lab_00

August 6, 2018

1 Geographic Data Science - Lab 01

Dani Arribas-Bel

2 Computational tools for Geographic Data Science

In this tutorial we will introduce the main tools we will be working with throughout the rest of the course. Although very basic and seemingly abstract, everything showed here will become the basis on top of which we will build more sophisticated (and fun) tasks. But, before, let us get to know the tools that will give us data super-powers.

2.1 Open Source

This course will introduce you to a series of computational tools that make the life of the Data Scientist possible, and much easier. All of them are open-source, which means the creators of these pieces of software have made available the source code for people to use it, study it, modify it, and re-distribute it. This has allowed a large eco-system that today represents the best option for scientific computing, and is used widely both in industry and academia. Thanks to this, this course can be taught with entirely freely available tools that you can install in any of your computers.

If you want to learn more about open-source and free software, here are a few links:

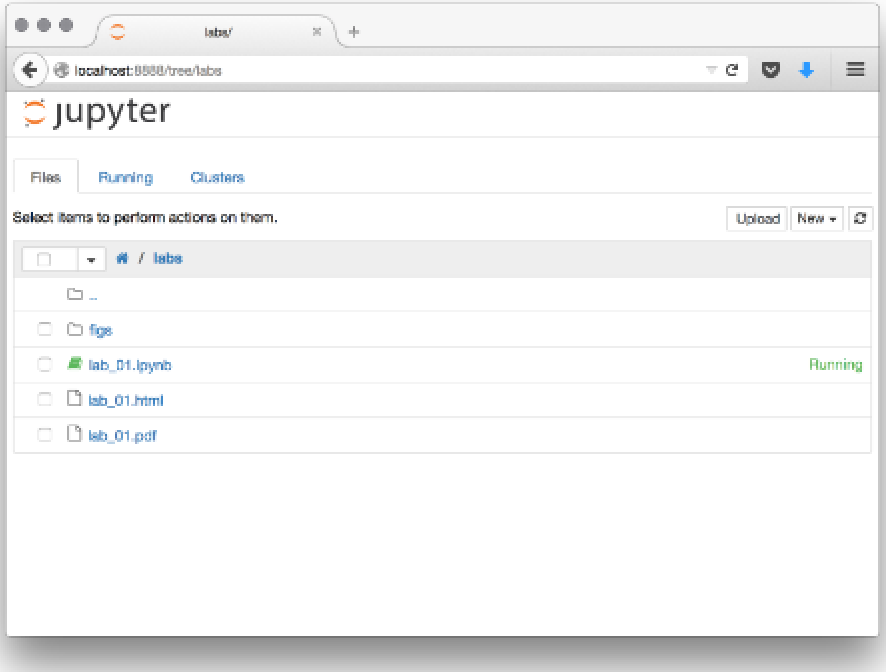
- **[Video]**: brief explanation of open source.
- **[Book]** The Cathedral and the Bazaar: classic book, freely available, that documents the benefits and history of open-source software.

2.2 `Jupyter` Notebook

The main computational tool you will be using during this course is the Jupyter notebook. Notebooks are a convenient way to thread text, code and the output it produces in a simple file that you can then share, edit and modify. You can think of notebooks as the Word document of Data Scientists, just better.

2.2.1 Start a notebook

In order to begin a notebook session, you need to do it from what is called the *command line*, a terminal window that allows you to interact with your computer through written commands. This is how you can fire up a terminal:



Select items to perform actions on them.

/ labs

..

figs

lab_01.ipynb Running

lab_01.html

lab_01.pdf

Jupyter home

- If you are on a **Windows** computer, you can start the "Anaconda Command Prompt" from the Start menu.
- On a **Mac**, fire up the Terminal.app utility.
- In **Linux**, use any of the terminals available.

Then type the following command

```
> source activate gds
```

NOTE: ignore `source` if you are on Windows and simply type `activate gds`.
Then launch `Jupyter` by typing on the same terminal:

```
> jupyter notebook
```

This should bring up a browser window with a home page that looks more or less like this (although with a different list of files probably):

Navigate until the folder where you have placed the `lab_01.ipynb` file for this tutorial and click on it. This will open the notebook on a different tab. You are now on the interactive version of the notebook!

When you are finished with the session, you can save the notebook with `File -> Save and Checkpoint`. Everything you do on the notebook (text, code and output) is saved into an `.ipynb` file that you can open later, share, etc.

2.2.2 Cells

The main building block of notebooks are cells. These are chunks of the same time of content which can be cut, pasted, and moved around in a notebook. Cells can be of two types:

- **Text**, like the one where this is written.
- **Code**, like the following one below:

```
In [1]: # This is a code cell
```

You can create a new cell by clicking `Insert` -> `Cell Above/Below` in the top menu. By default, this will be a code cell, but you can change that on the `Cell` -> `Cell Type` menu. Choose `Markdown` for a text cell. Once a new cell is created, you can edit it by clicking on it, which will create the cursor bar inside for you to start typing.

Pro tip!: cells can also be created with shortcuts. If you press `<escape>` and then `b` (`a`), a new cell will be created below (above). There is a whole bunch of shortcuts you can explore by pressing `<escape>` and `h` (press `<escape>` again to leave the help).

2.2.3 Code and its output

A particularly useful feature of notebooks is that you can save, in the same place, the code you use to generate any output (tables, figures, etc.). As an example, the cell below contains a snipet of Python that returns a printed statement. This statement is then printed below and recorded in the notebook as output:

```
In [2]: print("Hello world!!!")
```

```
Hello world!!!
```

Note also how the notebook has automatic syntax highlighting support for Python. This makes the code much more readable and understandable. More on Python below.

2.2.4 Markdown

Text cells in a notebook use the Github Flavored Markdown markup language. This means you can write plain text with some rules and the notebook renders a more visually appealing version of it. Let's see some examples:

- **BOLD**:

  `This is **bold**.`
  Is rendered:
  This is **bold**.

- **ITALIC**:

  `This is *italic*.`
  Is rendered:
  This is *italic*.

- **LISTS**:

You can create unnumbered lists:

```
* Item 1
* Item 2
* ...
```

Which will produce:

- Item 1
- Item 2
- ...

Or you can create numbered lists:

```
1. First element
1. Second element
1. ...
```

And get:

1. First element
2. Second element
3. ...

Note that you don't have to write the actual number of the element, just using `1.` always produces a numbered list.

You can also nest lists:

```
* First unnumbered element, which can be split into:

    1. One numbered element
    2. Another numbered element

* Second element.
* ...
```

- First unnumbered element, which can be split into:

    1. One numbered element
    2. Another numbered element

- Second element.

- ...

This creates many oportunities to combine things nicely.

- **LINKS**

```
You can easily create hyperlinks, for example to [WikiPedia](https://www.wikipedia.org/).
```

You can easily create hyperlinks, for example to WikiPedia.

- **HEADINGS**: including # before a line causes it to render a heading.

```
# This is Header 1
```

Turns into:

3 This is Header 1

```
## This is Header 2
```

Turns into:

3.1 This is Header 2

```
### This is Header 3
```

Turns into:

3.1.1 This is Header 3

And so on…

You can see a more in detail introduction in the following links:

https://help.github.com/articles/markdown-basics/

https://help.github.com/articles/github-flavored-markdown/

3.1.2 Rich content in a notebook

Notebooks can also include rich content from the web. For that, we need to import the `display` module:

```
In [3]: import IPython.display as display
```

This makes available additional functionality that allows us to embed rich content. For example, we can include a YouTube clip easily by passing it's ID:

```
In [4]: display.YouTubeVideo('iinQDhsdE9s')
```

```
Out[4]:
```

Or we can pass standard HTML code:

```
In [5]: display.HTML("""<table>
        <tr>
        <th>Header 1</th>
        <th>Header 2</th>
        </tr>
        <tr>
        <td>row 1, cell 1</td>
        <td>row 1, cell 2</td>
        </tr>
        <tr>
        <td>row 2, cell 1</td>
        <td>row 2, cell 2</td>
        </tr>
        </table>""")
```

```
Out[5]: <IPython.core.display.HTML object>
```

Note that this opens the door for including a large number of elements from the web, as an `iframe` is also allowed. For example, interactive maps can be included:

```
In [6]: osm = """
        <iframe width="425" height="350" frameborder="0" scrolling="no" marginheight="0" marginw
        """
        display.HTML(osm)

Out[6]: <IPython.core.display.HTML object>
```

Or sound content:

```
In [7]: sound = '''
        <iframe width="100%" height="450" scrolling="no" frameborder="no" src="https://w.soundcl
        '''
        display.HTML(sound)

Out[7]: <IPython.core.display.HTML object>
```

A more thorough exploration of them is available in this notebook.

3.1.3 Exercise to work on your own

Try to reproduce, using markdown and the different tools the notebook affords you, the following WikiPedia entry:

> https://en.wikipedia.org/wiki/Chocolate_chip_cookie_dough_ice_cream

```
In [8]: display.IFrame('https://en.wikipedia.org/wiki/Chocolate_chip_cookie_dough_ice_cream',
                       700, 500)

Out[8]: <IPython.lib.display.IFrame at 0x10dc9f8d0>
```

Pay special attention to getting the bold, italics, links, headlines and lists correctly formated, but don't worry too much about the overall layout. Bonus if you manage to insert the image as well!

3.2 Python

The main bulk of the course relies on the Python programming language. Python is a high-level programming language widely used today. To give a couple of examples of its relevance, it is underlying most of the Dropbox systems, but also heavily used to control satellites at NASA. A great deal of Science is also done in Python, from research in astronomy at UC Berkley, to courses in economics by Nobel Prize Professors.

This course uses Python because it has emerged as one of the main and most solid options for Data Science, together with other free alternatives such as R. Python is widely used for data processing and analysis both in academia and in industry. There is a vibrant and growing scientific community (example and example), working at both universities and companies, that supports and enhances its capabilities for data analysis by providing new and refining existing extensions (a.ka.a. libraries, see below). In the geospatial world, Python is also very widely adopted, being the selected language for scripting in both ArcGIS and QGIS. All of this means that, whether you are thinking of continuing in Higher Education or trying to find a job in industry, Python will be an importan asset that employers will significantly value.

Being a high-level language means that the code can be "dynamically interpreted", which means it is run on-the-fly without the need to be compiled. This is in contrast to "low-level" programming languages, which first need to be converted into machine code (i.e. compiled) before they can be run. With Python, one does not need to worry about compilation and can just write code, evaluate, fix it, re-evaluate it, etc. in a quick cycle, making it a very productive tool. The rest of this tutorial covers some of the basic elements of the language, from conventions like how to comment your code, to the basic data structures available.

The rest of the tutorial is partly inspired by the introductory lesson in this course by Lorena Barba's group.

3.2.1 Python libraries

The standard Python language includes some data structures (e.g. lists, dictionaries, etc. See below) and allows many basic operations (e.g. sum, product, etc.). For example, right out of the box, and without any further action needed, you can use Python as a calculator:

```
In [9]: 3 + 5
```

```
Out[9]: 8
```

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

```
Out[10]: 0.6666666666666666
```

```
In [11]: (3 + 5) * 2. / 3
```

```
Out[11]: 5.333333333333333
```

However, the strength of Python as a data analysis tool comes from the extensions provided separately that add functionality and provide access to much more sophisticated data structures and functions. These come in the form of packages, or libraries, that once installed need to be imported into a session.

In this course, we will be using many of the core libraries of what has been called the "PyData stack", the set of libraries that make Python a full-fledge system for Data Science. We will introduce them gradually as we need them for particular tasks but, for now, let us have a look at the foundational library, numpy (short for numerical Python). Importing it is simple:

```
In [12]: import numpy as np # we rename it in the session as `np` by convention
```

Note how we import it *and* rename it in the session, from `numpy` to `np`, which is shorter and more convenient.

Note also how comments work in Python: everything in a line *after* the # sign is ignored by Python when it evaluates the code. This allows you to insert comments that Python will ignore but that can help you and others better understand the code.

Once imports are out of the way, let us start exploring what we can do with `numpy`. One of the easiest tasks is to create a sequence of numbers:

```
In [13]: seq = np.arange(10)
         seq
```

```
Out[13]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first thing to note is that, in line 1, we create the sequence by calling the function `arange` and assign it to an object called `seq` (it could have been called anything else, pick your favorite) and, in line 2, we have it printed as the output of the cell.

Another interesting feature is how, since we are calling a `numpy` function called `arange` by adding `np.` in front. This is to note that the function comes explicitly from `numpy`. To find out how necessary this is, you can try generating the sequence without `np`:

```
In [14]: # NOTE: comment out to run the cell
         #seq = arange(10)
```

What you get instead is an error, also called a "traceback". In particular, Python is telling that it cannot find a function named `arange` in the core library. This is because that particular function is only available in `numpy`.

3.2.2 Variables

A basic feature of Python is the ability to assign a name to different "things", or objects. These can also be called sometimes "variables". We have already seen that in the example above but, to make it more explicit, let us make it even simpler. For example, an object can be a single number:

```
In [15]: a = 3
```

Or a name, also called "string":

```
In [16]: b = 'Hello World'
```

You can check what type an object is also easily:

```
In [17]: type(a)
```

```
Out[17]: int
```

`int` is short for "integer" which, roughly speaking, means an whole number. If you want to save a number with decimals, you will be using floats:

```
In [18]: c = 1.5
         type(c)
```

```
Out[18]: float
```

As mentioned, what we understand as letters in a wide sense (spaces and other signs count too) is called "strings" (`str` in short):

```
In [19]: type(b)
```

```
Out[19]: str
```

3.2.3 Help

A very handy feature of Python is the ability to access on-the-spot help for its different functions. This means that you can check what a function is supposed to do, or how to access it, right inside your Python session. Of course, this also works handsomely inside a notebook. There are a couple of ways to access the help.

Take the `numpy` function `arange` that we have used above. The easiest way to check interactively how to use it is by:

```
In [20]: np.arange?
```

As you can see, this brings up a sub-window in the browser with all the information you need.

If, for whatever reason, you needed to print that info into the notebook itself, you can do the following:

```
In [21]: help(np.arange)
```

```
Help on built-in function arange in module numpy.core.multiarray:

arange(...)
    arange([start,] stop[, step,], dtype=None)

    Return evenly spaced values within a given interval.

    Values are generated within the half-open interval ``[start, stop)``
    (in other words, the interval including `start` but excluding `stop`).
    For integer arguments the function is equivalent to the Python built-in
    `range <http://docs.python.org/lib/built-in-funcs.html>`_ function,
    but returns an ndarray rather than a list.

    When using a non-integer step, such as 0.1, the results will often not
    be consistent.  It is better to use ``linspace`` for these cases.

    Parameters
    ----------
    start : number, optional
        Start of interval.  The interval includes this value.  The default
        start value is 0.
    stop : number
        End of interval.  The interval does not include this value, except
        in some cases where `step` is not an integer and floating point
        round-off affects the length of `out`.
    step : number, optional
        Spacing between values.  For any output `out`, this is the distance
        between two adjacent values, ``out[i+1] - out[i]``.  The default
        step size is 1.  If `step` is specified, `start` must also be given.
    dtype : dtype
        The type of the output array.  If `dtype` is not given, infer the data
        type from the other input arguments.
```

```
Returns
-------
arange : ndarray
    Array of evenly spaced values.

    For floating point arguments, the length of the result is
    ``ceil((stop - start)/step)``.  Because of floating point overflow,
    this rule may result in the last element of `out` being greater
    than `stop`.

See Also
--------
linspace : Evenly spaced numbers with careful handling of endpoints.
ogrid: Arrays of evenly spaced numbers in N-dimensions.
mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples
--------
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

3.2.4 Control flow (a.k.a. `for` loops and `if` statements)

Although this does not intend to be a comprehensive introduction to computer programming or general purpose Python (check the references for that, in particular Allen Downey's book), it is important to be aware of two building blocks of almost any computer program: `for` loops and `if` statements. It is possible that you will never require them for this course, as all that is used here is based on existing methods and functions, but it is always useful to know they exist and to be able to recognize them. They can also come in very handy in cases where you some extra functionality out of standard methods. Without further ado, let us have a look and the two single most relevant tools of computer programming.

- `for` loops

These allow you to repeat a particular action or task over a sequence. As an example, you can print your name ten times without having to type it yourself every single time:

```
In [22]: for i in np.arange(10):
             print('my name')
```

```
my name
my name
my name
my name
my name
my name
my name
my name
my name
my name
```

Note a couple of features in the loop:

1. You loop *over* a sequence, in this particular case the sequence of ten numbers created by `np.arange(10)`.
2. In every step, for every element of the sequence in this case, you repeat an action. Here we are printing the same text, `my name`.
3. Although not used in this simple loop, each of the elements you loop over can be accessed inside the loop. This can be irrelevant, as in the loop above, or extremely useful, it depends on the context. For example, see a case where you use the value of the sequence in each step:

```
In [23]: for i in np.arange(10):
             print("I am at step ", i)
```

```
I am at step  0
I am at step  1
I am at step  2
I am at step  3
I am at step  4
I am at step  5
I am at step  6
I am at step  7
I am at step  8
I am at step  9
```

One more note: for convention, we are calling the element of the sequence `i`, but this could be named anything. In fact, in many cases, more meaningful names make code much more readable. For example, you could think of a re-write of the loop above as:

```
In [24]: for step in np.arange(10):
             print("I am at step ", step)
```

```
I am at step  0
I am at step  1
I am at step  2
I am at step  3
I am at step  4
```

```
I am at step  5
I am at step  6
I am at step  7
I am at step  8
I am at step  9
```

- `if` statements

We have just seen how `for` loops allow you to repeat an action over a sequence. In the case of `if` statements, these allow you to select or restrict such actions to only those cases that meet a condition(s) you specify in the statement.

For example, if you think of the loops written above, you might want to only print those that are odd, skipping those that are even:

```
In [25]: for i in np.arange(10):
             if i%2:
                 print(i)
```

```
1
3
5
7
9
```

Ignore for the moment the part `i%2`, just remember this is one way Python has to check if a number is odd. The important bit in this loop, as compared to the simpler one above, is that we are using an `if` statement to select only those candidates that meet the condition. In other words, what we are doing it looping over every number in the sequence from zero to nine (`for i in np.arange(10)`) and checking if they are even or odd (`if i%2`). If they meet the condition, they are odd, then we proceed and print them on the screen.

A full `if` statement also allows for an action to be taken if the original condition is not satisfied. This is called an "ifelse" statement. For example, you can think of a loop that prints the type of each number in a sequence:

```
In [26]: for i in np.arange(10):
             # Check if it is odd
             if i%2:
                 print(i, ' is odd')
             # If not odd (even), then do the following
             else:
                 print(i, ' is even')
```

```
0  is even
1  is odd
2  is even
3  is odd
4  is even
```

```
5  is odd
6  is even
7  is odd
8  is even
9  is odd
```

3.2.5 Data structures

The standard python you can access without importing any additional libraries contains a few core data structures that is very handy to know. Most of data analysis is done on top of other structures specifically designed for the purpose (numpy arrays and pandas dataframes, mostly. See the following sessions for more details), but some understanding of these core Python structures is very useful. In this context, we will look at three: values, lists, and dictionaries.

- **Values**: these are the most basic elements to organize data and information in Python. You can think of them as numbers (integers or floats) or words (strings). Typically, these are the elements that will be stored in lists and dictionaries.

An integer is a whole number:

```
In [27]: i = 5
         type(i)
```

```
Out[27]: int
```

A float is a number that allows for decimals:

```
In [28]: f = 5.2
         type(f)
```

```
Out[28]: float
```

Note that a float can also not have decimals and still be stored as such:

```
In [29]: fw = 5.
         type(fw)
```

```
Out[29]: float
```

However, they are different representations:

```
In [30]: f == fw
```

```
Out[30]: False
```

- **Lists**: a list is an ordered sequence of values that can be of mixed types. They are represented between squared brackets (`[]`) and, although not very efficient in memory terms, are very flexible and useful to "put things together".

For example, the following list of integers:

```
In [31]: l = [1, 2, 3, 4, 5]
         l

Out[31]: [1, 2, 3, 4, 5]

In [32]: type(l)

Out[32]: list
```

Or the following mixed one:

```
In [33]: m = ['a', 'b', 5, 'c', 6, 7]
         m

Out[33]: ['a', 'b', 5, 'c', 6, 7]
```

Lists can be queried and sliced. For example, the first element can be retrieved by:

```
In [34]: l[0]

Out[34]: 1
```

Or the second to the fourth:

```
In [35]: m[1:4]

Out[35]: ['b', 5, 'c']
```

Lists can be added:

```
In [36]: l + m

Out[36]: [1, 2, 3, 4, 5, 'a', 'b', 5, 'c', 6, 7]
```

New elements added:

```
In [37]: l.append(4)
         l

Out[37]: [1, 2, 3, 4, 5, 4]
```

Or modified:

```
In [38]: l[1]

Out[38]: 2

In [39]: l[1] = 'two'
         l[1]

Out[39]: 'two'
```

```
In [40]: l

Out[40]: [1, 'two', 3, 4, 5, 4]
```

- **Dictionaries**: dictionaries are unordered collections of “keys” and “values”. A key, which can be of any kind, is the element associated with a “value”, which can also be of any kind. Dictionaries are used when order is not important but you need fast and easy lookup. They are expressed in curly brackets, with keys and values being linked through columns.

For example, we can think of a dictionary to store a series of names and the ages of the people they represent:

```
In [41]: ages = {'Ana': 24, 'John': 20, 'Li': 27, 'Ivan': 40, 'Tali':33}
         ages

Out[41]: {'Ana': 24, 'Ivan': 40, 'John': 20, 'Li': 27, 'Tali': 33}

In [42]: type(ages)

Out[42]: dict
```

Dictionaries can then be queried and values retrieved easily by using their keys. For example, if we quickly want to know Li’s age:

```
In [43]: ages['Li']

Out[43]: 27
```

Similarly to lists, you can modify and assign new values:

```
In [44]: ages['Juan'] = 73
         ages

Out[44]: {'Ana': 24, 'Ivan': 40, 'John': 20, 'Juan': 73, 'Li': 27, 'Tali': 33}
```

Using this property, you can create entirely empty dictionaries and populate them later on:

```
In [45]: newdict = {}
         newdict['key1'] = 1
         newdict['key2'] = 2
         newdict

Out[45]: {'key1': 1, 'key2': 2}
```

3.2.6 Functions

The last part of this whirlwind tour on Python relates to functions, or more properly termed, methods. The motivation is that, so far, we have only seen how you can create Python code that, if you want to run again somewhere else, you need to copy and paste entirely. However, as we will see in more detail later in the course, one of the main reasons why you want to use Python for data analysis, instead of a point-and-click graphical interface like SPSS, for instance, is that you can easily reuse code and re-run analyses easily. Methods help us accomplish this by encapsulating snippets of code that perform a particular task and making them available to be called.

We have already *used* methods here. When we call `np.arange`, we are using one of them. Now, we will see how to *create* a method of our own that performs the specific task we want it to do. For example, let us create a very simple method to reproduce the first loop we created above:

```
In [46]: def run_simple_loop():
             for i in np.arange(10):
                 print(i)
             return None
```

Already with this simple method, there is a bunch of interesting things going on:

- First, note how we define a bit of code is a method, as oposed to plain Python: we use `def` followed by the name of our function (we have chosen `run_simple_loop`, but we anything could have done).
- Second, we append `()` after the name, and finish the line with a colon (`:`). This is necessary and will allow us to specify requirements for the function (see below).
- Third, realize that everything inside a function needs to be indented. This is a core property of Python and, although some people find it odd, it enhances readibility greatly.
- Fourth, the piece of code to do the task we want, printing the sequence of numbers, is inside the function in the same way it was outside, only properly indented.
- Fifth, we finish the method with a line starting by `return`. In this case, we follow it with `None`, but this will change as methods become more sophisticated. Essentially, this is the part of the method where you specify which elements you want it to return and save for later use.

Once we have paid attention to these elements, we can see how the method can be *called* and hence the code inside it executed:

```
In [47]: run_simple_loop()
```

```
0
1
2
3
4
5
6
7
8
9
```

This is the same way that we called `np.arange` before. Note how we do not include the colon (`:`) but simply use the name of the method followed by the parenthesis.

This is the simplest possile method you can write: you do not require anything, just executing it, and the code produces and output (the printout) but it is not saved anywhere. The rest of this section relaxes these two aspects to allow us to build more complex, but also more useful, methods.

First, you can specify "arguments" to be passed that modify the behaviour of the method. Remember how we called `np.arange` with a number that implied the length of the sequence we wanted returned. We can do the same thing in our own function. The main aspect to pay attention to in this context is that the arguments need to be variables, not particular values. Let us see a modified example of our method:

```
In [48]: def run_simple_loopX(x):
             for i in np.arange(x):
                 print(i)
             return None
```

We have replaced the fixed length of the sequence (10) by a variable named `x` that allows us to specify *any value we want* when we call the method:

```
In [49]: run_simple_loopX(3)

0
1
2


In [50]: run_simple_loopX(2)

0
1
```

Another way you can build more flexibility into a method is by allowing it to return an output of the computation. In the previous examples, the function performs a computation (i.e. printing values on the screen), but it does not return any value. This is in contrast with, for example, `np.arange` which does return an output, the sequence of values:

```
In [51]: a = np.arange(10)

In [52]: a

Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Our function does not save anything:

```
In [53]: b = run_simple_loopX(3)

0
1
2
```

```
In [54]: b
```

We can modify this using the last line of a method. For example, let us assume we want to return a sequence as long as the series of numbers we print on the screen. The method should be:

```
In [55]: def run_simple_loopXout(x):
             for i in np.arange(x):
                 print(i)
             return np.arange(x)
```

Note the main difference: instead of returning `None`, we are telling Python to return a sequence, which has the same length as the one used to specify the loop. Now, there is an alternative way of being more efficient in this method, and that is assigning the sequence to a new object and using it when necessary later on. The results are exactly the same, but there are less computations performed and, more critically, we minimize the chances of making mistakes.

```
In [56]: def run_simple_loopXout(x):
             seq = np.arange(x)
             for i in seq:
                 print(i)
             return seq
```

Either of these two new versions of the method return an output:

```
In [57]: a = run_simple_loopX(3)
         b = run_simple_loopXout(3)
```

```
0
1
2
0
1
2
```

```
In [58]: a
```

```
In [59]: b
```

```
Out[59]: array([0, 1, 2])
```

The advantage of methods, as oposed to straight code, is that they force us to think in a modular way, helping us identify exactly what it needs to be done, in what order, and what it is required. Encapsulating these atomic bits of functionality in methods allows us to write things once and flexibly use them everywhere, saving us time (and headaches) in the long run.

A final note on functions. It is important that, whenever you create a function, you include some documentation about what it expects, what it does, and what it returns. Although there are many ways of doing this, the typical convention is as follows:

```
In [60]: def run_simple_loopXout(x):
             """
             Print out the values of a sequence of certain length
             ...

             Arguments
             ---------
             x      : int
                     Length of the sequence to be printed out

             Returns
             -------
             seq    : np.array
                     Sequence of values printed out
             """
             seq = np.arange(x)
             for i in seq:
                 print(i)
             return seq
```

Documentation, as any string, are highlighted in red on a notebook. Let us have a look at the structure and components of a well-made documentation (also called "docstring"):

- It is encapsulated between triple commas (`"""`).
- Begins with a short description of what the method does. The shorter the better, the more concise, the even better.
- There is a section called "Arguments" that lists the elements that the function expects.
- Each argument is then listed, followed by its type. In this case it is an object `x` that, as we are told, needs to be an integer.
- The arguments are followed by another section that specifies what the function returns, and of what type the output is.

Documentation in this way is very useful to remember what a function does, but also to force yourself to write clearer code. A bonus is that, if you include documentation in this way, it can be checked with the standard `help` or `?` systems reviewed above:

```
In [61]: run_simple_loopXout?

In [62]: help(run_simple_loopXout)
```

```
Help on function run_simple_loopXout in module __main__:

run_simple_loopXout(x)
    Print out the values of a sequence of certain length
    ...

    Arguments
    ---------
    x      : int
```

```
            Length of the sequence to be printed out

    Returns
    -------
    seq   : np.array
            Sequence of values printed out
```

3.2.7 Exercise to work on your own

Write a properly documented function with the following behaviour:

- It has a meaningful name (i.e. it is related to its behaviour).
- It takes an integer that represents the length of a sequence that will be created.
- It creates a dictionary that is empty.
- It loops over the sequence and stores each number as the key of an entry in the dictionary, assigning either “odd” or “even” to the value, depending on the type of number.
- Returns the dictionary.

This notebook, as well as the entire set of materials, code, and data included in this course are available as an open Github repository available at" https://github.com/darribas/gds17

Geographic Data Science’17 - Lab 0 by Dani Arribas-Bel is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.