Python (Semester 2 2024)

Introduction to Python (II)

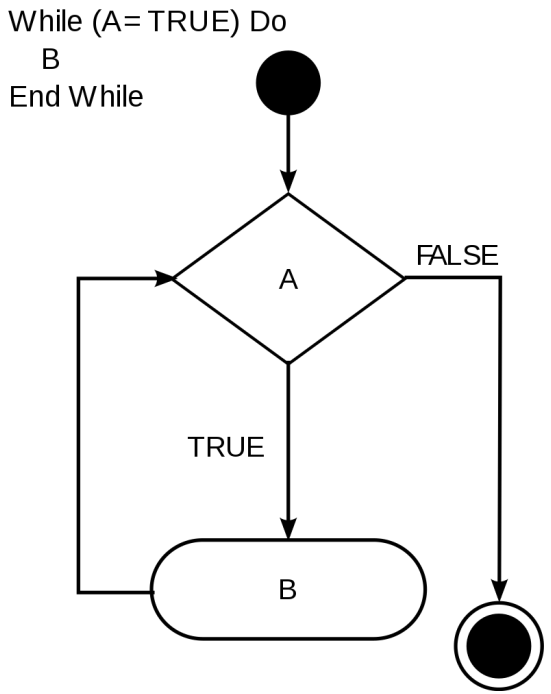
Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

August 26, 2024

Motivation

We have seen how to write simple Python commands.

To put those commands together into Python scripts, we will see how to use **control flow statements**, and also how to access more complex algorithms from **libraries**.



Writing a Program

So far, we have interacted with Python entirely using the iPython interpreter and a Jupyter notebook. While both are quick and easy ways to interact with Python, and Jupyter notebook especially an excellent way to demonstrate functionality, it is unsuitable for **realistic projects**.

Most of the time we will write what is known as scripts, or programs.

Definition

A program is a self-contained list of commands that are stored in a file that can be read by Python.

Writing a Program

So far, we have interacted with Python entirely using the iPython interpreter and a Jupyter notebook. While both are quick and easy ways to interact with Python, and Jupyter notebook especially an excellent way to demonstrate functionality, it is unsuitable for **realistic projects**.

Most of the time we will write what is known as scripts, or programs.

Definition

A program is a self-contained list of commands that are stored in a file that can be read by Python.

Essentially, it is a text file, with each line being the exact syntax you would have typed into the terminal.

Python then opens your program and runs it through the interpreter, line by line.

Writing a Program

For example, if this is what you did in interpreter before:

```
[IN]: import numpy as np
[IN]: import matplotlib.pyplot as plt
[IN]: x = np.arange(100)
[IN]: y = x**2 + np.sin(3*x)
```

then you could write a program in a text file that looked like this:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(100)
y = x**2 + np.sin(3*x)
```

To start the program from the terminal:

```
$ python3 simple_program.py
```

Control Flow Statements

So far, we have Python programs that are interpreted line by line in the order of their line number: sequential commands.

Control Flow Statements

So far, we have Python programs that are interpreted line by line in the order of their line number: sequential commands.

The real power of programming, however, lies in our ability to write programs that don't just contain a list of sequential commands but which execution depends on various inputs. This is done by **control flow statements**.

Conditional Statements

Definition

A conditional statement begins a defined, separated block of code which only executes (runs) if the conditional statement is evaluated by the interpreter to be true. Essentially, you are telling the computer to *only run this block of code IF some condition is true*. The condition itself is determined by the programmer.

Conditional Statements

Definition

A conditional statement begins a defined, separated block of code which only executes (runs) if the conditional statement is evaluated by the interpreter to be true. Essentially, you are telling the computer to *only run this block of code IF some condition is true*. The condition itself is determined by the programmer.

In Python, there are three forms of the `if...else` statement:

- `if` statement
- `if...else` statement
- `if...elif...else` statement

Conditional Statements

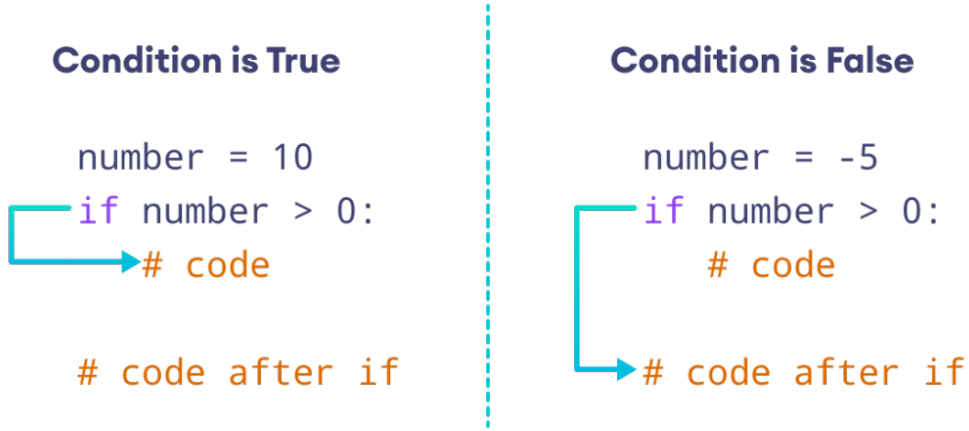
The syntax of the `if` statement in Python is:

```
if condition:
    # body of if statement
```

The `if` statement evaluates `condition`:

If `condition` is evaluated to `True`, the code inside the body of if is executed.

If `condition` is evaluated to `False`, the code inside the body of if is skipped.



Conditional Statements

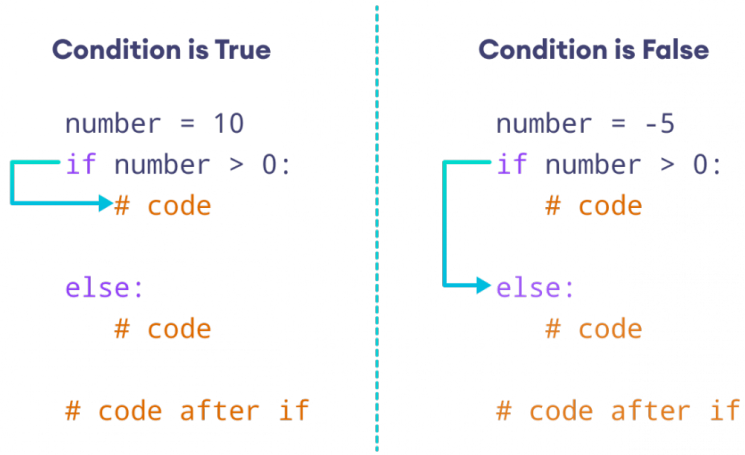
An `if` statement can have an optional **else** clause.
The syntax of the `if...else` statement is:

```
if condition:
    # block of code if condition is True
else:
    # block of code if condition is False
```

The `if...else` statement evaluates the given condition:

If the `condition` evaluates to `True`,
the code inside `if` is executed
the code inside `else` is skipped

If the `condition` evaluates to `False`,
the code inside `else` is executed
the code inside `if` is skipped

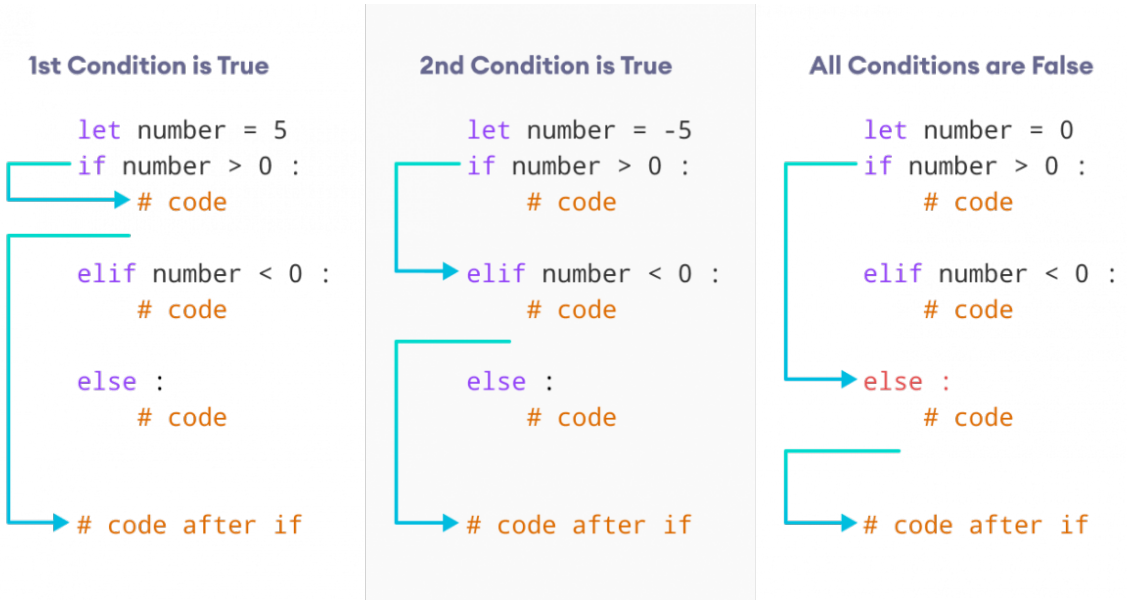


Conditional Statements

The `if...else` statement is used to execute a block of code among two alternatives.
However, if we need to make a choice between more than two alternatives, then we use the `if...elif...else` statement.

The syntax of the `if...elif...else` statement is:

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```



Conditional Statements

We can also use an if statement inside of an if statement. This is known as a nested if statement.

The syntax of nested if statement is:

```
# outer if statement
if condition1:
    # statement(s)

    # inner if statement
    if condition2:
        # statement(s)
```

Notes:

- We can add `else` and `elif` statements to the inner `if` statement as required.
- We can also insert inner `if` statement inside the outer `else` or `elif` statements (if they exist).
- We can nest multiple layers of if statements.

Conditional Statements

An example:

```
#Example: A Simple Conditional
x = 5
y = 6

if (2*x**2 > y**2):
    print(''condition holds'')
```

The `if` keyword tells the interpreter to evaluate the truthiness of the rest of the line, up to the colon. In the case above, the if-statement would indeed print `condition holds`, because $2 \times 5^2 = 50 > 36$.
Like for functions, all lines to be considered part of the conditional must be indented.

Conditional Statements

Python also provides other conditionals:

==	equal
>	greater than
<	less than
>=	greater or equal
<=	less or equal
!=	not equal

caution:
In Python, a single = sign is reserved for setting the values of variables.

Combining Conditionals

We are not limited to one conditional per statement; we can combine as many as we need (within reason).

```
#Example: Multiple Conditionals
x = input('Enter a number:')
x = float(x)
y = 15
z = 20

if (x > y) and (x != z):
     print ('cannot evaluate z')
if (z > x) or (x != y):
     z =x +y +z
     print ('z was evaluated and is ', z)
```

Combining Conditionals

We are not limited to one conditional per statement; we can combine as many as we need (within reason).

```
#Example: Multiple Conditionals
x = input('Enter a number:')
x = float(x)
y = 15
z = 20

if (x > y) and (x != z):
     print ('cannot evaluate z')
if (z > x) or (x != y):
     z =x +y +z
     print ('z was evaluated and is ', z)
```

Here we have 2 `if`-statements, with the two possible combinations of conditionals, `or` and `and`. These statements can be combined indefinitely (for example, `if ((a and b and c) or (d and f))`.

Loops

The two primary loops in Python are the `while` and `for` loops.

Loops

The two primary loops in Python are the `while` and `for` loops.

Definition

A `while`-loop is repeats a specific block of code sequentially as long as a certain condition is met.

Loops

The two primary loops in Python are the `while` and `for` loops.

Definition

A `while`-loop is repeats a specific block of code sequentially as long as a certain condition is met.

```
#Example: A while-loop

x = 100 # initialize x
while x > 5: #as long as x is greater than 5 run the indented code
     print x
     x = x -1
print('loop finished')
```

Eventually, after 95 times through the loop (and 95 prints), `x` would become 6-1=5, which would no longer satisfy the `while` statement. The interpreter would then move on to the next line of code outside of the loop.

Loops

Definition

A `for`-loop is a set off block of code that contains a temporary variable known as an iterator, and runs the block of code over and over for different specified values of that iterator.

Loops

Definition

A `for`-loop is a set off block of code that contains a temporary variable known as an iterator, and runs the block of code over and over for different specified values of that iterator.

```
#Example: A for-loop
arr = [1,2,3,4,5,6,7,8,9,10]
for i in arr:
     if i %2 ==0:
          print i
```

Loops

Definition

A `for`-loop is a set off block of code that contains a temporary variable known as an iterator, and runs the block of code over and over for different specified values of that iterator.

```
#Example: A for-loop
arr = [1,2,3,4,5,6,7,8,9,10]
for i in arr:
     if i %2 ==0:
          print i
```

The % sign means "modulo", and the conditional would read "if `i` divided by two has a remainder of 0" (the even numbers). The letter `i` is a generalized iterator: with `for i in arr` you are telling the computer to run the block of code, replacing `i` in the block with the first second, third, etc. element in the array. (You could use any character/combination of characters, but `i` is standard practice (followed by `j`, and `k` if necessary).

Loops

We have seen that there is a condition to end the loop, the **break condition** (also known as *test condition*). If we had not included the `x = x-1` part of the code, `x` would never end up being 5 or less.

For this it is important when using loops to **not forget the break condition**. Otherwise the loop will not end.

Loops

We have seen that there is a condition to end the loop, the **break condition** (also known as *test condition*). If we had not included the `x = x-1` part of the code, `x` would never end up being 5 or less.

For this it is important when using loops to **not forget the break condition**. Otherwise the loop will not end.

In case a loop won't finish or will simply take too long and you decide to interrupt it:

Python interpreters have built-in keyboard shortcuts to interrupt a program. (Usually this is Ctrl + C).

Loops

Multiple loops can be **nested** in case of iterating over more than one value in your code. This often happens when dealing with two-dimensional arrays.

Loops

Multiple loops can be **nested** in case of iterating over more than one value in your code. This often happens when dealing with two-dimensional arrays.

```
# Example: Iterating a 2D Array
for i in range(len(x)-1):
     for j in range(len(y)-1):
          if arr[i,j]<1500.:
               arr[i,j]=0
```

Loops

Multiple loops can be **nested** in case of iterating over more than one value in your code. This often happens when dealing with two-dimensional arrays.

```
# Example: Iterating a 2D Array
for i in range(len(x)-1):
     for j in range(len(y)-1):
          if arr[i,j]<1500.:
               arr[i,j]=0
```

In the above example, `x` and `y` would be variables representing the coordinates in the array. This particular block of code would run through every combination of `i`, `j` reaching each element in the 2D array, and if the value at any given point was below the 1500 threshold, it would just set that element to be 0.

List Comprehension

Loops and if statements are both **computationally expensive operations**, so look for ways to reduce the number you use to **speed up your code**.

Use list comprehension instead:

List Comprehension

Loops and if statements are both **computationally expensive operations**, so look for ways to reduce the number you use to **speed up your code**.

Use list comprehension instead:

In general: List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list. List comprehensions can utilize conditional statement to modify existing list (or other tuples).

This becomes especially efficient when using `numpy` for list comprehension:

One of the main benefits of libraries such as `numpy` is that they are designed for efficiency in mathematical operations on arrays.

Thus: Do not use any other technique if you can use list comprehension.

List Comprehension

For example, here is a code to list all the numbers between 1 and 1000 that is the multiplier of 3:

```
L = []
for i in range (1, 1000):
    if i%3 == 0:
        L.append(i)
```

Using list comprehension, it would be:

```
L = [i for i in range (1, 1000) if i%3 == 0]
```

The list L will be populated by the items in range from 0-1000 if the item's value is divisible by 3.
List comprehension works faster than using the append method.

Loops

We can also replace the code from above for iterating a 2D array:

```
# Example: Iterating a 2D Array
for i in range(len(x)-1):
     for j in range(len(y)-1):
          if arr[i,j]<1500.:
               arr[i,j]=0
```

Using list comprehension with a `where` statement, we get:

```
array_name[np.where(array_name < 1500)[0]] = 0
```

List Comprehension

Key Points to Remember about List Comprehension

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in a `for` loop, but not every `for` loop can be rewritten in the form of list comprehension.

List Comprehension

Key Points to Remember about List Comprehension

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in a `for` loop, but not every `for` loop can be rewritten in the form of list comprehension.

more performance tips:

`https://wiki.python.org/moin/PythonSpeed/PerformanceTips`

Libraries

Earlier we saw how to do basic math with Python. We also used various data types that are available already within Python (such as integers, floats, strings, lists).

However, once a code requires more **sophisticated analytical tools** (especially for astronomical processes), it becomes apparent that the built-in Python functions are not sufficient. Luckily, there are hundreds of functions that have been written to accomplish these tasks, most of which are organized into what are called libraries.

Definition

A library is a maintained collection of functions which can be installed and imported into a Python code to be used.

Numpy (which we already used) and Scipy are examples of libraries.

Most Python distributions come with a lot of these libraries included, and the installation of new libraries is generally straightforward.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python itself can't. For example, Python doesn't provide trigonometric functions.
That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python can't. For example, Python doesn't provide trigonometric functions. That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Astropy is a library with functions specifically for astronomical applications:
we will e.g. use it to import `fits` files (containing science images and tables).

Libraries

There are 4 key libraries that we will discuss in detail in this course: numpy, matplotlib, astropy, scipy.

NumPy is an extremely versatile library of functions to do things Python can't. For example, Python doesn't provide trigonometric functions. That's where NumPy comes in.

Matplotlib is a library with functions dedicated to plotting data and making graphs.

Astropy is a library with functions specifically for astronomical applications:
we will e.g. use it to import `fits` files (containing science images and tables).

Scipy is a library that contains special functions that are often used in science.

Libraries

How to find the right library and function within?

Since there are thousands of these functions, instead of memorizing them all, the best way to learn is to Google or query Stack Exchange for the type of function you are looking for, and you'll find a library containing the function you need. The ones you use most often will then become second nature.

Libraries

One of the key features of Python is that the actual core language is fairly small. This is an intentional design feature to maintain simplicity. Much of the powerful functionality comes through external modules and packages.

Libraries

One of the key features of Python is that the actual core language is fairly small. This is an intentional design feature to maintain simplicity. Much of the powerful functionality comes through external modules and packages. The main work of installation so far has been to supplement the core Python with useful modules for science analysis.

Module

A module is simply a file containing Python definitions, functions, and statements. Putting code into modules is useful because of the ability to import the module functionality into your code, for instance:

```
import astropy
import astropy.table
data = astropy.table.Table.read('my_table.fits')
```

You will find `import` in almost every Python script.

Libraries

Package

A package is just a way of collecting related modules together within a single tree-like hierarchy. Very complex packages like NumPy or SciPy have hundreds of individual modules so putting them into a directory-like structure keeps things organized and avoids name collisions. For example here is a partial list of sub-packages available within SciPy:

`scipy.fftpack`	Discrete Fourier Transform algorithms
`scipy.stats`	Statistical Functions
`scipy.lib`	Python wrappers to external libraries
`scipy.lib.blas`	Wrappers to BLAS library
`scipy.lib.lapack`	Wrappers to LAPACK library
`scipy.integrate`	Integration routines
`scipy.linalg`	Linear algebra routines
`scipy.sparse.linalg`	Sparse Linear Algebra
`scipy.sparse.linalg.eigen`	Sparse Eigenvalue Solvers

Importing Libraries


```
#Example: Importing libraries
import numpy
import astropy
import astropy.table
import astropy.io.fits
```

The dot notation of some imports is associated with classes. Some libraries are huge, so it is best to only load the functions you really need.

When using those functions, again the dot notation is used to let Python know from which libary the function you are calling is coming from.

```
#Example: sin function
import numpy
x = numpy.arange(100)
y = numpy.sin(x)
```

Importing Libraries

When importing libraries, one can name them whatever we want for the purposes of our code. A standard choices is:

```
import numpy as np
```

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

The easiest way to see if a package is in `pip` is to just try to `pip install` it, if it works then you're done:

```
$ pip install packagename
```

Installing Libraries

As mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way is to use `pip`, a package installer already available on most computers.

The easiest way to see if a package is in `pip` is to just try to `pip install` it, if it works then you're done:

```
$ pip install packagename
```

If it says "not found" then you might have to look up online how to install it. Usually it is required to download an archive and running a Python script like the following:

```
$ python setup.py install
```

Data Input/Output

A file is a container in computer storage devices used for storing data.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a **file operation** takes place in the following order:

- Open a file
- Read or write (perform operation)
- Close the file

Data Input/Output

To open a file for reading, we use:

```
file1 = open('filename.txt','r')
```

where `'r'` indicates we plan to write to the file.
A final `close` statement above tells Python to save and close the file.
When we are done with performing operations on the file, we need to properly close the file to free up the resources that were tied with the file.

```
#Example: Reading from a File
# open a file
file1 = open('test.txt', 'r')

# read the file
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

Data Input/Output

In a similar way we can **write** to files.

There are two things we need to remember while writing to a file.

- If we try to open a file that doesn't exist, a new file is created.
- If a file already exists, its content is erased, and new content is written.

In order to write into a file in Python, we need to open it in write mode by passing `'w'` inside `open()` as a second argument.
Suppose, we don't have a file named `test2.txt`. Let's see what happens if we write contents to that file.

```
#Example: Writing to a File

with open(test2.txt', 'w') as file2:
    # write contents to the test2.txt file
    file2.write('This is a test file.')
    fil2.write('Added second line.')
```

Data Input/Output

Different modes to open a file in Python

mode	description
`r`	Open a file for reading. (default)
`w`	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`x`	Open a file for exclusive creation. If the file already exists, the operation fails.
`a`	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
`t`	Open in text mode. (default)
`b`	Open in binary mode.
`+`	Open a file for updating (reading and writing)

Data Input/Output

So far, we were ignoring issues like columns of varying lenght and possible inproper values. A better option in such cases is provided by `numpy` as the function `np.genfromtxt`.

What `genfromtxt` does is the following: Internally it runs two main loops. The first loop converts each line of the file in a sequence of strings. The second loop converts each string to the appropriate data type. This mechanism is slower than a single loop, but gives more flexibility. In particular, `genfromtxt` is able to take missing data into account.

Data Input/Output

As example, suppose we have the following text file called `my_data.txt`:

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

We import this file, while assigning different types to different columns:

```
#Example: Importing a file with numpy

import numpy as np

a = np.genfromtxt("my_data.txt",
dtype=[np.int, 32 int, np.float, 32 float])
print(a)
```

Data Input/Output

We get the following output:

```
array([(1, 2, 3., 4.), (5, 6, 7., 8.)],
  dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f4'), ('f3', '<f8')])
```

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

We can take a look at the documentation of that library or function (and I strongly recommend that).
We can, however, also do someting straight within the interpreter. Simply type

A final thought on working with packages

How do we know how a function actually works, what its inputs and outputs are?

We can take a look at the documentation of that library or function (and I strongly recommend that).
We can, however, also do someting straight within the interpreter. Simply type

```
[IN]: help(np.genfromtxt)
```

(plug in your function of choice) and Python will give you a helpful rundown of how the function works. To advance through the documentation, keep hitting Enter , or hit Q to exit out of it.

An Outlook: Data Exploration

With now knowing about control flow statements, the usage of libraries and data I/O, you are now well equipped to put all of this together to write more complex code for data exploration.

next week:

September 2 Q&A Session
September 4 Project Idea Presentation