Numpy and pyplot exercises

In these exercises, “array” means a “numpy ndarray”. If the type is not specified, assume it’s the default floating point type (most likely it will be `np.float64`).

Don’t bother with argument checks in these exercises.

Use slicing and built-in array methods as much as possible (in other words, try to avoid `for` loops).

1. Numpy integer types

Create a variable storing the number 100 as a 16-bit integer. Then another variable which stores 100 as a 16-bit **unsigned** integer (generally abbreviated “uint”). From the lecture notes, compute which are the minimum and maximum values that these two types can store. Verify your computations by “testing the boundaries”: perform operations that reach the boundaries, then go beyond and observe that the numbers start cycling around.

Also experiment with mixing different number types and observe what happens (e.g. if you sum a signed integer with an unsigned integer, etc.). Observe the output result, but also the output type. Put other integer types in the mix too, e.g. 64-bit integers, signed and unsigned, standard python integers (try both small and very large values). If you can, try to infer some (potentially vague) general underlying principle that is used by Python to decide which output type to use if you mix different input types.

2. Numpy array creation basics (I)

Write a python list of integer numbers, then create a 1-d numpy array out of it. Read out the element type from the array. (There is a specific array attribute for that, make sure that you know what it is! The mnemonic to remember what it is is “data type”.)

Then do the same for a list of floating point numbers. Again, read out the element type.

Then create a list in which there is a mix of integers and floating point numbers, e.g. `[1, 2.0, 3]`, and create a numpy array from it. What do you expect to happen in such case? Verify your expectations by checking in practice.

What if you wanted to ensure that the element type is floating point even when your input list only contains integers?

What happens if your starting list contains floating point values but you try to force it to a `bool` element type?

What happens if your list has elements that are not numbers, e.g. if some elements are strings? Check this. Inspect your results carefully. Get an element of the array out (e.g. with `a[0]`) and then check its type with the `type` built-in function. Is the result an instance of a string? Check it with `isinstance`.

What about if one or more elements are sets, or dicts?

With the results of the last test, consider again a list with a mix of integers and floating points, and consider an unlikely case in which you didn’t want any implicit conversion to happen - you want your array to retain the type of each element even inside a numpy array. How can you do that?

Finally, a corner case: what happens if you pass an empty list to the constructor, and you don’t specify the element type?

3. Numpy array creation basics (II)

Write a function that takes a single argument `n` and creates a 1-d array of size `n`, filled with ones. Make it so that

the element type is floating point. Then, make it so that it is 64-bit integer. Then, make it 32-bit integer. Then, make it an array of bools (what the is equivalent of `1` when using bool values?). Then make it an array of strings (what does this even mean? Can you make sense of what's happening?). Make sure that you know how to read out the element type from the array. Also look at the type of the first element of the array.

Then, make the function create an `n` × `n` matrix, still filled with ones, of some integer type. Observe what happens when you multiply it by an integer number. Then observe what happens when you multiply it by a floating point number.

4. Some broadcasting basics

Write a function that takes a single argument `n` and creates a 1-d array of size `n`, filled with the value `-1` (integer). Do it in a single line of code. There is a dedicated function for this, but you can also exploit the simplest of the broadcasting rules.

Note: there are at least 4 ways to do this. Can you figure them all out?

5. Reading out array chunks

Write a function that takes a list as its argument, let's say that it is of length `n`. The function should first create a 1-d array from this list. Then, it should read out the first half of the array. Then it should read out the second half. Both these operations must be performed with one line of code each. If `n` is odd, then the middle element should be put in the second list.

If you were working with Python lists, you could concatenate back the two halves using `+`. With numpy arrays you can't do that though. Why? (Try it; use input lists with even and odd `n`, and see what happens.)

Look up the function `np.hstack` in the numpy documentation (maybe also look at `np.stack` and `np.vstack` while you're at it, although they are not necessary for this exercise). Using that function, you should be able to concatenate the two halves and get back an array like the original one. Do that, and return the result.

Here are some additional questions, to be done after you have completed the task above. You should answer by experimentation - try to answer without testing at first, but then **always** check your intuition with some concrete examples!

1. Is the resulting array identical to the original one, or just equal? How do you make sure, and what difference does this make?
2. You expect the final array to be equal to the initial array, so maybe you could use `==` to check that. Can you? Try it and see what happens. Look up the documentation of `np.all` function, or equivalently of the `.all` array method. With that, find a way to write a check in your function that ensures that everything is as it's supposed to be, by using an `assert` statement. After that, check the documentation of `np.array_equal` too.
3. If you did things correctly, each chunk of code was obtained with a slicing expression. Now take one of the halves that you have obtained and change one element, say the first. After that, print out the original array. Would you expect it to be changed? Can you tell what's happening here? Print out the `.base` attribute of the two halves. What is it identical to?

6. Slicing in strides (I)

Write a function that takes a list as its argument, let's say it is of length `n`. Let's also assume that `n>0` for simplicity (the input list is not empty). The function should first create a 1-d array from this list. Then, it should read out all the elements with even indices, in a new array. Then all the elements with odd indices. Both these operations must be performed with one line of code each, using slicing expressions.

What happens if you change one of the new sub-lists? Does this affect the original array or not? Why? Check your intuitions by experimenting!

7. Slicing in strides (II)

Write a function that takes a list as its argument, let's say it is of length `n`. The function should first create a 1-d array from this list.

After this, read out the following sub-arrays from it, in each case using a single slicing expression (no list methods, no array methods, just indexing with slicing!):

1. The whole array, but in reverse. E.g. if the input is `[4,3,6,8,9]` you should read it from `9` to `4`.
2. The first half of the array, but in reverse. Then the second half, also in reverse. If `n` is odd, then the middle element should go into the second half. This is similar to ex. 5 combined with the previous point 1. In fact, you could do each of these in two ways. One way (easier) is to do them in two steps, first like in ex. 5 and then reverting the result like in point 1. The other way is to do everything with a single expression, in which case there is a subtle change that you need to make in the indexing compared to the first way. Be extra careful when checking your results, and make sure that you understand what's going on with the indices and why.
3. All elements with even indices but in reverse. Then the elements with odd indices, also in reverse. This is similar to ex. 6 combined with the previous point 1, and again a simple way is to perform these tasks in two steps. However, you should then try to find a second way in which everything is done with a single slicing expression. In this case, you should realize that if `n` is odd, this is much more tricky than you may think at first. Why? As a starting point, you may use an `if` expression here. Verify that you got it right by experimenting with lists of even and of odd size. Then, once you get this correctly, still try to do each of these with one single indexing expression and no `if`s (this is **very** tricky!)
4. Take the result of point 1 and index it in reverse again. The result should be the same as the original array. Check this with `np.array_equal`, see also exercise 5. Are the two arrays identical? Check this with `is`. If you change the result, does it change the original array? Check this by doing an experiment. Can you make sense of what's going on? Use the `.base` attribute. Can you find a slicing expression that would give you the same result as this double-reversed array, i.e. something with the exact same properties?

For completeness, use the `.base` attribute also for the result of points 1-3 and check whether you are getting views or not. Also, in points 2 and 3, when you do the intermediate tests and do everything in two steps, have a look at the `.base` of the result too.

8. Indexing with lists (I)

Write a function that takes a list as its argument, let's say it is of length `n`. Let's also say for simplicity that `n` is odd and that `n>=1`. As for the previous exercises, the function should create a 1-d array from this list. Use an indexing expression that uses another list (of length 3) to extract a new array which contains, the first, the middle and the last element of the original array. (Since `n` is odd, the array will always have a middle element.) For example, if your input list is `[4,6,1,2,0,7,9]`, then your result should contain `4`, `2` and `9`.

Do you get a view of the original array or not from this? Write some code that verifies your answer.

Important note: If `n==1`, the 3 elements of your indexing list should be the same: this is no problem at all! When you use a list for indexing, there can be repetitions in the list. The result will just be an array with 3 identical values. To convince yourself, try to index an array with different lists and observe what happens, and observe what happens in particular if you have repetitions, e.g. take an array of length 3 and observe that you can use a list like `[2, 0, 1, 0, 0, 1]`, resulting in an output array of length 6: larger than the original!

9. Indexing with lists (II)

Write a function that takes two arguments. The first argument is a list, let's say it is of length `n`. For simplicity, the length should be `n>=1`. As for the previous exercises, the function should create an array from this list. The second argument should be a non-negative integer `m`. This will be the size of our output array.

Look up the specifications for the function `randint` in the `np.random` module. You want to use it to create an

array of `m` random integers, each of which is between `0` (included) and `n` (excluded). Hint: the code will be clearer if you use the `size` keyword argument.

Use the randomly generated array of indices to index into your original array. The output should be an array of length `m`. Running the function several times will produce different results every time. Make sure that everything works as intended by running it several times, with various combinations of list lengths and `m` (i.e. be sure to cover the cases `n<m`, `n==m`, `n>m`).

Important note: notice that in the previous exercise we were using Python lists as indexing expressions, while now we have been using numpy arrays of integers to the same effect. The result is in fact the same (the numpy version is slightly more efficient though).

10. Indexing with masks (I)

Write a function that takes a list as its argument, let's say it is of length `n`. Let's also say that the list may contain both positive and negative numbers. As for the previous exercises, the function should create a 1-d array from this list. Compare the whole array with `0`, i.e. do something like `a > 0`. What is the result? Check this.

Use the result as a mask in an indexing expression. You want to get a new array which contains only the elements of the original array that are non-negative. For example, if your input list is `[3, -1, -2, 5, 6, -3, 8]`, your output should contain `3`, `5`, `6` and `8`. Can you do this with a single indexing expression? (Yes you can!)

Do you get a view or not from this? Write some code that verifies your answer.

11. Indexing with masks (II)

Write a function that takes two arguments. The first argument is a list, let's say it is of length `n`. As for the previous exercises, the function should create an array from this list. The second argument should be a floating point number `r` between `0` and `1`.

Look up the specifications for the function `random` in the `np.random` module. You want to use it to create an array of `n` random floats, each of which is between `0.0` (included) and `1.0` (excluded). Let's call this list `s`.

You need to compare this random list of floats with `r`, e.g. `s < r`. As we have seen in exercise 10, the result is an array of bools. Use this array of bools to index into the original array.

Overall, the result will be that you will have created a new array by picking elements from the original one, each of which was chosen with probability `r` and discarded with probability `1-r`. Verify that if `r==0` your result is always empty, and that if `r==1` your result is always equal (but not identical!) to the original list.

Also verify this: that the length of the result is equal to the number of `True`s that is present in the random mask. In order to compute how many `True`s there are in a mask, use the following observation, which allows you to do it in one single line of code: if you sum boolean values, they get interpreted as `0` and `1`. Therefore, for example, `True + False == 1` and `True + True == 2`. Therefore, you just need to sum up all the values in the mask and you will have effectively counted the number of `True`s.

Finally, also write a second function that calls the first one repeatedly, with the same arguments, over and over. Make it such that it takes the same argument as the other one, a list and a float `r`, but also a new one with the number of iterations, with some large default value (e.g. 1000). Compute the average value of the length of the results. Verify that when the number of calls (controlled by the iterations argument) becomes very large, the average length of the output list tends to `n * r`. Note that this should happen even if `n * r` is a non-integer number like `3.43` or something. Make it print both the measured average length and the expected one, so that you can compare.

12. Automatic conversions

Create an array of float values. Then assign an integer to it. What happens? Try with various integers. Try with

very large integers too, something larger than `1.8e308` (why this value? what happens there?)

Now do the opposite: create an array of integer values, and assign a float to it. Observe what happens. Make it an array of 16-bit integers and assign a value out of the range of `np.int16`. Observe what happens. Try using the special values `np.inf` (representing infinity) and `np.nan` (representing the not-a-number special value).

Feel free to experiment with other combinations, e.g. use bools (in both roles). If you think that you got the idea when using integers, well think twice: you'll get some surprising results.

13. Assigning with slices

Slices work in assignments too (which is the case also for Python lists by the way…). Write a function that takes a list, converts it into an array, gets the first and second half with slicing (up to here it's the same as exercise 5), then reverts those two half slices (see also exercise 7), and finally writes back the two haves reversed into the original array.

For example, if the input list is `[3, 5, 6, 7, 8, 9]` you want the output to look like `[6, 5, 3, 9, 8, 7]`. As before, if the length is odd, put the middle element to the second half. For example, if the input is `[3, 5, 6, 7, 8]` the output should look like `[5, 3, 8, 7, 6]`.

There are several ways to do this actually. One is like described above. But you could also not revert the two halves, and use a reversed indexing expression in the assignment instead. Try it and verify that the result is the same.

You could also use `np.hstack` (see exercise 5 again), and use the indexing expression `[:]` to overwrite the whole original array. (Aside task: make sure that you full understand the difference between assigning to a variable `a` as opposed to assigning to a "view" like `a[:]`.) However, using `hstack` creates an intermediate temporary array. Can you understand why?

Important note: This way of assignment only works when the sizes match. So for examples `a[i:j]=something` only works if `something` has the same length as `a[i:j]`. This is also true for the next exercises, when using lists and bool masks in the assignments. The only exception is broadcasting (see later exercises).

14. Assigning with lists

Do the same as exercise 8, but this time set the first, middle and last element of the array to `0`, `1` and `2`, and return the array. For example, if the input list is `[9,9,9,9,9]` the output array should look like `[0,9,1,9,2]`. You must do this using a single line of code for the assignment (just mimic what you did in exercise 8).

What happens now if the input list has length 1? You are assigning 3 different values into a single element. What do you expect to happen? Verify your expectations.

Still about the case `n==1`. Suppose that you want to make sure that in this case the output will be `[1]`, i.e. you want to give precedence to the "middle" value. Instead of using an `if` expression, modify the indexing expression to ensure this.

15. Assigning with masks

Similar to exercise 10, but with a twist: use a single expression involving bool masks (both in reading and in assignment) in order to make all element of an array positive.

Hint: get the mask of the negative values, read those out, flip the sign, put them back (using the same mask).

At the end test your code by checking that all values are larger or equal than 0 using a one-line expression involving `np.all`. Also do another equivalent test but this time by checking that no values are smaller than 0, using a one-line expression involving `np.any`.

16. Broadcasted assignment (I)

Write a function that takes one argument `n`, and returns a 1-d array of length `n` with alternating zeros and ones, e.g. if `n==7` you want something like this:

```
0.0 1.0 0.0 1.0 0.0 1.0 0.0
```

Here however you need to use broadcasting: assigning a scalar (a single value) into an array slice. So in total it can be done with no more than 3 lines of code, including the return statement.

There are at least two ways to do this, can you write both?

17. Broadcasted assignment (II)

Write a function that takes one argument `n`, and returns a 1-d array of length `n` in which the first half is filled with ones and the second half is filled with zeros. If `n` is odd, then the middle element should be a zero. No more than 3 lines of code (including the `return` statement).

Write variants in which the resulting array contains integer numbers. Then another with bools (in the latter case we implicitly assume `True==1` and `False==0`).

18. More array creation tricks, broadcasted functions, and some plotting

Create a 1-d array with the elements between `0.0` and `2π` in it, at regular intervals. Use a variable `n` that determines the number of intervals. Include `0.0` but exclude `2π`. How do you write that in one line of code? You'll need a special numpy array constructor.

Now do the same, but have `2π` be the last element of the array (i.e. don't exclude it), still with the same total number of elements as before (i.e. the interval is going to be larger now). How do you still write that in 1 line of code? **Hint:** you'll need a *different* special numpy array constructor now.

Now compute both the `sin` and `cos` of the previous result. One line of code each. Exploit function broadcasting.

Plot the two `sin` and `cos` curves on top of each other. Make sure that the x axis is labeled "radians" (see the pyplot tutorial…)

Advanced: have a label in the figure which tells you which curve is the `sin` and which is the `cos`. (You'll need the `label` keyword argument and the `legend` function. The documentation is a little cryptic; it's easier if you look at some examples from the gallery section of the documentation too.)

Advanced: save the figure as a "png" image. This is much easier if you set your IPython console preferences in Spyder to not print the graphics inline, but in an external window instead (Note that you'll need to restart the kernel if you change that option). Otherwise use the `savefig` function (change the `dpi` option to control the image resolution).

19. Normal random numbers

Generate 1 million random numbers, each of which is extracted from a Normal (i.e. Gaussian) distribution with mean `0.0` and variance `1.0`. Do this in two ways:

1. create a list using the built-in Python `random` module and using a comprehension;
2. create a 1-d array using numpy's `random` module, with one line of code, no comprehensions. Figure out which function you should use by yourself, browsing the documentation.

You should observe a noticeable difference in the time required using the two methods. You can try to estimate the

performance difference by importing the `time` module and using the `time()` function. For example,

```
t0=time.time(); myfunction(); t1=time.time(); print(t1-t0)
```

20. Normal random numbers, 2-d arrays, computing statistics and more

Create a 2-d array of 10×20 random Gaussian numbers using numpy (like for exercise 19, just this time it's 2-d). Compute their mean (1 line of code).

Now take a sub-table of the first table in which only the elements whose row and column indices are both even are kept: e.g. elements like `[2,4]`, `[0,8]` are ok, while `[1,2]` or `[3,5]` are not. Do this with a single indexing expression. **Hint:** use two slices, one for the rows and one for the columns, and see also exercise 6.

Question: do you get a new independent array or a view? Check your answer by modifying the second array and verifying whether the first array is also changed, or not. How could you obtain the opposite result, if you wanted to? **Hint:** there is an array method that you should use after the slicing.

Compute the mean of this sub-array.

Now put everything in a function that does all of the above operations repeatedly for a number of times (say, 1000 times); i.e. produce a random array and compute its mean and the mean of the "even-indexed sub-table". The function should collect the results in two separate 1-d arrays.

Following this, the function whould plot a histogram of both these arrays of means. Remember that if you keep calling `plot` or `hist` the new plots will be superimposed on the previous ones, therefore you will need a command to clear the figure between tests. You should clearly see a roughly Gaussian shape of the resulting curves. Look at the available options for `hist` here, and try to have the two histograms plotted as lines on top of each other, i.e. not to have the histogram bars filled. Also try to increase the resolution of the steps by increasing the number of bins: what happens? How do you obtain smoother curves? Experiment!

Extra: If you know some statistics, you should be able to predict the shape of the two curves, and verify your prediction by plotting the analytical curves on top of the histograms. In this case, you may want to look at the option `density` of the `hist` function.

21. More 2-d array indexing patterns questions

Say you have a 2-d array, and you want to set to 0 all the elements whose row and column indices sum to an even number. E.g. elements `[0,0]`, `[2,4]`, `[1,1]`, `[1,3]` etc. Think of this as a checkered pattern in the table.

Can you do it in 1 line of code in general? Why? What's the minimal number of operations that works?

[An extra puzzle for the aspiring indexing virtuoso: say that the number of columns is odd, and the array is *not* a view. Do the above in 1 line of code.]

22. Stacking

Create a list (a standard Python list) of 1-d arrays. Each array has 5 elements. The first one has the numbers from 0 to 4, the second from 1 to 5, the third from 2 to 6 etc. Generate `n` such arrays (i.e. create a list of length `n`). Do this in 1 line of code. **Hint:** you'll need some special numpy array constructors (seen in ex. 18); you'll also need to use comprehensions (you can start with a `for` loop and then convert that into a comprehension if it helps).

Once you have this list of arrays, do the following (do each tasks using 1 line of code, see also ex. 5):

1. Create a 2-d array of size `n×5` in which the `i`-th row is the `i`-th element of the list. E.g. if `n==3` you want to end up with a 2-d array like this:

   ```
   0 1 2 3 4
   ```

```
1 2 3 4 5
2 3 4 5 6
```

2. Create a 1-d array of size `n*5` that is the concatenation of the arrays in the list. E.g. if `n == 3` you want to end up with a 1-d array like this:

```
0 1 2 3 4 1 2 3 4 5 2 3 4 5 6
```

23. A more advanced broadcasting puzzle

Produce the same final result of exercise 22 part 1, but this time do it using only two 1-d arrays and 1 line of code. You'll need to exploit broadcasting, and use `reshape` (or `newaxis`, or `expand_dims`… there are several equivalent expressions that could produce the same result).

24. Sieve of Eratosthenes (I)

Implement a slightly sub-optimal version of the Sieve of Eratosthenes, like this:

1. create an array of bools of length `n`, all set to `True`
2. set the `0`-th and `1`st element to `False`
3. loop over all numbers between `2` and `√(n-1)` (rounded down to the nearest integer, included). For each value `i`, set to `False` all the elements indexed by multiples of `i`, starting from `i**2`.

The resulting array will only contain `True`s in the positions corresponding to prime numbers.

Advanced: get the indices of those prime numbers with one line of code, using an `arange` and boolean mask indexing. (Alternatively, you can use `nonzero` or `flatnonzero`; alternatively, you can use `argwhere` and `ravel`)

25. Sieve of Eratosthenes (II) - Advanced

Same as exercise 24, but this time implement the true sieve, as described in the Wikipedia page (or any other resource). The difference is this: in point 3, instead of looping through all indices between `2` and `√n`, use a `while` loop. Start from `i=2`, but then at each cycle you should get the next `i` as the first index after the current `i` for which your array is `True`.

Since the array is made of bools, you can use the `argmax` function from numpy to get the first `True` index in the array. But beware, you want to find the next `True` after the current `i`. Can you do that without creating copies or using loops?

Try to measure the time difference between this version of the algorithm and the previous one, with `n == 10**7` or larger. It should be roughly 4 or 5 times faster (the gain increases with increasing `n`). You may also want to try a version which uses Python lists and for loops, and check how much slower that is (on my laptop, roughly 30 times slower for `n==10**7`).

26. Conway's game of life - Very advanced (!)

Implement Conway's game of life. Write a function that takes an initial configuration as a table of zeros and ones (could be bools or ints) and a number of steps `n`. The function runs the game of life for `n` steps and returns the final configuration.

Note: in the original game of life rules, it is supposed that the grid is infinite; instead, for simplicity, you should use a finite grid instead, and just apply the standard rules even at the borders — which means that the border cells have only 5 neighbors, and the corner cells only 3.

Tip: for a more straightforward implementation, you'll need 2 arrays, one for the current configuration and one for the next one; after each step overwrite the current with the next. For an advanced implementation, you may try a version that computes the table of neighbors of each cell first, and then uses comparison operators and logic boolean filters (the operations `&`, `|` and `~`) to update the configuration.

Plot the configuration at each step as an image (using `imshow`). Remember to use `plt.clf()` before each new plot or your plots will rapidly become slow (can you guess why?). This works best if you don't use inline plotting (see note in exercise 18) and if you put `plt.pause(0.05)` after each plot in order to actually see what's going on.

Tip: start with small grids and simple configurations when debugging, and check one step at a time. Here is for example an interesting self-sustaining pattern, called a glider:

```
n = 20
c = np.zeros((n, n), dtype=bool)
glider = np.array(
 [[0,0,1],
  [1,0,1],
  [0,1,1]])
m0,m1 = glider.shape
c[:m0,:m1] = glider
ex26_gameoflife(c, 50)
```

Then, once you have the function, you can experiment with more complex patterns. This one is called a Gosper glider gun:

```
n = 50
c = np.zeros((n, n), dtype=bool)
gun = np.array(
  [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0],
   [0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0],
   [0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
   [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]
  )
m0,m1 = gun.shape
c[:m0,:m1] = gun
ex26_gameoflife(c, 100)
```

One can also experiment with things like this:

```
c = np.random.rand(100, 100) < 0.2
ex26_gameoflife(c, 100)
```

This is kind of fun too, and gets better with larger and larger sizes (also, can you tell what the first three lines do?):

```
n = 200
c = np.zeros((n, n), dtype=bool)
c.ravel()[::n+1] = 1
c.ravel()[n-1::n-1] = 1
ex26_gameoflife(c, 1000)
```

Variation:

```
n = 200
c = np.zeros((n, n), dtype=bool)
c.ravel()[n::n+1] = 1
```

```
c.ravel()[n-2:n**2-2*n+1:n-1] = 1
c.ravel()[1::n+1] = 1
c.ravel()[2*n-1::n-1] = 1
plt.clf()
plt.imshow(c)
ex26_gameoflife(c, 1000)
```