Chapter 6

Data Types and Variable Assignment

6.1 Python data types

Python defines various data types, including `int` (integers), `float` (real numbers, also known as *floats*), `complex` (complex numbers), `list` (lists), and `str` (strings). The data type of any object `O` can be found using the command `type(O)`

The distinction between integers (`int`) and real numbers (`float`) is mostly transparent in Python, but can be important in other programming languages.

Complex numbers (`complex`) are new to us. Python uses `j` for $\sqrt{-1}$. For example, a complex number with real part 2.0 and imaginary part 3.0 is denoted `2.0 + 3.0j`. Note that there is no asterisk (no multiplication sign) between `3.0` and `j`.

When we combine an integer with a real number through addition, subtraction, multiplication or division, the result is a real number. When we combine an integer or a real number with a complex number, the result is a complex number. In Sec. 2.3 we observed that for Python versions 3.0 and beyond, the division of two integers gives a real number.

Exercise 6.1a

Choose an integer `i`, a real number `r`, and a complex number `c`. Compute the sums `i+r`, `i+c` and `r+c` and check their types using the `type()` function. Do the same for multiplication and division. What type is `i/i`?

A list (`list`) is a collection of numbers or strings, such as `L = [2.2, "tree", -4]`. Elements of a list are accessed with the square bracket no-

tation. Element `L[0]` is `2.2`, element `L[1]` is `"tree"`, *etc.*

A string (`str`) is a sequence of characters surrounded by single or double quotation marks. Thus, `"cat"` is a string. The characters of a string can be accessed with the square bracket notation, just like the elements of a list. For the string `S = "cat"`, element `S[0]` is `c`, element `S[1]` is `a`, *etc.* Like lists, strings can be added together or multiplied by an integer. For example, `"cat" + "dog"` is `"catdog"` and `2*"cat"` is `"catcat"`.

Exercise 6.1b

Consider the list `L = [2.2, "tree", -4]`. Verify that `L[1]` is `tree` and `L[2]` is `-4`. What is `L[1][2]`?

Exercise 6.1c

Let `C` and `D` denote two strings of equal length. Write a code that will interleave these strings together. For example, if `C = "cat"` and `D = "dog"`, the resulting string should be `"cdaotg"`. Your code should work with other pairs of strings, such as `C = "orange"` and `D = "purple"`.

6.2 NumPy data types

The NumPy library defines more data types, including `matrix` (matrices) and `ndarray` (n–dimensional arrays). We will discuss matrices in the chapter on linear algebra. Objects of the type `ndarray` include the 1–dimensional arrays discussed previously. Those arrays are created using NumPy functions such as `linspace()`, `array()` and `zeros()`.

By default, the elements of an `ndarray` are real numbers (floats). You can override the default by specifying the data type when the array is created. For example, the command `np.zeros(10,dtype=int)` will create a NumPy array of 10 integers, initially all 0's.

Exercise 6.2a

Experiment with various data types for NumPy arrays. What is the difference between `np.zeros(10)` and `np.zeros(10,dtype=int)`? What is the difference between `np.linspace(1,4,9)` and `np.linspace(1,4,9,dtype=int)`? What happens if you create an

array of integers, then try to change one of its elements to a float?

Here is a 2–dimensional NumPy array:

```
A = np.array([[1.0,2.0,3.0],[4.0,5.0,6.0]])
```

Pay close attention to the syntax. The round parentheses are present because `array()` is a function. Inside the round parentheses we have nested sets of square brackets. The inner brackets define the 1–dimensional arrays `[1.0,2.0,3.0]` and `[4.0,5.0,6.0]`. The outer brackets define a one–dimensional array whose first element is `[1.0,2.0,3.0]` and second element is `[4.0,5.0,6.0]`. In other words, we can view `A` as an array of arrays.

The elements of an n–dimensional array can be accessed with indices. For example, the second element of `A` (the element with index `1`) is

```
A[1] = [4.0,5.0,6.0]
```

which is itself an array. The third element of the array `A[1]` (the element with index `2`) is

```
A[1][2] = 6.0
```

Another notation for `A[1][2]` is

```
A[1,2] = 6.0
```

When printed, Python displays the array `A` as

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

Thus, we can view `A` as a 2×3 matrix whose rows are `A[0]` and `A[1]`.

Exercise 6.2b

Create two 3×3 NumPy arrays `A` and `B`. Compute:

- `3*A`
- `A + B`
- `A*B`

Note that 2–dimensional NumPy arrays do *not* multiply like matrices.

An n–dimensional NumPy array can be created using `zeros()` or `ones()`. For example, `zeros((3,4))` will produce a 2–dimensional array of size 3×4, filled with zero's. Note the nested round parentheses.

Exercise 6.2c

Write a program that will ask the user to input a positive integer n, then create an $n \times n$ multiplication table. For $n = 3$, the output should appear as

```
[[1. 2. 3.]
 [2. 4. 6.]
 [3. 6. 9.]]
```

6.3 Lists of lists

A two dimensional NumPy array can be viewed as an array of arrays. Likewise, we can have a list of lists. In other words, each element of a list can be a list. For example:

```
L = [[-3.3,"cat"], [2.5, "tree"], ["dog",1.7]]
```

is a list whose three elements are `L[0] = [-3.3,"cat"]`, `L[1] = [2.5, "tree"]`, and `L[2] = ["dog",1.7]`. The two elements of `L[1]` are `L[1][0] = 2.5` and `L[1][1] = "tree"`. Unlike a 2–dimensional array, we cannot access elements of a 2–dimensional list by enclosing the indices in a single pair of square brackets. That is, we can't write `L[1,1]` in place of `L[1][1]`.

Exercise 6.3

The elements of a list can include a mixture of numbers, strings and lists of varying lengths. Modify the code above by replacing `L[2]` with a list of length 3. Also replace `L[1][0]` with a list. How would you access an element of that list?

6.4 Type conversion

We can convert integers and real numbers to strings using the `str()` function:

```
n = 17                    # choose integer
x = 23.1                  # choose real number
print(type(n), type(x))
nstring = str(n)          # convert n to a string
xstring = str(x)          # convert x to a string
print(type(nstring), type(xstring))
```

If a string consists of digits without a decimal point, we can convert it to an integer using the `int()` function. If a string consists of digits with

or without a decimal point, we can convert it to a real number using the `float()` function:

```
s1 = "4321"              # string without a decimal point
s2 = "34.87"             # string with a decimal point
print(type(s1), type(s2))
s1int = int(s1)          # convert s1 to integer
s2float = float(s2)      # convert s2 to real number
print(type(s1int), type(s2float))
```

We previously saw (Sec. 2.8) that the `input` command interprets user input as a string. If the input is to be used as a number, it must be converted using `float()` or `int()`.

Exercise 6.4a

Run the codes above, and observe the output.

How many 0's are in the number 2^{53}? To answer this question, we need to convert the number 2^{53} to a string:

```
i = 2**53                # Create the integer 2**53
i = str(i)               # Convert the integer to a string
count = 0                # Initialize counter
for n in range(len(i)):  # Loop over elements of the string
    if i[n] == "0":      # Check if string element equals 0
        count = count + 1 # Increment the counter by 1
print(count)
```

Exercise 6.4b

Find all powers of 2 through 2^{100} that do *not* contain the digit 3. (Powers of 2 are numbers of the form 2^n with n a positive integer; that is, 2, 4, 8, 16, *etc.*) How many such numbers are there? What is the largest?

6.5 Variable assignment for numbers

Consider the following scenario. A variable `x` has been assigned to some value. We now set `y = x`. What happens to `y` if we modify `x`? What happens to `x` if we modify `y`?

Exercise 6.5a

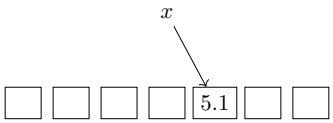
Run the program

```
x = 5.1     # assign value to x
y = x       # set y equal to x
x = 3.7     # modify x
print(x, y)
```

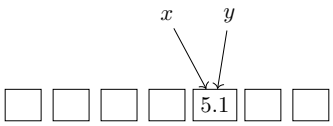
What are the final values of `x` and `y`? Are the results what you expected?

Here's how Python interprets this program. We start with a blank section of memory, depicted as follows:

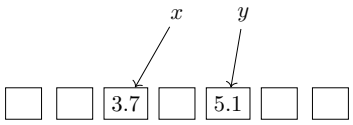
In the first line, `x = 5.1`, Python instructs the computer to write the number `5.1` into memory. The computer then creates the variable `x` that points to that memory location.



The second line is `y = x`. Python evaluates the right–hand side, which is `5.1`, then creates a new variable `y` that points to the location in memory containing `5.1`.



The third line is `x = 3.7`. Python instructs the computer to write the number `3.7` into memory and then redirect the variable `x` to that memory location.



As you can see, the final value of `x` is `3.7` and the final value of `y` is `5.1`. The reassignment of `x` from `5.1` to `3.7` does *not* change the value of `y`. The relation `y = x` no longer holds.

Exercise 6.5b

Predict the output of this program:

```
x = 5.1
y = x
y = 7.4     # modify y
print(x,y)
```

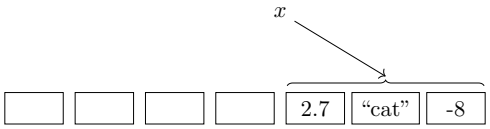
Were you correct?

6.6 Variable assignment for lists and arrays

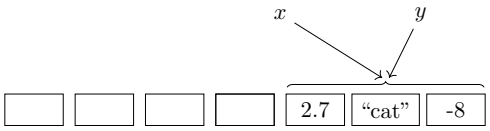
How does Python interpret `y = x` when the variable `x` is a list or array? Let's analyze the program

```
x = [2.7, "cat", -8]
y = x
x = [-3.3, 17, "cat"]
```

The first line instructs Python to write the list elements, `2.1`, `"cat"` and `-8` into memory, then create a variable `x` that points to that list.

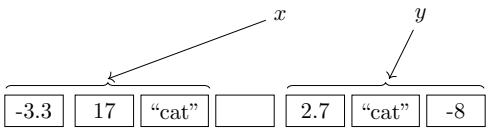


The second line is `y = x`. Python evaluates the right–hand side, which is the list `[2.7, "cat", -8]`, then creates a new variable `y` that points to the memory location containing the list.



The third line is `x = [-3.3, 17, "cat"]`. The computer writes the list elements `-3.3`, `17` and `"cat"` into memory and then redirects the variable

`x` to that memory location.

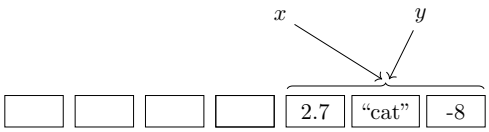


The final value of `x` is `[-3.3, 17, "cat"]` and the final value of `y` is `[2.7, "cat", -8]`. The reassignment of `x` does not change the value of `y`, so the relation `y = x` no longer holds.

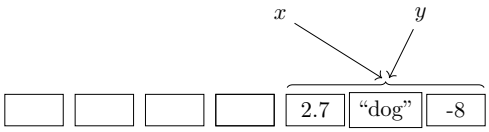
This isn't the end of the story. Consider the program

```
x = [2.7, "cat", -8]
y = x
x[1] = "dog"
```

The first line creates the list `[2.7, "cat", -8]` in memory and points `x` to that list. The second line points `y` to the same list.



The third line, `x[1] = "dog"`, tells the computer to modify the list by replacing `"cat"` with `"dog"`. The final values for *both* `x` and `y` is the list `[2.7, "dog", -8]`.



NumPy arrays behave the same way as lists. The statement `x = np.linspace(0.0,4.0,5)` places the array `[0. 1. 2. 3. 4.]` into memory and points `x` to that array. We can assign a new variable name to that same array using `y = x`. Subsequently:

- We can use the statement `x = np.linspace(10.0,20.0,11)` to redirect `x` to a new array. This does not change `y`, so `x` and `y` are now different arrays.

- We can use the statement `x[3] = 17.0` to modify the array from `[0. 1. 2. 3. 4.]` to `[0. 1. 2. 17. 4.]`. Both `x` and `y` point to the modified array, so `x` and `y` are still equal.

Exercise 6.6

Experiment with the program

```
x = np.linspace(0.0, 4.0, 5)
y = x
# modify or replace the x or y array
print(x,y)
```

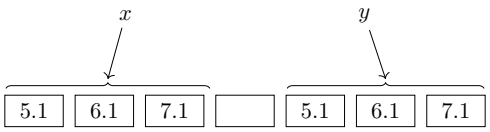
What happens if you replace every element of `x` individually? What happens if you replace one or more elements of `y`? What happens if you replace `y` with a new array?

6.7 Copying arrays and lists

If `x` is a NumPy array and you need to create another copy, you probably don't want to use `y = x`. As we saw in the last section, `y = x` simply creates a new name, `y`, for the array. Both `x` and `y` point to the same array. If you modify an individual element of `x`, this will also modify `y`. Likewise, if you modify an individual element of `y`, this will change `x`.

Rather than `y = x`, you should use the NumPy `copy()` function:

```
x = np.linspace(5.1,7.1,3)  # create an array
y = np.copy(x)              # make a copy of the array
```



Using `y = np.copy(x)` creates a new array `y` whose elements are equal to the elements of `x`. The arrays `x` and `y` are independent from one another.

Exercise 6.7a

Create a NumPy array `x` and use `copy()` to copy the array to `y`. What happens to each array if you change the value of `x[1]`? If you change the value of `y[3]`?

The same advice applies to lists: you should avoid using `y = x`. Note that the syntax for copying a list is a bit different from copying a NumPy array:

```
x = [2.7, "cat", -8]  # create a list
y = x.copy()          # copy the list
print(y)
x[1] = "dog"          # change element x[1]
print(x, y)
```

Exercise 6.7b

Run the code above and and check the final results for `x` and `y`. What happens if you replace the line `x[1] = "dog"` with `y[1] = "dog"`?

6.8 Cellular automata

A cellular automaton consists of a regular grid of cells. Each cell can be either *on* or *off*. These two states are represented by +1 (for on) and 0 (for off). The cells evolve, switching their states between on and off, according to some predefined rules.

Consider a simple example with a one–dimensional grid. We can number the cells by an index j. Observe that each cell has two neighbors, one to the left and one to the right. In particular, the jth cell has neighbors $j-1$ and $j+1$. Here is one possible set of rules:

- If the two neighbors of cell j are in different states, switch the state of j.
- If the two neighbors of cell j are in the same state, leave the state of j alone.

The picture below shows the jth cell and its nearest neighbors. Cell $j-1$ is off (0), cells j and $j+1$ are on (+1). Since the neighbors of j are in different states, cell j will switch from on (+1) to off (0) when the rules are applied.

$$\begin{array}{ccccccc} \cdots & \boxed{0} & \boxed{1} & \boxed{1} & \cdots \\ \cdots & j-1 & j & j+1 & \cdots \end{array}$$

Let's write a code to simulate the evolution of these cells. Begin by creating the grid of, say, 20 cells, all in the off (0) state. Then flip some

interior cells to on (+1):

```
C = np.zeros(20,dtype=int)
C[10] = 1
```

Now we implement the rules, repeating 5 times:

```
for n in range(5):
    D = np.copy(C)
    for j in range(1,19):
        if D[j-1] != D[j+1]:
            C[j] = 1 - C[j]
print(C)
```

Comments:

- The array `C` is created using `dtype=int`, and `D` is a copy of `C`. As a result the elements of `D` are integers. We chose to use integers because we want to avoid machine roundoff errors in the evaluation of the condition `D[j-1] != D[j+1]`.[1]
- The outer `for` loop causes the rules to be repeated 5 times, once for each value of `n` from 0 through 4. The `print(C)` statement prints the final results.
- The condition `D[j-1] != D[j+1]` evaluates to `True` if the nearest neighbors of cell `j` are in different states.
- The line `C[j] = 1 - C[j]` flips the state of cell `j`. If the old state of `C[j]` is 0, the new state is 1. If the old state of `C[j]` is 1, the new state is 0.
- In the `for j in range(1,19):` statement, `j` takes values from 1 through 18. Note that `C = np.zeros(20,dtype=int)` creates cells numbered from 0 through 19. We cannot implement our rules for cell number 0 or for cell number 19, since these cells don't have two nearest neighbors. We only implement the rules for the *interior* cells, those numbered 1 through 18.
- The statement `D = np.copy(C)` makes a copy of array `C` and calls it `D`. Why is this necessary? Why don't we use the array `C` throughout the code? Because the element `j` is being modified, or not, depending on the value of element `j-1`. We cannot change the state of element `j-1` before checking to see if `j-1` and `j+1` are in the same state.

[1] Roundoff errors do not actually affect this code, because Python can represent the real numbers 0.0 and 1.0 exactly. Nevertheless, it is important to remember that in many situations roundoff error can affect a comparison between real numbers.

The final result for this cellular automaton is:

```
[0 0 0 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0]
```

Exercise 6.8a

Run the cellular automaton code. Modify the code by replacing `D = np.copy(C)` with the simple statement `D = C`. Why doesn't this work?

Remember: The statement `D = C` simply creates another name for the array `C`. When the elements of `C` are modified, the elements of `D` are also modified. On the other hand, `D = np.copy(C)` creates a new array `D` whose elements are initially the same as the elements of `C`. In this case, `C` and `D` can be modifed without affecting each another.

How can we visualize the evolution of the cellular automaton? A simple way is to add the following lines of code inside the `for` loop over `n`:[2]

```
for j in range(0,20):
    if C[j] == 1:
        plt.plot(j,n,'r*')
```

You might need to include `plt.close()` at the beginning of your program and `plt.show()` at the end. These lines of code will plot the state of the cellular automaton after each application of the rules, beginning with `n = 0`. You can also display the initial state by adding the same lines of code before the loop over `n`, but with `plt.plot(j,n,'r*')` replaced by `plt.plot(j,-1,'r*')`

Exercise 6.8b

Modify the cellular automaton code to produce a plot of the evolution. Increase the number of cells and increase the number of iterations of the rules. Modify the initial state such that several cells are on (+1). See, for example, Fig. (6.1).

Exercise 6.8c

Create a cellular automaton code that uses these rules:

- If cell `j` is off (0), flip it to on (+1).

[2] Remember, `plt` is shorthand for `matplotlib.pyplot`.

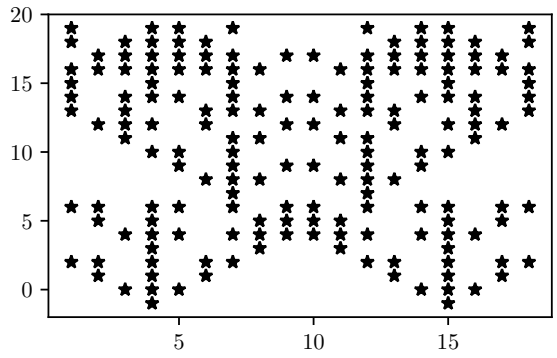


Fig. 6.1: Evolution of the cellular automaton using the rules of Sec. 6.8. Cells 4 and 15 are initially in the on (+1) state. The evolution is carried out for 20 iterations.

- If `j` is on and its nearest neighbors `j-1` and `j+1` are the same, leave `j` alone.
- If `j` is on and its nearest neighbors `j-1` and `j+1` are different, flip `j` to off.

Plot the evolution.