----  
Mike Barley  
Derived from Paul Denny  
May 2019