Chapter 5

Indexing Lists and Arrays

5.1 NumPy and Matplotlib

The NumPy and Matplotlib libraries are frequently used in scientific computing. In the example programs below (and in future chapters) these import commands are not shown. We will assume that, if needed, NumPy has been imported as `np` and `matplotlib.pyplot` has been imported as `plt`.

5.2 Lists and arrays

In the last chapter we learned how Python lists and NumPy arrays behave differently.

Exercise 5.2

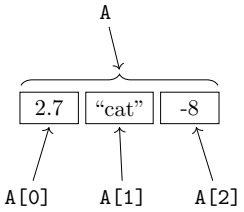
Create the following lists and arrays:

- `listone = [4,2,5,3]`
- `listtwo = [3,7,9,1]`
- `arrayone = np.array([4,2,5,3])`
- `arraytwo = np.array([3,7,9,1])`

Compute `listone + listtwo` and `arrayone + arraytwo`. What is the difference? Compute `2*listone` and `2*arrayone`. What is the difference? What is the result of `listone*listtwo`? `arrayone*arraytwo`? `listone` + `arrayone`?

5.3 Indexing

Given a list or array `A` we can reference an individual element by writing `A[i]`, where the index `i` is an integer. The first element in the list or array has index `0`, the second element has index `1`, and so on. As an example, consider the list `A = [2.7, "cat", -8]`. Obviously `A` has three elements, and we say that the "length" of `A` is three. The individual elements are `A[0] = 2.7`, `A[1] = "cat"` and `A[2] = -8`.



Exercise 5.3a

Consider these lists and arrays:

```
A = [2, 4, 6, 8, 10, 12]
B = np.linspace(0.1, 5.1, 6)
```

- How many elements are in `A`? In `B`?
- What is `A[0]`? `B[0]`?
- What is `A[2]`? `B[2]`?
- What is `A[5]`? `B[5]`?
- What is `A[6]`? `B[6]`?

These examples illustrate a very important aspect of Python's behavior: *Indices for lists and arrays begin with* 0. If there are N elements in a list or array called `A`, the first element is `A[0]` and the last element is `A[N-1]`.

Exercise 5.3b

Run this code:

```
C = np.linspace(2,4,5)
print(C)
C[3] = 17.0
```

```
print(C)
```

What's happening?

In the first line, `linspace()` creates the array `[2.0 2.5 3.0 3.5 4.0]` and assigns it to the variable `C`. In the statement `C[3] = 17.0`, the *fourth* array element is changed from its previous value, `3.5`, to the new value `17.0`. The array becomes `[2.0 2.5 3.0 17.0 4.0]`.

5.4 Accessing elements

You can use variables as indices to access the elements of a list or array. Consider the following code:

```
A = np.arange(6,15,1)             # create an array A
print("array length = ", len(A))  # print the length of A

i = 2                             # choose an index value
print("element ", i, "is ", A[i]) # print element i
```

This code uses the function `len()`, which gives the length of a list or array (the number of elements in the list or array).

Exercise 5.4a

Run the code above. Do you understand the output? What happens when you try to access an array element that is greater than or equal to the length of the array?

Another example of accessing array elements:

```
A = np.array([6.1, -15.3, -3.7, 7.4, 20.4, 11.8, -6.5])
N = len(A)
for i in range(N):
    if A[i] < 0:
        A[i] = -A[i]
print(A)
```

This code takes the array `A` and reverses the sign of every element that is negative.

- The first line creates `A`.
- The second line defines `N` as the length of `A`.

- In the third line, the command `range(N)` creates a list of integers beginning with `0` and ending with `N-1`.
- The `for` loop assigns `i` to each element of `range(N)`, one at a time, and executes the indented lines below.
- If the array element `A[i]` is negative, the `if` statement evaluates to `True`. The indented code below the `if` statement reverses the sign of `A[i]`.
- The print function prints the elements of the array with negative elements switched to positive.

Notice that we are using `N` rather than `7` inside the `range` command. This helps avoid errors, in particular if we decide to change the number of elements in `A`.

Exercise 5.4b

Write a code that will create a NumPy array consisting of numbers $1, 2, \ldots, 100$. Have your code search through the array using a `for` loop to cycle through the index values. Print the value of every array element that is divisible by 3, but not divisible by 2.

5.5 Using arrays

Let's create an array `A` of real numbers from 0 through 10, spaced by 0.5, then construct a new array containing the square roots of those numbers.

```
A = np.linspace(0,10,21)      # Create a real number array
roots = np.zeros(len(A))      # Create array for square roots

for i in range(len(A)):       # Loop over array indices
    roots[i] = np.sqrt(A[i])  # Compute the square root

print("roots =", roots)       # Print the answer
```

Make sure you understand each line of this code.

- Why is the last argument of `linspace()` equal to 21?
- What values does `i` take in the `for` loop?
- Why did we include the command `roots = np.zeros(len(A))` in the second line?

Take a close look at this last question. The command `roots = np.zeros(len(A))` creates the array `roots` and fills it with 0's. We could

create the array using a different NumPy function; for example, we could use `ones()` instead of `zeros()`. The initial values of the array don't matter since those values are overwritten in the `for` loop. Note, however, that the `roots` array should have the same length as `A`.

The real question is this: Why do we need to create the `roots` array at all?

Exercise 5.5a

Run the code above. Eliminate (or comment out) the line `roots = np.zeros(len(A))`, and run the code again. (If you are using a Jupyter notebook, you need to restart the kernel before running the code. This causes Python to forget that `A` is a NumPy array.) What sort of error message do you get?

Without the command `roots = np.zeros(len(A))`, the code doesn't run. Here's why: Without that line, the first appearance of the variable `roots` is in the statement `roots[i] = np.sqrt(A[i])`. Python doesn't know how to interpret `roots[i]` because Python doesn't know that `roots` is an array. It might seem obvious that `roots` is either a list or an array, since it carries an index. But Python isn't able to allocate memory for `roots` since it can't guess what type of object it is, or how many elements it should contain.

So we need to tell Python that `roots` is an array of length `len(A)` before we can assign values to the elements `roots[i]`. We do this with the statement `roots = np.zeros(len(A))`.

Another way to produce an array of square roots is to use the approach discussed in Sec. 4.2:

```
A = np.linspace(0,10,21)    # Create a real number array
roots = np.sqrt(A)          # Compute square roots
print("roots =",roots)      # Print the answer
```

This code is more efficient than the previous code, and more simple. In particular there is no need to create the `roots` array using `zeros()` or some other NumPy function. This is because Python already knows that `A` is a NumPy array. When Python sees the statement `roots = np.sqrt(A)`, it can infer that `roots` is an array of length `len(A)`.

Exercise 5.5b

Make an array `A = np.linspace(-10,10,25)`. Use a `for` loop and array indexing to create an array `B` whose elements are the same as `A`, but in reverse order.

Exercise 5.5c

A ball of clay is dropped from an initial height y_0 above the floor. While the ball is in the air, its height as a function of time t is

$$y(t) = y_0 - \frac{1}{2}gt^2 \ ,$$

where $g = 9.8\,\mathrm{m/s^2}$ is the acceleration due to gravity. When the ball hits the floor it sticks; its height is then $y(t) = 0.0$. Write a code to plot a graph of y versus t for $0.0 \leq t \leq 2.0\,\mathrm{s}$, using $y_0 = 2.0\,\mathrm{m}$. One way to do this: create an array for `t` using `linspace()` and an array for `y` using the formula above. Then loop through the elements of `y` and replace all negative values with 0.0.

5.6 Counters

How many numbers from N_1 through N_2 are divisible by 3, but not divisible by 2? For example, with $N_1 = 7$ and $N_2 = 28$ the numbers that are divisible by 3 but not by 2 are these: 9, 15, 21 and 27. The number of such numbers is 4.

The code below solves this problem using a *counter* variable called `ctr`:

```
N1 = 7              # choose value for N1
N2 = 28             # choose value for N2
ctr = 0             # initialize counter to 0
for n in range(N1, N2+1):
    if n%3 == 0 and n%2 != 0:
        ctr = ctr + 1
print(ctr)
```

After assigning values to N_1 and N_2, the variable `ctr` is initialized to 0. In the `for` loop `n` is assigned values from N_1 through N_2, one after the other. The `if` statement determines if `n` is divisible by 3, but not divisible by 2, and if so, increments `ctr` by 1. The variable `ctr` counts the number of times the condition `n%3 == 0 and n%2 != 0` evaluates to `True`.

Exercise 5.6

How many numbers from 1 through 10000 are divisible by 5, but not by 7?

5.7 Appending to an array

Rather than simply counting the number of numbers that are divisible by 3 but not 2, let's collect these numbers into an array `A`. Here is a somewhat unsatisfactory solution to the problem:

```
N1 = 7
N2 = 28
ctr = 0
A = np.zeros(4)     # create array A

# loop over n from N1 through N2
for n in range(N1, N2+1):
    # test to see if n is divisible by 3 but not by 2
    if n%3 == 0 and n%2 != 0:
        A[ctr] = n      # set array element to n
        ctr = ctr + 1   # increment counter
print(ctr, A)
```

This code unsatisfactory because the command `A = np.zeros(4)`, which is used to create `A`, requires prior knowledge that the array `A` should have 4 elements. What if we change N_1 and N_2? How can we tell how large the array `A` should be?

There is a better way to structure this code. Begin with an empty array, then use the NumPy `append()` function to add array elements as they are found.

```
N1 = 7
N2 = 28
A = []              # create an empty array A

# loop over n from N1 through N2
for n in range(N1, N2+1):
    # test to see if n is divisible by 3 but not by 2
    if n%3 == 0 and n%2 != 0:
        A = np.append(A,n)      # append n to A
print(A)
```

The NumPy command `append(A,n)` creates a new array with `n` appended to the end of `A`. Python then replaces the old `A` array with the new array.

Note that a counter is not needed for this code.

Exercise 5.7a

Write a program that creates an array of numbers from N_1 through N_2 that are divisible either by 3 or by 5, but not both. Use the NumPy `append()` function. Add a counter to determine the number of such numbers.

Exercise 5.7b

Write a code that produces an array containing the cumulative sum of integers up to a chosen integer N. For example, if $N = 5$, the result should be `[1 3 6 10 15]` . (These are 1, $1+2$, $1+2+3$, $1+2+3+4$, $1+2+3+4+5$.)

5.8 Appending to a list

Appending an element to a list is not the same as appending an element to a NumPy array. To add an element *newelement* to the end of a list `mylist`, simply issue the command

`mylist.append(`*newelement*`)`

Exercise 5.8a

Create a list containing your favorite animals. Start with an empty list and append the elements, one–by–one.

5.9 More exercises

Exercise 5.9a

The Fibonacci sequence is defined by the recurrence relation

$$F_0 = 0 \ ,$$
$$F_1 = 1 \ ,$$
$$F_n = F_{n-1} + F_{n-2} \ ,$$

where n is an integer greater than 1. The first few terms in the sequence are $0, 1, 1, 2, 3, 5, 8, \ldots$. Write a code to compute the Fibonacci sequence up to $n = 100$ and plot $\log(F_n)$ versus n. Find the

slope of the curve near $n = 100$. According to Binet's formula, the slope should approach $\log(\varphi)$ for large n, where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio.

Exercise 5.9b

Sterling's approximation to $n!$ is

$$S(n) = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \ .$$

Create a graph showing the percent relative error $|1 - S(n)/n!| \cdot 100$ in Sterling's approximation for positive integers n.