First introduction to Python

Installation, IPython (Jupyter) notebook, arithmetic and logical expressions, data types, variables, subprograms (functions), conditions, loops, NumPy arrays, MatPlotLib.

Installation and first launch

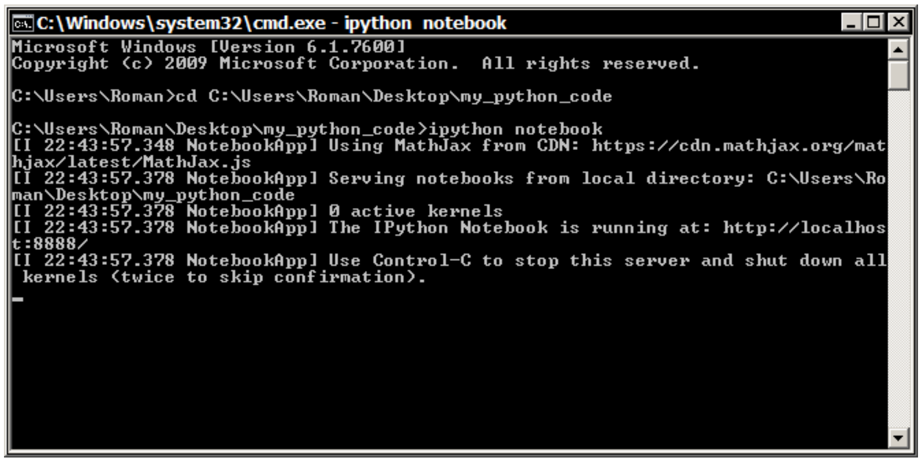
Being one of the most used programming languages in the world, Python comes in huge variety of distributions, each having its own unique flaws, virtues and compatibility issues. In this course, we recommend using the freeware distribution **Anaconda** that comes with a development environment called **IPython Notebook** (more recently, **Jupyter Notebook**). Anaconda is available for all major operating systems (Windows, Linux and Mac OS included). The rest of this manual assumes that the reader uses Anaconda to write and execute Python code.

Anaconda can be downloaded from its official website: https://www.anaconda.com/download/. You will have to choose your operating system first and then your preferred version of Python. As of today (01/06/2019), one has to choose between Python 3.7 and Python 2.7. Unless you have previous experience using Python 3.x, please download and install Anaconda with **Python 2.7**.

FYI: Python 2 VS Python 3

Python 3.x is the most up-to-date version of Python and is generally considered more self-consistent. This version of Python is under active maintenance and more members of the Python 3.x family will be released over the next few years. Python 2.x is older and most likely will not be developed beyond Python 2.7. On the other hand, Python 2.x has undergone more comprehensive testing due to being available for longer and there are still many actively used external modules that were developed for Python 2.x and may not be fully compatible with Python 3.x, especially in the field of astronomy. Furthermore, Python 2.x may be a little faster in some of its applications utilized within this course. As such, the instruction in this course will be aimed at Python 2.7. The syntax of Python 3.x is different enough that most of the code written for it will not run on Python 2.x or vice versa without some minor alterations, which is why we recommend installing and using Python 2.7. However, please rest assured that your TA will be happy to help with Python 3.x should you choose to ignore our recommendation either based on your prior experience or because you are eager to try out the most recent version of the software available.

Once installed, open the terminal (on Mac OS, search for `terminal`; on Windows, search for `cmd.exe`; on most Linux distributions `CTRL+ALT+T` will do), change your directory to where you would like your Python code to be saved (use the `cd` command regardless of your operating system) and launch IPython Notebook either by typing `ipython notebook` or `jupyter notebook` depending on your setup of Anaconda (most likely, both will work).

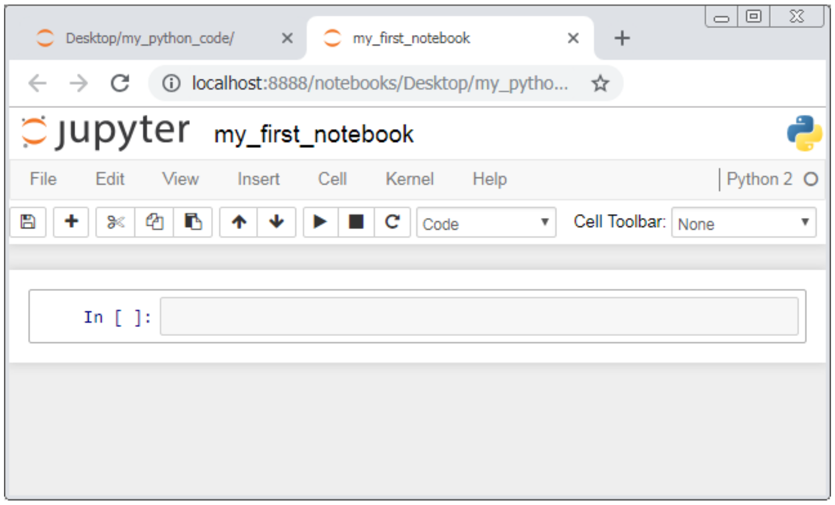


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Roman>cd C:\Users\Roman\Desktop\my_python_code

C:\Users\Roman\Desktop\my_python_code>ipython notebook
[I 22:43:57.348 NotebookApp] Using MathJax from CDN: https://cdn.mathjax.org/mat
hjax/latest/MathJax.js
[I 22:43:57.378 NotebookApp] Serving notebooks from local directory: C:\Users\Ro
man\Desktop\my_python_code
[I 22:43:57.378 NotebookApp] 0 active kernels
[I 22:43:57.378 NotebookApp] The IPython Notebook is running at: http://localhos
t:8888/
[I 22:43:57.378 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
_
```

It may take a few seconds for the environment to start up. Eventually, it should open a new tab in your browser, listing the files in your chosen folder. Keep the terminal session window open for as long as you are working on your code. When finished, save your changes, close the browser tab, switch back to the terminal window and press `CTRL+C` to stop the notebook server. Alternatively, you can simply close the terminal window. In some cases, it may take the environment a few seconds or minutes to end your session.

IPython (Jupyter) Notebook

IPython Notebook is an environment where you can write, organize and execute your Python code as well as add formatted comments that may help you or somebody else understand what is going on. The notebook will run entirely within your browser. Start by either clicking on an already existing notebook or creating a new one using the buttons provided. When a new notebook is created, it will automatically open in a new browser tab and receive a default name (usually, `Untitled` or `Untitled0`). You can (and should) rename your notebook to something more meaningful by clicking on its title at the top of the page. Your work will be saved automatically every now and then; however, it is a good idea to save everything manually by clicking the "floppy disk" icon before you close the browser tab at the end of your session.



A notebook is composed of cells. Originally, there will only be one, but you can create more with the "Plus" button in the toolbar. You can also copy, delete and reorganize your cells as you wish, using the other toolbar buttons. Every cell will be one of two types: a code cell or a markdown cell. You can change the type of any cell by activating it with a click and using the left dropdown in the toolbar (it is set to "Code" by default). As the name suggests, a code cell contains Python code that can be executed. A markdown cell contains arbitrary comments that can be formatted using a markup language called **Markdown**. Have a look at the Markdown Cheatsheet, https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet, to learn more about this language.

Should you happen to be familiar with other markup languages, such as **HTML** or **LaTeX**, you may find it useful to know that both are partially supported in markdown cells as well. Try creating a new cell, changing its type to markdown and typing in the following lines:

```
# Large header
## Smaller header

Some text. **Some bold text**. *Some italic text*. Here is a bullet list:

* Item 1
* Item 2

<table>
  <tr>
    <td>This is a table...</td>
    <td>...created with HTML</td>
  </tr>
</table>

And this is a LaTeX equation: $L=4\pi\sigma R^{2} T^{4}$.
```

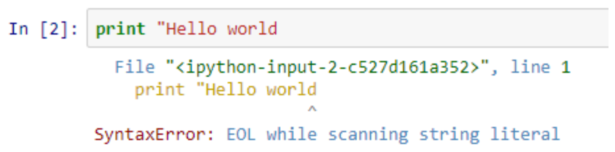
When finished, either click on the "Play" icon in the toolbar or press `SHIFT+ENTER`. Hopefully, you should see some basic formatting written in Markdown, followed by a table written in HTML and an equation written in LaTeX. At this stage, do not worry if you never encountered those markup languages before and the above makes little sense. Nonetheless, do try to annotate any Python code that you write with markdown cells and feel free to experiment with basic formatting either by using the Cheatsheet or by asking your TA for help!

Of course, the primary purpose of a notebook is to store and execute Python code. Create a new cell and make sure that it is a code cell (it should be by default). Type in the following line and execute the cell, as before, either by clicking on the "Play" icon or with `SHIFT+ENTER`:

```python
print "Hello world!"
```

This is a line of code written in Python that outputs (“prints”) “Hello world!” to the screen.

If the above worked successfully, try running invalid Python code to see how the environment reacts. For example, remove one of the quotation marks in the code and execute the cell again.



```
In [2]: print "Hello world

  File "<ipython-input-2-c527d161a352>", line 1
    print "Hello world
                      ^
SyntaxError: EOL while scanning string literal
```

The environment should say that a syntax error was found, give you a short description of the error (in my case, “EOL while scanning string literal”, where EOL means “end of line”) and state where the error is (in my case, it is “line 1”). No doubt, you will be running into a lot of errors and this information may help finding and fixing them. Your TA will also need to know this information to be of any assistance.

Basic expressions and data types

Try running the following lines of code, placing only one of them in your code cell at a time and executing the cell:

```python
1.  2                       # Outputs 2
2.  2 + 2                   # Outputs 4
3.  2 * 3                   # Outputs 6
4.  4 / 2                   # Outputs 2
5.  5 / 2                   # Outputs 2 (!!)
6.  float(5) / float(2)     # Outputs 2.5
7.  5.0 / 2                 # Outputs 2.5
8.  'Hello'                 # Outputs 'Hello'
9.  'Hello' + 'World'       # Outputs 'HelloWorld'
10. 'Hello' * 3             # Outputs 'HelloHelloHello'
```

The ultimate purpose of programming is to take some initial data and transform it into some final output. The data comes in many different types: numbers, strings of text, vectors, matrices and so forth. The type of data determines what operations can be performed on it (e.g. numbers can be added or multiplied, strings can be concatenated, matrices can be transposed etc.) The first line of code gives Python an item of data, which is the number 2. We do not ask Python to perform any processing of the data just yet, so it returns the input (2) back as is.

Note that # in Python means that everything that follows until the end of the line is a *comment*. Comments are ignored by Python and you do not need to type those in to make the code work. However, good commenting is necessary to make your code easier to organize and understand. While markdown cells are there to store a general description of what you are doing, # can be used to annotate a specific line or block of the code.

FYI: Good and bad commenting

As opposed to some of the early programming languages like C++, Python was developed with user-friendliness in mind. More often than not, Python code reads as plain English and can be understood with no prior knowledge of the language whatsoever. However, overreliance on default behavior (e.g. by using the standard order of operations in arithmetic expressions instead of brackets), unwillingness to give functions and variables (covered later) meaningful names and allowing excessive redundancy (e.g. through failing to utilize loops and functions where necessary) can easily render your Python code incomprehensible, defying its underlying philosophy.

Your commenting style is often a good indicator of the code quality. Well-written code rarely needs comments to explain how it works. Instead, one should be able to infer that by looking at the code itself. The primary purpose of comments is not to explain how the code works, but why it is there. If you ever find yourself writing comments that are longer than the code they are aiming to explain or comments that answer the “how” question instead of the “why” question, there almost certainly is a better way to rewrite your code that would not require this. Our aim is not just to write code that “works”, but also one that can be maintained and expanded by others in the future.

Lines 2 to 5 attempt to perform basic arithmetic operations. In Python, + means addition, − means subtraction, * means multiplication, / means division and ** (not ^!) means exponentiation. The output of line 5 is somewhat counterintuitive, as it suggests that $5/2 = 2$! This is because the initial data (5 and 2) are both integers, which makes Python assume that we expect the result to be an integer as well. It achieves so by rounding the result down, arriving at 2 instead of the expected 2.5. This behavior can be fixed if we tell Python that it should treat the input data as real numbers and not as integers.

One way of doing it is by wrapping 5 and 2 in `float()`, as in line 6. `float()` is a function that takes some value and attempts to convert it into a real number (float stands for floating-point arithmetic, which is a formulation that Python and many other languages use to store real numbers in memory). There are analogous functions for other data types. `int()` will produce an integer (rounding down if necessary), `str()` will produce a string of text and so forth. The process, where Python is forced to interpret some value as a specific data type is called *typecasting*.

Line 7 does the same as line 6, but in a different way. Instead of using `float()` to typecast the input, it adds an explicit fractional part to 5, making it 5.0. This way, Python will know that the number may have a fractional part and will treat is as a float by default. Note that 2 will remain an integer; however, Python only needs one of the numbers to be a float to interpret the entire expression as floating-point arithmetic. In fact, the second call of `float()` in line 6 was redundant.

FYI: Python 3 integer division
If you find Python's interpretation of integer VS float division confusing and unnecessary, you would not be alone. This behavior was changed in Python 3.x to make it more consistent with other programming languages and prevent silly mistakes. In Python 3.x, a single slash (`/`) always leads to float division regardless of the input data types. Integer division must be forced with a double slash (`//`). This change, however, does not affect Python 2.x, where one still needs to remain extra vigilant when dividing numbers.

The last three lines demonstrate a few operations that can be performed on strings. Just like the fractional part tells Python that the value should be interpreted as a float, the quotation marks around the value tell Python that the value should be interpreted as a string. When working with strings, + means concatenation and * means duplication. Operations like / are not supported and will produce errors when applied to strings.

FYI: Single VS double quotes
Python interprets single and double quoted strings with subtle differences when it comes to special characters within the strings and formatting. It is unlikely that advanced string operations will be necessary in this course, which is why this otherwise important feature of Python will be ignored for now. In this manual, single and double quotation marks will be used interchangeably. You can read more about special characters and escape sequences in the Python Reference Manual, https://docs.python.org/2.0/ref/strings.html.

Let us have a look at a few more advanced examples. As before, run the below lines one at a time in your code cell:

```
1.  2 + 2                     # Outputs 4
2.  str(2) + str(2)           # Outputs 22
3.  '2' + '2'                 # Outputs 22
4.  int(2.6) + float('3.5')   # Outputs 5.5
5.  3 ** (2 + 2)              # Outputs 81
6.  str(int(1e20 + 2))        # Outputs '100000000000000000000'
7.  str(int(1e20) + 2)        # Outputs '100000000000000000002'
8.  1+5j * 3                  # Outputs (1+15j)
9.  (1+5j).imag               # Outputs 5.0
```

At this point, you should be able to explain the behavior of the first four lines yourself. Line 5 demonstrates that multiple arithmetic operations can be performed in a single line. The order of execution is standard (first, exponentiation, followed by multiplication and division, followed by addition and subtraction), but can be altered with brackets.

Line 6 introduces scientific notation in Python, using `e`, meaning $\times 10^{\cdots}$. E.g. `1e20` means $1 \times 10^{20} = 10^{20}$. Scientific notation is typecast into float automatically.

Lines 6 and 7 draw an important difference between integers and floats. When integer arithmetic is performed, it will be carried out exactly regardless of the number of significant figures as long as the computational power of the system allows that. On the other hand, float numbers are trimmed at a certain number of significant figures so that they always take the same amount of RAM. This is why in line 6, $10^{20} + 2 = 10^{20}$: the added 2 is simply beyond the maximum level of precision and will be dropped. In line 7, 10^{20} is converted into an integer first, forcing integer arithmetic instead, where the full precision is maintained.

Line 8 introduces complex numbers. In Python, the imaginary unit is denoted with `j`. For example, `1+15j` means $1 + 15i$. Note how the order of operations (multiplication is done first) applies to line 8.

Finally, line 9 emphasizes a very important feature of Python: it is an *object-oriented* language. This means that every item of data (be it a string, an integer, a float or anything else) is treated as an *object*. Objects can have properties (aka attributes) that can be accessed by typing out their name after a period (`.`). `imag` is a property possessed by any object that is a complex number and will store its imaginary part. Analogically, `real` will store the real part. Note that these two specific properties are only possessed by complex numbers. An attempt to access them in, say, a string (e.g. by typing `'x'.real`) will result in an error.

Variables

So far we only considered single lines of code. Most tasks will need to be broken down into multiple steps, requiring a convenient way to save intermediate results and refer to them. With most languages, including Python, this is accomplished through *variables*. Consider the following snippet of code. This time, run every line of the snippet at the same time in a single code cell:

```
1. a = 2
2. b = 6
3. c = a * b + a
4. print c, float(c), complex(c)
```

A variable is defined with `=` (the assignment operator). The first line defines a variable called `a` that refers to the integer 2. From now on, we can refer to this integer using the name of the variable. Equivalently, line 2 defines a variable called `b` that stores a different integer. Line 3 performs some basic operations on `a` and `b` and defines a third variable, `c`, to store the result.

The last line invokes a `print` statement. It simply outputs to the screen one or more expressions separated with commas. In this case, it will output `c` as is, then typecast it into a float and a complex number and output those as well.

FYI: Functions VS statements

Previously, we encountered functions such as `float()`, `int()`, `complex()` and others.
The arguments of those functions are placed in brackets, just like they would be in mathematical notation. `print` is different, as it is not a function, but a statement. Being one, it does not require brackets and will continue reading its arguments until the end of the line. Or, at least, this is true in Python 2.x.
This inconsistency was removed in Python 3.x, where `print` IS a function and requires its arguments to be contained in brackets. This is likely the most obvious difference between the two versions and the most common reason why Python 2.x code will not run on 3.x interpreters.

Once a variable is defined, it can be redefined an unlimited number of times. The value of the variable will be preserved either until the end of your IPython notebook session or until you restart the kernel by going to `Kernel->Restart` in the main menu. It is a good idea to restart the kernel regularly when testing your code to make sure that its successful execution does not rely on left-over values of variables that will get lost once the notebook is closed.

Subprograms (aka functions or methods)

Often, the same calculation needs to be performed multiple times in the same program. Rather than repeating the same code multiple times, most languages provide means to define *subprograms*, or blocks of code that are given a special name, which can be used to “call” them from other parts of the code. We have already encountered subprograms before when typecasting data (e.g. `int()` is a subprogram).

Before continuing, an aside must be made regarding the terms used in this manual. A *function* is a subprogram that, when done running, returns a certain value as its output. For example, `int()` is a function, as it returns its argument typecast as integer that can be subsequently used in an expression. In Python, all subprograms are functions, because they always return a default value even when their creator did not code in any output explicitly. In object-oriented languages, such as Python, functions may be “owned” by specific objects. For example, an object representing a matrix may own a function that transposes it. Such functions are called *methods*. Because virtually every function in python is a method (which may sometimes be implicit), these terms are often used interchangeably even in official documentation!

The code snippet below serves as an example of defining a new function:

```
def square_root(x):
    result = x ** 0.5
    return result

print square_root(1), square_root(2), square_root(3)
```

Here, we use the `def` statement to define a new function called `square_root`. As with variables, function names are arbitrary, but giving them meaningful names is considered a good coding practice. Right after the name of the function, we need to list every argument that the function needs to run. In this case, only one argument, `x`, is listed. It will be the number, whose square root needs to be computed.

Lines 2 and 3 are the body of the function. Each line is indented with the same number of spaces to tell Python where the body of the function begins and ends. The number of spaces is arbitrary, but the convention is to use four. More often than not, the notebook will take care of indentation automatically. In the body, `x` is raised to the power of 0.5, thereby yielding its square root. The result is saved in a variable called `result`. Finally, a `return` statement is used that serves two purposes. First, it terminates the function and, second, it sets the variable name that follows it, `result`, as the output (the returned value) of the function.

In line 5, we call the function three times and print the results, outputting the numerical values of $\sqrt{1}$, $\sqrt{2}$ and $\sqrt{3}$. Now that the code cell containing the definition of the function has ran, the function will be accessible in any other code cell until the notebook session is closed or the kernel is restarted.

Functions can have more than one argument and return more than one value. Consider the following example of a function, designed to solve quadratic equations:

```
1. def solve_quadratic(a, b, c):
2.     """Solves a*x**2 + b*x + c = 0 and returns both roots"""
3.     d = b**2.0 - 4 * a * c
4.     d = complex(d)
5.     x1 = (-b + d**0.5) / (2*a)
6.     x2 = (-b - d**0.5) / (2*a)
7.     return x1, x2
8. 
9. print solve_quadratic(1, 1, 0), solve_quadratic(1, 0, 2)
```

This function has three arguments: `a`, `b` and `c` and returns two outputs: `x1` and `x2`. Line 2 of the snippet is the so-called *docstring*. It is made of arbitrary text, enclosed in three double quotes, that is usually pasted in the first line of the body and describes what the function does. The docstring is purely a matter of convenience and will not affect the execution of the function. It is considered good practice to provide a docstring for every function in the code. Docstrings are often used by software, automatically compiling documentation for the code.

In line 3, `d` is typecast to a complex number. This is done on the off chance that `d` ends up being `float` and negative after line 3, as Python does not allow raising negative floats to fractional powers. Such feature is only supported by complex numbers; hence the typecasting call.

Try running the snippet in a new code cell. On my machine, the output looks as follows:

```
 (0j,      (-1+0j))      ((8.659560562354934e-17+1.4142135623730951j),      (-8.659560562354934e-17-
1.4142135623730951j))
```

Note that the real part of the second pair of roots is not exactly equal to zero. Instead, it evaluated to a very small non-zero number due to the finite precision of float-point arithmetic. Every numerical calculation performed by a computer is bound to have a small error due to limited machine precision. Sometimes Python will be clever enough to identify that and perform appropriate rounding and sometimes it will not be, resulting in the output as above.

Conditions and logical expressions

Every program considered up to now had a linear execution flow. This means that every instruction was carried out in a specific order that remained the same every time the code cell was ran. Very often, it is necessary to only execute a block of code when a certain condition holds true or to repeat executing the same block of code until a certain condition is met. Have a careful look at the following snippet, illustrating non-linear flow:

```
1. a = -2+1j
2. 
3. # Check if a is purely real
4. a = complex(a)
5. if a.imag == 0:
6.     print "a is purely real"
7. a = a.real
8. 
9. # Check if a is between 5 (inclusive) and 10 (exclusive)
```

```
if a >= 5 and a < 10:
    print "a is between 5 and 10"
else:
    print "a is not between 5 and 10"

# Check if a is positive, negative or 0
if a > 0:
    print "a is positive"
elif a < 0:
    print "a is negative"
else:
    print "a is neither positive nor negative"
    print "So it is probably zero"
```

Run the cell multiple times, altering the first line to assign different values to `a`. The code checks three conditions and prints different strings of text depending on whether these conditions hold or not. The general structure of a condition is given below (this is not proper code, it will not run!):

```
if <conditional expression to check>:
    code
    that
    runs
    when
    condition holds
elif <some_other_conditional_expression_to_check>:
    code
    that
    runs when
    this other condition
    holds
else:
    code
    that
    runs
    when
    none of the above conditions
    hold
```

Note that only the first `if` statement and the block of code that follows are mandatory. `elif` can be used to check a second condition when the first one does not hold (if the first condition holds, Python will simply skip the rest of the structure). The `elif` statement and the block of code that follows can be omitted or used multiple times to check three or more conditions (as before, Python will ignore the rest of the structure once it finds a condition that holds). If none of the conditional expressions hold, Python will execute the block of code that follows the `else` statement, if provided.

Returning back to our snippet, the first conditional expression, `a.imag == 0`, holds when `a.imag` (the imaginary part of a) is equal to `0`. Note the double equals sign, ==. This distinction is very important to differentiate the equality operator, ==, from the assignment operator, =, that we used before when defining variables. Using the assignment operator, =, instead of the equality operator, ==, in a conditional expression will trigger an error (this is usually considered a virtue of Python, as many other languages would accept assignment operators in conditions, but evaluate them in unobvious ways producing errors that are hard to track down).

Other operators that can be used in conditional expressions include != (not equal), >= (bigger than or equal to), <= (less than or equal to), > (bigger than) and < (less than). Multiple conditional expressions can be combined together with `and`, `or` and `not`. To illustrate the concept, try running the following lines of code <u>one at a time</u>:

```
1 == 2                         # False
1 != 2                         # True
not (1 == 2)                   # True
(1 != 2) and (1 != 3)          # True
(1 == 2) and (1 != 3)          # False
(1 == 2) or  (1 != 3)          # True
(not (1 == 2)) and (1 != 3)    # True
```

Line 1 evaluates to `False`, because 1 is not equal to 2. Line 2 evaluates to `True` for the exact same reason. `not` is a unitary operator, meaning that it acts on a single expression that follows it. Its role is to turn `True` into `False` and `False` into `True` (i.e. it inverts the value). In line 3, `1==2` on its own would evaluate to `False`. With the `not` operator in front of it, it evaluates to `True` instead. Both `and` and `or` are binary operators, as they join two different conditional expressions into one. `or` evaluates to `True` when at least one of the expressions is `True`, while `and` evaluates to `True` only when both expressions are `True`. Finally, multiple operators can be combined together and the order of operations can be altered with brackets as demonstrated by line 7.

It is interesting that even non-conditional expressions can be used in `if` and `elif` statements. Consider the following example:

```python
if 42:
    print "Python thinks that 42 is True"

if 'Area 51':
    print "Python thinks that Area 51 is True"

if not 0:
    print "Python thinks that 0 is False"

if not "":
    print "Python thinks that an empty string is False"
```

When a non-conditional expression is encountered in an `if` or `elif` statement, it will be typecast into one (this kind of automatic typecasting is sometimes called *type-juggling*, emphasizing its unobvious nature and wide room for mistakes). The rules are that all non-zero numbers evaluate to `True` and so do all non-empty strings.

Conditional expressions may sometimes be referred to as logical expressions or Boolean expressions (after George Boole, who formalized a branch of algebra involving expressions that can only have one of two possible values, such as `True` and `False`). Any data type can be typecast into a `True` or `False` value with `bool()`.

While-loops

Loops allow running the same block of code multiple times until a certain condition is satisfied. This differentiates them from functions, which run only when “called”. Python supports two different types of loops: *while-loops* and *for-loops* (the third common type of loops in programming languages, *until-loops*, is not supported). Consider an example while-loop below:

```python
1.  i = 1
2.  while i <= 10:
3.      print i
4.      i += 1
```

The block of code after the `while` statement (lines 3 and 4) will run again and again until the conditional expression after `while` is satisfied. Every run of the loop is known as *iteration*. In every iteration, line 3 will print the current value of `i` and line 4 will increase the value of `i` by 1 (`i+=1` is a shorthand for `i=i+1`; `i-=1`, `i*=1` and `i/=1` will also work, but structures like `i--` and `i++` are not supported, despite being very common in many other languages). Ultimately, the loop will run 10 times, printing numbers from 1 to 10.

If the condition of the loop is never satisfied, the loop will continue iterating indefinitely and the program will “hang”. This behavior can be recreated if line 4 is removed. When this happens, the notebook may become unresponsive, consuming arbitrary amounts of RAM and CPU time. The evaluation of the cell can be stopped by choosing `Kernel->Interrupt` in the main menu.

Within the body of the loop, two additional statements are available: `continue` terminates the current iteration of the loop and skips to the next one, `break` exists the loop right away. An example application of these statements is in the code snippet below. Try running the code and explaining any output it produces.

```python
1.  i = 1
2.  while i <= 10:
3.      print "Iteration started"
4.      print "i =", i
```

```
    i += 1

    if i == 5:
        continue

    if i == 7:
        break

    print "Iteration finished"

print "End of loop"
```

Lists and for-loops

List is a special data type that can store multiple values of any data type (including other lists!) at the same time. Consider the example below:

```
1.  my_first_list = [1, 2.5, "Area 51", 1+5j]
2.  my_second_list = [1 == 3, 2 != 4]
3.
4.  print my_first_list                   #  [1, 2.5, 'Area 51', (1+5j)]
5.  print my_first_list + my_second_list  #  [1, 2.5, 'Area 51', (1+5j), False, True]
6.  print my_first_list[2]                #  Area 51
7.
8.  i = 0
9.  while i < len(my_first_list):
10.     print i, ':', my_first_list[i]
11.     i += 1
```

Lines 1 and 2 define two different lists and save them in two variables called `my_first_list` and `my_second_list`. Lists are defined by comma-separating their values and enclosing all of them in `[]`. The first list contains 4 elements: an integer, a float, a string and a complex number. The second list is made of two conditional expressions, one of which evaluates to `False` and the other one to `True`.

Lines 4 prints the first list, while line 5 adds the two lists together (for lists, + results in concatenation) and prints the result. Line 6 demonstrates how an individual element of the list can be accessed, where 2 is the index of the element. Indexing begins with 0, so `[2]` refers to the third element of the list. An attempt to access a non-existent element of the list (e.g. `my_first_list[4]`) will trigger an error. Negative indices can be used to refer to elements in a reversed order. For example, `my_first_list[-1]` will access the last element of the list, `my_first_list[-2]` will access the second last element etc.)

Finally, lines 8 through 11 iterate over every element of the list in a while loop and print all of them as well as their indices. `len()` is a special function that returns the total number of elements in the list (in our case, 4).

Although it is perfectly valid to use while-loops to iterate over lists, Python offers a different type of loops that are designed specifically for this purpose. They are called *for-loops*. Consider the same program as above rewritten as a *for-loop* instead of a *while-loop*:

```
my_first_list = [1, 2.5, "Area 51", 1+5j]

for element in my_first_list:
    print element
```

A `for…in…:` statement expects a list to be iterated over after `in` and a variable (previously defined or not) where the iterated element of the loop will be placed in each iteration (this variable is called *iterant*) after `for`. `continue` and `break` can be used in for-loops just as well as in while-loops.

In the example above, we do not have access to the index of the element being iterated. If this is necessary, refer to the example below instead:

```
my_first_list = [1, 2.5, "Area 51", 1+5j]

for i, element in enumerate(my_first_list):
    print i, ':', element
```

`enumerate()` is a function that returns a special data type, specifically designed to be used in a for-loop, such that both the element being iterated and its index are accessible within the body of the loop. With the index known, list elements can be changed, as in the example below:

```python
my_first_list = [1, 2.5, "Area 51", 1+5j]

for i, element in enumerate(my_first_list):
    if element == "Area 51":
        my_first_list[i] = "Aliens"

print my_first_list
```

The snipped above searches the element "`Area 51`" and replaces it with "`Aliens`". It is imperative that `my_first_list[i]` is being overridden and not `element`, as `element` is a copy of the iterated element and not the element itself.

Now consider the following somewhat more complicated program that takes a list of numbers and sorts it in an ascending order. Run the code yourself and try to explain what is happening. If necessary, add more `print` statements to the code to see intermediate values of variables.

```python
to_sort = [5, 9, 1, -4, 12, 0.5]

for i, element_1 in enumerate(to_sort):
    for j, element_2 in enumerate(to_sort):
        if i >= j:
            continue     # Avoid checking the same elements twice
        if element_1 > element_2:
            to_sort[i] = element_2
            to_sort[j] = element_1
            element_1 = to_sort[i]
            element_2 = to_sort[j]

print to_sort            # Prints [-4, 0.5, 1, 5, 9, 12]
```

NumPy

So far, we have only considered native built-in features offered by Python. The true power of this language in the field of scientific computing lies in external modules, containing predefined functions for performing specific tasks. Of those, *NumPy* is perhaps the most famous one, encompassing a suite of tools for linear algebra, statistical analysis, differential equations and more. The module comes with the standard distribution of Anaconda and needs not to be installed separately. However, we have to tell Python that we want to use this module in our code with the `import` statement.

```python
1.  import numpy as np
2.  
3.  print np.pi          # The circle constant
4.  print np.e           # Euler's constant
5.  
6.  # Trigonometric functions
7.  x = 5
8.  print np.sin(5), np.cos(5), np.tan(5)
9.  
10. # Statistics
11. my_list = [1, 5, 25, 50]
12. print np.mean(my_list), np.std(my_list)
```

Line 1 tells Python to load the NumPy module and make its content accessible in our code. `numpy` is the name of the module and `np` is an arbitrary alias that can be used instead of `numpy` throughout the code to make it shorter. Any alias can be chosen, but `np` is the standard convention.

Once imported, `np` acts as an object. It has properties (such as `pi` and `e`), similarly to how complex numbers have `imag` and `real`. It also has functions (technically, methods), such as `sin()`, `cos()` and many-many more. Both properties and functions are accessible through typing `np` followed by a period (`.`), followed by the name of the desired function or property.

NumPy has extensive online documentation, covering most of its features. If the purpose of any of the functions in the snippet above is not clear, look up their exact specification in the official documentation at https://docs.scipy.org/doc/numpy-1.15.0/genindex.html.

The central concept of NumPy is *NumPy arrays*. They are a data type that can hold multiple values at the same time, similarly to Python's native lists. The snippet below summarizes a few ways in which a NumPy array can be created:

```
print np.ones(5)                     # Create an array of five "ones"
print np.zeros(6)                    # Create an array of six "zeros"
print np.random.uniform(0, 1, 10)    # Create an array of ten random numbers between 0 and 1
print np.array([1, 3, 6])            # Create an array from a list
print np.linspace(1, 10, 19)         # Create an array of 19 evenly spaced numbers between 1 and 10

# Arrays can be saved in files for future use and loaded from them
a = np.random.uniform(0, 1, 10)      # Generate some random numbers
np.savetxt('random.dat', a)          # Save them in the file random.dat
b = np.loadtxt('random.dat')         # Load them back from the file into b
print b                              # Print them
```

The advantage of NumPy arrays over lists is in automatic iteration of any applied operations. For example, when a number is added to a NumPy array (using $+$), it will be automatically added to every element of that array. When two arrays of equal sizes are multiplied together, every element of the first array will be automatically multiplied by the corresponding element of the second array and so forth. With lists, similar manipulations must be performed in a for-loop or a while-loop for every element individually. The code snippet below exploits this feature of NumPy arrays to integrate $\sin(x)$ from $x = 0$ to $x = \pi$:

```
x = np.linspace(0, np.pi, 1000)        # Array of 1000 evenly spaced numbers between 0 and pi
                                       # sin(x) will be "sampled" at these points
y = np.sin(x)                          # Compute sin(x) for all 1000 numbers (no loops necessary)
dx = x[1] - x[0]                       # Difference between adjacent x values
print "Integral:", np.sum(dx * y)
```

The power of NumPy will be heavily utilized in this course and more of its features will be covered in subsequent lectures.

MatPlotLib

Just like NumPy, **MatPlotLib** is a module available within Anaconda. It accommodates a variety of tools for data visualization, including plots, histograms, image rendering and more. When working with large amounts of data (which we most often will be), simply printing individual values will not be sufficient to understand what is going on. This is where MatPlotLib and its functions become useful.

The example below produces a plot of $\ln(x)$ with essential labeling:

```
1.  import matplotlib.pyplot as plt
2.  %matplotlib inline
3.
4.  x = np.linspace(2, 5, 1000)
5.  y = np.log(x)
6.  plt.plot(x, y, 'k-', label = 'Plot', lw = 4)
7.  plt.xlabel('$x$', size = 20)
8.  plt.ylabel('$\ln(x)$', size = 20)
9.  plt.title('My first MatPlotLib plot')
10. plt.grid()
11. plt.legend(loc = 'upper left')
```

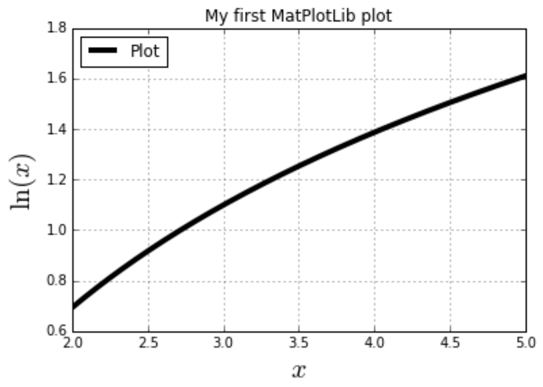
Line 1 imports MatPlotLib or, rather, a specific submodule within MatPlotLib called `pyplot`. Line 2 is not a Python line. Instead, it is a command to be interpreted by the environment (the notebook) that forces it to display all output figures within the code cell (without this command, the environment will open all plots in separate windows, which many programmers find less convenient).

Lines 4 and 5 precompute the values of $\ln(x)$ between $x = 2$ and $x = 5$ at 1000 evenly space points. This is identical to the previous example with integration. Line 6 is where MatPlotLib-driven features begin. `plot()` is a function within MatPlotLib that contains almost 50 arguments determining various parameters of the desired plot. Fortunately, most of them are not mandatory. The first two arguments are the $x$ and y (horizontal and vertical axes) of the data to be plotted.

The third argument is a string that determines the style of the plot. Namely, the first character determines the color (`k` for black, `r` for red, `b` for blue; more at https://matplotlib.org/2.0.2/api/colors_api.html). The second character determines the line style (− for solid; more at https://matplotlib.org/gallery/lines_bars_and_markers/line_styles_reference.html).

Python allows specifying values of arguments out of order by giving the names of desired arguments, separated from their values by the equals sign (=). Note that this is a special syntax that is only used when coding a function call and is not the assignment operator that we encountered before when creating and updating variables. We use this mechanism to provide values for two more arguments: `label` sets the legend label of the data series and `lw` specifies the width of the line (it is a shorthand of `linewidth`).

Analogically, lines 7 and 8 set the horizontal and vertical axes labels and their sizes. Note that both strings are wrapped in `$`, which signals to MatPlotLib that the values are written in the LaTeX markup language. This is merely an aesthetic feature and can be omitted. Line 9 gives the plot a title, line 10 enables the background grid (no arguments needed, unless the grid requires customizations!) and line 11 displays the legend in the top-left corner of the figure. Evaluating the cell should produce a plot similar to below:



As with NumPy, MatPlotLib has extensive online documentation at https://matplotlib.org/ and will be explored much further throughout the course.