`NumPy`: Using loops and logical operators with `arrays`

Learning outcomes:

- Advanced usage of the `NumPy arrays`
- Advanced usage of `for` and `while` loops
- Apply loops and logical operators on `numpy arrays`

In previous tutorials we have learned about `for` and `while` loops using simple Python datatypes such as `lists`.

Hereafter we will combine the use of loops with `NumPy` arrays. This will give us the opportunity to explore more advanced uses of `NumPy arrays`, how to create 2-dimensional arrays, index into them and combine them using methods.

Reminder. A method is a property of an object in python. Methods are invoked (the technical term for called, or simply used) as follows `<object>.<method>` for example `numpyarray1.append()`. We will learn later what `.append()` can do. For the moment, it is helpful to remember how to call a method: `functionName.methodName`.

More thinkering with `arrays`

In [1]:
```
import numpy as np
```

As introduced before, `numpy arrays` are multidimensional objects. This means that they can have one dimension:

In [2]:
```
oneDarray = np.array([1,2,3])
print(oneDarray)
```

```
[1 2 3]
```

We can investigate the shape of the array using the method `.shape`:

In [3]:
```
oneDarray.shape
```

Out[3]:
```
(3,)
```

Ok, that means that the array has one dimension (the first, as the second after the come is empty) and the size of the dimension is `3`.

In [4]:
```
Array can also have more dimensions, say 2:
```

```
  Input In [4]
    Array can also have more dimensions, say 2:
          ^
SyntaxError: invalid syntax
```

In [5]:
```
twoDarray = np.array([[1, 2, 3], [4, 5, 6]])
print(twoDarray)
```

```
[[1 2 3]
 [4 5 6]]
```

In [6]:
```
twoDarray.shape
```

Out[6]:
```
(2, 3)
```

Here we see that the first dimension is of size `2` and the second of size `3`. That is because the outer square brakets contain only two elements, two sets of square brakets, something like `[ [] , [] ]`. The outer square brakets are the first dimension. The inner squaure brakets are the second dimension.

What happens if the second dimension has a different number of elements? Let's try:

In [7]:
```
twoDarray = np.array([[1, 2, 3], [4, 5, 6, 7]])
print(twoDarray)
```

```
[list([1, 2, 3]) list([4, 5, 6, 7])]
/var/folders/yq/3rc62cqs3nn_n_c8mm6k56jw0000gn/T/ipykernel_871/2137918845.py:1: VisibleDeprecationWarnin
g: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndar
rays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=o
bject' when creating the ndarray.
  twoDarray = np.array([[1, 2, 3], [4, 5, 6, 7]])
```

OK What happened there? Why we cannot have a numpy array with elements of different size? Well as you can see the `print()` operation returns that the content of the array are lists. The array can only contain lists of equal size. The array creation method about (making lists and stiking thme into arrays) is suboptimal.

Indeed, we can try to multiple the array by `2` and the output is not what we wanted.

In [8]:
```
twoDarray*2
```

Out[8]:
```
array([list([1, 2, 3, 1, 2, 3]), list([4, 5, 6, 7, 4, 5, 6, 7])],
      dtype=object)
```

We wish that by multiplying the array we would double the values in the array, instead, the operations duplicated the numbers (i.e., [1, 2, 3] became [1, 2, 3, 1, 2, 3] and [4, 5, 6, 7] became [4, 5, 6, 7, 4, 5, 6, 7]).

Numpy provides better ways to create arrays.

Complete the following exercise.

- Create a 2-D array with 3 numbers in each dimension and then multiply the array by `2`. Explain what is happens.

  [*It now works. Each array is successfully multiply by 2.*]

In [9]:
```
twoDarray2 = np.array([[1, 2, 3], [4, 5, 6]])
twoDarray2*2
```

Out[9]:
```
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

General methods to create `numpy` arrays

Oftentimes, as we go about doing our nice data sciency stuff, we will need to create arrays of a certain dimension. For this reasons numpy provides a few methods to initialize arrays, create empty arrays of the proper size. These arrays can later on be filled up with data.

The functions `np.zeros()` and `np.ones()` allow creating arrays with the proper dimensions. Whereas the first initializes an array with, ahem, zeros inside, the second, well you know the drill... Let's see how to use them:

In [10]:
```
AnArrayOfZeros = np.zeros(2)
print(AnArrayOfZeros)
```

```
[0. 0.]
```

The above created a 1-D array of size 2.

In [11]:
```
AnotherArrayOfZeros = np.zeros((2,2))
print(AnotherArrayOfZeros)
```

```
[[0. 0.]
 [0. 0.]]
```

The above created a 2-D array of size 2 in size 1 and size 2 in size 2. Let's create one with 3 elements in the first dimension and 4 in the second dimension:

In [12]:
```
myArray = np.zeros((3,4))
print(myArray)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

We can also use `np.ones()` in a similar way, let's create an array with 2 elements in the first dimension and 5 in the second:

In [13]:
```
myArray = np.ones((2,5))
print(myArray)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

The arrays come with a series of methoss, built-in! For example, number of dimensions can be returned by the array using the method `.ndim`:

In [14]:
```
myArray.ndim
```

Out[14]:
```
2
```

The shape of the array can be returned as follows:

In [15]:
```
myArray.shape
```

Out[15]:
```
(2, 5)
```

The totla number of the elements can be returned using the method `.size`:

In [16]:
```
myArray.size
```

Out[16]:
```
10
```

Complete the following exercise.

- List the number of methods of a the numpy array myArray and return the list of number here: [7 Methods]

- Create a 3-D array and show the use of two methods (excluding the methods shown above).

[Report your results below, use `print()` when needed to show the results of the operation of the method]

In [20]:
```
Myarray3_1 = np.array([[1, 2, 3], [4, 5, 6], [7,8,9]])
print(Myarray3_1)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In [23]:
```
Myarray3_2 = np.random.randn(3,2,5)
print(Myarray3_2)
```

```
[[[ 2.28417442  0.40458509  1.28552375  0.03691788 -1.20157218]
  [ 0.05651258 -1.12072256  0.81383909  1.27186134  0.259727  ]]

 [[ 1.82529772  0.28570376  0.12663984  0.01478739  0.78760873]
  [-0.81613038  1.31800757 -0.05575175  0.82523281 -0.76297369]]

 [[-0.05866506 -2.76815037  1.08095871  0.08124658 -0.44909124]
  [ 0.11441332  0.35229233  0.10158182 -1.27196319  1.1339563 ]]]
```

`for` loops on arrays

We have learned before about for loops. We have learned about arrays. Here we will practice with using for loops to perform operations on numpy arrays.

Now on we will use a handy method to create arrays, we will:

- Define the dimensions an size of elements
- Create a set of dimensions (remember sets use `()`)
- Use the set in combination with the function `.zeros()` to create our array.

Now on, we will use this method for initializing arrays several time sin the future.

In [24]:
```
nRows, nCols = 10, 5   # Python let's us do this!
myArraySize = (nRows, nCols)  # we'll make a 10x5 array. Rows always come first!
anArray = np.zeros(myArraySize)
print(anArray)
```

```
[[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.]]
```

Complete the following exercise.

- Create an 3-D array called `threeDarray` filled with `2`'s with max four elements in each dimension:

  [Show your code in the next cell]

In [36]:
```
nRows, nCols, nOther = 3,2,2
myArraySize2 = (nRows, nCols, nOther)
threeDarray1 = np.ones(myArraySize2)
threeDarray = threeDarray1 * 2
print(threeDarray)
```

```
[[[2. 2.]
  [2. 2.]]

 [[2. 2.]
  [2. 2.]]

 [[2. 2.]
  [2. 2.]]]
```

Let's think how we can start filling up arrays with data.

One great thing about `for` loops is that we can use them to go through the rows or columns of an array (or both!) in turn, repeating some operation on each one. Let's say we need to put the numbers of the binary sequence (2, 4, 8, 16...) in the columns of a 10x5 array for some future simulation.

HEREHRHEH

We could do that this way:

In [37]:
```
anArray[:,0] = 2
anArray[:,1] = 4
anArray[:,2] = 8
anArray[:,3] = 16
anArray[:,4] = 32

anArray
```

Out[37]:
```
array([[ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.]])
```

That works, no doubt. But

1. there's a lot of "hand coding", which is prone to mistakes
2. it would be a pain to scale up to huge arrays (as we already know)
3. it's ugly

Now let's do this a cleaner and much more scalable way using a `for` loop.

In [38]:
```
nRows, nCols = 10, 5   # make variables for length and width of our array
myArraySize = (nRows, nCols)    # we'll make a 10x5 array. Rows always come first!
ourNumbers = [2, 4, 8, 16, 32]  # numbers that we'll set each column to
anArray = np.zeros(myArraySize) # make an array to hold our numbers

for i in range(nCols) :
    anArray[:,i] = ourNumbers[i]

anArray
```

Out[38]:
```
array([[ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.]])
```

And we get the same result.

So we've swapped this:

```
anArray[:,0] = 2
anArray[:,1] = 4
anArray[:,2] = 8
anArray[:,3] = 16
anArray[:,4] = 32
```

(Yuk.)

for this:

```
for i in range(nCols) :
    anArray[:,i] = ourNumbers[i]
```

(Nice.)

which is already a huge improvement. But imagine if we were working with a 1000 or 10,000 element array! Doing it the first way – well – you can imagine. But doing it the second way, all we would have to do is change `nCols` and be a bit clever and compute `ourNumbers` automatically.

Wait, what? How would we compute the binary sequence – the powers of 2 – automatically?

With a `for` loop of course! Let's do that!

In [39]:
```
ourNumbers = list()                 # Make an empty Python list
for i in range(nCols) :
    thisNumber = 2**(i+1)           # compute 2 to the right power
    ourNumbers.append(thisNumber)   # and append it to our list
ourNumbers
```

Out[39]:
```
[2, 4, 8, 16, 32]
```

Complete the following exercise.

- Above we have used `range()`, was this the first time the function was used? [~~Yes~~/**No**]
- What does the function do?

  [For every number in the empty list within the rnage of nCols. Those number that being add by (`i+1`) is multiply by 2 (`2**`).]

- Rewrite the above code avoiding defining (using) the `thisNumber` variable (so there should only be one line inside the `for` loop).

  [Use the cell below to show your code]

In [40]:
```
ourNumbers2 = list()
for i in range(nCols) :
    ourNumbers2.append(2**(i+1))
ourNumbers2
```

Out[40]:
```
[2, 4, 8, 16, 32]
```

Okay, now we can write our code to populate the numpy array in a way that is completely scalable (this word means that the operations can work well for small as well as big datasets) using a single `for` loop:

In [41]:
```
nRows, nCols = 10, 5    # Python let's us do this!
myArraySize = (nRows, nCols)  # we'll make a 10x5 array. Rows always come first!
anArray = np.zeros(myArraySize)

for i in range(nCols) :
    anArray[:,i] = 2**(i+1)

anArray
```

Out[41]:
```
array([[ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.],
       [ 2.,  4.,  8., 16., 32.]])
```

Notice that, now, the ***only*** thing we need to change to compute and add more or fewer powers of 2 to our array is a single value – nCols in this case – *everything else is done automatically!*

Complete the following exercise.

- Write code (using a `for` loop of course) to compute the cube of the odd numbers from 1 to 9. (Remember that `range()` can take a step argument.)

  [Use the cell below to show your code]

In [46]:
```
for i in range(1,10,2):
    root = i**3
    print('The cube of ', i, ' is ', root,'(',i,"*",i,'*',i,')')
```

```
The cube of  1  is  1 ( 1 * 1 * 1 )
The cube of  3  is  27 ( 3 * 3 * 3 )
The cube of  5  is  125 ( 5 * 5 * 5 )
The cube of  7  is  343 ( 7 * 7 * 7 )
The cube of  9  is  729 ( 9 * 9 * 9 )
```

Write scalable code to compute the first "n" numbers of the Fibonacci sequence. The Fibonacci sequence (named for the famous 13th century mathematician) starts with the numbers 0 and 1, and each number after that is the sum of the previous two numbers. (Galileo, da Vinci, and Franco aren't the only famous Italian scientists/mathematicians!).

In [6]:
```
a = 0
b = 1

print("My Fibonacci sequence:")

print(a)
print(b)

# The stop is at 15.
for i in range(2,15):
    c = a + b
    a  = b
    b = c
    print(c)

print("The End!")
```

```
My Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
The End!
```

Nested for loops

A great thing about `for` loops is that they can be *nested* inside one another. This is best illustrated by example, so let's look at one and disect it.

In [57]:
```
nRows, nCols = 4, 3            # (easily changeble) array height and width
myArraySize = (nRows, nCols)    # handy list of the size
anArray = np.zeros(myArraySize) # make the array

for i in range(nRows) :
    for j in range(nCols) :
        anArray[i,j] = i + j*nRows
        print('Hi! I\'m in row ', j, ' and column', i, '!')

anArray
```

```
Hi! I'm in row  0  and column 0 !
Hi! I'm in row  1  and column 0 !
Hi! I'm in row  2  and column 0 !
Hi! I'm in row  0  and column 1 !
Hi! I'm in row  1  and column 1 !
Hi! I'm in row  2  and column 1 !
Hi! I'm in row  0  and column 2 !
Hi! I'm in row  1  and column 2 !
Hi! I'm in row  2  and column 2 !
Hi! I'm in row  0  and column 3 !
Hi! I'm in row  1  and column 3 !
Hi! I'm in row  2  and column 3 !
```

Out[57]:
```
array([[ 0.,  4.,  8.],
       [ 1.,  5.,  9.],
       [ 2.,  6., 10.],
       [ 3.,  7., 11.]])
```

So what's happening? In the first or "outer" loop, `for i in range(nRows) :` we're going to step through the numbers 0 to three, corresponding to the row indexes.

At each value of `i`, the entire second or "inner" loop, `for j in range(nCols) :` is going to run, stepping through each value of `j`, corresponding to the column indexes.

At each value of `j`, we stick a number in the `[i, j]` cell (`anArray[i,j] = i + j*nRows`), print a little message, and move on the the next value of `j`.

Once the inner loop is complete, we jump out into the outer loop, increment `i` by 1, and then jump back into the inner loop and do the whole thing again! After i has run its course from 0 to `nRows`, we say farewell to that loop and go on our way!

Complete the following exercise.

- Change the above loop so that it numbers the cells from from left-to-right, top-to-bottom. Resist the temptation to cut and paste and write your code from scratch!

  [Use the cell below to show your code]

In [2]:
```
nRows, nCols = 3, 4
myArraySize = (nRows, nCols)
anArray = np.zeros(myArraySize)

for i in range(nRows) :
    for j in range(nCols) :
        anArray[i,j] = i + j*nRows
        print('Hi! I\'m in row ', j, ' and column', i, '!')

anArray
```

```
Hi! I'm in row  0  and column 0 !
Hi! I'm in row  1  and column 0 !
Hi! I'm in row  2  and column 0 !
Hi! I'm in row  3  and column 0 !
Hi! I'm in row  0  and column 1 !
Hi! I'm in row  1  and column 1 !
Hi! I'm in row  2  and column 1 !
Hi! I'm in row  3  and column 1 !
Hi! I'm in row  0  and column 2 !
Hi! I'm in row  1  and column 2 !
Hi! I'm in row  2  and column 2 !
Hi! I'm in row  3  and column 2 !
```

Out[2]:
```
array([[ 0.,  3.,  6.,  9.],
       [ 1.,  4.,  7., 10.],
       [ 2.,  5.,  8., 11.]])
```

The loop you just wrote numbers the cells of your array in "row-major" order, or "row wise", while the original loop numbered the cells in "column-major" order, or "column wise".

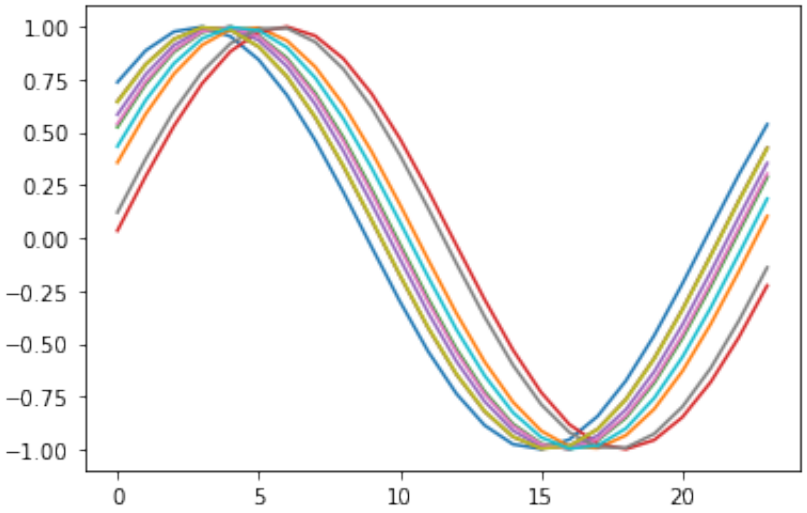
Nested loops give you tremendous power! You go through any array element-by-element and get or set individual values. You can even do things like loading a series of data files in turn (in an outer loop), and then chewing through each data file in an inner loop.

As a final example, let's say we want to simulate a diurnal rhythm, like the cortisol level the body for several people. Since differnt people have different schedules, we want to add a bit of randomness to when each persons cortisole level waxes and wanes.

In [59]:
```
hours, person = 24, 10              # (easily changeble) array height and width
myArraySize = (hours, person)       # handy list of the size
cortLevel = np.zeros(myArraySize) # make the array

myFreq = 2*np.pi/hours              # make the frequency once per 24 hrs

for j in range(person) :              # we'll go person by person
    myPhase = np.random.rand(1, 1)  # get a random phase for this person
    for i in range(hours) :           # go down current column (person) row-by-row
        cortLevel[i,j] = np.sin((myFreq*i + myPhase)) # set val. for this [time, person]
```

(Here we will go ahead of us and use `matplotlib` one of the major python libraries for data visualization. We will dedicate a tutorial on this library later on.)

In [60]:
```
import matplotlib.pyplot as plt

plt.plot(cortLevel);
```



Cool!

Complete the following exercise.

- Describe what the plot above shows. To do so, first describe the numbers in the `y` axis then those in the `x` axis, finally describe the function and the colors, what does this all mean?

  [The plot is interesting. It appears that the plot will move to the opposite slope once they reach certain threshold. The estimate threshold is that the value should stay in the range of (1,-1). Then,the x-axis will move forward regardless of the y-axis. The shape of the graph is resulted from `np.sin` which will create the graph to correlate with the shape of `sin` function. Also, I believe that the color represents number of a person. There are 8 colors. Thus, there should be 8 people in the data.]

`While` loops

Sometimes we wish to repeat a calculation (or something), not for a predetermined number of times like in a `for` loop, but until some critereon is reached. This is accomplished using a `while` loops, which just keeps running and running until a critereon is reached. One dangerous thing about a `while` loop is that if the criteon can't be reached because we made a mistake in our code, then the loop runs forever – an infinite loop!

As a simple example, let's see how many tries it takes to get a number from the standard normal distribution that is above 2 - the upper 2.5% tail of the distribution!

In [69]:
```
x, cutOff, myCounter = 0, 2, 0

while x < cutOff :
    x = np.random.randn()
    myCounter += 1

myCounter
```

Out[69]:
```
18
```

The dissection of the code is as follows.

- the first line sets some useful variables
  - a "test" variable `x` that will contain our candidate random numbers
  - our "cut off" variable that we will test x against
  - a "counter" variable that we'll use to count the number of tries
- the `while x < cutOff :` says "keep trying *while* `x` is less than `cutOff`
- `x = np.random.randn()` gets a random number and assignes it to `x`
- `myCounter +=` increments our counter

Once we get a random number above 2, the `x < 2` returns `False` and the loop ends. Whatever value is then in `myCounter` is our answer!

Run the above code cell several times! Does it always take the same number of times? Based on what you know about the standard normal distribution, how many times should it take?

Now here's an interesting puzzle... How many times does it take to get a big random number on average? What does the distribution look like?

How would we answer those questions?

Let's use...

a ***for loop!***

(Here we are going to go ahead of us and use two the python library `seaborn`, we will learn more about this library in later tutorials.)

In [62]:
```
import seaborn as sns  # for making a histogram/kde
```

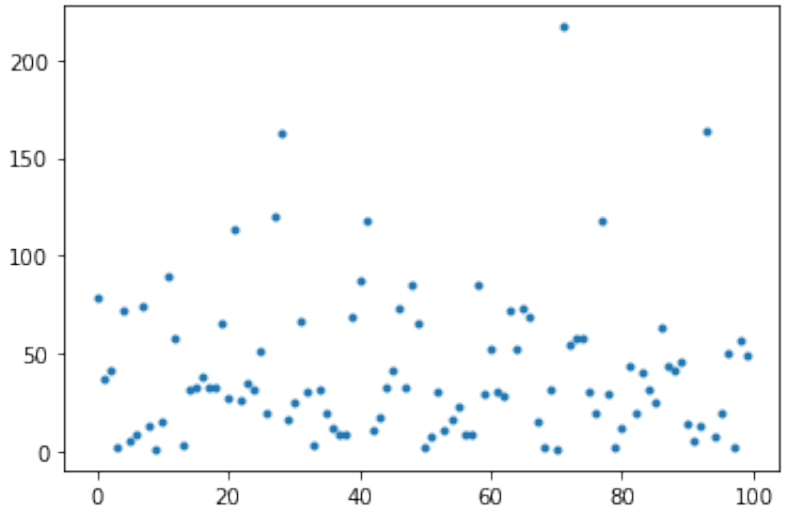
In [75]:
```
nExperiments = 100  # how many times we'll do our little experiment
nSamplesNeeded = np.zeros((nExperiments, 1))
x, cutOff, = 0, 2

for i in range(nExperiments) :
    myCounter = 0
    x = 0
    while x < cutOff :
        x = np.random.randn()
        myCounter += 1
    nSamplesNeeded[i, 0] = myCounter
```

That looks like a lot of code, but go through it carefully. All we have done is nest our `while` loop inside a `for` loop, so that we can do our "How many times?" experiment as often as we wish. On each pass through the `for` loop, we store the answer from a single experiment in the `i` th row of a numpy array!

Let's look at the number of tries it took on each experiment:

In [76]:
```
plt.plot(nSamplesNeeded, '.')
```

Out[76]:
```
[<matplotlib.lines.Line2D at 0x7fc9b8d1e3a0>]
```



Complete the following exercise.

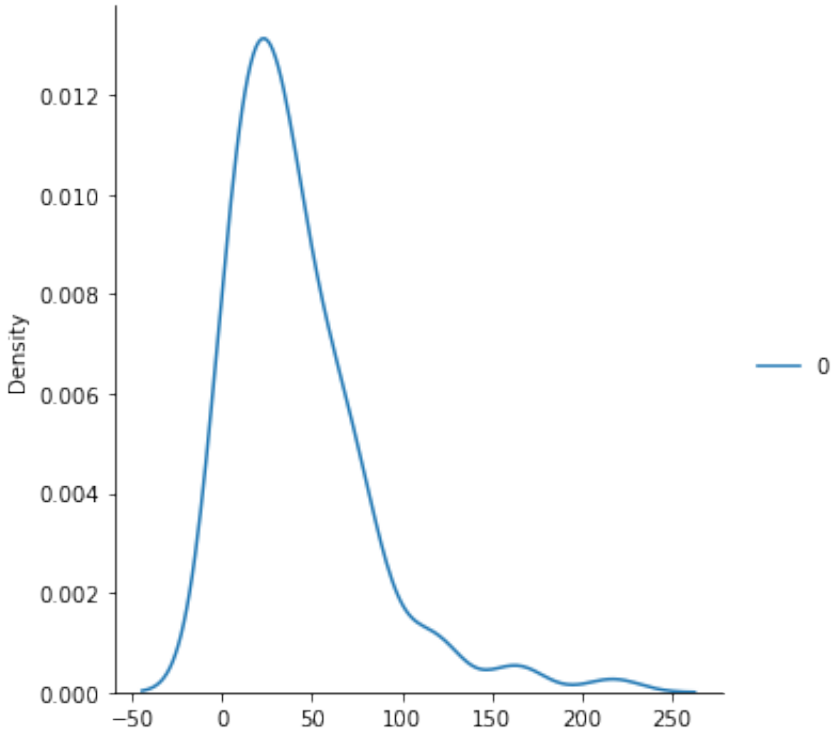
- Describe what the plot above shows. To do so, first describe the numbers in the `y` axis then those in the `x` axis, finally describe the function and the colors, what does this all mean?

  [The number on y-axis is ranged from 0 to 200, which result from the `for` loop function. Meanwhile, the number on x-axis is ranged from 0 to 100. The graph appears to have the shape similar to normal distribution where the mean would fall around 50. The function is to make random distribution graph where n = 100. And, the color was randomly selected by `plt.plot` function.]

Okay, cool! So it looks like we usually get a "big" number in under 50 tries, but it occasionally takes a lot longer. Let's look at the distribution of these numbers!

In [77]:
```
sns.displot(nSamplesNeeded, kind='kde')
```

Out[77]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc9b8ab3340>
```



Okay, I think that, while pretty, this plot is misleading. Can you see why?

Complete the following exercise.

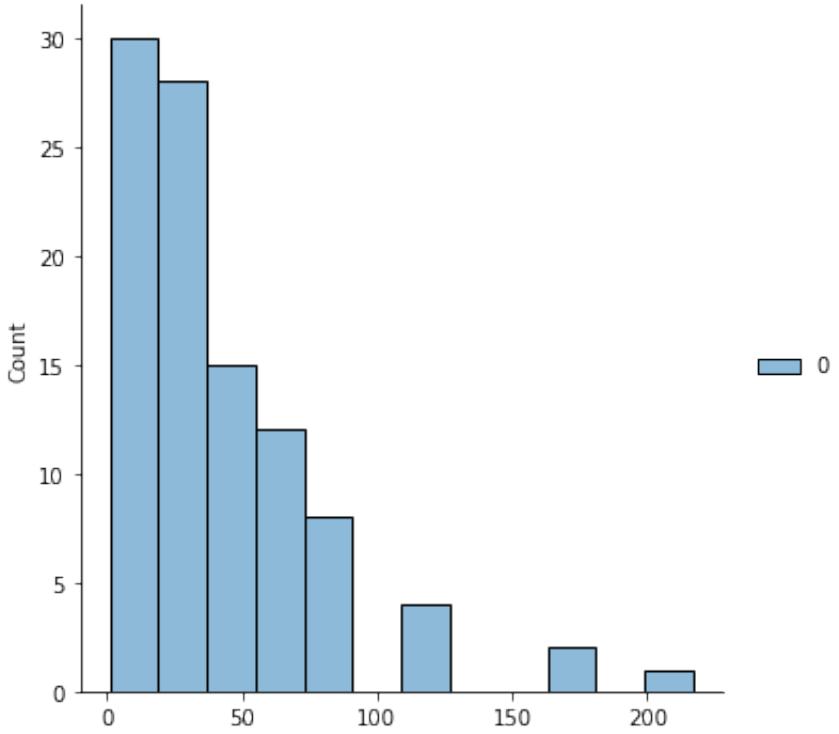
- Describe what the plot above shows. To do so, first describe the numbers in the `y` axis then those in the `x` axis, finally describe the function and the colors, what does this all mean?

  [`sns.displot` with `kind='dge'` represents the distribution of a numeric variable, a smooth version of histogram. Thus, the y-axis represents the probability density function for the kernel density estimation regarding x values. Meanwhile, the x-axis represents individual value ranged from -50 to 250. Blue color/line is representing the connectd data point for the whole graph.]

Let's do a plain old histogram.

In [78]:
```
sns.displot(nSamplesNeeded, kind='hist')
```

Out[78]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc9b8aab790>
```



Now this make more sense, because we can't have a negative number of tries!

Complete the following exercise.

- Describe what the plot above shows. To do so, first describe the numbers in the `y` axis then those in the `x` axis, finally describe the values in the plot, what does this all mean?

  [The y-axis represents the counts amounts of x values. For instance, at x = 0, there is 30 counts/data points in total. Meanwhile, x is the generated value ranged of `nSamplesNeeded` from 0 - 200. `sns.displot` with `kind='hist'` is the function to create gram in histrogram shape. And, again, the blue color is the default color of this function.]

So it looks like, on average, it took us about – what? – 40 tries to get a number in the upper 2 1/2% tail of the distribution. Let's do a quick calculation.

In [79]:
```
100 / 2.5
```

Out[79]:
```
40.0
```

Logic operators on NumPy arrays

There are other types of operators that do not come standard with Python but that are part of other packages and need to be imported. These operators behave differently.

When dealing with arrays, instead of individual numbers, things look slightly different. For example, if we wanted to perform a logical operation between two sets of numbers, e.g., two arrays, operatiors (`=` , `>` , etc) will work sometimes but not others.

Let's take a look at how we would perform comparisons and logical operations with NumPy arrays.

In [80]:
```
myRnds = np.random.randn(1, 5) # we create an array of random numbers
myRnds
```

Out[80]:
```
array([[-0.37024648, -1.15462129, -1.25226976,  2.43427118, -1.50724999]])
```

Now, imagine we wanted to know whether each number stored in the Array `myRnds` is positive.

In [81]:
```
myRnds > 0
```

Out[81]:
```
array([[False, False, False,  True, False]])
```

If we wanted to find out whether any of the numbers in an array are positive, we would use the numpy array method `any` :

In [82]:
```
logical_array = (myRnds > 0)
np.any(logical_array)
```

Out[82]:
```
True
```

If we wanted to test whether all the values in an array are positive, we would use the method `all`.

In [83]:
```
np.all(logical_array)
```

Out[83]:
```
False
```

Because both `all` and `any` apply to numpy atrays, they can also be called as methods of a NumPy Arrays. For example:

In [84]:
```
logical_array.any()
```

Out[84]:
```
True
```

In [85]:
```
logical_array.all()
```

Out[85]:
```
False
```

Numpy arrays also allow comparing values element-wise. This means that we could compare each element of one array with the corresponding element of another array. If the twovectors have the same size.

`[1, 2, 3] = [1, 4, 3]`

Would compare 1 to 1, 2 to 4 and 3 to 3.

In [86]:
```
array_one = np.random.randn(1,5) > 0;
array_two = np.random.randn(1,5) > 0;
np.logical_and(array_one, array_two)
```

Out[86]:
```
array([[ True, False, False,  True, False]])
```

What happens if the two arrays have different size, though?

In [87]:
```
vector_one = np.random.randn(1,6) > 0;
vector_two = np.random.randn(1,5) > 0;
np.logical_and(vector_one, vector_two)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [87], in <cell line: 3>()
      1 vector_one = np.random.randn(1,6) > 0;
      2 vector_two = np.random.randn(1,5) > 0;
----> 3 np.logical_and(vector_one, vector_two)

ValueError: operands could not be broadcast together with shapes (1,6) (1,5)
```

The not and or operators also exist for numpy arrays:

In [88]:
```
vector_one = np.random.randn(1,5) > 0;
vector_two = np.random.randn(1,5) > 0;
np.logical_or(vector_one, vector_two)
```

Out[88]:
```
array([[False,  True,  True,  True,  True]])
```

In [89]:
```
vector_one = np.random.randn(1,5) > 0;
vector_two = np.random.randn(1,5) > 0;
np.logical_not(vector_one, vector_two)
```

Out[89]:
```
array([[False, False,  True, False,  True]])
```

Complete the following exercise.

- Create an array of uniform random numbers with dimensions rows=1 and columns=8.

  [Use the cell below to write your code]

In [93]:
```
np.random.randn(1,8)
```

Out[93]:
```
array([[-0.31274946,  0.58765035,  1.50123305,  0.235506  ,  1.16522398,
        -0.97469564, -1.38938876,  1.4328365 ]])
```

- how many numbers were generated with a value larger than 0.5?

  [3]

In [94]:
```
np.random.randn(1,8) > 0.5
```

Out[94]:
```
array([[ True, False, False,  True, False, False, False,  True]])
```

So in this tutorial we have shown how to organize and manipulate data using Python `numpy` `arrays`.

We have also worked through the very important element of code called ***loops*** and how they apply to NumPy Arrays.

The most frequently used loop is the `for` loop, which allows us to do a computation a number of times. It can be used to do things like crawl through the rows and columns of a numpy array. With a pair of nested `for` loops, we can even crawl through each cell of an array in either row-major or column-major order. We could even use a `for` loop to chew through a series of data files, etc.

A `while` loop is used when we don't know ahead of time how many times we'll need to do the calculation. The while loop allows us to compute or look at thing as many times as necessary until some condition as met. We just have to be careful that we don't make a dreaded *infinite* loop (what sort of cut off would make the `while` loop above essentially infinite?).