Lesson 2: Play with data in Jupyter

This is the second lesson of our course in *"Engineering Computations."* In the first lesson, *Interacting with Python*, we used **IPython**, the interactive Python shell. It is really great to type single-line Python expressions and get the outputs, interactively. Yet, believe it or not, there are greater things!

In this lesson, you will continue playing with data using Python, but you will do so in a **Jupyter notebook**. This very lesson is written in a Jupyter notebook. Ready? You will love it.

1 What is Jupyter?

Jupyter is a set of open-source tools for interactive and exploratory computing. You work right on your browser, which becomes the user interface through which Jupyter gives you a file explorer (the *dashboard*) and a document format: the **notebook**.

A Jupyter notebook can contain: input and output of code, formatted text, images, videos, pretty math equations, and much more. The computer code is *executable*, which means that you can run the bits of code, right in the document, and get the output of that code displayed for you. This interactive way of computing, mixed with the multi-media narrative, allows you to tell a story (even to yourself) with extra powers!

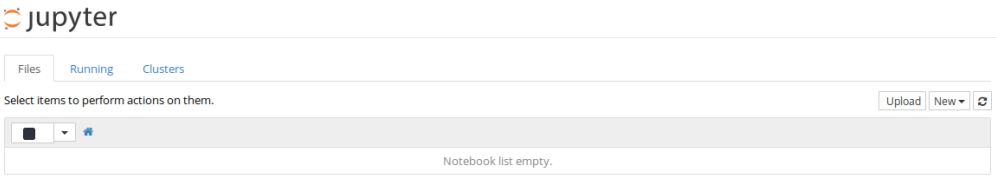
2 Working in Jupyter

Several things will seem counter-intuitive to you at first. For example, most people are used to launching apps in their computers by clicking some icon: this is the first thing to "unlearn." Jupyter is launched from the *command line* (like when you launched IPython). Next, we have two types of content—code and markdown—that handle a bit differently. The fact that your browser is an interface to a compute engine (called "kernel") leads to some extra housekeeping (like shutting down the kernel). But you'll get used to it pretty quick!

2.1 Start Jupyter

The standard way to start Jupyter is to type the following in the command-line interface:

```
jupyter notebook
```

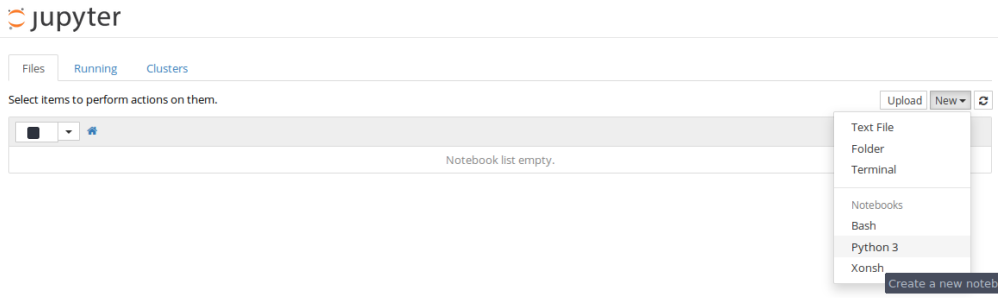


Screenshot of the Jupyter dashboard, open in the browser.

Hit enter and tadah!! After a little set up time, your default browser will open with the Jupyter app. It should look like in the screenshot below, but you may see a list of files and folders, depending on the location of your computer where you launched it.

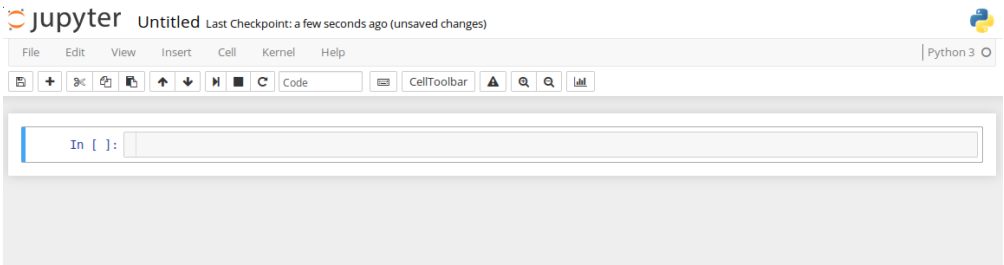
Note: Don't close the terminal window where you launched Jupyter (while you're still working on Jupyter). If you need to do other tasks on the command line, open a new terminal window.

To start a new Jupyter notebook, click on the top-right, where it says **New**, and select `Python 3`. Check out the screenshot below.



Screenshot showing how to create a new notebook.

A new tab will appear in your browser and you will see an empty notebook, with a single input line, waiting for you to enter some code. See the next screenshot.



Screenshot showing an empty new notebook.

The notebook opens by default with a single empty code cell. Try to write some Python code there and execute it by hitting `[shift] + [enter]`.

2.2 Notebook cells

The Jupyter notebook uses *cells*: blocks that divide chunks of text and code. Any text content is entered in a *Markdown* cell: it contains text that you can format using simple markers to get headings, bold, italic, bullet points, hyperlinks, and more.

Markdown is easy to learn, check out the syntax in the "Daring Fireball" webpage (by John Gruber). A few tips:

- to create a title, use a hash to start the line: `# Title`
- to create the next heading, use two hashes (and so on): `## Heading`
- to italicize a word or phrase, enclose it in asterisks (or underdashes): `*italic*` or `_italic_`
- to make it bold, enclose it with two asterisks: `**bolded**`
- to make a hyperlink, use square and round brackets: `[hyperlinked text](url)`

Computable content is entered in code cells. We will be using the IPython kernel ("kernel" is the name used for the computing engine), but you should know that Jupyter can be used with many different computing languages. It's amazing.

A code cell will show you an input mark, like this:

```
In [ ]:
```

Once you add some code and execute it, Jupyter will add a number ID to the input cell, and produce an output marked like this:

```
Out [1]:
```

A bit of history: Markdown was co-created by the legendary but tragic Aaron Swartz. The biographical documentary about him is called "The Internet's Own Boy," and you can view it in YouTube or Netflix. Recommended!

2.3 Interactive computing in the notebook

Look at the icons on the menu of Jupyter (see the screenshots above). The first icon on the left (an old floppy disk) is for saving your notebook. You can add a new cell with the big **+** button. Then you have the cut, copy, and paste buttons. The arrows are to move your current cell up or down. Then you have a button to "run" a code cell (execute the code), the square icon means "stop" and the swirly arrow is to "restart" your notebook's kernel (if the computation is stuck, for example). Next to that, you have the cell-type selector: Code or Markdown (or others that you can ignore for now).

You can test-drive a code cell by writing some arithmetic operations. Like we saw in our first lesson, the Python operators are:

```
+   -   *   /   **   %   //
```

There's addition, subtraction, multiplication and division. The last three operators are *exponent* (raise to the power of), *modulo* (divide and return remainder) and *floor division*.

Typing `[shift] + [enter]` will execute the cell and give you the output in a new line, labeled `Out[1]` (the numbering increases each time you execute a cell).

Try it! Add a cell with the plus button, enter some operations, and `[shift] + [enter]` to execute.

```
In [ ]:
```

Everything we did using IPython we can do in code cells within a Jupyter notebook. Try out some of the things we learned in lesson 1:

```
In [1]: print("Hello World!")
```

```
Hello World!
```

```
In [2]: x = 2**8
        x < 64
```

```
Out[2]: False
```

2.4 Edit mode and Command mode

Once you click on a notebook cell to select it, you may interact with it in two ways, which are called *modes*. Later on, when you are reviewing this material again, read more about this in Reference 1.

Edit mode:

- We enter **edit mode** by pressing `Enter` or double-clicking on the cell.
- We know we are in this mode when we see a green cell border and a prompt in the cell area.
- When we are in edit mode, we can type into the cell, like a normal text editor.

Command mode:

- We enter in **command mode** by pressing `Esc` or clicking outside the cell area.
- We know we are in this mode when we see a grey cell border with a left blue margin.
- In this mode, certain keys are mapped to shortcuts to help with common actions.

You can find a list of the shortcuts by selecting `Help->Keyboard Shortcuts` from the notebook menu bar. You may want to leave this for later, and come back to it, but it becomes more helpful the more you use Jupyter.

2.5 How to shut down the kernel and exit

Closing the browser tab where you've been working on a notebook does not immediately "shut down" the compute kernel. So you sometimes need to do a little housekeeping.

Once you close a notebook, you will see in the main Jupyter app that your notebook file has a green book symbol next to it. You should click in the box at the left of that symbol, and then click where it says **Shutdown**. You don't need to do this all the time, but if you have a *lot* of notebooks running, they will use resources in your machine.

Similarly, Jupyter is still running even after you close the tab that has the Jupyter dashboard open. To exit the Jupyter app, you should go to the terminal that you used to open Jupyter, and type `[Ctrl] + [c]` to exit.

2.6 Nbviewer

Nbviewer is a free web service that allows you to share static versions of hosted notebook files, as if they were a web page. If a notebook file is publicly available on the web, you can view it by entering its URL in the nbviewer web page, and hitting the **Go!** button. The notebook will be rendered as a static page: visitors can read everything, but they cannot interact with the code.

3 Play with Python strings

Let's keep playing around with strings, but now coding in a Jupyter notebook (instead of IPython). We recommend that you open a clean new notebook to follow along the examples in this lesson, typing the commands that you see. (If you copy and paste, you will save time, but you will learn little. Type it all out!)

```
In [3]: str_1 = 'hello'
        str_2 = 'world'
```

Remember that we can concatenate strings ("add"), for example:

```
In [4]: new_string = str_1 + str_2
        print(new_string)
```

```
helloworld
```

What if we want to add a space that separates `hello` from `world`? We directly add the string `' '` in the middle of the two variables. A space is a character!

```
In [5]: my_string = str_1 + ' ' + str_2
        print(my_string)
```

```
hello world
```

Exercise: Create a new string variable that adds three exclamation marks to the end of `my_string`.

3.1 Indexing

We can access each separate character in a string (or a continuous segment of it) using *indices*: integers denoting the position of the character in the string. Indices go in square brackets, touching the string variable name on the right. For example, to access the 1st element of `new_string`, we would enter `new_string[0]`. Yes! in Python we start counting from 0.

```
In [6]: my_string[0]

Out[6]: 'h'

In [7]: #If we want the 3rd element we do:
        my_string[2]

Out[7]: 'l'
```

You might have noticed that in the cell above we have a line before the code that starts with the # sign. That line seems to be ignored by Python: do you know why?

It is a *comment*: whenever you want to comment your Python code, you put a # in front of the comment. For example:

```
In [8]: my_string[1] #this is how we access the second element of a string

Out[8]: 'e'
```

How do we know the index of the last element in the string?

Python has a built-in function called `len()` that gives the information about length of an object. Let's try it:

```
In [9]: len(my_string)

Out[9]: 11
```

Great! Now we know that `my_string` is eleven characters long. What happens if we enter this number as an index?

```
In [10]: my_string[11]

         ---------------------------------------------------------------------------

         IndexError                                Traceback (most recent call last)

         <ipython-input-10-19e2c11e7861> in <module>()
    ----> 1 my_string[11]


         IndexError: string index out of range
```

Oops. We have an error: why? We know that the length of `my_string` is eleven. But the integer 11 doesn't work as an index. If you expected to get the last element, it's because you forgot that Python starts counting at zero. Don't worry: it takes some getting used to.

The error message says that the index is out of range: this is because the index of the *last element* will always be: `len(string) - 1`. In our case, that number is 10. Let's try it out.

```
In [11]: my_string[10]

Out[11]: 'd'
```

Python also offers a clever way to grab the last element so we don't need to calculate the lenghth and substract one: it is using a negative 1 for the index. Like this:

```
In [12]: my_string[-1]
```

```
Out[12]: 'd'
```

What if we use a `-2` as index?

```
In [13]: my_string[-2]
```

```
Out[13]: 'l'
```

That is the last `l` in the string `hello world`. Python is so clever, it can count backwards!

3.2 Slicing strings

Sometimes, we want to grab more than one single element: we may want a section of the string. We do it using *slicing* notation in the square brackets. For example, we can use `[start:end]`, where `start` is the index to beging the slice, and `end` is the (non-inclusive) index to finish the slice. For example, to grab the word `hello` from our string, we do:

```
In [14]: my_string[0:5]
```

```
Out[14]: 'hello'
```

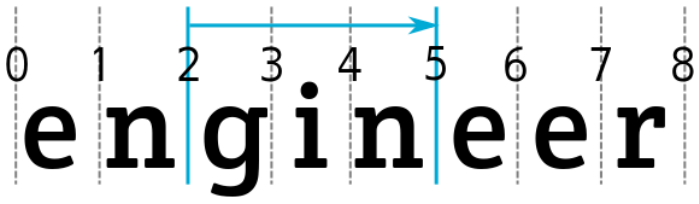
You can skip the `start` index, if you want to slice from the beginning of the string, and you can skip the `end` of a slice, indicating you want to go all the way to the end of the string. For example, if we want to grab the word `'world'` from `my_string`, we could do the following:

```
In [15]: my_string[6:]
```

```
Out[15]: 'world'
```

A helpful way to visualize slices is to imagine that the indices point to the spaces *between* characters in the string. That way, when you write `my_string[i]`, you would be referring to the "character to the right of `i`" (Reference 2).

Check out the diagram below. We start counting at zero; the letter `'g'` is to the right of index 2. So if we want to grab the sub-string `'gin'` from `'engineer'`, we need `[start:end]=[2:5]`.



Try it yourself!

```
In [16]: # Define your string
         eng_string = 'engineer'

         # Grab 'gin'slice
         eng_string[2:5]
```

```
Out[16]: 'gin'
```

Exercises:

1. Define a string called `'banana'` and print out the first and last `'a'`.
2. Using the same string, grab the 2 possible slices that correspond to the word `'ana'` and print them out.
3. Create your own slicing exercise and ask your classmates to give it a try (work in groups of 3).

The following lines contain the solutions; to reveal the answer, select the lines with the mouse:

Solution Exercise 1:

b = 'banana' print(b[1]) print(b[-1])

Solution Exercise 2:

print(b[1:4]) print(b[3:])

3.3 What else we can do with strings?

Python has many useful built-in functions for strings. You'll learn a few of them in this section. A technical detail: in Python, some functions are associated with a particular class of objects (e.g., strings). The word **method** is used in this case, and we have a new way to call them: the dot operator. It is a bit counter-intuitive in that the name of the method comes *after the dot*, while the name of the particular object it acts on comes first. Like this: `mystring.method()`.

If you are curious about the many available methods for strings, go to the section "Built-in String Methods" in this tutorial.

Let's use a quote by Albert Einstein as a string and apply some useful string methods.

```
In [17]: AE_quote = "Everybody is a genius. But if you judge a fish by its ability to
 climb a tree, it will live its whole life believing that it is stupid."
```

The `count()` method gives the number of ocurrences of a substring in a range. The arguments for the range are optional.

Syntax:

```
str.count(substring, start, end)
```

Here, `start` and `end` are integers that indicate the indices where to start and end the count. For example, if we want to know how many letters `'e'` we have in the whole string, we can do:

```
In [18]: AE_quote.count('e')
```

```
Out[18]: 10
```

If we want to know how many of those `'e'` charachters are in the range `[0:20]`, we do:

```
In [19]: AE_quote.count('e', 0, 20)
```

```
Out[19]: 2
```

We can look for more complex strings, for example:

```
In [20]: AE_quote.count('Everybody')
```

```
Out[20]: 1
```

The **find()** method tells us if a string `'substr'` occurs in the string we are applying the method on. The arguments for the range are optional.

Syntax:

```
str.find(substr, start, end)
```

Where `start` and `end` are indices indicating where to start and end the slice to apply the `find()` method on.

If the string `'substr'`is in the original string, the `find()` method will return the index where the substring starts, otherwise it will return `-1`.

For example, let's find the word "fish" in the Albert Einstein quote.

```
In [21]: AE_quote.find('fish')
```

```
Out[21]: 42
```

If we know the length of our sub-string, we can now apply slice notation to grab the word "fish".

```
In [22]: len('fish')
```

```
Out[22]: 4
```

```
In [23]: AE_quote[42: 42 + len('fish')]
```

```
Out[23]: 'fish'
```

Let's see what happens when we try to look for a string that is not in the quote.

```
In [24]: AE_quote.find('albert')
```

```
Out[24]: -1
```

It returns `-1`... but careful, that doesn't mean that the position is at the end of the original string! If we read the documentation, we confirm that a returned value of `-1` indicates that the sub-string we are looking for is *not in the string* we are searching in.

A similar method is `index()`: it works like the `find()` method, but throws an error if the string we are searching for is not found.

Syntax:

```
str.index(substr, start, end)
```

```
In [25]: AE_quote.index('fish')
```

```
Out[25]: 42
```

```
In [26]: AE_quote.index('albert')
```

```
ValueError                                Traceback (most recent call last)

<ipython-input-26-19ab4543c577> in <module>()
----> 1 AE_quote.index('albert')


ValueError: substring not found
```

In the example above, we used the `len()` function to calculate the length of the string `'fish'`, and we used the result to calculate the ending index. However, if the string is too long, having a line that calculates the length might be inconvenient or may make your code look messy. To avoid this, we can use the `find()` or `index()` methods to calculate the end position. In the `'fish'` example, we could look for the index of the word `'by'` (the word that follows `'fish'`) and subtract 1 from that index to get the index that corresponds to the space right after `'fish'`. There are many ways to slice strings, only limited by your imagination!

Note: Rememeber that the ending index is not inclusive, which is why we want the index of the space that follows the string `'fish'`.

```
In [27]: idx_start = AE_quote.index('fish')
         idx_end = AE_quote.index('by') - 1 # -1 to get the index of the space after 'fish'

In [28]: AE_quote[idx_start:idx_end]

Out[28]: 'fish'
```

Exercises:

1. Use the `count()` method to count how many letters `'a'` are in `AE_quote`?
2. Using the same method, how many isolated letters `'a'` are in `AE_quote`?
3. Use the `index()` method to find the position of the words `'genius'`, `'judge'` and `'tree'` in `AE_quote`.
4. Using slice syntax, extract the words in exercise 3 from `AE_quote`.

Two more string methods turn out to be useful when you are working with texts and you need to clean, separate or categorize parts of the text.

Let's work with a different string, a quote by Eleanor Roosevelt:

```
In [29]: ER_quote = "   Great minds discuss ideas; average minds discuss events; small
 minds discuss people.  "
```

Notice that the string we defined above contains extra white spaces at the beginning and at the end. In this case, we did it on purpose, but bothersome extra spaces are often present when reading text from a file (perhaps due to paragraph indentation).

Strings have a method that allows us to get rid of those extra white spaces.

The `strip()` method returns a copy of the string in which all characters given as argument are stripped from the beginning and the end of the string.

Syntax:

```
str.strip([chars])
```

The default argument is the space character. For example, if we want to remove the white spaces in the `ER_quote`, and save the result back in `ER_quote`, we can do:

```
In [30]: ER_quote = ER_quote.strip()

In [31]: ER_quote

Out[31]: 'Great minds discuss ideas; average minds discuss events; small minds
discuss people.'
```

Let's supose you want to strip the period at the end; you could do the following:

```
ER_quote = ER_quote.strip('.')
```

But if we don't want to keep the changes in our string variable, we don't overwrite the variable as we did above. Let's just see how it looks:

```
In [32]: ER_quote.strip('.')

Out[32]: 'Great minds discuss ideas; average minds discuss events; small minds
discuss people'
```

Check the string variable to confirm that it didn't change (it still has the period at the end):

```
In [33]: ER_quote

Out[33]: 'Great minds discuss ideas; average minds discuss events; small minds
discuss people.'
```

Another useful method is `startswith()`, to find out if a string starts with a certain character. Later on in this lesson we'll see a more interesting example; but for now, let's just "check" if our string starts with the word 'great'.

```
In [34]: ER_quote.startswith('great')

Out[34]: False
```

The output is `False` because the word is not capitalized! Upper-case and lower-case letters are distinct characters.

```
In [35]: ER_quote.startswith('Great')

Out[35]: True
```

It's important to mention that we don't need to match the character until we hit the white space.

```
In [36]: ER_quote.startswith('Gre')

Out[36]: True
```

The last string method we'll mention is `split()`: it returns a **list** of all the words in a string. We can also define a separator and split our string according to that separator, and optionally we can limit the number of splits to `num`.

Syntax:

```
str.split(separator, num)
```

```
In [37]: print(AE_quote.split())
```

```
['Everybody', 'is', 'a', 'genius.', 'But', 'if', 'you', 'judge', 'a', 'fish', 'by',
'its', 'ability', 'to', 'climb', 'a', 'tree,', 'it', 'will', 'live', 'its', 'whole',
'life', 'believing', 'that', 'it', 'is', 'stupid.']
```

```
In [38]: print(ER_quote.split())
```

```
['Great', 'minds', 'discuss', 'ideas;', 'average', 'minds', 'discuss', 'events;',
'small', 'minds', 'discuss', 'people.']
```

Let's split the `ER_quote` by a different character, a semicolon:

```
In [39]:  print(ER_quote.split(';'))
```

```
['Great minds discuss ideas', ' average minds discuss events', ' small minds discuss
people.']
```

Think... Do you notice something new in the output of the `print()` calls above? What are those `[ ]`?

4 Play with Python lists

The square brackets above indicate a Python **list**. A list is a built-in data type consisting of a sequence of values, e.g., numbers, or strings. Lists work in many ways similarly to strings: their elements are numbered from zero, the number of elements is given by the function `len()`, they can be manipulated with slicing notation, and so on.

The easiest way to create a list is to enclose a comma-separated sequence of values in square brackets:

```
In [40]: # A list of integers
         [1, 4, 7, 9]

Out[40]: [1, 4, 7, 9]

In [41]: # A list of strings
         ['apple', 'banana', 'orange']

Out[41]: ['apple', 'banana', 'orange']

In [42]: # A list with different element types
         [2, 'apple', 4.5, [5, 10]]

Out[42]: [2, 'apple', 4.5, [5, 10]]
```

In the last list example, the last element of the list is actually *another list*. Yes! we can totally do that.

We can also assign lists to variable names, for example:

```
In [43]: integers = [1, 2, 3, 4, 5]
         fruits = ['apple', 'banana', 'orange']

In [44]: print(integers)

[1, 2, 3, 4, 5]


In [45]: print(fruits)

['apple', 'banana', 'orange']


In [46]: new_list = [integers, fruits]

In [47]: print(new_list)

[[1, 2, 3, 4, 5], ['apple', 'banana', 'orange']]
```

Notice that this `new_list` has only 2 elements. We can check that with the `len()` function:

```
In [48]: len(new_list)

Out[48]: 2
```

Each element of `new_list` is, of course, another list. As with strings, we access list elements with indices and slicing notation. The first element of `new_list` is the list of integers from 1 to 5, while the second element is the list of three fruit names.

```
In [49]: new_list[0]

Out[49]: [1, 2, 3, 4, 5]

In [50]: new_list[1]

Out[50]: ['apple', 'banana', 'orange']

In [51]: # Accessing the first two elements of the list fruits
         fruits[0:2]

Out[51]: ['apple', 'banana']
```

Exercises:

1. From the `integers` list, grab the slice `[2, 3, 4]` and then `[4, 5]`.
2. Create your own list and design an exercise for grabbing slices, working with your classmates.

4.1 Adding elements to a list

We can add elements to a list using the **append()** method: it appends the object we pass into the existing list. For example, to add the element 6 to our `integers` list, we can do:

```
In [52]: integers.append(6)
```

Let's check that the `integer` list now has a 6 at the end:

```
In [53]: print(integers)
```

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

4.2 List membership

Checking for list membership in Python looks pretty close to plain English!

Syntax

To check if an element is **in** a list:

```
element in list
```

To check if an element is **not in** a list:

```
element not in list
```

```
In [54]: 'strawberry' in fruits
```

```
Out[54]: False
```

```
In [55]: 'strawberry' not in fruits
```

```
Out[55]: True
```

Exercises

1. Add two different fruits to the `fruits` list.
2. Check if `'mango'` is in your new `fruits` list.
3. Given the list `alist = [1, 2, 3, '4', [5, 'six'], [7]]` run the following in separate cells and discuss the output with your classmates:

```
4 in alist
5 in alist
7 in alist
[7] in alist
```

4.3 Modifying elements of a list

We can not only add elements to a list, we can also modify a specific element. Let's re-use the list from the exercise above, and replace some elements.

```
In [56]: alist = [1, 2, 3, '4', [5, 'six'], [7]]
```

We can find the position of a certain element with the `index()` method, just like with strings. For example, if we want to know where the element `'4'` is, we can do:

```
In [57]: alist.index('4')
```

```
Out[57]: 3

In [58]: alist[3]

Out[58]: '4'
```

Let's replace it with the integer value 4:

```
In [59]: alist[3] = 4

In [60]: alist

Out[60]: [1, 2, 3, 4, [5, 'six'], [7]]

In [61]: 4 in alist

Out[61]: True
```

Exercise Replace the last element of `alist` with something different.

Being able to modify elements in a list is a "property" of Python lists; other Python objects we'll see later in the course also behave like this, but not all Python objects do. For example, you cannot modify elements in a a string. If we try, Python will complain.

Fine! Let's try it:

```
In [62]: string = 'This is a string.'
```

Suppose we want to replace the period ('.') by an exaclamation mark ('!'). Can we just modify this string element?

```
In [63]: string[-1]

Out[63]: '.'

In [64]: string[-1] = '!'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-64-dbf68e37fb66> in <module>()
----> 1 string[-1] = '!'

TypeError: 'str' object does not support item assignment
```

Told you! Python is confirming that we cannot change the elements of a string by item assignment.

5 Next: strings and lists in action

You have learned many things about strings and lists in this lesson, and you are probably eager to see how to apply it all to a realistic situation. We created a full example in a separate notebook to show you the power of Python with text data.

But before jumping in, we should introduce you to the powerful ideas of **iteration** and **conditionals** in Python.

5.1 Iteration with `for` statements

The idea of *iteration* (in plain English) is to repeat a process several times. If you have any programming experience with another language (like C or Java, say), you may have an idea of how to create iteration with `for` statements. But these are a little different in Python, as you can read in the documentation.

A Python `for` statement iterates over the items of a sequence, naturally. Say you have a list called `fruits` containing a sequence of strings with fruit names; you can write a statement like

```
for fruit in fruits:
```

to do something with each item in the list.

Here, for the first time, we will encounter a distinctive feature of the Python language: grouping by **indentation**. To delimit *what* Python should do with each `fruit` in the list of `fruits`, we place the next statement(s) *indented* from the left.

How much to indent? This is a style question, and everyone has a preference: two spaces, four spaces, one tab... they are all valid: but pick one and be consistent!

Let's use four spaces:

```
In [65]: fruits = ['apple', 'banana', 'orange', 'cherry', 'mandarin']

         for fruit in fruits:
             print("Eat your", fruit)
```

```
Eat your apple
Eat your banana
Eat your orange
Eat your cherry
Eat your mandarin
```

Pay attention:

- the `for` statement ends with a colon, :
- the variable `fruit` is implicitly defined in the `for` statement
- `fruit` takes the (string) value of each element of the list `fruits`, in order
- the indented `print()` statement is executed for each value of `fruit`
- once Python runs out of `fruits`, it stops

- we don't need to know ahead of time how many items are in the list!

Challenge question: — What is the value of the variable `fruit` after executing the `for` statement above? Discuss with your neighbor. (Confirm your guess in a code cell.)

A very useful function to use with `for` statements is `enumerate()`: it adds a counter that you can use as an index while your iteration runs. To use it, you implicitly define *two* variables in the `for` statement: the counter, and the value of the sequence being iterated on.

Study the following block of code:

```
In [66]: names = ['sam', 'zoe', 'naty', 'gil', 'tom']

         for i, name in enumerate(names):
             names[i] = name.capitalize()
         print(names)
```

```
['Sam', 'Zoe', 'Naty', 'Gil', 'Tom']
```

Challenge question: — What is the value of the variable `name` after executing the `for` statement above? Discuss with your neighbor. (Confirm your guess in a code cell.)

Exercise: Say we have a list of lists (a.k.a., a *nested* list), as follows:

```
fullnames = [['sam','jones'], ['zoe','smith'],['joe','cheek'],['tom','perez'] ]
```

Write some code that creates two simple lists: one with the first names, another with the last names from the nested list above, but capitalized.

To start, you need to create two *empty* lists using the square brackets with nothing inside. We've done that for you below. *Hint*: Use the `append()` list method!

```
In [67]: fullnames = [ ['sam','jones'], ['zoe','smith'],['joe','cheek'],['tom','perez'] ]
         firstnames = []
         lastnames = []

         # Write your code here
```

5.2 Conditionals with `if` statements

Sometimes we need the ability to check for conditions, and change the behavior of our program depending on the condition. We accomplish it with an `if` statement, which can take one of three forms.

(1) **If** statement on its own:

```
In [68]: a = 8
         b = 3
```

```
        if a > b:
            print('a is bigger than b')
```

```
a is bigger than b
```

(2) **If-else** statement:

```
In [69]: # We pick a number, but you can change it
         x = 1547
```

```
In [70]: if x % 17 == 0:
             print('Your number is a multiple of 17.')
         else:
             print('Your number is not a multiple of 17.')
```

```
Your number is a multiple of 17.
```

Note: The % represents a modulo operation: it gives the remainder from division of the first argument by the second

Tip: You can uncomment this following cell, and learn a good trick to ask the user to insert a number. You can use this instead of assigning a speceifc value to x above.

```
In [71]: #x = float(input('Insert your number: '))
```

(3) **If-elif-else** statement:

```
In [72]: a = 3
         b = 5

         if a > b:
             print('a is bigger than b')
         elif a < b:
             print('a is smaller than b')
         else:
             print('a is equal to b')
```

```
a is smaller than b
```

Note: We can have as many `elif` lines as we want.

Exercise Using `if`, `elif` and `else` statements write a code where you pick a 4-digit number, if it is divisible by 2 and 3 you print: 'Your number is not only divisible by 2 and 3 but also by 6'. If it is divisible by 2 you print: 'Your number is divisible by 2'. If it is divisible by 3 you print: 'Your number is divisible by 3'. Any other option, you print: 'Your number is not divisible by 2, 3 or 6'

6 What we've learned

- How to use the Jupyter environnment.
- Playing with strings: accessing values, slicing and string methods.
- Playing with lists: accessing values, slicing and list methods.
- Iteration with `for` statements.
- Conditionals with `if` statements.

7 References

1. Notebook Basics: Modal Editor
2. "Indices point between elements," blog post by Nelson Elhage (2015).
3. *Python for Everybody: Exploring Data Using Python 3* (2016). Charles R. Severance. PDF available
4. *Think Python: How to Think Like a Computer Scientist* (2012). Allen Downey. Green Tea Press. PDF available