The Evolution of Picobot

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/PicobotProject on 3/22/2017

Milestone requirements for the Picobot project

For the milestone, you should submit a version for which the following pieces are implemented and work:

- You have a `Program` class with a working constructor and `__repr__` and the method
- `randomize` runs correctly. Also,
- you should have a `World` class with a working constructor and `__repr__` and the methods
- `step` and `run` both work correctly...
- This means you'll have built a Picobot simulator -- in ASCII, at least
- The genetic algorithm part of the project need not be complete for the milestone, but if it is, all the better!

Be sure to zip up the folder containing your Picobot project file (or files) and name it `picobot_milestone.zip` -- then submit that in the right spot for Picobot in Hw#12!

Picobot: *background*

In our first assignment of this course, we programmed a virtual robot named Picobot. In this project, you will write a Python program that ***writes*** Picobot programs!

What is more, your Python software will automatically "evolve" better and better Picobot programs using a biologically-inspired technique called "Genetic Algorithms." You will use techniques from your object-oriented programming work in Python. As a result, your final result will be relatively short and elegant!

The big idea is to begin with a ***population*** of random Picobot programs.

- Then, your code will evaluate the "fitness" of each of the individual Picobot programs by running it for some number of steps and seeing

what fraction of the maze it fills -- that fraction will be somewhere between 0.0 and 1.0.
- Your software will then select a subset of the ***most fit*** programs.
- From that subset, your software will randomly choose pairs of programs to "mate". This mating process will produce a new, "child" Picobot program with characteristics of both of the "parent" Picobot programs.
- Additionally, your software should introduce a small number of random mutations to some of the programs. Such random mutations help preserve diversity, which is important for the overall fitness of a population.
- In this way, you will have constructed a next generation of programs. This next generation will be of the same size as the previous generation, but -- hopefully -- more "fit".
- You'll repeat this process some number of times; in the end, you'll choose the most fit program from your final generation and *that* will be your fully evolved picobot program!

How Does Picobot Work Again?

Take a moment to refresh your memory of the first Picobot assignment and the syntax for Picobot rules.

You can also go directly to the Picobot simulator.

Getting started...

All of your code should be in a file named `milestone.py` for the milestone and `final.py` for the final deliverable.

At the top of the file, be sure to include

- your name
- a short description of the project
- the date

In addition, be sure to import libraries you'll need, e.g.,

- `import random`
- and maybe others...

Finally, you may use some global variables to avoid "magic numbers" in your code.

"Magic numbers" are simply constants within a program that seem "magic" because they have no explanatory context. They're bad practice because if you (or someone else) returns to your code later, it's very difficult to understand what those values are doing!

For instance, the Picobot empty-room world is 25x25 (including walls). Rather than littering your code with `25`'s, include a few global constants at the top of your file:

```
HEIGHT = 25
WIDTH = 25
NUMSTATES = 5
```

That last one is the limit of the number of states for Picobot programs... more on that below. Other global constants are welcome, too -- by tradition, they're all uppercase (and they're constant for any one run of the program).

Another HUGE advantage of these global constants is that you can make a change to their value -- and, since you're using them throughout the code -- that single change will work throughout (to change every individual `25` would be error-prone (you might miss one) and too time consuming!)

The `Program` and `World` classes:

Next, there are two basic components that you'll need:

- One is a Picobot program (or rule set) and
- another is a Picobot world.

You'll write a class for each of these two things:

- `class Program` to represent a single Picobot program
- `class World` to represent a single Picobot world

You don't need a separate class for Picobot itself -- it's simply going to be a component of the `World`, above.

You may have other classes of your own design, especially if you add other facets to the project, but these two are the foundation... .

About the `Program` class

Your `Program` class should begin this way:

```
class Program:
    def __init__(self):
        self.rules = {}
```

What is that `self.rules`? It's a dictionary that stores the actual details of the program! Remember that a Picobot rule looks like this:

```
state pattern -> move newState
```

For example, it could be:

```
0 xExx -> N 1
```

This example rule says: "If I'm in state 0 with surrounding pattern `xExx` then I should move North and go to state 1."

This can (and should!) by represented in a dictionary where the key is the tuple `(0, "xExx")` and its corresponding value is the tuple `("N", 1)`.

A `Program` object will store this rule in its `self.rules` dictionary so that the key `(0, "xExx")` has associated value `("N", 1)`. In other words

```
self.rules[ (0, "xExx") ] = ("N", 1)
```

Of course, this is just an example of the representation of one possible Picobot rule -- this particular rule may or may not be a part of an actual program.

Empty room (to begin)

For the standard project, you should start with the empty Picobot room. (Evolving programs to solve the Maze is more difficult -- that or other Picobot worlds is an optional extra part of the project.)

Start with only 5 states

You should limit the number of states your Picobot programs (objects of class `Program`) will deal with.

Experience has shown that 5 states is a good initial value. For more complicated environments (extra credit), more states can help.

Evolving programs with more states takes additional time, however, since the search space (the set of possible Picobot programs) is so much larger!

No wildcards!

At least to start, do not use the wildcard character `*` in your Picobot rules.

The wildcard adds considerable complexity both to the Picobot simulation and to evolving programs, because it represents many rules with a single line. Instead, your Picobot programs should include fully-specified surroundings.

Fortunately, this is feasible, since there are only 9 possible surroundings in the empty room:

Possible *surroundings*

In an empty room, note that these 9 are the only surrounding patterns that Picobot needs to consider:

```
xxxx
Nxxx
NExx
NxWx
xxxS
xExS
xxWS
xExx
xxWx
```

Remember that we're restricting Picobot programs to use exactly 5 states: 0, 1, 2, 3, and 4. (At least to begin with.)

That was what the `NUMSTATES` global constant at the top of the file is for.

Then, a Picobot program will have exactly 5 * 9 = 45 rules, one for each combination of state and pattern. For each combination of state and pattern, we have a key `(state, pattern)` that has a corresponding value `(nextMove, nextState)`.

What methods are needed in the `Program` class?

Your `Program` class should have, at a minimum, the following methods:

- The `__init__(self)` method as we saw earlier should simply set `self.rules` to be an empty dictionary.

- You need a `__repr__(self)` method that **returns** the string representation of the program - Python needs it to return a string! If you want to insert a new line in a string, use `\n`. *Remember* that `__repr__` should **return** a string rather than print the string itself.

  In particular, your `__repr__` method should create a string that has all of the rules in `self.rules` ***in sorted order*** by the dictionary's keys. This is helpful, because it means that programs are more easily compared to each other and to our expectations.

  ... but *how* to print the rules in sorted order? Notice that `self.rules` is a dictionary. So, if you let

  ```
  Keys = list( self.rules.keys() )
  ```

  then `Keys` will be a list of all *keys* in that dictionary. That is, `Keys` is a list of the form `[(state, pattern), (state, pattern), ...]`. Then, you can use the built-in `sorted` function, i.e., let

  ```
  SortedKeys = sorted( Keys )
  ```

  and this will produce the ***sorted*** list of keys named `SortedKeys`. Sweet!

  Equally Important!

  Your `__repr__` method should print out its full Picobot program ***in a way that can be directly copied-and-pasted*** into the Picobot Simulator. That is, each rule should be formatted as follows:

  ```
  2 Nxxx -> W 3
  ```

  Here is a link to an example output to use as a template.

- `randomize(self)` should generate a random, full set of rules for the program's `self.rules` dictionary. Remember that the dictionary is initially empty. However, this method makes random rules, one for each combination of the `NUMSTATES` states - 5 to start - and 9 possible surroundings. Note that for 5 states and 9 surroundings, there should

be 45 randomly-generated rules.

Hint! The keys (startingstate, surroundingspattern) of the dictionary are ***not*** random! You should simply loop through each possible state and each possible surroundings pattern. For each combination, *then* you want to generate a random next state and a random next move, `movedir`. One challenge is that `movedir`, the next move, *must be legal, that is, **not** into a wall*!

Hint #2 In order to ensure you only select a legal move for a given `pattern`, consider adapting a while loop like this one:

```
movedir = random.choice( POSSIBLE_MOVES )
while movedir in pattern:
    movedir = random.choice( POSSIBLE_MOVES )
```

This does depend on your possible moves and your patterns both using capital letters for their possible values, but this shouldn't be a problem. This `while` loop will then continue randomly selecting a value for `movedir` until it *is* a legal move!

- `getMove(self, state, surroundings)` should take as input an integer state and a surroundings(e.g., `"xExx"`) and it should return a tuple that contains the next move and the new state. This method will simply use the dictionary `self.rules` to accomplish this.

- `mutate(self)` should choose a single rule from `self.rules` and it should changes the value (the move and new state) for that rule. For example, if the program contained the rule that associated the `(state, pattern) (0, "xExx")` with the `(move, new state) ("N", 1)`, it might replace this with a new randomly chosen `(move, new state)` like `("S", 2)`. Notice that it would not be valid to move "E" in this case, because the pattern shows that there is a wall to the east -- your code will need to randomly choose one of the *valid* directions to move.

- `crossover(self, other)` is a method that takes an `other` object of type `Program` as input. It should return a new "offspring" `Program` that contains *some* of the rules from `self` and the rest from `other`. Notice that both `self` and `other` are programs of exactly the same length! For example, if we are using 5 states, then since there are only 9 possible surrounding patterns, there will be exactly 45 lines in a program.

  The **most effective** way to crossover is to choose a "crossover state" at random, from 1 to 3 inclusive. Suppose state 2 is chosen. Then, the offspring will get all rules from one parent involving states 0, 1, and 2

and will get all the rules from the *other* parent involving the remaining states, 3 and 4.

It turns out that if you choose every rule from a parent at random, it does too much scrambling of the "genetic code" of the parents, resulting in offspring that don't preserve any of the good features of their parents.

Common pitfall warning: A common mistake here is to *modify* the program of one of the parents rather than return a brand-new offspring `Program`. This is a bad idea biologically and will not work in your program either! Why? Imagine that mating changed the genome of a parent... . If a parent has high fitness, it is likely to have multiple children, hopefully many of whom inherit beneficial parts of the parent's genome. However, if the parent's genome *changes* the first time it mates, then it is very likely that it's fitness will drop! Its subsequent children will therefore be likely to have lower fitness than that first sibling.

- *Helpful, but not required* Many Picobotters, when they get to writing the genetic algorithm, find it useful to sort lists of `(fitness value, program)` pairs. That is, they have a length-200 list of this form: `[ (fitness0, prog0), (fitness1, prog1), ... (fitness199, prog199) ]` and they would like to sort the list *by the floating-point numeric value of fitness*. The built-in `sorted` function will work great - *except when there is a tie for two fitnesses*, which certainly happens! Then, the `sorted` function tries to break the tie by sorting the programs, e.g., `prog0` vs `prog1`. The problem is, sorting programs isn't defined.

  In order to define sorting by programs, you can paste in these definitions of `__gt__` and `__lt__`, the greater-than and less-than operators. They don't actually decide between two programs - they only return `True` or `False` at random - but that's all you'll need to make the `sorted` function work on lists as described above.

```
        def __gt__(self,other):
            """ greater than operator - works randomly, but works! """
            return random.choice( [ True, False ] )

        def __lt__(self,other):
            """ less than operator - works randomly, but works! """
            return random.choice( [ True, False ] )
```

Representing a Picobot world

The other fundamental component of your program will be a `World` class that can simulate a whole Picobot environment (and Picobot's location and actions within it). For this final project, your `World` class only needs to contain and simulate the 25x25 empty-room environment.

Picobot, the one-grid-square "robot," does not need its own class: it can simply be simulated within the `World` class!

The `World` class

The `World` class will need several pieces of data in order to maintain the state of the world and simulate Picobot within it:

- `self.prow`, the current row in which Picobot is located (will change)
- `self.pcol`, the current column in which Picobot is located (will change)
- `self.state`, the current state for Picobot (will change)
- `self.room`, a list-of-lists that holds the 2d room in which Picobot is located (very similar to the C4 `Board`'s `self.data`)
- `self.prog`, an object of type `Program` that controls the Picobot simulation

Other data members are welcome, if you'd like to define them... .

Here's how the class should start:

```
class World:
    def __init__(self, initial_row, initial_col, program)
        self.prow = initial_row
        self.pcol = initial_col
        self.state = 0
        self.prog = program
        self.room = [ [' ']*WIDTH for row in range(HEIGHT) ]
```

That empty list is not the final value for `self.room`!

You might want to refer to the Connect Four `Board` class as a reference for your representation of your room for Picobot -- the starting point above borrows from it! Notice that, just as in Connect Four's `self.data` representation, a reasonable approach to building `self.room` is to create a list of rows, each of which is a list of columns, each of which is a one-character string.

In addition, you will need to make sure that the ***outer edge*** of the `self.room` consists of walls, perhaps using the `'W'` character -- or whatever you'd like. For those who like their ASCII extra-fancy, you could use `'-'` for horizontal walls, `'|'` for vertical walls, and `'+'` for intersections!

Hint: If you stick with a simpler approach, making all of the walls a `'+'` character, then it makes life far easier ***to place the plus signs into the room itself*** using code similar to this in your `__init__` constructor, *after* you've created the room to be full of blank space characters:

```
        for col in range(WIDTH):
            self.room[0][col] = '+'
```

Note that this only puts the top-row walls into the room. You'll want to place the other three walls, too -- and Picobot's `'P'`, as well!

This approach of keeping all of the contents of every room square in `self.room` makes the `__repr__` method far easier to write!

In essence, an object of type `World` is storing an entire map of the room `self.room`, the program `self.prog` that it will use to try to explore the room, and Picobot's current state, row, and column `self.state`, `self.row`, and `self.col`. Other information is OK, too.

Methods for the `World` class

We leave some of the methods in your `World` class up to you, at a minimum it should have the following:

- A `__repr__(self)` method that returns a string that shows the maze with the space character to unvisited cells, the walls with whatever character(s) you've chosen, the Picobot's position in the maze, and `o` (lower-case `o`) characters for empty but previously-visited locations in the maze. This method will be crucial for debugging your Picobot simulator.

- A `getCurrentSurroundings(self)` method, which should return a surroundings string, such as `"xExx"`, for the current position of Picobot, that is, `self.prow` and `self.pcol`.

- A `step(self)` method that moves the Picobot one step, updates the `self.room`, and updates the state, row, and col of Picobot, all using the program `self.prog`. This method doesn't return anything!

Instead, it uses the Picobot's `self.prow` and `self.pcol` to determine the current surroundings (using `getCurrentSurroundings`. Then, it uses those surroundings and `self.state` to ask the program, `self.prog` to determine the `nextMove` and `nextState`. This would be done using the program's `getMove` method, as described above in the `Program` class. Finally, this `step` method should change Picobot's position and state and should also change the room (for example, marking a cell as empty but visited and marking another as the current location of Picobot's `P`).

- Also, create a `run(self, steps)` method that takes as input the number of steps, named `steps`, to move. The `run` method should execute that number of steps using the `step` method. This is a Picobot simulator!

- You will need a `fractionVisitedCells(self)` method that returns the floating-point fraction of cells in `self.room` that are marked as *visited* (including Picobot's current location). This is not the number of steps, because many cells may have been visited more than once. Instead, this is the number of distinct grid squares in Picobot's environment that it touched during its run. This is the basic fitness score for a Picobot program!

What Now?

Believe it or not, that's the bulk of the program!

All of the rest can be done in a few short functions (outside of both the `Program` and `World` classes).

Here are some of the pieces you'll need to build:

- A very short function that takes as input a population size and returns a population (a Python list) of that many random Picobot programs.

- A very short fitness evaluation function that measures the fitness of a given Picobot program.

  This function should be called `evaluateFitness(program, trials, steps)` and takes as input a Picobot program, a positive integer `trials` that indicates the number of random starting points that are to be tested, and a positive integer `steps` that indicates how many steps of the simulation each trial should be allowed to take. The

function returns a fitness (a floating point number between 0.0 and 1.0) that is the fraction of the cells visited by this Picobot program, averaged over the given number of `trials`. You'll want to use the `fractionVisitedCells` here.

For example, imagine that `p` refers to a Picobot program and we run `evaluateFitness(p, 42, 1000)`. Then this function chooses a random starting point for the picobot and runs the given program `p` for 800 steps. The code tracks the fraction of cells that were visited and does this 42 times for 42 different (random) starting points in this example. Afterwards, the code averages the fraction of cells visited over the 42 runs. Note that if the room has 529 cells (23 by 23 empty cells) then even a relatively fit program may need well over 529 steps to fill a large portion of the room, because it's unlikely that the program will be optimally efficient. That's why you might use 1000 `steps` here, or even a larger number, rather than 529.

- **The main function** should be named after our theme, genetic algorithms: `GA(popsize, numgens)`.

  In particular, `GA(popsize, numgens)` should create `popsize` random Picobot programs as the initial population (200 has worked well in the past). Then, it should
  - evaluate the fitness of all of those programs
  - sort a list of [fitness, program] pairs
    - Hint on sorting: if `L = [ (.4,prog1), (.2,prog2), (.3,prog3) ]`, you can then call

      ```
      SL = sorted(L)
      ```

      and you will see that `SL` is `[ (.2,prog2), (.3,prog3), (.4,prog1) ]`
  - extract the most-fit programs -- these are the one that will survive and be "parents" for the next generation
    - in the past, using 10% (for example, 20 out of 200) has worked well for some folks...
  - you should ***keep*** those most-fit programs as part of the next generation - plus create some children programs!
  - to create the children programs, one approach is to
    - select two parents at random from the most-fit pool
    - use `crossover` to create a child program from those two parents

        - use `mutate` once in a while (this you'll need to tune...). Too much mutation and fitness will not be retained; too little and the lack of novelty will cause evolution to get stuck... .
        - ***The details of creating children will determine how effective your genetic algorithm is -- this is where you'll need to experiment!***
    - Then, with this new, more fit population -- the next generation -- repeat the whole process
        - It's best to keep each generation at the same size `popsize`, perhaps 200...

- At the end, your program should return (and print, if you like) the best program from the last generation. You can then copy-and-paste that program into the Picobot simulator to see it run!
    - Some people prefer to print the best program from each generation to a file (less messy and easier to preserve). Here's a snippet to make this easier:

```
def saveToFile( filename, p ):
    """ saves the data from Program p
        to a file named filename """
    f = open( filename, "w" )
    print(p, file=f)        # prints Picobot program from __repr__
    f.close()

# here's how to call this... probably within GA's main loop...
saveToFile( "gen1.txt", best_p_in_gen_1 )
saveToFile( "gen2.txt", best_p_in_gen_2 ) # and so on...
```

- Whether you save your programs to file or print them to the screen, as `GA` runs each iteration of its generations loop, it should print both the ***average*** and ***maximum*** fitness among the programs in that generation of the simulation. Here is an example (in which the best programs don't appear because they're going to separate files):

```
>>> GA(200, 15)
Fitness is measured using 20 random trials and running for 800 steps per trial:

Generation  0
  Average fitness:  0.0575675
  Best fitness:  0.217125

Generation  1
  Average fitness:  0.08800625
  Best fitness:  0.453125

Generation  2
  Average fitness:  0.1041525
  Best fitness:  0.343625
```

```
Generation  3
  Average fitness:  0.113141875
  Best fitness:  0.41525

...

Generate 14:
  Average fitness:  0.869975625
  Best fitness:  0.880875

Best Picobot program (also in gen14.txt):
3 xExx -> W 3
1 xxWx -> E 2
1 NExx -> W 4
1 xxxS -> N 0
0 xxWS -> N 1
3 xxxx -> S 2
 ... rest of the program omitted ...
```

Certainly you may want to introduce a small number of additional helper functions as well -- this is up to you.

You might also want to read a bit about how we chose the fittest programs to reproduce and the choices of parameter values that we used (although you are certainly welcome to experiment and use different ones).

How Fast and How "Good" Should My Program Be?

Using the parameter values mentioned above, running `GA(200, 20)` should take about 60 seconds per generation. Your initial population will probably have an average fitness of around 1-2%. The average fitness should slowly increase at every generation -- exceptions will be relatively few. The maximum fitness will normally increase, but can be some dips here, as well.

By the end of 20 generations you should see a maximum fitness of 80-90% -- in some cases even higher than that.

Milestone deliverable

For the milestone, you should submit a version for which the following pieces are implemented and work:

- You have a `Program` class with a working constructor and `__repr__`
- You have a `World` class with a working constructor and `__repr__` and the methods
- `step` and
- `run` work
- In essence, this means you've built an ASCII Picobot simulator...
- The genetic algorithm part of the project need not be complete for the milestone, but if it is, all the better!

Be sure to zip up the folder containing your Picobot project file (or files) and name it `picobot_milestone.zip` -- then submit that in the right spot for Picobot in Hw#12!

Final deliverable

For the final project submission, you'll need to finish the genetic algorithm that evolves Picobot programs. Remember that we will test your program using your `GA` function as described above.

Test your program carefully. You may determine that your crossover (mating) mechanism is not very effective, in which case you may need to experiment with other ways to define crossover.

Zip up all of your files and submit them in `final.zip` Include

- your final project in a file called `final.py`.
- Include a text file named `final.txt` that describes
    - how you tested your code and chose the parameters of your genetic algorithm, e.g.,
        - what fraction of each population you decided to be fit enough to "mate"
        - how you arranged the mating and mutations
    - how well your overall fittest evolved Picobot program performed and *how* it did so
    - please include your most-fit picobot program, too, in a comment -- along with its fitness -- that way, we can run it!

Want More? Optional BONUS extensions...

If you'd like to explore the genetic-algorithm facets of this problem further, you have the option to investigate some additional facets of the problem:

- How good a *maze-solving* program can you evolve (using a different `World` room...)?
  - The maze is quite difficult to learn to solve -- you might try the diamond-shaped room as a medium-difficulty challenge.
- It's possible to evolve quite a fit Picobot program *without any crossover at all!* The idea is simply to keep the fittest from the previous generation and use mutation to create a next generation, but it typically takes longer:
  - Design and implement the necessary software scaffolding to test *how much larger a population* and/or *how many more generations* are required in order to evolve as fit a program using only mutation, relative to the use of both mutation and crossover?

Help for debugging...

Wow! Debugging Python program that evolve Picobot programs can be challenging...

Here I'll include some things I use to help (in small font)...

This is a method, `working`, for the `Program` class that is analogous to `randomize`. The difference is that `working` sets the rules to a known, working program that covers the whole empty room. Good for making sure `step`, `run`, and `evaluateFitness` all work...

```
def working(self):
    """
    This method will set the current program (self) to a working
    room-clearing program. This is super-useful to make sure that
    methods such as step, run, and evaluateFitness are working!
    """
    possiblekeys = [(0, "NExx"), (0, "NxWx"), (0, "Nxxx"), (0, "xExS"),
     (0, "xExx"), (0, "xxWS"), (0, "xxWx"), (0, "xxxS"), (0, "xxxx"),
     (1, "NExx"), (1, "NxWx"), (1, "Nxxx"), (1, "xExS"), (1, "xExx"),
     (1, "xxWS"), (1, "xxWx"), (1, "xxxS"), (1, "xxxx"), (2, "NExx"),
     (2, "NxWx"), (2, "Nxxx"), (2, "xExS"), (2, "xExx"), (2, "xxWS"),
     (2, "xxWx"), (2, "xxxS"), (2, "xxxx"), (3, "NExx"), (3, "NxWx"),
```

```
 (3, "Nxxx"), (3, "xExS"), (3, "xExx"), (3, "xxWS"), (3, "xxWx"),
 (3, "xxxS"), (3, "xxxx"), (4, "NExx"), (4, "NxWx"), (4, "Nxxx"),
 (4, "xExS"), (4, "xExx"), (4, "xxWS"), (4, "xxWx"), (4, "xxxS"),
 (4, "xxxx")]
POSSIBLE_MOVES = ['N', 'E', 'W', 'S']
POSSIBLE_STATES = [0, 1, 2, 3, 4]
for k in possiblekeys:
    st = k[0]
    surr = k[1]
    if st == 0 and ('N' not in surr):   val = ( 'N', 0 )
    elif st == 0 and ('W' in surr):     val = ( 'E', 2 )
    elif st == 0:                       val = ( 'W', 1 )
    elif st == 1 and ('S' not in surr): val = ( 'S', 1 )
    elif st == 1 and ('W' in surr):     val = ( 'E', 2 )
    elif st == 1:                       val = ( 'W', 0 )
    elif st == 2 and ('E' not in surr): val = ( 'E', 2 )
    elif st == 2 and ('N' in surr):     val = ( 'S', 1 )
    elif st == 2:                       val = ( 'N', 0 )
    else:
        stepdir = surr[0]
        while stepdir in surr: stepdir = random.choice(POSSIBLE_MOVES)
        val = ( stepdir, st )  # keep the same state
    self.rules[k] = val
```