Tutorial: Introduction to Python

Dr. Thomas Gumbsch
thomas.gumbsch@bsse.ethz.ch

Dr. Carlos Oliver
carlos.oliver@bsse.ethz.ch

Dr. Sarah Brüningk
sarah.brueningk@bsse.ethz.ch

Bowen Fan
bowen.fan@bsse.ethz.ch

Prof. Dr. Karsten Borgwardt
karsten.borgwardt@bsse.ethz.ch

Introduction

Python is a programming language whose popularity has been rapidly increasing over the last few years. It has now become the *de-facto standard* for numerous disciplines, including machine learning and data mining.

Objectives

The goals of this introduction to Python are:

- to introduce Python and how to use it for simple mathematical operations
- to demonstrate how to create a Python script and call it from command line
- to demonstrate the syntax for functions, 'if' statements and loops
- to demonstrate how to import different Python modules
- to demonstrate how to perform mathematical operations on arrays

Throughout this tutorial, we will be using Python 3. There are a few key differences to previous versions, which will be denoted by a special block. Note that Python 2 has reached its 'end-of-life', so it is a good idea to learn the new version *now*.

1 Mathematical Operations

This section goes through the basics of mathematical operations in Python, defining variables, and printing strings to screen. For this section, all the commands will be performed on the command line. Python code will be shown in grey listings, and '>>>' represents the Python command prompt—so don't type it.

1.1 Addition, subtraction, multiplication

Python is great for doing quick calculations on the command line:

```
>>> 2+3
5
>>> 8-5.3
2.7
>>> 4*5
20
>>> 1+2-3*4
-9
>>> 1+2-(3*4)
-9
```

Above, you see examples of *floats* (numbers with a decimal expansion, e.g. `2.7`) and *integers* (numbers without a decimal expansion, e.g. `5`.) The second-last example shows that operations are not always carried out 'left to right', and that some operators have *precedence*. If in doubt, it is a good idea to use parentheses.

1.2 Exponentiation and square roots

Exponentiation (raising a value to a power) is performed using `**`, so $2^3 = 2 \times 2 \times 2 = 8$ will be computed by:

```
>>> 2**3
8
```

Obviously, 2^3 is different to 3^2:

```
>>> 2**3
8
>>> 3**2
9
```

Since taking a square root of a value is the same as raising that value to the power of $1/2$, e.g. $3 = \sqrt{9} = 9^{\frac{1}{2}}$, we can calculate square roots in this way:

```
>>> 9**0.5
3.0
>>> 2**0.5
1.4142135623730951
```

In order to use the `sqrt` function, you need to first *import* the `math` module, and then call the function using `math.sqrt`, as in:

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

There will be a longer discussion on modules in Section 4.

1.3 `int` and `float`

It is possible to extract only the integer part of a float using the `int`function:

```
>>> int(3.7)
3
```

It is also possible to turn an integer into a floating point number using the `float` function:

```
>>> float(2)
2.0
```

This will be useful when doing division or 'casting' types into each other.

In previous versions of Python, the division of two numbers depended on their type. Hence, dividing two integers would result in another integer. This can introduce subtle bugs, so in Python 3, the result of such an operation will *always* be a `float`.

1.4 Creating variables and printing strings

Creating variables (objects that store values) in Python is extremely easy:

```
>>> x=3
>>> x
3
>>> x+5
8
>>> y=4
>>> x*y
12
>>> x**y
81
```

Variables need not be numbers; we can also store collections of characters (words), which we call *strings*:

```
>>> today='Wednesday'
>>> today
'Wednesday'
```

Later, it will be useful to print values to screen in the middle of a string. In order to do this for an `int`, we need to use the `print` function:

```
>>> print('the value of x is:', x)
the value of x is: 3
```

Note how Python automatically converted the number into a string that can be presented. Using this syntax, it is also possible to *chain* multiple variables:

```
>>> print('two times', x, 'is:', 2*x)
two times 3 is: 6
```

As you can see, Python automatically adds spaces between different arguments.

In Python 3, `print` is a *function*. This means that we have to use parentheses when using it. This used to be different in Python 2, and there might be a lot of tutorials out there that still show the old syntax. Do *not* use it!

If we want to print floating-point numbers with a specific format, i.e. with a certain number of decimal digits, we can use the `format` function:

```
>>> y=12.345
>>> print('y is: {:.3f}'.format(y))
y is: 12.345
>>> print('y is: {:.2f}'.format(y))
y is: 12.35
>>> print('y is: {:.1f}'.format(y))
y is: 12.3
```

We can also use a format string that employs the variable directly:

```
>>> y=12.345
>>> print(f'y is: {y:.2f}')
y is: 12.35
```

There is also an alternative syntax for formatting strings that uses the `%` operator. This syntax is deprecated; do *not* use it!

One final remark: the *case* of the variable matters. It is good practice to use lower-case identifier names only:

```
>>> today='Wednesday'
>>> Today='Monday'
>>> today
'Wednesday'
>>> Today
'Monday'
```

If your variable uses multiple words, separate them using an 'underline':

```
>>> work_from_home=True
>>> n_coffee_to_wake_up=3
```

2 Writing scripts

Although the command line is great for trying out one or two commands quickly, there will be times when we shall wish to execute the same commands over and over[1]. In these cases, it may be useful to create a *script* that contains all the commands, and then the commands can be executed simply be running the script.

2.1 A simple script

This is best illustrated with an example. Create a file called `eg1.py`, and place the following text into it:

[1]You can also use the 'up arrow' on the keyboard to cycle through all the commands that have already been executed.

```python
x = 3
y = 4
z = 5
answer = x*y + z
print(f'the answer is: {answer:.2f}')
```

Then, on a (Unix) command line in the same folder, run the script using: `python eg1.py`. The output should appear as below ('>' denotes the Unix command prompt—do not type it):

```
> python eg1.py
the answer is: 17.00
```

Depending on the configuration of your operating system, you may have to use `python3` in the example above because the old Python interpreter may be set as the default.

2.2 Passing arguments from the command line

However, suppose we want to run the script again, but for different values of `x`, `y` and `z`. Then we would need to edit the script, and change these values. However, there is another solution. We could create a function whose arguments are `x`, `y` and `z`, and then pass different values to the function. For example, create a script called `eg2.py` which contains the following:

```python
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('-x', action='store', dest='ax', type=int)
parser.add_argument('-y', action='store', dest='ay', type=float)
parser.add_argument('-z', action='store', dest='az', type=int)

args = parser.parse_args()

x = args.ax
y = args.ay
z = args.az

answer = x*y + z
print(f'the answer is: {answer:.2f}')
```

Then the script can be run, for different values of `x`, `y` and `z` using:

```
> python eg2.py -x 2 -y 4.5 -z 6
the answer is: 15.00
> python eg2.py -x 1 -y 0.5 -z 8
the answer is: 8.50
```

Several of the tutorials will require you to write scripts that take parameters from the command line. `argparse` is a Python module for simplifying this task. The example above only scratches the surface of the capabilities of this module; refer to its documentation for more showcases.

By default, Python will execute all commands that appear in a Python script as long as they are not contained inside a function. It is thus good coding practice to *wrap* the commands that your script should execute when being called on the command line. This can be achieved by the following construction:

```python
if __name__ == '__main__':
    # Code that should be executed whenever you call the script from
    # the command line.
    print('Hello, World!')
```

We will *not* use this in the scripts below to keep the code simple. However, you are supposed to use this construction in the homework you submit.

2.3 Functions

Often there will be blocks of code that we wish to run several times within a program. Instead of repeating the code, we can assign a name to that particular block of code. In this way, we create a *function*. As an example, create a file called eg3.py with the following content:

```python
def times_two(x):
    return 2*x


# There should be two blank lines after each function. Python works
# anyway, but this is the accepted style.
y = 3.4
answer = times_two(y)
print(f'two times {y:.2f} is {answer:.2f}')
```

This is a simple script that computes a value times two:

```
> python eg3.py
two times 3.40 is 6.80
```

The following, eg4.py, presents an example which has more features worth discussing:

```python
def compute_mean_and_sd(x, y, z):
    """Calculate mean and standard deviation of three numbers.

    This function computes the mean and the standard deviation of three
    numbers.  This comment is a so-called `docstring`. It is good style
    to use them at the top of a function.
    """
    my_sum = x + y + z
    my_mean = my_sum / 3
    residual_sum = (x - my_mean)**2 + (y - my_mean)**2 + (z - my_mean)**2
    my_variance = residual_sum / 2
    my_sd = my_variance**0.5

    # Return the answer as a tuple of values. In contrast to other
    # programming languages, Python supports n-tuples as well.
    return my_mean, my_sd


# This should be the main function (see note above). We directly assign
# the return value to two variables here.
mean, sd = compute_mean_and_sd(2, 4, 7)
print(f'1: mean is {mean:.2f} and the sdev is {sd:.2f}')
mean, sd = compute_mean_and_sd(3, 4, 8)
print(f'2: mean is {mean:.2f} and the sdev is {sd:.2f}'.format(mean, sd))
```

Calling this script yields:

```
> python eg4.py
1: mean is 4.33 and the sdev is 2.52
2: mean is 5.00 and the sdev is 2.65
```

There are several features of the script worth noticing:

- A function is *declared* using: `def function_name(parameters):`. It is good practice in Python to use underscores instead of 'camel case' (hence, prefer `function_name` to `functionName`). Make sure that your function names speak for themselves!
- The body of the function is **indented** using four spaces or one tabulator. Indentation is very important in Python—without the correct indentation, a script will not run correctly because the Python interpreter will not know how to partition your code into 'blocks'.
- Lines are normally indented using the `<TAB>` key. Configure your editor accordingly.
- Lines beginning with '#' are not run by Python. These lines are *comments* that describe what blocks of code do. It is good practice to comment your code to allow others (and yourself) to read it easily: 'always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live'.
- Our script can be separated into a two sections: one section contains our function(s), while the other section contains is the script's `main` function. The main section is the part of the script that is executed (once), and where functions are *called*.
- A function can be called multiple times in the main function.
- More than one value can be returned by a function.
- If a function returns n values, then n variables should be allocated to the function's output. In this example, `compute_mean_and_sd` returns two values.

3 Conditional and control statements

3.1 `if-else` statements and booleans

There will be times when you want your script to take different actions, depending on a particular calculation. In these cases, an `if-else` statement provides a solution. For example, create a script called `eg5.py` that contains the following:

```python
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('-x', action='store', type=int)
args = parser.parse_args()
x = args.x  # default name based on `add_argument`

# check whether x is even or odd
if x % 2 == 0:
    print(x, 'is even')
else:
    print(x, 'is odd')
```

This simple script checks if a number is even or odd:

```
$ python eg5.py -x 21
21 is odd
$ python eg5.py -x 24
24 is even
```

We can also create *nested* `if-else` statements. Save the following in `eg6.py`:

```python
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('-x', type=int)
args = parser.parse_args()
x = args.x

if x % 2 == 0:
    if x % 3 != 0:
        print(x, 'is even and not divisible by 3')
    else:
        print(x, 'is even and divisible by 3')
else:
    print(x, 'is odd')
```

Now the script tells us if an even number is also divisible by three (or not):

```
> python eg6.py -x 24
24 is even and divisible by 3
```

Notice the following:

- The use of '==' is used for checking equality, while '!=' checks 'is *not* equal to'.
- Nested `if-else` statements must be **indented**. Without the correct indentation, a Python script will not run correctly!
- Instead of `else`, there is also `elif`, which starts another `if` statement.

It is also possible to check for multiple conditions such that (a) at least one condition is satisfied, or (b) all conditions are satisfied. This is accomplished by using the 'or' and 'and' operators. The following script, which can be saved as `eg7.py`, provides an example (which also shows how `elif` can be useful):

```python
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('-x', type=int)
args = parser.parse_args()
x = args.x

if (x % 2 == 0) and (x % 3 == 0):
    print(x, 'is divisible by 2 and 3')
elif (x % 2 == 0) or (x % 3 == 0):
    print(x, 'is divisible by either 2 or 3')
else:
    print(x, 'is not divisible by either 2 or 3')
```

There is another data type that is not an integer, float or string that is called a *boolean*, and can only have one of two values: `True` or `False`. These variables are useful for storing 'decisions'. For example, on the command line, try:

```
>>> want_coffee = True
>>> want_coffee
True
>>> is_even = (5 % 2 == 0)
>>> is_even
False
>>> want_coffee = not want_coffee
>>> want_coffee
False
```

Note that *case* is very important here: 'True' is a boolean, but 'true' is not. Booleans can be used in `if` statements. For example, try the following in a script:

```python
my_bool = True
if my_bool:
    print('the condition is True')
else:
    print('the condition is False')
```

If the condition/boolean is True, the part directly following `if` will execute. If the script has `my_bool = False` instead, then the part following the `else` will be executed.

3.2 Loops

There are times when we may want to execute a command N times, or until a certain condition is met. The way to do this is to use a *loop*. We shall discuss two types of loops here: *for-loops* and *while-loops*.

For-loops

In a script called eg8.py, save the following text:

```python
N = 4
for i in range(0, N):
    print(i, 'squared is:', i**2)
```

Running this script will produce the following output:

```
> python eg8.py
0 squared is: 0
1 squared is: 1
2 squared is: 4
3 squared is: 9
```

Notice that:

- As in functions and `if` statements, **indentation** is important in loops.
- We use `range(a, b)` to give the iterator variable `i` values from `a` (**inclusive**) to `b` (**exclusive**). The last part is very important. It is a typical beginner's mistake to think that `range(0, 10)` goes from 0 to 10, but this is not true. The syntax for `range` follows the common practice of iterating over a list `l`, by specifying `range(0, len(l))`. You can also drop the first number, and Python will automatically assume it to be `0`.

It is also possible to iterate, i.e. *traverse*, over a complex object, such as a list of strings:

```python
names = ['Alice', 'Bob', 'Carol']
for name in names:
    print(name)
```

Here, Python will automatically assign the variable `name` the corresponding value. If you are familiar with other programming language, you will see that Python does not need any explicit indices here, making it really easy to write *understandable* code. Whenever possible, try to iterate directly over an object without accessing it via indices. If you need the indices for some reason (keep in mind that indices in Python, just like most programming languages, start with zero), you can use `enumerate`:

```python
names = ['Alice', 'Bob', 'Carol']
for index, name in enumerate(names):
    print(f'Index {index} belongs to {name}')
```

In the previous snippet, you can see that `enumerate` returns a *tuple* of two values, namely an index and a value. It is also possible to write *nested* for-loops. The following script (eg9.py) is a rudimentary method for finding all the primes less than N:

```python
N = 20
for p in range(2, N):
    is_prime = True
    for divisor in range(2, p):
        if p % divisor == 0:
            is_prime = False
    if is_prime:
        print(f'{p:2d} is prime')
```

As usual in Python, the indentation is very important. Running the script will produce:

```
> python eg9.py
 2 is prime
 3 is prime
 5 is prime
 7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Notice how the "`{p:2d}`" specification in the last line of `eg9.py` pads the single digit numbers with an extra space, to allow them to line up with the two-digit numbers in the output. It is good practice to try to make numeric outputs line up easily for human consumption.

While-loops

Instead of running some commands for a pre-specified number of times, we may wish to run the commands until a condition is met. *While-loops* are well-suited to this task. In a script called `eg10.py`, save the following:

```
x = 0
cube = 0
max_cube = 100
while cube < max_cube:
    x += 1
    cube = x**3
    print(f'{x} cubed is: {cube:3d}')
```

This loop runs while `cube` is less than 100, and then stops as soon as `cube` is greater than this limit, as the following output shows:

```
> python eg10.py
1 cubed is:   1
2 cubed is:   8
3 cubed is:  27
4 cubed is:  64
5 cubed is: 125
```

Note that for-loops automatically increment the iterator (e.g. `i`) from 1 to 2 to 3, etc., but when we use while-loops, we need to do this ourselves. The line '`x += 1`' is shorthand for '`x = x + 1`'. You can try to change the second for-loop in `eg9.py` to a while-loop in order to make the program more efficient.

4 Using Python modules

Python has a lot of functionality on its own, but there are additional modules such as: `numpy`, `scipy`, and textttsklearn that offer greater functionality and data structures such as matrices, numerical algorithms, machine learning algorithms, etc. In fact, we have already seen two modules above:

- `math`, which provides implementations of functions such as the square root, i.e. `math.sqrt()`, and constants such as π and e (`math.pi` and `math.e`).
- `argparse`, which is used to allow the passing of values from the command line to a Python script.

Modules are imported into a Python (command line) session using `import`, e.g.

```
>>> import math
>>> math.pi
3.141592653589793
```

or

```
>>> import numpy as np
>>> matrix = np.array([[1, 2], [3, 4]])
>>> matrix
matrix([[1, 2],
        [3, 4]])
>>> matrix[0, 1]
2
```

We shall discuss arrays (and matrices) with `numpy` in the next section. Some modules, such as `math` and `argparse` are usually automatically installed with Python. However, other modules such as `numpy` and `sklearn` usually need to be installed manually. At the end of this tutorial we give directions for how to install these additional modules.

5 Arrays and matrices using numpy

The `numpy` module offers many useful functions related to arrays. First, let us look at several ways for creating one-dimensional arrays:

```
>>> import numpy as np
>>> np.zeros(4)
array([ 0.,  0.,  0.,  0.])
>>> np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> np.arange(1, 10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1, 2, 0.2)
array([ 1. ,  1.2,  1.4,  1.6,  1.8])
>>> np.linspace(1, 3, 5)
array([ 1. ,  1.5,  2. ,  2.5,  3. ])
>>> len(np.linspace(1, 3, 5))
5
>>> np.array([2, 3, 5, 7, 11])
array([ 2,  3,  5,  7, 11])
>>> np.array([2, 3.1, 5, 7.5, 11])
array([  2. ,   3.1,   5. ,   7.5,  11. ])
```

From now on, we shall assume that the line '`import numpy as np`' has already been executed in the command line session. Notice the following:

- We use '`import numpy as np`' so that we can write functions as `np.zeros(4)` instead of `numpy.zeros(4)`, which is marginally longer. This is a convention that is often used, even in the official `numpy` documentation. It is common practice to use this abbreviation.
- `np.zeros(N)` creates an array of length `N`, filled with zeros.
- `np.arange(a, b)` is similar to `range(a, b)` and creates an array from `a` (inclusive) to `b` (exclusive), in increments of 1.
- `np.arange(a, b, c)` creates an array from `a` (inclusive) to `b` (exclusive), in increments of `c`.
- `np.linspace(a, b, N)` creates an array of length `N` of equally spaced values from `a` to `b` (**both inclusive**: note the difference to `np.arange(a,b)` which excluded `b`).
- the `len(x)` function returns the length of the vector `x`.
- the `np.array([a1, a2, a3])` function creates an array containing the elements `a1`, `a2`, `a3`, etc. Note that if all the elements are integers, the array is represented as an array of integers. However, if one of the elements is a float, all of the elements of the array are represented as floats.

Now let us try saving a vector to a variable name and accessing its elements:

```
>>> x = np.linspace(1, 3, 5)
>>> x
array([ 1. ,  1.5,  2. ,  2.5,  3. ])
>>> x[0]
1.0
>>> x[1]
1.5
```

The important thing to notice is that **the first element of an array is indexed by 0**, just like for regular objects. `numpy` can also easily create 2-dimensional arrays and matrices:

```
>>> np.zeros(shape=(2, 3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> mat[0,1]
2
>>> mat[1, :]
matrix([[4, 5, 6]])
>>> mat[:,1]
matrix([[2],
        [5]])
>>> mat[1, 1:3]
matrix([[5, 6]])
```

Note that:

- For `np.zeros()`, the `shape=(n, m)` argument creates a matrix with `n` rows and `m` columns
- When initialising a matrix with particular values, we need to enclose the one-dimensional arrays in brackets '`[`' and '`]`'. This is required because those brackets indicate Python lists, and we can just as well consider a matrix to be a 'list of lists'.
- An element of a matrix is extracted using the index `[r, c]`, where `r` is the row number and `c` is the column number, and indexing starts from 0, like for arrays.
- Whole rows or columns can be extracted by using '`:`' in place of the column/row index
- A subset of a row or column can be extracted using the range `a:b`, which extracts elements from index `a` (inclusive) to index `b` (exclusive)

The difference between 2-dimensional arrays and matrices is in the way behave under **multiplication**. For arrays, multiplication is element-wise, but for matrices the multiplication is the usual matrix multiplication, as illustrated in the examples below:

```
>>> a1 = np.array([[0, 1], [1, 0]])
>>> a2 = np.array([[1, 2], [3, 4]])
>>> a1
array([[0, 1],
       [1, 0]])
>>> a2
array([[1, 2],
       [3, 4]])
>>> a1 * a2
array([[0, 2],
       [3, 0]])
>>> m1 = np.array([[0, 1], [1, 0]])
>>> m2 = np.array([[1, 2], [3, 4]])
>>> m1
matrix([[0, 1],
        [1, 0]])
>>> m2
matrix([[1, 2],
        [3, 4]])
>>> m1 * m2
matrix([[3, 4],
        [1, 2]])
```

Notice how matrices and arrays are treated the same (in the sense of using the same class for creating them). They only differ by their `shape` variable; you can try this out for yourself.

6 Extracting and counting values in an array

Using numpy arrays allows one to easily extract certain values from the arrays. In particular, it is possible to use boolean expressions to index arrays and easily extracts subsets:

```
>>> x = np.array([2, 4, 2, 8, 5, 3])
>>> index = x < 5
>>> index
array([ True,  True,  True, False, False,  True], dtype=bool)
>>> x[index]
array([2, 4, 2, 3])
>>>
>>> ~index
array([False, False, False,  True,  True, False], dtype=bool)
>>> x[~index]
array([8, 5])
>>>
>>> np.min(x)
2
>>> np.argmin(x)
0
>>> np.max(x)
8
>>> np.argmax(x)
3
>>> x[x > 3]
array([4, 8, 5])
```

Note that '∼' is used to take the 'not' operation of a boolean numpy array, while argmin and argmax give the indices of the minimum and maximum elements of a numerical array, respectively. It is also very easy to count the occurrence of certain values in an array:

```
>>> y = np.array([2, 0, 1, 2, 0, 2, 1, 0, 0, 0])
>>> y
array([2, 0, 1, 2, 0, 2, 1, 0, 0, 0])
>>> len(y[y==1])
2
>>> count = np.array( [len(y[y==i]) for i in range(0, 3)] )
>>> count
array([5, 2, 3])
>>> proportion = count / len(y)
>>> proportion
array([ 0.5,  0.2,  0.3])
```

This will become very useful for the classification tutorials.

7 Generating random variables

It will often be useful to generate random values. Fortunately, `numpy` makes this easy to do.

Uniform random variables

Suppose we wish to generate random numbers (floats) uniformly distributed between 0 and 1—specifically, in the interval $[0, 1)$. This can be easily done using the `numpy` function `random.uniform`:

```
>>> np.random.uniform(0, 1)
0.6762549019801313
>>> np.random.uniform(low=0, high=1)
0.5908628174163508
>>> np.random.uniform(low=0, high=1, size=3)
array([ 0.02398188,  0.55885409,  0.25925245])
>>> np.random.uniform(low=5, high=10, size=4)
array([ 6.41762541,  8.46568959,  7.20226859,  5.78433869])
```

If you run this on your computer, the values will differ, because `numpy` uses a pseudo-random number generator whose initialization depends on various inputs. As can be seen above, this function has three parameters, `low` (lower limit), `high` (upper limit) and `size` (size of sample to generate, if more than one value is required).

Setting the seed for the random number generator

Before we go further, suppose we wish to generate the same random number(s) multiple times, perhaps for different experiments. This can be easily done by **setting the seed** of the random number generator, using the `numpy` function `random.seed`, which takes an argument `seed`, which should be a positive integer (or 0). For example, below we use `seed=3`:

```
>>> np.random.seed(3)
>>> np.random.uniform(low=0, high=1, size=3)
array([ 0.5507979 ,  0.70814782,  0.29090474])
>>> np.random.uniform(low=0, high=1, size=3)
array([ 0.51082761,  0.89294695,  0.89629309])
>>>
>>> np.random.seed(3)
>>> np.random.uniform(low=0, high=1, size=3)
array([ 0.5507979 ,  0.70814782,  0.29090474])
>>>
>>> np.random.seed(3)
>>> np.random.uniform(low=0, high=1)
0.5507979025745755
>>> np.random.uniform(low=0, high=1, size=4)
array([ 0.70814782,  0.29090474,  0.51082761,  0.89294695])
```

Above we have set the seed and then generated random numbers between 0 and 1 three times. Notice that the random numbers which are generated immediately after the seed is set are always the same.

Random integers

The numpy function random.choice is very useful for generating random integers. Suppose that we wish to generate non-negative (positive or 0) random integers less than a.

```
>>> np.random.seed(5)
>>> np.random.choice(a=10, size=5)
array([3, 6, 6, 0, 9])
>>> np.random.seed(5)
>>> np.random.choice(a=10, size=5, replace=False)
array([9, 5, 2, 4, 7])
>>> np.random.seed(5)
>>> np.random.choice(a=10, size=5, replace=True)
array([3, 6, 6, 0, 9])
```

The first argument, a, indicates that integers will be randomly chosen from the set $\{0, 1, \ldots, a-1\}$. By default, duplicates are allowed. If you do not want duplicates, the replace parameter should be set to False. This function can also be used to randomly select elements from a particular numpy array:

```
>>> x = np.array([0, 1.1, 5])
>>> np.random.choice(x, size=7)
array([ 1.1,  5. ,  0. ,  1.1,  0. ,  1.1,  5. ])
```

By default, every element (in the specified array, or the set $\{0, 1, \ldots, a-1\}$) has an equal probability of being selected. However, it is also possible to select elements with nonuniform probabilty, as the following example shows:

```
>>> np.random.choice(3, size=10, p=[0.1, 0.3, 0.6])
array([1, 2, 2, 2, 0, 1, 2, 1, 1, 2])
```

To confirm the frequency of the elements matches the specified probabilities, we can run a small experiment, by generating a large sample and computing the sample frequencies:

```
>>> x = np.random.choice(3, size=10000, p=[0.1, 0.3, 0.6])
>>> count = np.array( [len(x[x==i]) for i in range(0,3)] )
>>> proportion = count / float(len(x))
>>> proportion
array([ 0.0983,  0.3012,  0.6005])
```

Random subset or permutation

Suppose that we have an array from which we wish to select a subset. The numpy function random.choice can also be used for this task:

```
>>> fib = np.array([1, 2, 3, 5, 8])
>>> fib
```

```
array([1, 2, 3, 5, 8])
>>> index = np.random.choice(len(fib), size=3, replace=False)
>>> index
array([4, 0, 2])
>>> fib[index]
array([8, 1, 3])
```

In the same way, a permutation (rearrangement) of an array can be obtained, simply by selecting a subset with the same size as the array:

```
>>> z = np.array([1, 2, 3, 4, 5])
>>> index = np.random.choice(len(z), size=len(z),replace=False)
>>> z[index]
array([3, 2, 5, 4, 1])
```

Finally, it is worthwhile learning a trick to obtain the complement of a subset. This is only *one* way to solve this. It would also be possible to use set-theoretic calculations here, using the builtin set class.

```
>>> y = np.array([10, 20, 30, 40, 50])
>>> np.random.seed(1)
>>> index = np.random.choice(len(y), size=3, replace=False)
>>> index
array([2, 1, 4])
>>> y[index]
array([30, 20, 50])
>>>
>>> all_indices = np.arange(len(y))
>>> all_indices
array([0, 1, 2, 3, 4])
>>>
>>> other = np.ones(len(y), dtype=bool)
>>> other
array([ True,  True,  True,  True,  True], dtype=bool)
>>> other[[index]] = False
>>> other
array([ True, False, False,  True, False], dtype=bool)
>>> complement_index = all_indices[other]
>>> complement_index
array([0, 3])
>>>
>>> y[index]
array([30, 20, 50])
>>> y[complement_index]
array([10, 40])
```

8 Creating copies of variables and arrays

We need to briefly say a few words about how Python copies variables. For those with programming experience in other languages, the following behaviour is familiar:

```
>>> x = 1
>>> y = x
>>> x = 2
>>> x
2
>>> y
1
```

In other words, if we initialise `y` to have `x`'s value, and then later change the value of `x`, this will not change the value of `y`. However, this is not quite the case with `numpy` arrays:

```
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
>>> x[1] = 9
>>> x
array([1, 9, 3])
>>> y
array([1, 9, 3])
>>> z
array([1, 2, 3])
```

Now, when an element of the array `x` is changed, the corresponding element in `y` will also changed. However, if we use the `numpy` function `copy`, as we did for defining the array `z`, then the corresponding element is not changed. Python generally handles 'complex' objects, such as arrays, matrices, and so on, slightly differently than simple types, such as `int`. If this is confusing to you now, don't worry: most of the time, you will not have to consciously think about these rules, it will 'just work'.

9 Reading and writing files using numpy

Usually your data will be stored in files on a disk. Before you can do any data mining, you will first need to read the data from the files into memory. After analysing the data, you will probabaly want to record the results you obtained in another file. There are several ways to read and write files in Python.

The `numpy` method `loadtxt` loads data from a text file into a `numpy` array. Suppose the following text is saved in the file `data1.txt`:

```
0.8147    0.9134    0.2785
0.9058    0.6324    0.5469
0.1270    0.0975    0.9575
```

Note that the values in each row are separated by a space (or several spaces). In such cases, the data can be loaded into memory simply by using:

```
>>> import numpy as np
>>> np.loadtxt('data1.txt')
array([[ 0.8147,  0.9134,  0.2785],
       [ 0.9058,  0.6324,  0.5469],
       [ 0.127 ,  0.0975,  0.9575]])
```

If your data is in the CSV format (or other formats), the `delimiter` parameter of `loadtxt` allows you to specify this. Suppose that the following text is saved in `data2.txt`:

```
0.8147,    0.9134,    0.2785
0.9058,    0.6324,    0.5469
0.1270,    0.0975,    0.9575
```

The data can now be loaded using:

```
>>> import numpy as np
>>> np.loadtxt('data2.txt', delimiter=',')
array([[ 0.8147,  0.9134,  0.2785],
       [ 0.9058,  0.6324,  0.5469],
       [ 0.127 ,  0.0975,  0.9575]])
```

If your data contains not just numbers, but also strings and categorical values, you can specify them for `loadtxt` using parameter `dtype`. Suppose the following text is saved in the file `data3.txt`:

```
M,    35,    78.3
F,    28,    52.2
F,    37,    56.2
```

Then this data set can be loaded using the following code:

```
>>> import numpy as np
>>> names = ('gender', 'age', 'weight')
>>> formats = ('S1', 'i4', 'f4')
>>> data = {'names': names, 'formats': formats}
>>> np.loadtxt('data3.txt', delimiter=',', dtype=data)
array([('M', 35, 78.3), ('F', 28, 52.2), ('F', 37, 56.2)],
  dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])
```

The first character of a format in `dtype` specifies the kind of data (eg. `S` for string, `i` for integer and `f` for float) and the remaining characters specify the number of bytes per item (eg. `i4` means 32-bit signed integer, since there are 8 bits in one byte). For even more complicated files, you can use the function `genfromtxt()`. Alternatively, a good package for data wrangling is `pandas`, as it is compatible with `numpy` and offers seamless interoperability. Now, suppose you want to save data to a file. The following commands

```
>>> name = ('a', 'b', 'c')
>>> age = (20, 24, 27)
>>> height = (5.8, 6.1, 5.6)
>>> data = np.rec.fromarrays((name, age, height))
>>> data
rec.array([('a', 20, 5.8), ('b', 24, 6.1), ('c', 27, 5.6)],
      dtype=[('f0', 'S1'), ('f1', '<i8'), ('f2', '<f8')])
>>> np.savetxt(fname='data4.txt', X=data, fmt='%s %d %.2e')
```

will produce a file named `data4.txt` which contains the following text:

```
a 20 5.80e+00
b 24 6.10e+00
c 27 5.60e+00
```

There are a few points to note about the code which produced this file:

- Each of the arrays, `name`, `age` and `height` form a column of the txt file.
- Each column can be a different type (string, integer or float).
- We use the `numpy` function `rec.fromarrays` to combine the arrays, column-wise, into a 'matrix'.
- The `numpy` function `savetxt` saves the data, specified by parameter `X`, to a file, specified by the parameter `fname`.
- It is necessary to specify the format of each column in a single string separated by spaces. For example, `fmt='%s %d %.2e'` specifies that the first column contains strings, the second column contains integers and the third column should be formatted as a floating point number with two decimal places.

In many cases, the dimensionality of your data could be quite high (with many columns to be read) and then manually specifying the data types and formats for each column will not be an option. In such cases, the `pandas` module will help you dealing with them automatically. Although this is beyond the scope of this tutorial, there are many online resources containing more information on the `numpy` and `pandas` modules.

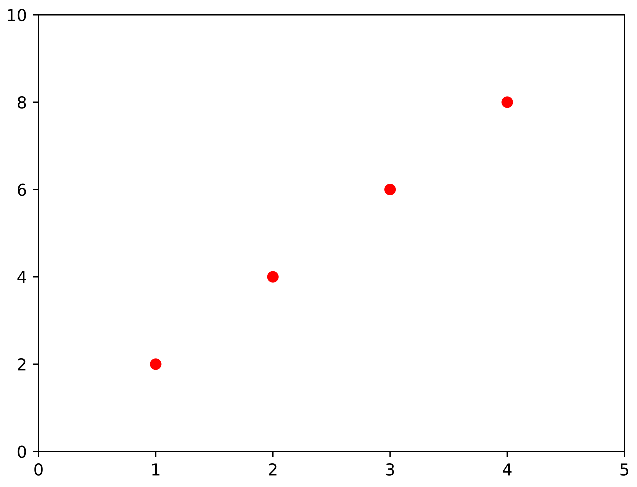
10 Plotting using `matplotlib`

Sometimes you need to plot your results for better visualization. The `matplotlib` module is very useful for easily creating plots of your data. We describe some basic operations here; for more information please check the documentation of `matplotlib`. For simple plotting the `pyplot` interface provides a MATLAB-like interface. You will find it familiar, if you are a MATLAB user. Enter the following commands:

```
>>> import matplotlib.pyplot as plt
>>> x = [1, 2, 3, 4]
>>> y = [2, 4, 6, 8]
>>> plt.plot(x, y, 'ro')
>>> plt.scatter(x[2], y[2], s=80, facecolors='none')
>>> plt.axis([0, 5, 0, 10])
>>> plt.show()
```

This produces the following figure: Some points to note:

- The `plot` argument `'ro'` specifies that the points should be plotted as red dots. There are many more options available.
- The command `plt.scatter(...)` has placed a circle around the third point (remember: the first element of an array is indexed by 0).
- The `axis([a, b, c, d])` function specifies the x-axis to be between `a` and `b`, and the y-axis to be between `c` and `d`. In the figure above, the x-axis is between 0 and 5, and the y-axis is between 0 and 10.
- After executing the above code, a figure will pop up. It is possible to save the figure as a png file by clicking on the 'save' icon below the figure.

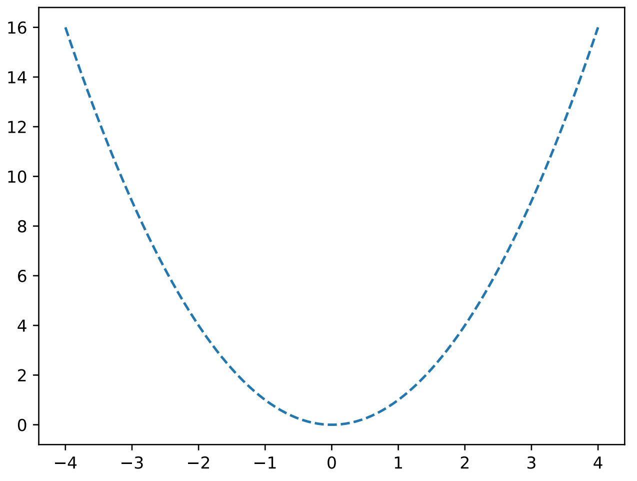


- While the figure is showing above, you will not be able to add any more commands in the Python session until the figure is closed. However, if you use the command `plt.show(block=False)` instead, you will still have access to the session while the figure is showing.

If you would like to save a figure in pdf format, the following code will help you:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> plt.clf()
>>> x = np.linspace(-4, 4, 100)
>>> y = x**2
>>> plt.plot(x, y, '--')
>>> plt.savefig('fig2.pdf')
```

saves the following figure to `fig2.pdf`: Note that:



- The command to save the figure is: `plt.savefig('filename.filetype')`.

- The command `plt.clf()` clears any previous plot.
- The plot type `'-'` creates a blue dashed line.

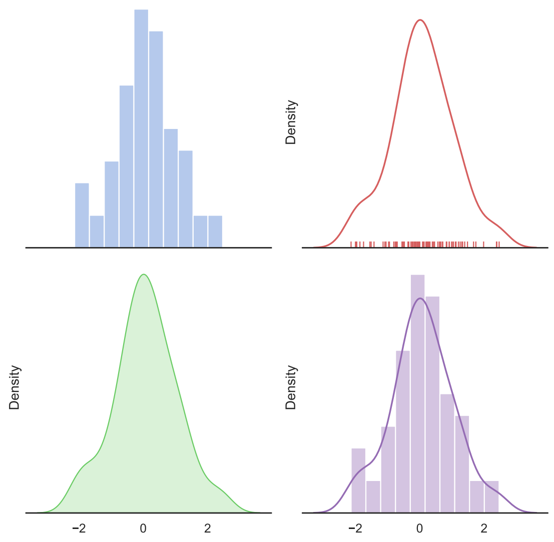
As already mentioned, there are far more options for creating plots with `matplotlib`, and the reader is encouraged to look at the online documention. There are also other plotting packages that build on `matplotlib`. One particularly good example is `seaborn`. It focuses on creating powerful statistical graphics. The following code (taken from the official documentation) will create plots of the same distribution with different modalities:

```python
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

sns.set(style='white', palette='muted', color_codes=True)
rs = np.random.RandomState(10)

# Set up the matplotlib figure
f, axes = plt.subplots(2, 2, figsize=(7, 7), sharex=True)
sns.despine(left=True)

# Generate a random univariate dataset
d = rs.normal(size=100)

# Plot a simple histogram with binsize determined automatically
sns.distplot(d, kde=False, color='b', ax=axes[0, 0])

# Plot a kernel density estimate and rug plot
sns.distplot(d, hist=False, rug=True, color='r', ax=axes[0, 1])

# Plot a filled kernel density estimate
sns.distplot(d,
  hist=False,
  color="g",
  kde_kws={'shade': True},
  ax=axes[1, 0]
)

# Plot a historgram and kernel density estimate
sns.distplot(d, color='m', ax=axes[1, 1])

plt.setp(axes, yticks=[])
plt.tight_layout()
```

This will result in the following figure:

11 Installation

We briefly discuss how to install Python and Python modules.



11.1 Linux and OS X

If you are running Linux or Mac OS X, it is likely that Python is already installed. This can be easily checked by opening a terminal window and typing:

```
> python
```

(do not type the ">", it represents the terminal command prompt). You should then see something like:

```
Python 3.7.0 (default, Jun 29 2018, 20:13:13)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

where the '»>' represents the Python command prompt.

11.2 Windows

For Windows, it is suggested that the reader go to `https://docs.python.org/3/using/index.html`, which contains links for the Windows Python installer, as well is information on how to install `pip`, a program that will allow one to easily install modules.

11.3 Installing modules

To install the Python modules `numpy`, `matplotlib` and `scikit-learn`, we can use `pip`. The following commands can be typed into a Unix terminal:

```
> pip3 install numpy
> pip3 install matplotlib
> pip3 install scikit-learn
```

11.4 Integrated Development Environments

Python scripts can be written in any text editor, and then called on the command line. However, it may be worthwhile to consider using an integrated development environment (IDE) such as Eclipe or VisualStudio, which has a Python plugin. Emacs and vim are great editors, but there is a learning curve, and will require an initial investment of time. If you do not already have a preferred editor for Python, we recommend trying one of:

- **Pycharm**: free ('Community') version available from: `https://www.jetbrains.com/pycharm/download/`
- **IEP**: free and available from: `http://www.iep-project.org/downloads.html`
- **Canopy**: free version available from: `https://store.enthought.com/downloads/`
- **vim**: free version available from: `https://www.vim.org`

which are all available on Windows, Mac and Linux. The benefits of using an IDE/specialised text editor include:

- Syntax colouring
- Auto-indentation (very important in Python)
- Code-completion (in some cases)
- Faster editing (in some cases)

12 Resources

This tutorial only offers a few highlights of what Python can do in order the reader to be able to solve the tutorials. There are far more functions which can be discovered via the online documentation:

- `https://www.python.org/doc/`
- `http://docs.scipy.org/doc/numpy/reference/`
- `http://scikit-learn.org/stable/index.html`
- `http://matplotlib.org/`

Another very useful resource is the **StackOverflow** forum. If you Google something like 'python create a matrix of zeros', it is often the case that the first few results will be from `stackoverflow.org`, e.g. `http://stackoverflow.com/questions/568962/how-do-i-create-an-empty-array-matrix-in-numpy`

Acknowledgements

This tutorial was created by Dean Bodenham and Karsten Borgwardt. It was updated in 2018, 2019, and 2020 by Max Horn and Bastian Rieck.