Engineering Computations

Module 1: Get data off the ground

Lorena A. Barba and Natalia C. Clementi

Lesson 1: Interacting with Python

This is the first lesson in our course *"Engineering Computations"*, a one-semester course for second-year university students. The course uses Python, and assumes no prior programming experience. Our first step will be to get you interacting with Python. But let's also learn some background.

1 What is Python?

Python is now 26 years old. Its creator, Guido van Rossum, named it after the British comedy "Monty Python's Flying Circus." His goals for the language were that it be an "an easy and intuitive language just as powerful as major competitors," producing computer code "that is as understandable as plain English."

It is a general-purpose language, which means that you can use it for anything: organizing data, scraping the web, creating websites, analyzing sounds, creating games, and of course *engineering computations*.

Python is an interpreted language. This means that you can write Python commands and the computer can execute those instructions directly. Other programming languages—like C, C++ and Fortran—require a previous *compilation* step: translating the commands into machine language. A neat ability of Python is to be used *interactively*. Fernando Perez famously created **IPython** as a side-project during his PhD. We're going to use IPython (the I stands for "interactive") in this lesson.

2 Why Python?

Because it's fun! With Python, the more you learn, the more you *want* to learn. You can find lots of resources online and, since Python is an open-source project, you'll also find a friendly community of people sharing their knowledge.

Python is known as a *high-productivity language*. As a programmer, you'll need less time to develop a solution with Python than with most languages. This is important to always bring up whenever someone complains that "Python is slow." Your time is more valuable than a machine's! (See the Recommended Readings section at the end.) And if we really need to speed up our program, we can re-write the slow parts in a compiled language afterwards. Because Python plays well with other languages :–)

The top technology companies use Python: Google, Facebook, Dropbox, Wikipedia, Yahoo!, YouTube… And this year, Python took the No. 1 spot in the interactive list of The 2017 Top

Programming Languages, by *IEEE Spectrum* (IEEE is the world's largest technical professional society).

***Python is a versatile language, you can analyze data, build websites (e.g., Instagram, Mozilla, Pinterest), make art or music, etc. Because it is a versatile language, employers love Python: if you know Python they will want to hire you.* —Jessica McKellar, ex Director of the Python Software Foundation, in a 2014 tutorial.**

3 Let's get started

In this first lesson, we will be using IPython: a tool for working with Python interactively. If you have it installed in the computer you're using for this lesson, you enter the program by typing

```
ipython
```

on the command-line interface (the **Terminal** app on Mac OSX, and on Windows possibly the **PowerShell** or **git bash**). You will get a few lines of text about your IPython version and how to get help, and a blinking cursor next to the input line counter:

```
In[1]:
```

That input line is ready to receive any Python code to be executed interactively. The output of the code will be shown to you next to `Out[1]`, and so on for successive input/output lines.

Note: Our plan for this course is to work in a computer lab, where everyone will have a computer with everything installed ahead of time. For this reason, we won't discuss installation right now. Later on, when you're eager to work on your personal computer, we'll help you install everything you need. *It's all free!*

3.1 Your first program

In every programming class ever, your first program consists of printing a *"Hello"* message. In Python, you use the `print()` function, with your message inside quotation marks.

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

```
Hello world!!
```

Easy peasy!! You just wrote your first program and you learned how to use the `print()` function. Yes, `print()` is a function: we pass the *argument* we want the function to act on, inside the parentheses. In the case above, we passed a *string*, which is a series of characters between quotation marks. Don't worry, we will come back to what strings are later on in this lesson.

Key concept: function A function is a compact collection of code that executes some action on its *arguments*. Every Python function has a *name*, used to call it, and takes its arguments inside round brackets. Some arguments may be optional (which means they have a default value defined inside the function), others are required. For example, the `print()` function has one required argument: the string of characters it should print out for you.

Python comes with many *built-in* functions, but you can also build your own. Chunking blocks of code into functions is one of the best strategies to deal with complex programs. It makes you more efficient, because you can reuse the code that you wrote into a function. Modularity and reuse are every programmer's friend.

3.2 Python as a calculator

Try any arithmetic operation in IPython. The symbols are what you would expect, except for the "raise-to-the-power-of" operator, which you obtain with two asterisks: `**`. Try all of these:

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

The `%` symbol is the *modulo* operator (divide and return remainder), and the double-slash is *floor division*.

```
In [2]: 2 + 2

Out[2]: 4

In [3]: 1.25 + 3.65

Out[3]: 4.9

In [4]: 5 - 3

Out[4]: 2

In [5]: 2 * 4

Out[5]: 8

In [6]: 7 / 2

Out[6]: 3.5

In [7]: 2**3

Out[7]: 8
```

Let's see an interesting case:

```
In [8]: 9**1/2

Out[8]: 4.5
```

Discuss with your neighbor: *What happened?* Isn't $9^{1/2} = 3$? (Raising to the power $1/2$ is the same as taking the square root.) Did Python get this wrong?

Compare with this:

```
In [9]: 9**(1/2)

Out[9]: 3.0
```

Yes! The order of operations matters!

If you don't remember what we are talking about, review the Arithmetics/Order of operations. A frequent situation that exposes this is the following:

```
In [10]: 3 + 3 / 2

Out[10]: 4.5

In [11]: (3 + 3) / 2

Out[11]: 3.0
```

In the first case, we are adding 3 plus the number resulting of the operation $3/2$. If we want the division to apply to the result of $3+3$, we need the parentheses.

Exercises: Use IPython (as a calculator) to solve the following two problems:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with diameter 6.65 cm?

   For the value of π use 3.14159 (for now). Compare your answer with the solution up to 4 decimal numbers.

   Hint: 523.5983 is wrong and 615.9184 is also wrong.

2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies? Compare your answer with the solution up to 2 decimal numbers.

```
In [ ]:
```

To reveal the answers, highlight the following line of text using the mouse:

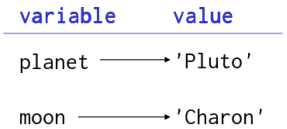
Answer exercise 1: 153.9796 Answer exercise 2: 945.45

3.3 Variables and their type

Variables consist of two parts: a name and a value. When we want to give a variable its name and value, we use the equal sign: `name = value`. This is called an *assignment*. The name of the variable goes on the left and the value on the right.

The first thing to get used to is that the equal sign in an assignment has a different meaning than it has in Algebra! Think of it as an arrow pointing from `name` to `value`.

```
In[1]: planet = 'Pluto'
In[2]: print(planet)
Pluto
In[3]: moon = 'Charon'
In[4]: print(moon)
Charon
```



We have many possibilities for variable names: they can be made up of upper and lowercase letters, underscores and digits... although digits cannot go on the front of the name. For example, valid variable names are:

```
x
x1
X_2
name_3
NameLastname
```

Keep in mind, there are reserved words that you can't use; they are the special Python keywords.

OK. Let's assign some values to variables and do some operations with them:

```
In [12]: x = 3
```

```
In [13]: y = 4.5
```

Exercise: Print the values of the variables `x` and `y`.

```
In [ ]:
```

Let's do some arithmetic operations with our new variables:

```
In [14]: x + y
```

```
Out[14]: 7.5
```

```
In [15]: 2**x
```

```
Out[15]: 8
```

```
In [16]: y - 3
```

```
Out[16]: 1.5
```

And now, let's check the values of `x` and `y`. Are they still the same as they were when you assigned them?

```
In [17]: print(x)
```

```
3
```

```
In [18]: print(y)
```

```
4.5
```

3.4 String variables

In addition to name and value, Python variables have a *type*: the type of the value it refers to. For example, an integer value has type `int`, and a real number has type `float`. A string is a variable consisting of a sequence of characters marked by two quotes, and it has type `str`.

```
In [19]: z = 'this is a string'
```

```
In [20]: w = '1'
```

What if you try to "add" two strings?

```
In [21]: z + w

Out[21]: 'this is a string1'
```

The operation above is called *concatenation*: chaining two strings together into one. Insteresting, eh? But look at this:

```
In [22]: x + w


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-22-bc79f95cdaf2> in <module>()
    ----> 1 x + w


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Error! Why? Let's inspect what Python has to say and explore what is happening.

Python is a *dynamic language*, which means that you don't *need* to specify a type to invoke an existing object. The humorous nickname for this is "duck typing":

"If it looks like a duck, and quacks like a duck, then it's probably a duck." In other words, a variable has a type, but we don't need to specify it. It will just behave like it's supposed to when we operate with it (it'll quack and walk like nature intended it to).

But sometimes you need to make sure you know the type of a variable. Thankfully, Python offers a function to find out the type of a variable: `type()`.

```
In [23]: type(x)

Out[23]: int

In [24]: type(w)

Out[24]: str

In [25]: type(y)

Out[25]: float
```

3.5 More assignments

What if you want to assign to a new variable the result of an operation that involves other variables? Well, you totally can!

```
In [26]: sum_xy = x + y
         diff_xy = x - y

In [27]: print('The sum of x and y is:', sum_xy)
         print('The difference between x and y is:', diff_xy)

The sum of x and y is: 7.5
The difference between x and y is: -1.5
```

Notice what we did above: we used the `print()` function with a string message, followed by a variable, and Python printed a useful combination of the message and the variable value. This is a pro tip! You want to print for humans. Let's now check the type of the new variables we just created above:

```
In [28]: type(sum_xy)

Out[28]: float

In [29]: type(diff_xy)

Out[29]: float
```

Discuss with your neighbor: Can you summarize what we did above?

3.6 Special variables

Python has special variables that are built into the language. These are: `True`, `False`, `None` and `NotImplemented`. For now, we will look at just the first three of these.

Boolean variables are used to represent truth values, and they can take one of two possible values: `True` and `False`. *Logical expressions* return a boolean. Here is the simplest logical expression, using the keyword `not`:

```
not True
```

It returns... you guessed it... `False`.

The Python function `bool()` returns a truth value assigned to any argument. Any number other than zero has a truth value of `True`, as well as any nonempty string or list. The number zero and any empty string or list will have a truth value of `False`. Explore the `bool()` function with various arguments.

```
In [30]: bool(0)

Out[30]: False

In [31]: bool('Do we need oxygen?')

Out[31]: True

In [32]: bool('We do not need oxygen')

Out[32]: True
```

None is not Zero: `None` is a special variable indicating that no value was assigned or that a behavior is undefined. It is different than the value zero, an empty string, or some other nil value.

You can check that it is not zero by trying to add it to a number. Let's see what happens when we try that:

```
In [33]: a = None

         b = 3

In [34]: a + b


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

         TypeError                                 Traceback (most recent call last)

         <ipython-input-34-f96fb8f649b6> in <module>()
    ----> 1 a + b


         TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

3.7 Logical and comparison operators

The Python comparison operators are: `<`, `<=`, `>`, `>=`, `==`, `!=`. They compare two objects and return either `True` or `False`: smaller than, smaller or equal, greater than, greater or equal, equal, not equal. Try it!

```
In [35]: x = 3
         y = 5

In [36]: x > y

Out[36]: False
```

We can assign the truth value of a comparison operation to a new variable name:

```
In [37]: z = x > y

In [38]: z

Out[38]: False

In [39]: type(z)

Out[39]: bool
```

Logical operators are the following: `and`, `or`, and `not`. They work just like English (with the added bonus of being always consistent, not like English speakers!). A logical expression with `and` is `True` if both operands are true, and one with `or` is `True` when either operand is true. And the keyword `not` always negates the expression that follows.

Let's do some examples:

```
In [40]: a = 5
         b = 3
         c = 10

In [41]: a > b and b > c

Out[41]: False
```

Remember that the logical operator `and` is `True` only when both operands are `True`. In the case above the first operand is `True` but the second one is `False`.

If we try the `or` operation using the same operands we should get a `True`.

```
In [42]: a > b or b > c

Out[42]: True
```

And the negation of the second operand results in ...

```
In [43]: not b > c

Out[43]: True
```

What if we negate the second operand in the `and` operation above?

Note: Be careful with the order of logical operations. The order of precedence in logic is:

1. Negation
2. And
3. Or

If you don't rememeber this, make sure to use parentheses to indicate the order you want.

Exercise: What is happening in the case below? Play around with logical operators and try some examples.

```
In [44]: a > b and not b > c

Out[44]: True
```

4 What we've learned

- Using the `print()` function. The concept of *function*.
- Using Python as a calculator.
- Concepts of variable, type, assignment.
- Special variables: `True`, `False`, `None`.
- Supported operations, logical operations.
- Reading error messages.

5 References

Throughout this course module, we will be drawing from the following references:

1. *Effective Computation in Physics: Field Guide to Research with Python* (2015). Anthony Scopatz & Kathryn D. Huff. O'Reilly Media, Inc.
2. *Python for Everybody: Exploring Data Using Python 3* (2016). Charles R. Severance. PDF available
3. *Think Python: How to Think Like a Computer Scientist* (2012). Allen Downey. Green Tea Press. PDF available

Recommended Readings

- "Yes, Python is Slow, and I Don't Care" by Nick Humrich, on Hackernoon. (Skip the part on microservices, which is a bit specialized, and continue after the photo of moving car lights.)
- "Why I Push for Python", by Prof. Lorena A. Barba (2014). This blog post got a bit of interest over at Hacker News.

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!

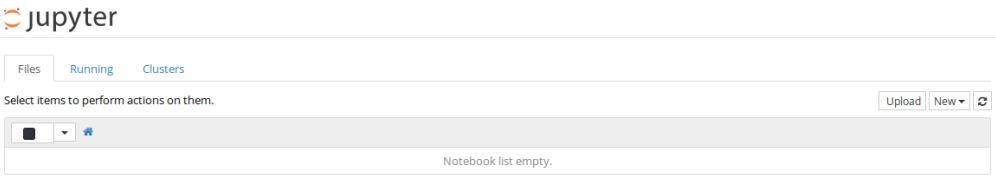
2 Working in Jupyter

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
```

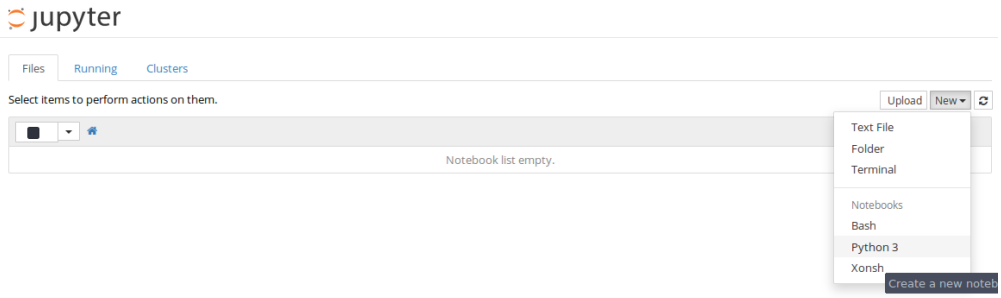


Screenshot of the Jupyter dashboard, open in the browser.

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.

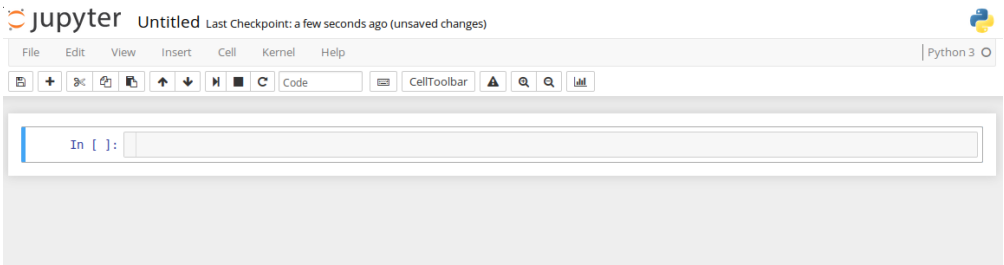
Note: Don't close the terminal window where you launched Jupyter (while you're still working on Jupyter). If you need to do other tasks on the command line, open a new terminal window.

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'
```

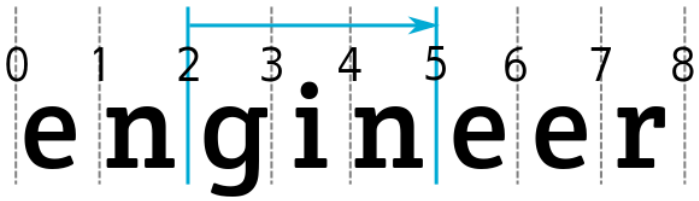
You can skip the `start` index, if you want to slice from the beginning of the string, and you can skip the `end` of a slice, indicating you want to go all the way to the end of the string. For example, if we want to grab the word `'world'` from `my_string`, we could do the following:

```
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

Lesson 3: Strings and lists in action

After completing Lesson 1 and Lesson 2 of this course in *"Engineering Computations*," here we have a full example using all that you've learned.

You may be wondering why we're dedicating the first lessons of the course to playing around with strings and lists. *"Engineering computations involve numbers, formulas and equations!"*, you may be thinking. The reason is that this course assumes no programming experience at all, so we want to get everyone used to Python first, without adding the extra complexity of number-crunching. The idea is to get acquainted with programming constructs first, applying them to situations that involve no mathematics ...for now!

1 Play with the MAE bulletin

We are going to play with text from a file containing a copy of the MAE Bulletin for 2017-2018. We'll create different lists to allow us to locate the title, credits and description of a course based on the course code.

The data file for this example should be located in a folder named `data`, two levels above the location of this lesson—if you copied the course materials as they were stored. If you have the data tile elsewhere, you should edit the full path below.

We'll start by reading the data file into the Jupyter notebook, then we'll clean the data a bit, and finally work out ways to play with it.

2 Read data from a file

We know that we have a data file, and we'd like to read its contents into the Jupyter notebook. Usually, it's a good idea to first peek into the file, to see what its contents look like. This also gives us a chance to teach you a pretty neat trick.

Recall that code cells in a Jupyter notebook can handle any valid **IPython** statement. Well, IPython is able to do a bit more than just Python: it can also run any system command. If you know some Unix, this can be very useful—for example, you could list all the files in the working directory (your location in the computer's file system).

To execute a system (a.k.a., shell) command, you prepend `!`—a "bang." The command we need is `head`: it prints out the first few lines of a file.

Our data folder is found two directories up from the lessons (this Jupyter notebook): in Unix, going *up* a directory is indicated by two dots; so we need to have `../../data/` before the file name,`mae_bulletin.txt`, as part of the path.

Let's call `head` with a bang:

```
In [1]: !head ../../data/mae_bulletin.txt
```

```
MAE 1004. Engineering Drawing and Computer Graphics. 0-3 Credits.

Introduction to technical drawing, including use of instruments, lettering,
geometric construction, sketching, orthographic projection, section view,
dimensioning, tolerancing, and pictorial drawing. Introduction to computer
graphics, including topics covered in manual drawing and computer-aided
drafting.   (Fall and spring).

MAE 2117. Engineering Computations. 3 Credits.

Numerical methods for engineering applications. Round-off errors and discretization
errors. Methods for solving systems of linear equations, root finding, curve fitting,
numerical Fourier transform, and data approximation. Numerical differentiation and
integration and numerical solution of differential equations. Computer applications.
Prerequisite: MATH 1232. (Fall, Every Year).

MAE 2124. Linear Systems Analysis for Robotics. 3 Credits.
```

That looks good! The next step is to *open the file* and save its contents into a Python variable that we can use.

The `open()` function with the file name as a *string* argument (note the quotes) returns a Python file object. We have several options next, and you should definitely read the documentation about reading and writing files.

If you use the `read()` file method, you get a (big) string with all the file's contents. If you use instead the `readlines()` method, you get a list of strings, where each string contains one line in the file. Yet another option is to use the `list()` function to create a list of lines from the file contents.

```
In [2]: mae_bulletin_file = open('../../data/mae_bulletin.txt')
```

```
In [3]: mae_bulletin_text = mae_bulletin_file.readlines()
```

```
In [4]: type(mae_bulletin_text)
```

```
Out[4]: list
```

```
In [5]: len(mae_bulletin_text)
```

```
Out[5]: 431
```

3 Cleaning and organizing text data

When manipulating text data, one of the typical actions is to get rid of extra white spaces and lines. Here, we'll remove the spaces at the beginning and end of each line.

Note that there are also some blank lines: we'll skip them. The goal is to get two new lists: one with the course ID line, and another with the course descriptions.

Study the following block of code:

```
In [6]: courses = []
        descriptions = []

        for line in mae_bulletin_text:
            line = line.strip()            #Remove white spaces
            if line == '':                 #Skip the empty lines
                continue
            elif line.startswith('MAE'):
                courses.append(line)       #Save lines that start with MAE in list
            else:
                descriptions.append(line)  #Save descriptions in other list
```

Be sure to also visit the documentation for string methods, to get a glimpse of all the things you can do with Python! Here, we used the `strip()` method to get rid of leadign and trailing spaces, and we also used `startswith()` to identify the course ID lines.

Let's check what we ended up with by printing a few items in each list:

```
In [7]: #print first 5 elements of courses
        print(courses[0:5])
```

```
['MAE 1004. Engineering Drawing and Computer Graphics. 0-3 Credits.', 'MAE 2117.
Engineering Computations. 3 Credits.', 'MAE 2124. Linear Systems Analysis for
Robotics. 3 Credits.', 'MAE 2131. Thermodynamics. 3 Credits.', 'MAE 2170. History
and Impact of the US Patent System. 3 Credits.']
```

```
In [8]: #print first 3 elements of descriptions
        print(descriptions[0:3])
```

```
['Introduction to technical drawing, including use of instruments, lettering,
geometric construction, sketching, orthographic projection, section view,
dimensioning, tolerancing, and pictorial drawing. Introduction to computer graphics,
including topics covered in manual drawing and computer-aided drafting.  (Fall and
spring).', 'Numerical methods for engineering applications. Round-off errors and
discretization errors. Methods for solving systems of linear equations, root finding,
curve fitting, numerical Fourier transform, and data approximation. Numerical
differentiation and integration and numerical solution of differential equations.
Computer applications. Prerequisite: MATH 1232. (Fall, Every Year).', 'Properties of
linear systems. Mathematical modeling of dynamic systems. State space, state variables,
and their selection. Linearization of non-linear behavior. Matrix functions. Solution
of state equations in the time domain and using transformations. System stability and
```

```
frequency response.']
```

We should also check that both lists are the of the same length!

```
In [9]: print(len(courses))
        print(len(descriptions))
```

```
108
108
```

4 Separate `courses` list into course id, title, and credits

We may want to have the information of the course id, title, and credits in separate lists. Here's how we could do that:

```
In [10]: course_id = []
         course_title = []
         course_credits = []

         for course in courses:
             course_info = course.split('. ')
             course_id.append(course_info[0])
             course_title.append(course_info[1])
             course_credits.append(course_info[2])
```

Note that we split using the string `'. '` (period + space) to avoid having an extra white space at the beginning of each string in the lists for course title and credits.

Let's print out the first elements of the new lists.

```
In [11]: print(course_id[0:5])
```

```
['MAE 1004', 'MAE 2117', 'MAE 2124', 'MAE 2131', 'MAE 2170']
```

```
In [12]: print(course_title[0:5])
```

```
['Engineering Drawing and Computer Graphics', 'Engineering Computations',
'Linear Systems Analysis for Robotics', 'Thermodynamics', 'History and
Impact of the US Patent System']
```

```
In [13]: print(course_credits[0:5])
```

```
['0-3 Credits.', '3 Credits.', '3 Credits.', '3 Credits.', '3 Credits.']
```

5 Tracking course information

The lists we have created are aligned: that is each item on the same index location corresponds to the same course. So by finding the location of a course id, we can access all the other information.

We use the `index()` method to find the index of the course id and track the rest of the info. Try it out!

```
In [14]: course_id.index('MAE 3190')
```

```
Out[14]: 17
```

```
In [15]: print(course_id[17])
         print(course_title[17])
         print(course_credits[17])
         print(descriptions[17])
```

```
MAE 3190
Analysis and Synthesis of Mechanisms
3 Credits.
Kinematics and dynamics of mechanisms. Displacements, velocities, and accelerations
in linkage, cam, and gear systems by analytical, graphical, and computer methods.
Synthesis of linkages to meet prescribed performance requirements. Prerequisite:
APSC 2058.  (Fall).
```

6 How many courses have prerequisites?

Here's an idea: we could look for all the courses that have prerequisites using a `for` statement. In this case, we'll iterate over the *index* of the elements in the list `descriptions`.

It gives us a chance to introduce to you a most helpful Python object: `range`: it creates a sequence of numbers in arithmetic progression to iterate over. With a single argument, `range(N)` will create a sequence of length `N` starting at zero: `0, 1, 2, ..., N-1`.

```
In [16]: for i in range(4):
             print(i)
```

```
0
1
2
3
```

The funny thing with `range` is that it's created on-the-fly, when iterating over it. So it's not really a list, although for most practical purposes, it behaves like one.

A typical way to use it is with an argument that's output by the `len()` function. Study this block of code:

```
In [17]: course_with_pre = []

         for i in range(len(descriptions)):
             if 'Prerequisite' in descriptions[i]:
                 course_with_pre.append(course_id[i])
```

Now we have a list named `course_with_pre` that contains the id of all the courses that have prerequisites.

```
In [18]: print(course_with_pre)
```

```
['MAE 2117', 'MAE 2131', 'MAE 3120', 'MAE 3126', 'MAE 3128', 'MAE 3134', 'MAE 3145',
'MAE 3155', 'MAE 3162', 'MAE 3166W', 'MAE 3167W', 'MAE 3187', 'MAE 3190', 'MAE 3191',
'MAE 3192', 'MAE 3193', 'MAE 3195', 'MAE 3197', 'MAE 4129', 'MAE 4149', 'MAE 4157',
'MAE 4163', 'MAE 4168', 'MAE 4172', 'MAE 4182', 'MAE 4193', 'MAE 4194', 'MAE 4198',
'MAE 4199', 'MAE 6201', 'MAE 6207', 'MAE 6220', 'MAE 6221', 'MAE 6222', 'MAE 6223',
'MAE 6224', 'MAE 6225', 'MAE 6226', 'MAE 6227', 'MAE 6228', 'MAE 6229', 'MAE 6230',
'MAE 6231', 'MAE 6232', 'MAE 6233', 'MAE 6234', 'MAE 6237', 'MAE 6238', 'MAE 6239',
'MAE 6240', 'MAE 6241', 'MAE 6243', 'MAE 6244', 'MAE 6245', 'MAE 6246', 'MAE 6247',
'MAE 6249', 'MAE 6251', 'MAE 6252', 'MAE 6253', 'MAE 6254', 'MAE 6255', 'MAE 6257',
'MAE 6258', 'MAE 6260', 'MAE 6261', 'MAE 6262', 'MAE 6270', 'MAE 6271', 'MAE 6274',
'MAE 6276', 'MAE 6280', 'MAE 6281', 'MAE 6282', 'MAE 6283', 'MAE 6284', 'MAE 6286',
'MAE 6287', 'MAE 6288', 'MAE 6290', 'MAE 6291', 'MAE 6292', 'MAE 8350', 'MAE 8351',
'MAE 8352']
```

Exercise:

1. Save in a new list named `course_with_cor` all the courses that have a corequisite, and print out the list.
2. Using a `for` statement and `if-elif-else` statements, separate the courses that are offered in the Fall semester, those offered in the Spring semester, those offered in both semesters, and those that don't specify a semester. Create 4 lists: `fall_and_spring`, `fall`, `spring` and `not_spec`.

To check your answers uncomment the following lines by deleting the # symbol and running the cell. If there is no output you got it right!

```
In [19]:
```

7 What we've learned

- System commands in a code cell start with a bang (`!`).
- Opening a text file and saving its contents into a string or list variable.
- Cleaning up text data using string methods.
- Manipulating text into lists.

Lesson 4: Play with NumPy Arrays

Welcome to **Lesson 4** of the first course module in *"Engineering Computations*." You have come a long way!

Remember, this course assumes no coding experience, so the first three lessons were focused on creating a foundation with Python programming constructs using essentially *no mathematics*. The previous lessons are:

- Lesson 1: Interacting with Python
- Lesson 2: Play with data in Jupyter
- Lesson 3: Strings and lists in action

In engineering applications, most computing situations benefit from using *arrays*: they are sequences of data all of the *same type*. They behave a lot like lists, except for the constraint in the type of their elements. There is a huge efficiency advantage when you know that all elements of a sequence are of the same type—so equivalent methods for arrays execute a lot faster than those for lists.

The Python language is expanded for special applications, like scientific computing, with **libraries**. The most important library in science and engineering is **NumPy**, providing the *n-dimensional array* data structure (a.k.a, `ndarray`) and a wealth of functions, operations and algorithms for efficient linear-algebra computations.

In this lesson, you'll start playing with NumPy arrays and discover their power. You'll also meet another widely loved library: **Matplotlib**, for creating two-dimensional plots of data.

1 Importing libraries

First, a word on importing libraries to expand your running Python session. Because libraries are large collections of code and are for special purposes, they are not loaded automatically when you launch Python (or IPython, or Jupyter). You have to import a library using the `import` command. For example, to import **NumPy**, with all its linear-algebra goodness, we enter:

```
import numpy
```

Once you execute that command in a code cell, you can call any NumPy function using the dot notation, prepending the library name. For example, some commonly used functions are:

- `numpy.linspace()`
- `numpy.ones()`
- `numpy.zeros()`

- `numpy.empty()`
- `numpy.copy()`

Follow the links to explore the documentation for these very useful NumPy functions!

Warning: You will find *a lot* of sample code online that uses a different syntax for importing. They will do:

```
import numpy as np
```

All this does is create an alias for `numpy` with the shorter string `np`, so you then would call a **NumPy** function like this: `np.linspace()`. This is just an alternative way of doing it, for lazy people that find it too long to type `numpy` and want to save 3 characters each time. For the not-lazy, typing `numpy` is more readable and beautiful.

We like it better like this:

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

2 Creating arrays

To create a NumPy array from an existing list of (homogeneous) numbers, we call `numpy.array()`, like this:

```
In [2]: numpy.array([3, 5, 8, 17])
```

```
Out[2]: array([ 3,  5,  8, 17])
```

NumPy offers many ways to create arrays in addition to this. We already mentioned some of them above.

Play with `numpy.ones()` and `numpy.zeros()`: they create arrays full of ones and zeros, respectively. We pass as an argument the number of array elements we want.

```
In [3]: numpy.ones(5)
```

```
Out[3]: array([ 1.,  1.,  1.,  1.,  1.])
```

```
In [4]: numpy.zeros(3)
```

```
Out[4]: array([ 0.,  0.,  0.])
```

Another useful one: `numpy.arange()` gives an array of evenly spaced values in a defined interval.

Syntax:

```
numpy.arange(start, stop, step)
```

where `start` by default is zero, `stop` is not inclusive, and the default for `step` is one. Play with it!

```
In [5]: numpy.arange(4)
```

```
Out[5]: array([0, 1, 2, 3])
```

```
In [6]: numpy.arange(2, 6)
```

```
Out[6]: array([2, 3, 4, 5])

In [7]: numpy.arange(2, 6, 2)

Out[7]: array([2, 4])

In [8]: numpy.arange(2, 6, 0.5)

Out[8]: array([ 2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5])
```

`numpy.linspace()` is similar to `numpy.arange()`, but uses number of samples instead of a step size. It returns an array with evenly spaced numbers over the specified interval.

Syntax:

```
numpy.linspace(start, stop, num)
```

`stop` is included by default (it can be removed, read the docs), and `num` by default is 50.

```
In [9]: numpy.linspace(2.0, 3.0)

Out[9]: array([ 2.        ,  2.02040816,  2.04081633,  2.06122449,  2.08163265,
                2.10204082,  2.12244898,  2.14285714,  2.16326531,  2.18367347,
                2.20408163,  2.2244898 ,  2.24489796,  2.26530612,  2.28571429,
                2.30612245,  2.32653061,  2.34693878,  2.36734694,  2.3877551 ,
                2.40816327,  2.42857143,  2.44897959,  2.46938776,  2.48979592,
                2.51020408,  2.53061224,  2.55102041,  2.57142857,  2.59183673,
                2.6122449 ,  2.63265306,  2.65306122,  2.67346939,  2.69387755,
                2.71428571,  2.73469388,  2.75510204,  2.7755102 ,  2.79591837,
                2.81632653,  2.83673469,  2.85714286,  2.87755102,  2.89795918,
                2.91836735,  2.93877551,  2.95918367,  2.97959184,  3.        ])

In [10]: len(numpy.linspace(2.0, 3.0))

Out[10]: 50

In [11]: numpy.linspace(2.0, 3.0, 6)

Out[11]: array([ 2. ,  2.2,  2.4,  2.6,  2.8,  3. ])

In [12]: numpy.linspace(-1, 1, 9)

Out[12]: array([-1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

3 Array operations

Let's assign some arrays to variable names and perform some operations with them.

```
In [13]: x_array = numpy.linspace(-1, 1, 9)
```

Now that we've saved it with a variable name, we can do some computations with the array. E.g., take the square of every element of the array, in one go:

```
In [14]: y_array = x_array**2
         print(y_array)
```

```
[ 1.      0.5625  0.25    0.0625  0.      0.0625  0.25    0.5625  1.    ]
```

We can also take the square root of a positive array, using the `numpy.sqrt()` function:

```
In [15]: z_array = numpy.sqrt(y_array)
         print(z_array)
```

```
[ 1.    0.75  0.5   0.25  0.    0.25  0.5   0.75  1.  ]
```

Now that we have different arrays `x_array`, `y_array` and `z_array`, we can do more computations, like add or multiply them. For example:

```
In [16]: add_array = x_array + y_array
         print(add_array)
```

```
[ 0.     -0.1875 -0.25   -0.1875  0.      0.3125  0.75    1.3125  2.    ]
```

Array addition is defined element-wise, like when adding two vectors (or matrices). Array multiplication is also element-wise:

```
In [17]: mult_array = x_array * z_array
         print(mult_array)
```

```
[-1.     -0.5625 -0.25   -0.0625  0.      0.0625  0.25    0.5625  1.    ]
```

We can also divide arrays, but you have to be careful not to divide by zero. This operation will result in a `nan` which stands for *Not a Number*. Python will still perform the division, but will tell us about the problem.

Let's see how this might look:

```
In [18]: x_array / y_array
```

```
//anaconda/envs/future/lib/python3.5/site-packages/ipykernel/__main__.py:1:
RuntimeWarning: invalid value encountered in true_divide
  if __name__ == '__main__':
```

```
Out[18]: array([-1.        , -1.33333333, -2.        , -4.        ,         nan,
                 4.        ,  2.        ,  1.33333333,  1.        ])
```

4 Multidimensional arrays

4.1 2D arrays

NumPy can create arrays of N dimensions. For example, a 2D array is like a matrix, and is created from a nested list as follows:

```
In [19]: array_2d = numpy.array([[1, 2], [3, 4]])
         print(array_2d)
```

```
[[1 2]
 [3 4]]
```

2D arrays can be added, subtracted, and multiplied:

```
In [20]: X = numpy.array([[1, 2], [3, 4]])
         Y = numpy.array([[1, -1], [0, 1]])
```

The addition of these two matrices works exactly as you would expect:

```
In [21]: X + Y
```

```
Out[21]: array([[2, 1],
                [3, 5]])
```

What if we try to multiply arrays using the `'*'`operator?

```
In [22]: X * Y
```

```
Out[22]: array([[ 1, -2],
                [ 0,  4]])
```

The multiplication using the `'*'` operator is element-wise. If we want to do matrix multiplication we use the `'@'` operator:

```
In [23]: X @ Y
```

```
Out[23]: array([[1, 1],
                [3, 1]])
```

Or equivalently we can use `numpy.dot()`:

```
In [24]: numpy.dot(X, Y)
```

```
Out[24]: array([[1, 1],
                [3, 1]])
```

4.2 3D arrays

Let's create a 3D array by reshaping a 1D array. We can use `numpy.reshape()`, where we pass the array we want to reshape and the shape we want to give it, i.e., the number of elements in each dimension.

Syntax

```
numpy.reshape(array, newshape)
```

For example:

```
In [25]: a = numpy.arange(24)
```

```
In [26]: a_3D = numpy.reshape(a, (2, 3, 4))
         print(a_3D)
```

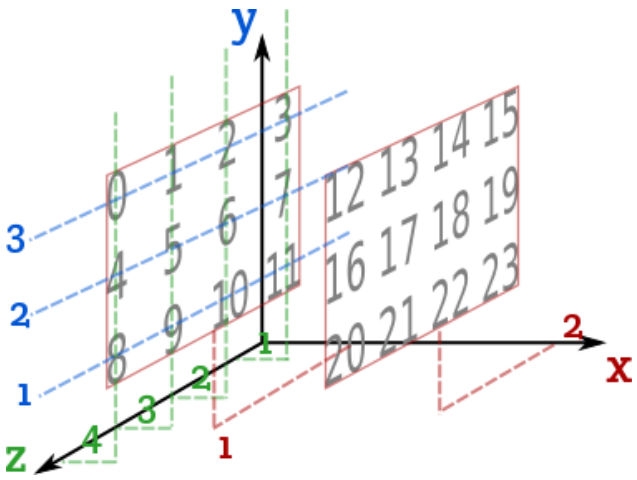
```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

We can check for the shape of a NumPy array using the function `numpy.shape()`:

```
In [27]: numpy.shape(a_3D)
```

```
Out[27]: (2, 3, 4)
```

Visualizing the dimensions of the `a_3D` array can be tricky, so here is a diagram that will help you to understand how the dimensions are assigned: each dimension is shown as a coordinate axis. For a 3D array, on the "x axis", we have the sub-arrays that themselves are two-dimensional (matrices). We have two of these 2D sub-arrays, in this case; each one has 3 rows and 4 columns. Study this sketch carefully, while comparing with how the array `a_3D` is printed out above.



When we have multidimensional arrays, we can access slices of their elements by slicing on each dimension. This is one of the advantages of using arrays: we cannot do this with lists.

Let's access some elements of our 2D array called `X`.

```
In [28]: X
```

```
Out[28]: array([[1, 2],
                [3, 4]])
```

```
In [29]: # Grab the element in the 1st row and 1st column
         X[0, 0]
```

```
Out[29]: 1
```

```
In [30]: # Grab the element in the 1st row and 2nd column
         X[0, 1]
```

```
Out[30]: 2
```

Exercises: From the X array:

1. Grab the 2nd element in the 1st column.
2. Grab the 2nd element in the 2nd column.

Play with slicing on this array:

```
In [31]: # Grab the 1st column
         X[:, 0]
```

```
Out[31]: array([1, 3])
```

When we don't specify the start and/or end point in the slicing, the symbol `':'` means "all". In the example above, we are telling NumPy that we want all the elements from the 0-th index in the second dimension (the first column).

```
In [32]: # Grab the 1st row
         X[0, :]
```

```
Out[32]: array([1, 2])
```

Exercises: From the X array:

1. Grab the 2nd column.
2. Grab the 2nd row.

Let's practice with a 3D array.

```
In [33]: a_3D
```

```
Out[33]: array([[[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]],

                [[12, 13, 14, 15],
                 [16, 17, 18, 19],
                 [20, 21, 22, 23]]])
```

If we want to grab the first column of both matrices in our `a_3D` array, we do:

```
In [34]: a_3D[:, :, 0]
```

```
Out[34]: array([[ 0,  4,  8],
                [12, 16, 20]])
```

The line above is telling NumPy that we want:

- first `':'` : from the first dimension, grab all the elements (2 matrices).
- second `':'`: from the second dimension, grab all the elements (all the rows).
- `'0'` : from the third dimension, grab the first element (first column).

If we want the first 2 elements of the first column of both matrices:

```
In [35]: a_3D[:, 0:2, 0]
```

```
Out[35]: array([[ 0,  4],
                [12, 16]])
```

Below, from the first matrix in our `a_3D` array, we will grab the two middle elements (5,6):

```
In [36]: a_3D[0, 1, 1:3]
```

```
Out[36]: array([5, 6])
```

Exercises: From the array named `a_3D`:

1. Grab the two middle elements (17, 18) from the second matrix.
2. Grab the last row from both matrices.
3. Grab the elements of the 1st matrix that exclude the first row and the first column.
4. Grab the elements of the 2nd matrix that exclude the last row and the last column.

5 NumPy == Fast and Clean!

When we are working with numbers, arrays are a better option because the NumPy library has built-in functions that are optimized, and therefore faster than vanilla Python. Especially if we have big arrays. Besides, using NumPy arrays and exploiting their properties makes our code more readable.

For example, if we wanted to add element-wise the elements of 2 lists, we need to do it with a `for` statement. If we want to add two NumPy arrays, we just use the addtion `'+'` symbol!

Below, we will add two lists and two arrays (with random elements) and we'll compare the time it takes to compute each addition.

5.1 Element-wise sum of a Python list

Using the Python library `random`, we will generate two lists with 100 pseudo-random elements in the range [0,100), with no numbers repeated.

```
In [37]: #import random library
         import random
```

```
In [38]: lst_1 = random.sample(range(100), 100)
         lst_2 = random.sample(range(100), 100)
```

```
In [39]: #print first 10 elements
         print(lst_1[0:10])
         print(lst_2[0:10])
```

```
[69, 21, 55, 9, 12, 57, 75, 81, 15, 17]
[57, 29, 94, 67, 51, 71, 78, 55, 41, 72]
```

We need to write a `for` statement, appending the result of the element-wise sum into a new list we call `result_lst`.

For timing, we can use the IPython "magic" `%%time`. Writing at the beginning of the code cell the command `%%time` will give us the time it takes to execute all the code in that cell.

```
In [40]: %%time
         res_lst = []
         for i in range(100):
             res_lst.append(lst_1[i] + lst_2[i])
```

```
CPU times: user 36 µs, sys: 1 µs, total: 37 µs
Wall time: 38.9 µs
```

```
In [41]: print(res_lst[0:10])
```

```
[126, 50, 149, 76, 63, 128, 153, 136, 56, 89]
```

5.2 Element-wise sum of NumPy arrays

In this case, we generate arrays with random integers using the NumPy function `numpy.random.randint()`. The arrays we generate with this function are not going to be like the lists: in this case we'll have 100 elements in the range [0, 100) but they can repeat. Our goal is to compare the time it takes to compute addition of a *list* or an *array* of numbers, so all that matters is that the arrays and the lists are of the same length and type (integers).

```
In [42]: arr_1 = numpy.random.randint(0, 100, size=100)
         arr_2 = numpy.random.randint(0, 100, size=100)
```

```
In [43]: #print first 10 elements
         print(arr_1[0:10])
         print(arr_2[0:10])
```

```
[31 13 72 30 13 29 34 64 26 56]
[ 3 57 63 51 35 75 56 59 86 50]
```

Now we can use the `%%time` cell magic, again, to see how long it takes NumPy to compute the element-wise sum.

```
In [44]: %%time
         arr_res = arr_1 + arr_2
```

```
CPU times: user 20 µs, sys: 1 µs, total: 21 µs
Wall time: 26 µs
```

Notice that in the case of arrays, the code not only is more readable (just one line of code), but it is also faster than with lists. This time advantage will be larger with bigger arrays/lists.

(Your timing results may vary to the ones we show in this notebook, because you will be computing in a different machine.)

Exercise

1. Try the comparison between lists and arrays, using bigger arrays; for example, of size 10,000.
2. Repeat the analysis, but now computing the operation that raises each element of an array/list to the power two. Use arrays of 10,000 elements.

6 Time to Plot

You will love the Python library **Matplotlib**! You'll learn here about its module `pyplot`, which makes line plots.

We need some data to plot. Let's define a NumPy array, compute derived data using its square, cube and square root (element-wise), and plot these values with the original array in the x-axis.

```
In [45]: xarray = numpy.linspace(0, 2, 41)
         print(xarray)
```

```
[ 0.    0.05  0.1   0.15  0.2   0.25  0.3   0.35  0.4   0.45  0.5   0.55
  0.6   0.65  0.7   0.75  0.8   0.85  0.9   0.95  1.    1.05  1.1   1.15
  1.2   1.25  1.3   1.35  1.4   1.45  1.5   1.55  1.6   1.65  1.7   1.75
  1.8   1.85  1.9   1.95  2.  ]
```

```
In [46]: pow2 = xarray**2
         pow3 = xarray**3
         pow_half = numpy.sqrt(xarray)
```

To plot the resulting arrays as a function of the orginal one (`xarray`) in the x-axis, we need to import the module `pyplot` from **Matplotlib**.

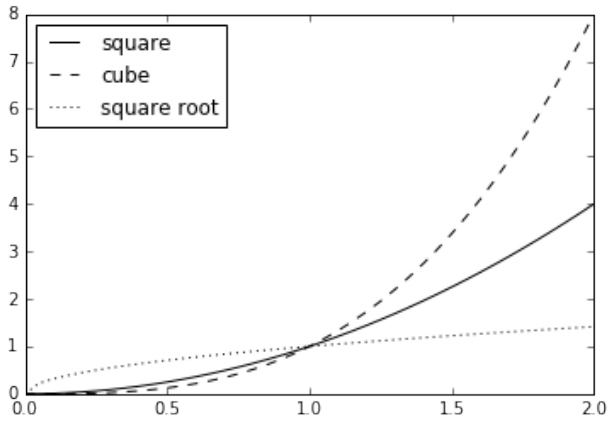
```
In [47]: from matplotlib import pyplot
         %matplotlib inline
```

The command `%matplotlib inline` is there to get our plots inside the notebook (instead of a pop-up window, which is the default behavior of `pyplot`).

We'll use the `pyplot.plot()` function, specifying the line color (`'k'` for black) and line style (`'-'`, `'--'` and `':'` for continuous, dashed and dotted line), and giving each line a label. Note that the values for `color`, `linestyle` and `label` are given in quotes.

```
In [48]: #Plot x^2
         pyplot.plot(xarray, pow2, color='k', linestyle='-', label='square')
         #Plot x^3
         pyplot.plot(xarray, pow3, color='k', linestyle='--', label='cube')
         #Plot sqrt(x)
         pyplot.plot(xarray, pow_half, color='k', linestyle=':', label='square root')
         #Plot the legends in the best location
         pyplot.legend(loc='best')
```

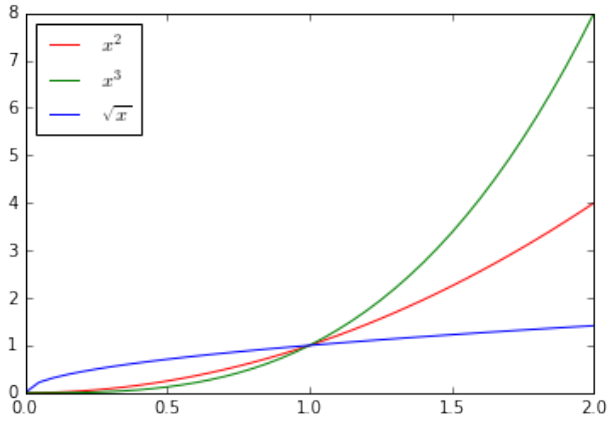
```
Out[48]: <matplotlib.legend.Legend at 0x10b224ac8>
```



To illustrate other features, we will plot the same data, but varying the colors instead of the line style. We'll also use LaTeX syntax to write formulas in the labels. If you want to know more about LaTeX syntax, there is a quick guide to LaTeX available online.

Adding a semicolon (';') to the last line in the plotting code block prevents that ugly output, like `<matplotlib.legend.Legend at 0x7f8c83cc7898>`. Try it.

```
In [49]: #Plot x^2
         pyplot.plot(xarray, pow2, color='red', linestyle='-', label='$x^2$')
         #Plot x^3
         pyplot.plot(xarray, pow3, color='green', linestyle='-', label='$x^3$')
         #Plot sqrt(x)
         pyplot.plot(xarray, pow_half, color='blue', linestyle='-', label='$\sqrt{x}$')
         #Plot the legends in the best location
         pyplot.legend(loc='best');
```



That's very nice! By now, you are probably imagining all the great stuff you can do with Jupyter notebooks, Python and its scientific libraries **NumPy** and **Matplotlib**. We just saw an introduction

to plotting but we will keep learning about the power of **Matplotlib** in the next lesson.

If you are curious, you can explore all the beautiful plots you can make by browsing the Matplotlib gallery.

Exercise: Pick two different operations to apply to the `xarray` and plot them the resulting data in the same plot.

7 What we've learned

- How to import libraries
- Multidimensional arrays using NumPy
- Accessing values and slicing in NumPy arrays
- `%%time` magic to time cell execution.
- Performance comparison: lists vs NumPy arrays
- Basic plotting with `pyplot`.

8 References

1. *Effective Computation in Physics: Field Guide to Research with Python* (2015). Anthony Scopatz & Kathryn D. Huff. O'Reilly Media, Inc.
2. *Numerical Python: A Practical Techniques Approach for Industry*. (2015). Robert Johansson. Appress.
3. "The world of Jupyter"—a tutorial. Lorena A. Barba - 2016

Lesson 5: Linear regression with real data

1 Earth temperature over time

In this lesson, we will apply all that we've learned (and more) to analyze real data of Earth temperature over time.

Is global temperature rising? How much? This is a question of burning importance in today's world!

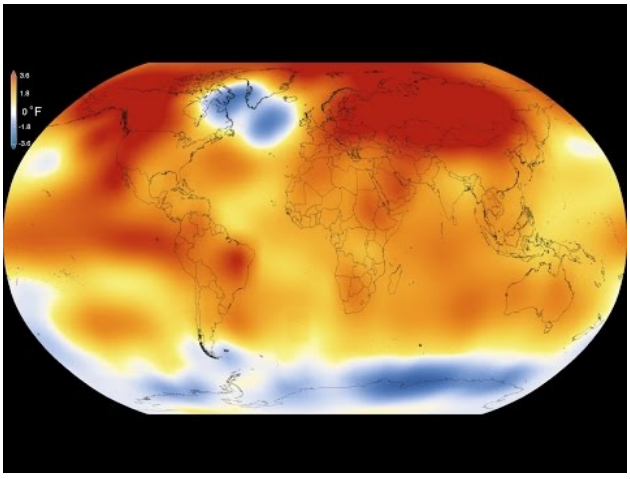
Data about global temperatures are available from several sources: NASA, the National Climatic Data Center (NCDC) and the University of East Anglia in the UK. Check out the University Corporation for Atmospheric Research (UCAR) for an in-depth discussion.

The NASA Goddard Space Flight Center is one of our sources of global climate data. They produced the video below showing a color map of the changing global surface **temperature anomalies** from 1880 to 2015.

The term global temperature anomaly means the difference in temperature with respect to a reference value or a long-term average. It is a very useful way of looking at the problem and in many ways better than absolute temperature. For example, a winter month may be colder than average in Washington DC, and also in Miami, but the absolute temperatures will be different in both places.

```
In [1]: from IPython.display import YouTubeVideo
        YouTubeVideo('gGOzHVUQCw0')
```

```
Out[1]:
```



How would we go about understanding the *trends* from the data on global temperature?

The first step in analyzing unknown data is to generate some simple plots using **Matplotlib**. We are going to look at the temperature-anomaly history, contained in a file, and make our first plot to explore this data.

We are going to smooth the data and then we'll fit a line to it to find a trend, plotting along the way to see how it all looks.

Let's get started!

2 Step 1: Read a data file

We took the data from the NOAA (National Oceanic and Atmospheric Administration) webpage. Feel free to play around with the webpage and analyze data on your own, but for now, let's make sure we're working with the same dataset.

We have a file named `land_global_temperature_anomaly-1880-2016.csv` in our `data` folder. This file contains the year on the first column, and averages of land temperature anomaly listed sequentially on the second column, from the year 1880 to 2016. We will load the file, then make an initial plot to see what it looks like.

Note: If you downloaded this notebook alone, rather than the full collection for this course, you may not have the data file on the location we assume below. In that case, you can download the data if you add a code cell, and execute the following code in it:

```
from urllib.request import urlretrieve
URL = 'http://go.gwu.edu/engcomp1data5?accessType=DOWNLOAD'
urlretrieve(URL, 'land_global_temperature_anomaly-1880-2016.csv')
```

The data file will be downloaded to your working directory, and you will then need to remove the path information, i.e., the string `'../../data/'`, from the definition of the variable `fname` below.

Let's start by importing NumPy.

```
In [2]: import numpy
```

To load our data from the file, we'll use the function `numpy.loadtxt()`, which lets us immediately save the data into NumPy arrays. (We encourage you to read the documentation for details on how the function works.) Here, we'll save the data into the arrays `year` and `temp_anomaly`.

```
In [3]: fname = '../../data/land_global_temperature_anomaly-1880-2016.csv'

        year, temp_anomaly = numpy.loadtxt(fname, delimiter=',', skiprows=5, unpack=True)
```

Exercise Inspect the data by printing `year` and `temp_anomaly`.

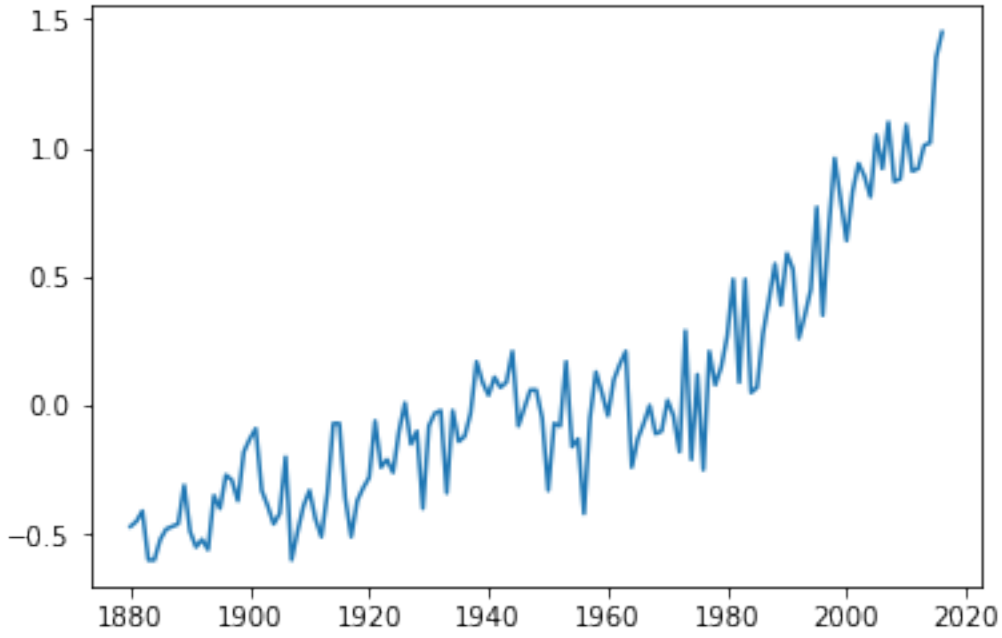
3 Step 2: Plot the data

Let's first load the **Matplotlib** module called `pyplot`, for making 2D plots. Remember that to get the plots inside the notebook, we use a special "magic" command, `%matplotlib inline`:

```
In [4]: from matplotlib import pyplot
        %matplotlib inline
```

The `plot()` function of the `pyplot` module makes simple line plots. We avoid that stuff that appeared on top of the figure, that `Out[x]: [< ...>]` ugliness, by adding a semicolon at the end of the plotting command.

```
In [5]: pyplot.plot(year, temp_anomaly);
```



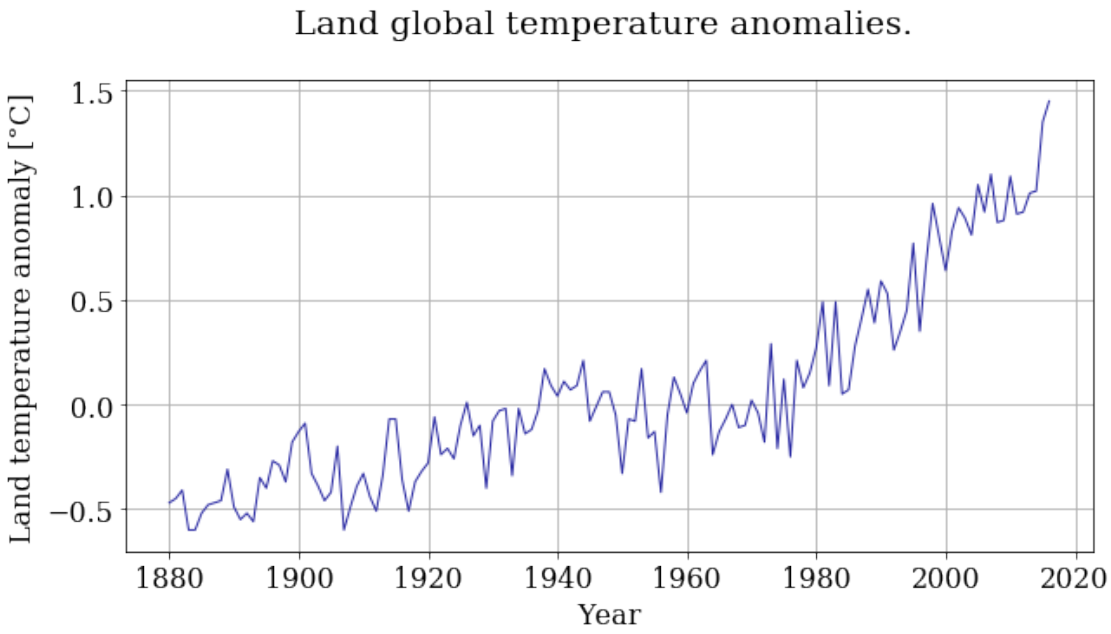
Now we have a line plot, but if you see this plot without any information you would not be able to figure out what kind of data it is! We need labels on the axes, a title and why not a better color, font and size of the ticks. **Publication quality** plots should always be your standard for plotting. How you present your data will allow others (and probably you in the future) to better understand your work.

We can customize the style of our plots using **Matplotlib**'s `rcParams`. It lets us set some style options that apply for all the plots we create in the current session. Here, we'll make the font of a specific size and type. You can also customize other parameters like line width, color, and so on (check out the documentation).

```
In [6]: from matplotlib import rcParams
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 16
```

We'll redo the same plot, but now we'll add a few things to make it prettier and **publication quality**. We'll add a title, label the axes and, show a background grid. Study the commands below and looks at the result!

```
In [7]: #You can set the size of the figure by doing:
        pyplot.figure(figsize=(10,5))

        #Plotting
        pyplot.plot(year, temp_anomaly, color='#2929a3', linestyle='-', linewidth=1)
        pyplot.title('Land global temperature anomalies. \n')
        pyplot.xlabel('Year')
        pyplot.ylabel('Land temperature anomaly [°C]')
        pyplot.grid();
```



Better, no? Feel free to play around with the parameters and see how the plot changes. There's nothing like trial and error to get the hang of it.

4 Step 3: Least-squares linear regression

In order to have an idea of the general behavior of our data, we can find a smooth curve that (approximately) fits the points. We generally look for a curve that's simple (e.g., a polynomial), and does not reproduce the noise that's always present in experimental data.

Let $f(x)$ be the function that we'll fit to the $n+1$ data points: (x_i, y_i), $i = 0, 1, ..., n$:

$$f(x) = f(x; a_0, a_1, ..., a_m)$$

The notation above means that f is a function of x, with $m+1$ variable parameters $a_0, a_1, ..., a_m$, where $m < n$. We need to choose the form of $f(x)$ a priori, by inspecting the experimental data and knowing something about the phenomenon we've measured. Thus, curve fitting consists of two steps:

1. Choosing the form of $f(x)$.
2. Computing the parameters that will give us the "best fit" to the data.

5 What is the "best" fit?

When the noise in the data is limited to the y-coordinate, it's common to use a **least-squares fit**, which minimizes the function

$$S(a_0, a_1, ..., a_m) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 \tag{1}$$

with respect to each a_j. We find the values of the parameters for the best fit by solving the following equations:

$$\frac{\partial S}{\partial a_k} = 0, \quad k = 0, 1, ..., m. \tag{2}$$

Here, the terms $r_i = y_i - f(x_i)$ are called residuals: they tell us the discrepancy between the data and the fitting function at x_i.

Take a look at the function S: what we want to minimize is the sum of the squares of the residuals. The equations (2) are generally nonlinear in a_j and might be difficult to solve. Therefore, the fitting function is commonly chosen as a linear combination of specified functions $f_j(x)$,

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + ... + a_m f_m(x)$$

which results in equations (2) being linear. In the case that the fitting function is polynomial, we have have $f_0(x) = 1,\ f_1(x) = x,\ f_2(x) = x^2$, and so on.

6 Linear regression

When we talk about linear regression we mean "fitting a straight line to the data." Thus,

$$f(x) = a_0 + a_1 x \tag{3}$$

In this case, the function that we'll minimize is:

$$S(a_0, a_1) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 = \sum_{i=0}^{n} (y_i - a_0 - a_1 x_i)^2 \tag{4}$$

Equations (2) become:

$$\frac{\partial S}{\partial a_0} = \sum_{i=0}^{n} -2(y_i - a_0 - a_1 x_i) = 2\left[a_0(n+1) + a_1 \sum_{i=0}^{n} x_i - \sum_{i=0}^{n} y_i\right] = 0 \tag{5}$$

$$\frac{\partial S}{\partial a_1} = \sum_{i=0}^{n} -2(y_i - a_0 - a_1 x_i)x_i = 2\left[a_0 \sum_{i=0}^{n} x_i + a_1 \sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i\right] = 0 \tag{6}$$

Let's divide both equations by $2(n+1)$ and rearrange terms.

Rearranging (6) and (7):

$$2\left[a_0(n+1) + a_1 \sum_{i=0}^{n} x_i - \sum_{i=0}^{n} y_i\right] = 0$$

$$\frac{a_0(n+1)}{n+1} + a_1 \frac{\sum_{i=0}^{n} x_i}{n+1} - \frac{\sum_{i=0}^{n} y_i}{n+1} = 0 \tag{7}$$

$$\tag{8}$$

$$a_0 = \bar{y} - a_1 \bar{x} \tag{9}$$

where $\bar{x} = \frac{\sum_{i=0}^{n} x_i}{n+1}$ and $\bar{y} = \frac{\sum_{i=0}^{n} y_i}{n+1}$.

Rearranging (7):

$$2\left[a_0 \sum_{i=0}^{n} x_i + a_1 \sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i\right] = 0 \tag{10}$$

$$a_0 \sum_{i=0}^{n} x_i + a_1 \sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i = 0 \tag{11}$$

$$\tag{12}$$

Now, if we replace a_0 from equation (8) into (9) and rearrange terms:

$$(\bar{y} - a_1 \bar{x}) \sum_{i=0}^{n} x_i + a_1 \sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i = 0$$

Replacing the definitions of the mean values into the equation,

$$\left[\frac{1}{n+1}\sum_{i=0}^{n} y_i - \frac{a_1}{n+1}\sum_{i=0}^{n} x_i\right]\sum_{i=0}^{n} x_i + a_1\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i = 0$$
$$\frac{1}{n+1}\sum_{i=0}^{n} y_i \sum_{i=0}^{n} x_i - \frac{a_1}{n+1}\sum_{i=0}^{n} x_i \sum_{i=0}^{n} x_i + a_1\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i = 0$$

Leaving everything in terms of $\bar{x}$,

$$\sum_{i=0}^{n} y_i \bar{x} - a_1\sum_{i=0}^{n} x_i \bar{x} + a_1\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i = 0$$

Grouping the terms that have a_1 on the left-hand side and the rest on the right-hand side:

$$a_1\left[\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i \bar{x}\right] = \sum_{i=0}^{n} x_i y_i - \sum_{i=0}^{n} y_i \bar{x}$$
$$a_1\sum_{i=0}^{n}(x_i^2 - x_i\bar{x}) = \sum_{i=0}^{n}(x_i y_i - y_i\bar{x})$$
$$a_1\sum_{i=0}^{n} x_i(x_i - \bar{x}) = \sum_{i=0}^{n} y_i(x_i - \bar{x})$$

Finally, we get that:

$$a_1 = \frac{\sum_{i=0}^{n} y_i(x_i - \bar{x})}{\sum_{i=0}^{n} x_i(x_i - \bar{x})} \tag{13}$$

Then our coefficients are:

$$a_1 = \frac{\sum_{i=0}^{n} y_i(x_i - \bar{x})}{\sum_{i=0}^{n} x_i(x_i - \bar{x})} \quad , \quad a_0 = \bar{y} - a_1\bar{x} \tag{14}$$

7 Let's fit!

Let's now fit a straight line through the temperature-anomaly data, to see the trend over time. We'll use least-squares linear regression to find the slope and intercept of a line

$$y = a_1 x + a_0$$

that fits our data.

In our case, the x-data corresponds to `year`, and the y-data is `temp_anomaly`. To calculate our coefficients with the formula above, we need the mean values of our data. Sine we'll need to compute the mean for both `x` and `y`, it could be useful to write a custom Python *function* that computes the mean for any array, and we can then reuse it.

It is good coding practice to *avoid repeating* ourselves: we want to write code that is reusable, not only because it leads to less typing but also because it reduces errors. If you find yourself doing the same calculation multiple times, it's better to encapsulate it into a *function*.

Remember the *key concept* from Lesson 1: A function is a compact collection of code that executes some action on its arguments.

Once *defined*, you can *call* a function as many times as you want. When we *call* a function, we execute all the code inside the function. The result of the execution depends on the *definition* of the function and on the values that are *passed* into it as *arguments*. Functions might or might not *return* values in their last operation.

The syntax for defining custom Python functions is:

```
def function_name(arg_1, arg_2, ...):
    '''
    docstring: description of the function
    '''
    <body of the function>
```

The **docstring** of a function is a message from the programmer documenting what he or she built. Docstrings should be descriptive and concise. They are important because they explain (or remind) the intended use of the function to the users. You can later access the docstring of a function using the function `help()` and passing the name of the function. If you are in a notebook, you can also prepend a question mark `'?'` before the name of the function and run the cell to display the information of a function.

Try it!

```
In [8]: ?print
```

Using the `help` function instead:

```
In [9]: help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Let's define a custom function that calculates the mean value of any array. Study the code below carefully.

```
In [10]: def mean_value(array):
             """ Calculate the mean value of an array

             Arguments
             ---------
             array: Numpy array

             Returns
             -------
             mean: mean value of the array
             """
             sum_elem = 0
             for element in array:
                     sum_elem += element # this is the same as sum_elem = sum_elem + element

             mean = sum_elem / len(array)

             return mean
```

Once you execute the cell above, the function`mean_value()` becomes available to use on any argument of the correct type. This function works on arrays of any length. We can try it now with our data.

```
In [11]: year_mean = mean_value(year)
         print(year_mean)
```

```
1948.0
```

```
In [12]: temp_anomaly_mean = mean_value(temp_anomaly)
         print(temp_anomaly_mean)
```

```
0.0526277372263
```

Neat! You learned how to write a Python function, and we wrote one for computing the mean value of an array of numbers. We didn't have to, though, because NumPy has a built-in function to do just what we needed: `numpy.mean()`.

Exercise Calculate the mean of the `year` and `temp_anomaly` arrays using the NumPy built-in function, and compare the results with the ones obtained using our custom `mean_value` function.

```
In [ ]:
```

Now that we have mean values, we can compute our coefficients by following equations (12). We first calculate a_1 and then use that value to calculate a_0.

Our coefficients are:

$$a_1 = \frac{\sum_{i=0}^{n} y_i(x_i - \bar{x})}{\sum_{i=0}^{n} x_i(x_i - \bar{x})} \quad , \quad a_0 = \bar{y} - a_1\bar{x}$$

We already calculated the mean values of the data arrays, but the formula requires two sums over new derived arrays. Guess what, NumPy has a built-in function for that: `numpy.sum()`. Study the code below.

```
In [13]: a_1 = numpy.sum(temp_anomaly*(year - year_mean)) / numpy.sum(year*(year - year_mean))

In [14]: print(a_1)

0.0103702839435


In [15]: a_0 = temp_anomaly_mean - a_1*year_mean

In [16]: print(a_0)

-20.1486853847
```

Exercise Write a function that computes the coefficients, call the function to compute them and compare the result with the values we obtained before. As a hint, we give you the structure that you should follow:

```
def coefficients(x, y, x_mean, y_mean):
    """
    Write docstrings here
    """

    a_1 =
    a_0 =

    return a_1, a_0
```

We now have the coefficients of a linear function that best fits our data. With them, we can compute the predicted values of temperature anomaly, according to our fit. Check again the equations above: the values we are going to compute are $f(x_i)$.

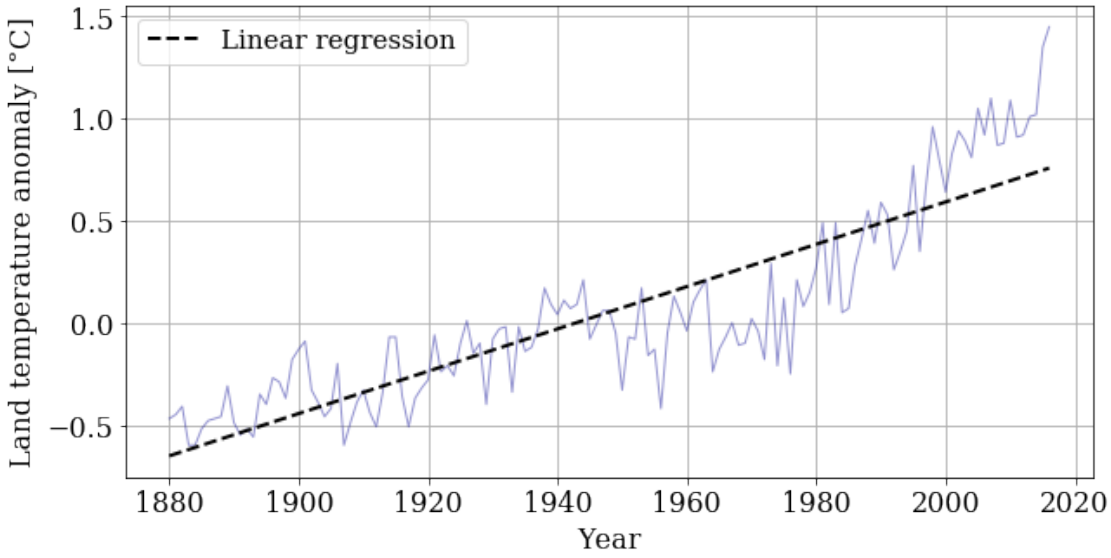
Let's call `reg` the array obtined from evaluating $f(x_i)$ for all years.

```
In [17]: reg = a_0 + a_1 * year
```

With the values of our linear regression, we can plot it on top of the original data to see how they look together. Study the code below.

```
In [18]: pyplot.figure(figsize=(10, 5))
```

```
pyplot.plot(year, temp_anomaly, color='#2929a3', linestyle='-', linewidth=1,
alpha=0.5)
pyplot.plot(year, reg, 'k--', linewidth=2, label='Linear regression')
pyplot.xlabel('Year')
pyplot.ylabel('Land temperature anomaly [°C]')
pyplot.legend(loc='best', fontsize=15)
pyplot.grid();
```



8 Step 4: Apply regression using NumPy

Above, we coded linear regression from scratch. But, guess what: we didn't have to because NumPy has built-in functions that do what we need!

Yes! Python and NumPy are here to help! With `polyfit()`, we get the slope and y-intercept of the line that best fits the data. With `poly1d()`, we can build the linear function from its slope and y-intercept.

Check it out:

```
In [19]: # First fit with NumPy, then name the coefficients obtained a_1n, a_0n:
         a_1n, a_0n = numpy.polyfit(year, temp_anomaly, 1)

         f_linear = numpy.poly1d((a_1n, a_0n))
```

```
In [20]: print(a_1n)
```

```
0.0103702839435
```

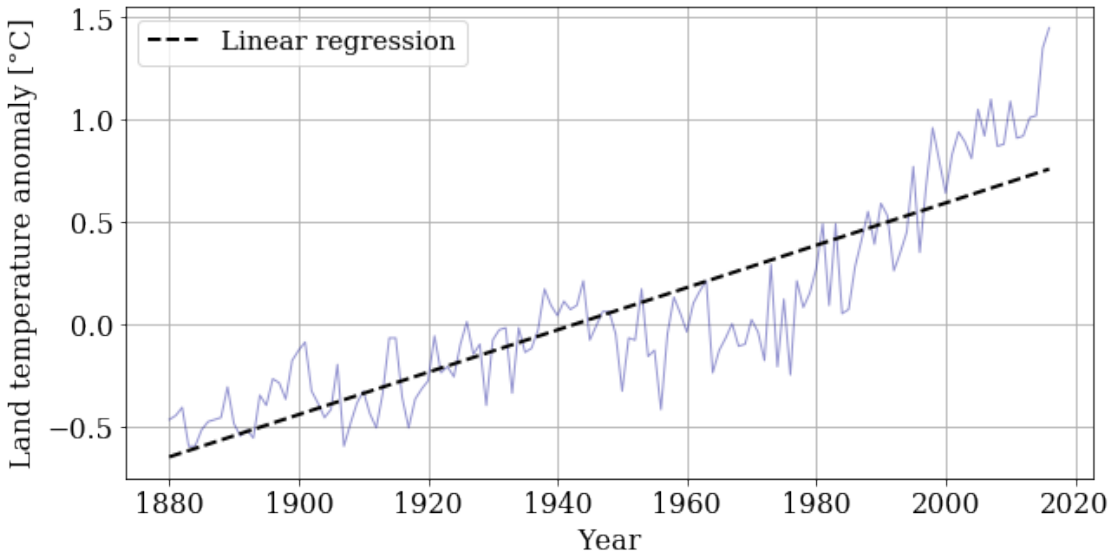
```
In [21]: print(a_0n)
```

```
-20.1486853847
```

```
In [22]: print(f_linear)
```

```
0.01037 x - 20.15
```

```
In [23]: pyplot.figure(figsize=(10, 5))

         pyplot.plot(year, temp_anomaly, color='#2929a3', linestyle='-', linewidth=1,
          alpha=0.5)
         pyplot.plot(year, f_linear(year), 'k--', linewidth=2, label='Linear regression')
         pyplot.xlabel('Year')
         pyplot.ylabel('Land temperature anomaly [°C]')
         pyplot.legend(loc='best', fontsize=15)
         pyplot.grid();
```



9 "Split regression"

If you look at the plot above, you might notice that around 1970 the temperature starts increasing faster that the previous trend. So maybe one single straight line does not give us a good-enough fit.

What if we break the data in two (before and after 1970) and do a linear regression in each segment?

To do that, we first need to find the position in our `year` array where the year 1970 is located.

Thankfully, NumPy has a function called `numpy.where()` that can help us. We pass a condition and `numpy.where()` tells us where in the array the condition is `True`.

```
In [24]: numpy.where(year==1970)
```

```
Out[24]: (array([90]),)
```

To split the data, we use the powerful instrument of *slicing* with the colon notation. Remember that a colon between two indices indicates a range of values from a `start` to an `end`. The rule is that `[start:end]` includes the element at index `start` but excludes the one at index `end`. For example, to grab the first 3 years in our `year` array, we do:

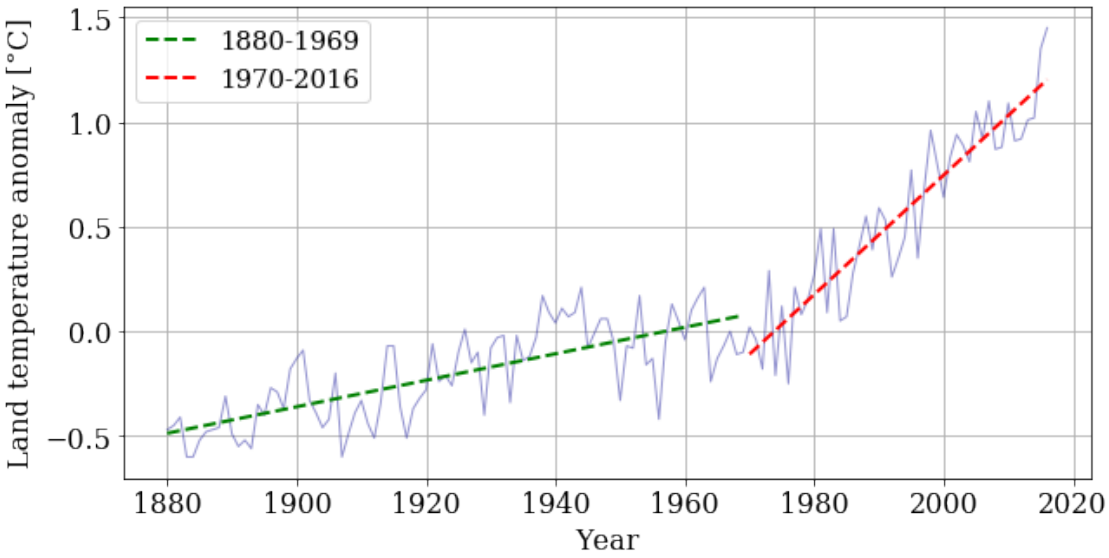
```
In [25]: year[0:3]
```

```
Out[25]: array([ 1880.,  1881.,  1882.])
```

Now we know how to split our data in two sets, to get two regression lines. We need two slices of the arrays `year` and `temp_anomaly`, which we'll save in new variable names below. After that, we complete two linear fits using the helpful NumPy functions we learned above.

```
In [26]: year_1 , temp_anomaly_1 = year[0:90], temp_anomaly[0:90]
         year_2 , temp_anomaly_2 = year[90:], temp_anomaly[90:]

         m1, b1 = numpy.polyfit(year_1, temp_anomaly_1, 1)
         m2, b2 = numpy.polyfit(year_2, temp_anomaly_2, 1)

         f_linear_1 = numpy.poly1d((m1, b1))
         f_linear_2 = numpy.poly1d((m2, b2))
```

```
In [27]: pyplot.figure(figsize=(10, 5))

         pyplot.plot(year, temp_anomaly, color='#2929a3', linestyle='-', linewidth=1,
         alpha=0.5)
         pyplot.plot(year_1, f_linear_1(year_1), 'g--', linewidth=2, label='1880-1969')
         pyplot.plot(year_2, f_linear_2(year_2), 'r--', linewidth=2, label='1970-2016')

         pyplot.xlabel('Year')
         pyplot.ylabel('Land temperature anomaly [°C]')
         pyplot.legend(loc='best', fontsize=15)
         pyplot.grid();
```



We have two different curves for two different parts of our data set. A little problem with this and is that the end point of our first regression doesn't match the starting point of the second regression. We did this for the purpose of learning, but it is not rigorously correct. We'll fix in in the next course module when we learn more about different types of regression.

10 We learned:

- Making our plots more beautiful
- Defining and calling custom Python functions
- Applying linear regression to data
- NumPy built-ins for linear regression
- The Earth is warming up!!!

11 References

1. *Essential skills for reproducible research computing* (2017). Lorena A. Barba, Natalia C. Clementi, Gilbert Forsyth.
2. *Numerical Methods in Engineering with Python 3* (2013). Jaan Kiusalaas. Cambridge University Press.
3. *Effective Computation in Physics: Field Guide to Research with Python* (2015). Anthony Scopatz & Kathryn D. Huff. O'Reilly Media, Inc.