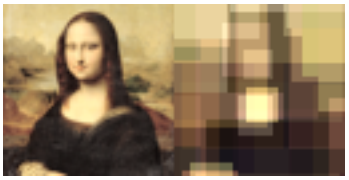
CS51A—Assignment 9
Evolving Art

Due: Tuesday, 4/14 at 11:59pm



In this assignment, you'll implement major portions of a *genetic algorithm* whose goal is to approximate a color image using only superimposed colored rectangles. By the end of this assignment, you should be able to:

- Install extra packages for Python using the `pip` package manager or through PyCharm.
- Implement and compare alternatives for an evolutionary search algorithm's selection, crossover, and mutation strategies
- Implement the main loop of an evolutionary search from pseudocode
- Apply your search algorithm to a new target image and evaluate how hyperparameter changes affect the algorithm's performance

This assignment is inspired by a couple[1] of demonstrations[2] of using genetic algorithms to approximate images. These types of algorithms are very useful if it's easy to come up with examples of a thing and evaluate its quality, but hard to know specifically how to improve a thing to make it better. They have applications in computer-aided design of everything from radio antennae to integrated circuit layout as well as virtual life in video games and even automated generation of video game rules.

Starter

Like many of the previous assignments, this assignment also has a starter code that contains several things:

[1] `http://datagenetics.com/blog/april2009/index.html`
[2] `https://chriscummins.cc/s/genetics/`

- `picture.py`, a module for reading and writing images in `ppm` format.
- A starter file called `assign9.py` that has some initial code to get you started.
- `images/`, a directory of ppm images used for evaluating your genetic algorithm.

Create a PyCharm project called `assign9` and put those two files and one folder into it.

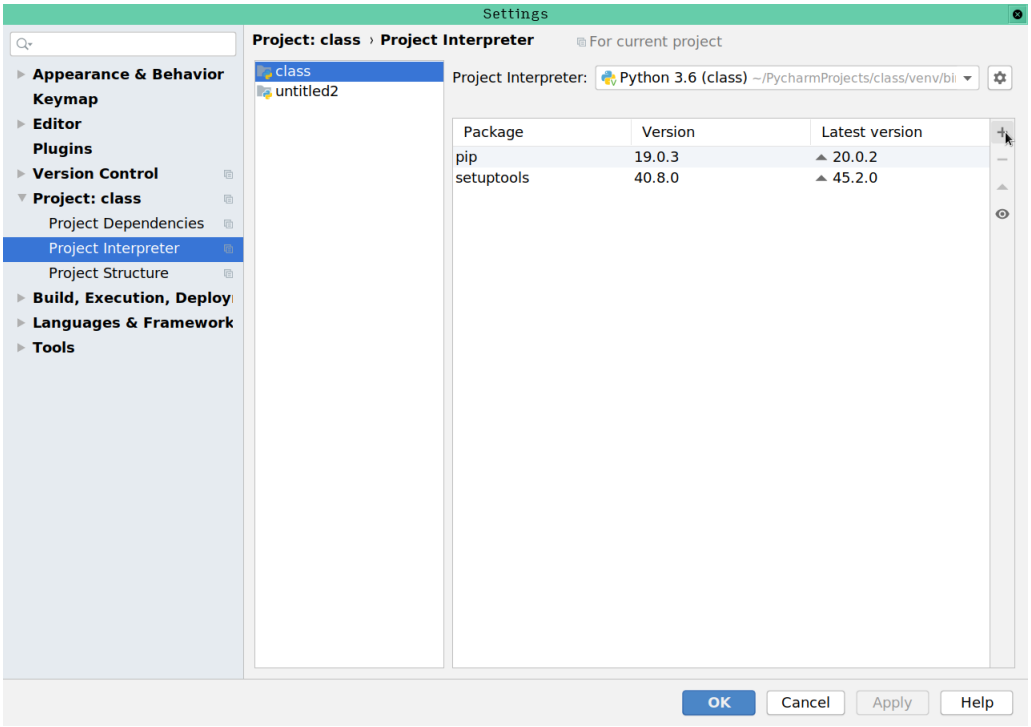
```
https://cs.pomona.edu/classes/cs51a/assignments/assign9-starter.zip
```

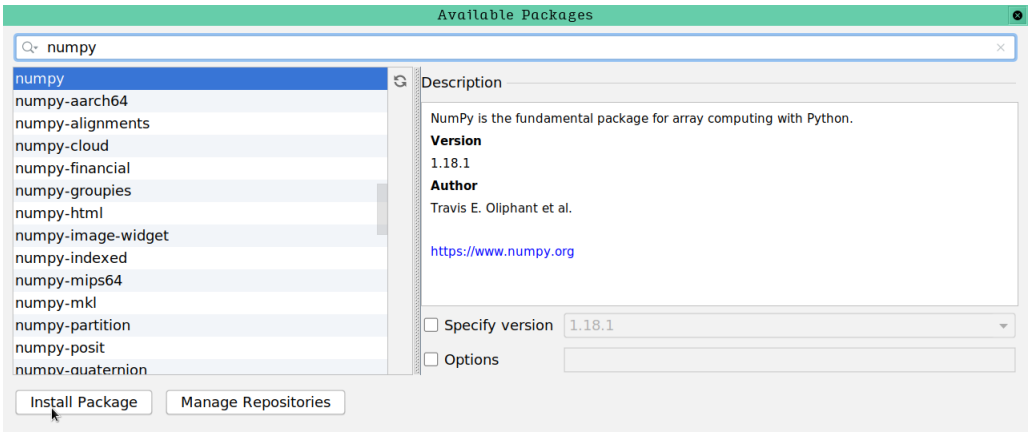
Put your name, etc. at the top of the file in comments.

Primordial Soup

Before diving into the code, it would be good to familiarize yourself with the starter code and `picture.py`. First, we need to install a *dependency*: `numpy`, a library for doing computations with nested numerical arrays of arbitrary dimensions. We'll internally treat images as arrays of arrays of arrays: `height` rows of `width` pixels each, where each pixel is comprised of three floating point 0.0–1.0 color values (red, green, and blue). This is just another module of Python code, but you don't have it on your computer (or in your project's *virtual environment* yet). We'll install that now.

In PyCharm, pop open the menu item `File>Settings` and choose `Project: assign9>Project Interpreter` in the left hand column. Next, hit the plus button near the top right corner of the table of packages.





In the window you see next, search for `numpy`, select it from the list, and click `Install Package`. If all goes well you'll see a green box in this window stating `Package 'numpy' installed successfully` and you can close this search window and the settings window.

Go ahead and open `picture.py` and explain in one or two sentences each of these three methods of the `Picture` class; write your answers in a multiline string at the beginning of your `assign9.py` file.

- **[1 point]** `clear`: What does it do and how? The syntax will be new, but what do you think it does given what is written earlier about how images are three-dimensional arrays?
- **[1 point]** `compare`: What does it do and how? You can look up numpy's documentation if it's helpful. What do you think the "difference" or subtraction of two images should be, and what's getting summed up at the end?
- **[1 point]** `blend_rect`: What does it do and how? `self.pixels` are accessed using two slices, but it's three dimensional; why is this okay? And what do you suppose that `clip` function does?

Every genetic algorithm starts from a definition of the *genome*: what data do we need to represent an individual? We may also map individual genomes onto *phenotypes*; in this case, our phenotype had better be an image (since that's what we're trying to produce), but the genome is *not* a bunch of red/green/blue pixels. Instead, each individual is made up of (by default) 50 *genes* representing rectangles. Each gene comprises a position (x and y coordinates), a size (w and h), and a color (r, g, b, and a for alpha (transparency)), so our genome is $50 * 8 = 400$ numbers. For convenience we'll keep each number between 0 and 1, and later scale up the coordinates and position by the target image's width and height.

To sum up, we are approximating the target image by generating lots of different images; each image we make is made up of 50 colored rectangles described by a genome of 50 genes, where each gene in each genome specifies one rectangle. Our *population* might consist of 30, 50, or 100 individuals, and each individual is a complete genome representing one image.

In our biologically inspired model, we want the "strongest" to survive; we therefore need a way of quantifying the strength of each individual. Add answers to these questions to your multiline comment from before:

- **[1 point]** In your own words, how does the starter code determine the fitness of an individual image genome? Is a higher or lower fitness better?
- **[1 point]** If you wanted to make a new random individual with a given number of genes, how would you do that given the starter code?

We cram these `Individuals` into a `Population`, and iterate the following process for a given number of steps:

1. Initialize a new list of individuals (the next "generation") with some of the very strongest individuals from the current generation.
2. As many times as we care to, select a pair of individuals from the current iteration and breed them together (and mutate the result) to create a new individual in the new generation.
3. Fill out the new generation with fresh randomly initialized individuals.
4. Sort the individuals by fitness.

Finally, glance at `run_ga` and answer the following questions in your top-of-file comment. When you're done, clear your answers with a TA or Prof. Ye:

- **[1 point]** What are the parameters for a `Population` and what do they mean?
- **[1 point]** What should we expect to happen when we call the `run_ga` function for, say, 1000 steps? What kind of output will it generate and what work will it do?

Building Blocks of Life

Now it's time to start writing code! We'll start with two implementations of `select`, a method on `Population` that grabs two individuals from the current generation (it takes no parameters besides `self`. You can use several different criteria:

- Select two individuals at random. (Consider `random.choice`)
- Select 10 (or 8, or 12, or...) individuals at random and pick the best one; repeat. (Consider `random.sample` or `random.choices`; but how to pick a certain number, and then how to pick the best one? `calculate_fitnesses` may provide a good hint)
- Select 10 individuals at random and pick the best two. (See hint above)

- Make a weighted random choice among all individuals, where each weight is determined by the individual's fitness. (`random.choices` can be used for this too! But it wants a probability distribution, so this is trickier than it seems)
- Something else?

[4 points] Implement two methods `select_a` and `select_b` with docstrings describing your strategy, then define `select` to call whichever one you like. You can test them individually using a setup like this:

```
# We need a picture to calculate fitnesses
picture = picture_from_ppm("images/losangeles64.ppm")
# Just get a random population
pop = Population(20, 0, 0, picture, 50)
print(pop.select()) # Should return two Individual objects
```

You can run your selection function repeatedly to see whether strong individuals are preferred the way you might expect.

We'll implement `crossover_with`, a method on `Individual`, next. This method is called on an Individual and an Individual is passed in as an argument; the output is a new Individual and the receiver and argument should remain unchanged (i.e., its type is `crossover_with(self, ind:Individual) -> Individual`. Again there are several good strategies:

- For each parameter of each gene, randomly decide whether the new individual gets the receiver's value or the mate's value. (You can iterate through the individual's genes if you like)
- Pick a position within the genome at random; everything to the left should come from one individual and everything to the right can come from the other. (Consider using slicing here to update the new individual's genome)
- Pick e.g. two positions within the genome at random; everything to the left of the first or the right of the second comes from one individual, and the middle part comes from the other.
- Pick some genes from one individual and some genes from the other (instead of individual parameters, cross over whole rectangles)

[**4 points**] Implement two methods `crossover_with_a` and `crossover_with_b` with docstrings describing your strategy, then define `crossover_with` to call whichever one you like. You can test them individually using a setup like this:

```
ind1 = Individual(2)
for i in range(len(ind1.genes)):
    ind1.genes[i] = 0
ind2 = Individual(2)
for i in range(len(ind2.genes)):
    ind2.genes[i] = 1
print(ind1.crossover_with(ind2).genes)
# Examine the genes that come out;
# should be neither all zeroes nor all ones.
# Exactly what the mix is should depend on your strategy,
# but it would be good to aim for about half and half.
```

The last part we need is a way to *mutate* an Individual in-place (`mutate(self, rate:float, amount:float)`. Lots of different types of mutation are possible:

- For each parameter of each gene, with some probability randomize it to a new value. (You can iterate through the individual's genes if you like)
- For each parameter of each gene, with some probability randomly increase or decrease it by some (random?) amount.
- For each gene or parameter, with some probability randomly swap it with another one in this individual.
- With some probability reverse each gene (or the whole genome!)

[**3 points**] Let's commit to the second one for now and implement it as method `mutate`. You can test it like so:

```
ind = Individual(2)
for i in range(len(ind.genes)):
    ind.genes[i] = i/len(ind.genes)
print(ind.genes) # Should be increasing floats
ind.mutate(0.1, 0.2)
print(ind.genes)
# Examine the genes that come out;
# if they're just increasing floating point numbers,
# that means mutation isn't doing much.  You should see
# that about 1/10 numbers break up the pattern:
# they're much bigger or smaller than the number before them.
# No number should be <0 or >1.
```

Note that it is *extremely important* that the value of each parameter remain between 0 and 1 even after mutation!

Life Finds a Way

Now that all of the messy choices are out of the way, we can implement the natural selection and evolution step. Recall that an evolutionary algorithm repeats this sequence of steps over and over:

1. Initialize a new list of individuals (the next "generation") with some of the very strongest individuals from the current generation. (Creating a new list, adding some of the best individuals from the current sorted list to it)
2. As many times as we care to, select a pair of individuals from the current iteration and breed them together (and mutate the result) to create a new individual in the new generation. (Some number of times: select, crossover, mutate)
3. Fill out the new generation with fresh randomly initialized individuals. (Add some new random individuals)
4. Replace the current generation by the new generation.
5. Sort the individuals by fitness. (Call `calculate_fitnesses`)

[**4 points**] We have all the pieces we need, so go ahead and implement `evolve_step` on `Population` to do each of those things (this function shouldn't take any arguments besides `self`. Be sure to use the appropriate instance variables of `Population` to know for example how many individuals to keep around from the previous generation—and be sure the new generation has exactly as many individuals as the previous one!

At this point we're ready to try it out! Call `run_ga(50, 50, 5, 20, PICTURES["mona64"], 2000)` and wait for the result. You'll see some diagnostic output in the console every five steps, printing out the amount of time the most recent evolution step took (DT), the net time spent so far (Net), and the predicted time to complete the given number of steps (Pred). You'll also see the step number and the fitnesses of the best and worst individuals in the population. You want the DT value—the time of the most recent step—to be pretty small, e.g. comfortably less than 0.1 seconds. If you're doing 2000 iterations, that means you'll finish in fewer than 200 seconds which is about as long as you'd want to wait. If it's running much slower than that, ask a TA or professor for help speeding things up (it may point to a bug somewhere). Otherwise you won't be able to test enough times to debug your program.

You'll also find a file next to `assign9.py` called `best.ppm` will be rewritten many times with progressively better approximations. If you have an image viewer that will reload the file automatically when it changes that would be ideal, or you can just close and reopen it from time to time to check in. Record and write down in a new multiline comment at the end of the file:

- [**1 point**] What was the best fitness you obtained after 2000 steps?

- **[1 point]** Did it keep decreasing significantly the whole time, or did it plateau at some earlier step?
- **[1 point]** What are the effects of choosing different *selection* strategies on convergence rate, final accuracy, and computational efficiency?
- **[1 point]** What are the effects of choosing different *crossover* strategies on convergence rate, final accuracy, and computational efficiency?
- **[1 point]** What are the effects of choosing different *mutation* parameters on convergence rate, final accuracy, and computational efficiency?
- **[1 point]** Rename `best.ppm` to `best-mona.ppm` and try out the algorithm on different images. Do you need to tweak the parameters (population count, gene count, keep and select counts, iteration count, mutation parameters, selection or crossover strategies) to make it work acceptably? Save out an image resulting from these experiments as `best-experiment.ppm`.
- **[1 point]** What do you anticipate the tradeoff might be between having more or fewer genes in an individual, with respect to e.g. accuracy or convergence time?

Extra credit

For up to **[2 points]**, try this with your own image! Convert it into a "binary PPM" image (the open source art program Krita can do this) and find a good set of parameters for it (you may need to tweak the `PICTURES` dictionary). Submit your target image and best obtained image from the algorithm, and write the best parameter settings you found for your image in the multiline comment at the end of the file.

When you're done

Make sure that your program is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- Each function should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

In addition, make sure that you've used good *style*. This includes:

- Following naming conventions, e.g. all variables and functions should be lowercase.
- Using good variable names.

- Proper use of whitespace, including indenting and use of blank lines to separate chunks of code that belong together.

- Make sure that none of the lines are too long

Submit your `assign9.py`, `best-mona.ppm` and `best-experiment.ppm` for one of the example images. If you're doing extra credit also submit your own ppm image along with its corresponding `best.ppm` using the course's submission mechanism.

Grading

	points
Primordial Soup	7
`select`	4
`crossover`	4
`mutate`	3
`evolve_step`	4
Analysis	7
Comments, style	3
extra credit	2
total	32 (+2)