Astroinformatics I (Semester 1 2025)

Python Introduction

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

May 19, 2025

Motivation

We have seen how to write shell commands.

Shell commands are efficient tools for working with files and directories, such as creating and deleting files, merging files, removing columns from tables...

Once we want to implement more **sophisticated analytical tools**, other programming languages such as **Python** are a better choice.

In many programming languages, so-called **libraries** maintain functions that accomplish such complex tasks. Python offers libraries for tasks such as plotting, astronomical calculations, statistics, machine learning... and much more.

Why use Python?

The **Python ecosystem** provides a single environment that is sufficient for the vast majority of astronomical analysis. It does so on several levels:

Why use Python?

The **Python ecosystem** provides a single environment that is sufficient for the vast majority of astronomical analysis. It does so on several levels:

- open source, which means it is free (as opposed to proprietary languages (like IDL) which require you to buy a license)
- powerful language, many scientific libraries available.
- strong set of 3rd party analysis tools that are professionally and actively developed.
- robust methods for binding with C, C++ and FORTRAN libraries (speed, legacy)
- standard library support for web, GUI, databases, process management, etc.
- very active user and developer community

Why use Python?

The Python 3 documentation can be found here:
`https://docs.python.org/3/`

In addition, Python libraries typically are very well documented, such as `mathplotlib`'s documentation:
`https://matplotlib.org/`.

Python in Astronomy

Python has become the language of choice for astronomers and astrophysicists working with data analysis and visualization.

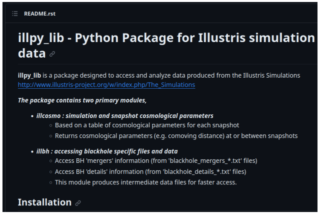
Python in Astronomy

Python has become the language of choice for astronomers and astrophysicists working with data analysis and visualization.

examples:

Cosmologists running Large-scale simulations (usually with C++ or Fortran for efficiency) use Python for **analyzing and visualizing** the results of their simulations.

Alert brokers allow **rapid access to astronomical observations** for the astronomical community world-wide.


README.rst

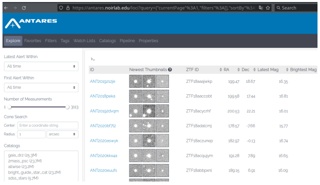
illpy_lib - Python Package for Illustris simulation data

illpy_lib is a package designed to access and analyze data produced from the Illustris Simulations http://www.illustris-project.org/w/index.php/The_Simulations

The package contains two primary modules,

- ***illcosmo : simulation and snapshot cosmological parameters***
  - Based on a table of cosmological parameters for each snapshot
  - Returns cosmological parameters (e.g. comoving distance) at or between snapshots
- ***illbh : accessing blackhole specific files and data***
  - Access BH 'mergers' information (from 'blackhole_mergers_*.txt' files)
  - Access BH 'details' information (from 'blackhole_details_*.txt' files)
  - This module produces intermediate data files for faster access.

Installation


https://antares.noirlab.edu/loci?query={"currentPage"%3A1,"filters"%3A[],"sortBy"%3A

Latest Alert Within: All time
First Alert Within: All time
Number of Measurements: 1 – 3113
Cone Search
Center: Enter a coordinate string
Radius: 1 arcsec
Catalogs
gaia_dr2 (25.3M)
2mass_psc (23.7M)
allwise (23.2M)
bright_guide_star_cat (23.2M)
sdss_stars (5.7M)

ID	Newest Thumbnails	ZTF ID	RA	Dec	Latest Mag	Brightest Mag
ANT2019zszje		ZTF18aaajwkp	199.47	18.67	16.35	
ANT2018peka		ZTF18aaccobt	199.58	17.44	16.81	
ANT2019zdvqm		ZTF18acycrhf	200.53	22.21	16.01	
ANT2020bf7t2		ZTF18adalcmj	178.57	-7.66	15.77	
ANT2020aswyk		ZTF18aczunep	182.97	-0.13	16.74	
ANT2020kkx4a		ZTF18acqujym	191.28	7.89	16.65	
ANT2020euufs		ZTF18abbpxnl	189.15	6.91	16.09	

The Python Language

Python is an **interpreted high-level programming language** for general-purpose programming.

The Python Language

Python is an **interpreted high-level programming language** for general-purpose programming.

Definitions:

An *interpreted* language is a language in which the implementations execute instructions directly without earlier compiling a program into machine language.

The Python Language

Python is an **interpreted high-level programming language** for general-purpose programming.

Definitions:

An *interpreted* language is a language in which the implementations execute instructions directly without earlier compiling a program into machine language.

High-level programming languages are designed to allow humans to write computer programs without having to have specific knowledge of the processor or hardware that the program will run on.

The Python Language

Python is an **interpreted high-level programming language** for general-purpose programming.

Definitions:

An *interpreted* language is a language in which the implementations execute instructions directly without earlier compiling a program into machine language.

High-level programming languages are designed to allow humans to write computer programs without having to have specific knowledge of the processor or hardware that the program will run on.
Examples of high-level programming languages in active use today include:

- Python
- Java
- C++
- C#
- Visual Basic

Installing Python

Nowadays many Linux and UNIX distributions, and even some Windows computers come with Python already installed.

If you do need to install Python: it is easy, download packages can be found on the official website:

`https://www.python.org/`

`https://www.python.org/downloads/`

Installing Python

The Anaconda Repository

It features over 8,000 open-source data science and machine learning packages, built and compiled for all major operating systems and architectures.

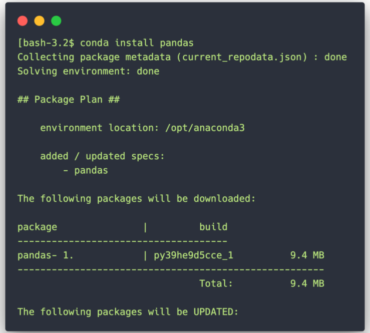
`https://www.anaconda.com/`

jupyter
SciPy
SPYDER
NumPy
DASK
bokeh
scikit learn
pandas
TensorFlow
Numba
Panel
matplotlib
PYTORCH
HoloViews
CONDA

Installing Python

Conda

Conda is an open-source package and environment management system for Windows, macOS, and Linux. Conda quickly installs, runs, and updates packages and their dependencies. It also easily creates, saves, loads, and switches between environments on your local computer. Despite created for Python programs, it can package and distribute software for any language.



```
[bash-3.2$ conda install pandas
Collecting package metadata (current_repodata.json) : done
Solving environment: done

## Package Plan ##

    environment location: /opt/anaconda3

    added / updated specs:
        - pandas

The following packages will be downloaded:

package               |         build
-------------------------------------
pandas- 1.            | py39he9d5cce_1          9.4 MB
------------------------------------------------------
                                Total:          9.4 MB

The following packages will be UPDATED:
```

Installing Python

Python can be used in a variety of ways: from the console (terminal), by loading program code previously written and saved in an editor, or as Jupyter notebooks.

We use Python 3.x. Python 2 is officially not supported as of January 1, 2020. In case you're still on Python 2.7, upgrade now as this course has a lot of code that won't run on Python 2.

Installing Python

Python can be used in a variety of ways: from the console (terminal), by loading program code previously written and saved in an editor, or as Jupyter notebooks.

We use Python 3.x. Python 2 is officially not supported as of January 1, 2020. In case you're still on Python 2.7, upgrade now as this course has a lot of code that won't run on Python 2.

After the installation process is done, go to your shell's command prompt via Terminal:

```
$ python --version
```

The response should look like:

```
Python 3.5.1 :: Anaconda 2.4.0 (x86_64)
```

Interacting with Python

Within Python, you have access to two different terminals, the Python terminal and the iPython terminal (interactive Python).

To begin an interactive iPython session, simply type

```
$ ipython
```

You should see your prompt change to

```
[IN]:
```

with a line number. You are now in iPython.

To instead run a Python script, simply type

```
$ python3 my_script.py
```

Interacting with Python

As it's a tradition to begin a programming course with a guide to showing the canonical phrase "Hello World" on the screen, we just do so.

All you have to do is type

```
print('hello world!')
```

in iPython and press `Enter`. The terminal will respond by showing your phrase in the output of the line below.

Interacting with Python

```
print('hello world!')
```

Definition

A print statement is a line of code which tells the interpreter (the program that turns your commands into computer bits and executes them) to output something to the screen.

In the above the phrase "hello world!" was placed inside quotation marks, and being surrounded by parenthesis.
Here, `print` is acting as a **function**, something that takes an argument and returns a result, similar to a mathematical function such as $sin(x)$.
The quotes, on the other hand, are described in the next section - they are the `string` data type, and can be either single or double quoted.

Data Types

Python, like most programming languages, uses **data types**.

While some data types are obvious, other data types seem more arbitrary at first glance: For example Python makes the distinction between integers (the counting numbers), and floats (numbers with decimals). It does so because of the way computer processors store information in bits, but it leads to the interesting (and important) characteristic that 42 and `42.` are different in Python, and take up different amounts of computer memory.

Data Types

Some **basic data types** are listed and defined below:

Data Types

Some **basic data types** are listed and defined below:

Integers: The counting numbers. Examples: `-1`, `0`, `1`, `2`, `3`, `4`, `5`, ...

Floats: Decimal numbers. Examples: `1.`, `2.345`, `6.3`, `999.99999`, ...

Strings: An iterable data type most commonly used to hold words/phrases or path locations. Denoted by single or double quotes. Examples: `"cat"` , `"/home/username/directory"`, `"1530"`, ...

Boolean: A data type with only two possible values: `True`, or `False`. They are used in conditional statements.

Lists: Stored lists of any combination of data types, denoted with brackets. Examples: `[1,2,'star','fish']` or `[1, 2, [3, 4, 5], 'star']` (notice that you can have lists within lists)

Numpy Arrays: Like lists, but can only contain one data type at a time, and have different operations. Defined in `numpy`, not native Python, but so ubiquitous we include them here.

Data Types

Python also allows for **composed data types** such as the ones below:

Tuples: Also like a list, but immutable (un-changable). These are defined with parentheses. Example: `tuple_1 = ('hi', 1, 4, 'bye')`

Dictionaries: A collection of pairs, where one is a *key* and the other is a *value*. One can access the value attached to a key by indexing the dictionary by key:
`dictionary_name['key']`

Basic Mathematical Operations

Within Python you can perform simple to very complex mathematical operations. Let's see how adding and subtracting works in iPython.

Basic Mathematical Operations

Within Python you can perform simple to very complex mathematical operations. Let's see how adding and subtracting works in iPython.

```
[IN]: 3 + 5
[OUT]: 8
[IN]: 9 - 3
[OUT]: 6
```

Basic Mathematical Operations

Within Python you can perform simple to very complex mathematical operations. Let's see how adding and subtracting works in iPython.

```
[IN]: 3 + 5
[OUT]: 8
[IN]: 9 - 3
[OUT]: 6
```

We can also test multiplication and division (denoted in Python with ∗ and /):

```
[IN]: 4 * 3
[OUT]: 12.0
[IN]: 1 / 2
[OUT]: 0.5
```

Basic Mathematical Operations

In Python 2.x the result of a division of integers was an integer. Now in Python 3.x the result is a float.

Basic Mathematical Operations

In Python 2.x the result of a division of integers was an integer. Now in Python 3.x the result is a float.
For your general knowledge, there is a function for converting integers to floats: `float(·)`, e.g. `float(2)`. This is called **hard casting**. Examples:

```
[IN]: x = float(x)
[IN]: x = int(x)
```

Basic Mathematical Operations

In Python 2.x the result of a division of integers was an integer. Now in Python 3.x the result is a float.
For your general knowledge, there is a function for converting integers to floats: `float(·)`, e.g. `float(2)`. This is called **hard casting**. Examples:

```
[IN]: x = float(x)
[IN]: x = int(x)
```

A much faster way to create floats when you are entering a number manually, which is simply to add a decimal (period) to any number, e.g. `12.`, `1./2`.

Basic Mathematical Operations

Another basic math operation in Python is exponentiation. Is denoted with a double asterisk (∗∗). An example:

```
[IN]: 2**3
[OUT]: 8
```

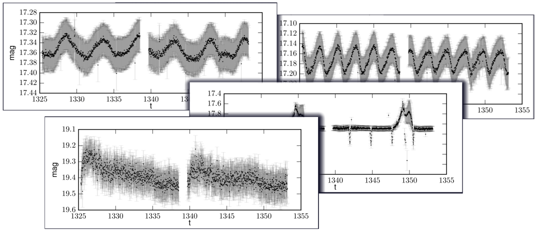
To perform operations like sin, cos, sqrt, etc., the use of some additional **packages** is required.

Storing and Manipulating Data in Python

Our primary goal for using Python as astronomers is as a tool with which to explore and manipulate data.

different types of data typically found in **astronomy**:

astronomical images, light curves, spectra, the output files of a supercomputer simulation, etc.

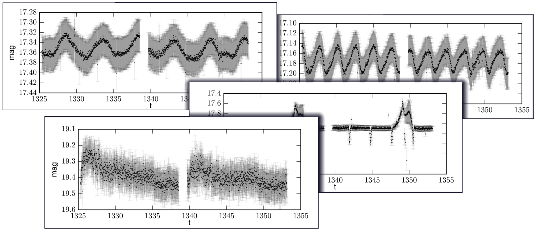


Storing and Manipulating Data in Python

Our primary goal for using Python as astronomers is as a tool with which to explore and manipulate data.

different types of data typically found in **astronomy**:

astronomical images, light curves, spectra, the output files of a supercomputer simulation, etc.



what all of these share in common is that they represent the plural inherent in the word "data": we might have a collection of 10,000, or even over a million stars or other objects

Variables

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Variables

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Declaring variables in Python is easy. For example:

```
[IN]: x = 5.0
[IN]: y = 'cat'
[IN]: header = 'Signal ' + 'Analysis'
```

Variables

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Declaring variables in Python is easy. For example:

```
[IN]: x = 5.0
[IN]: y = 'cat'
[IN]: header = 'Signal ' + 'Analysis'
```

The latter is a way to concatenate a string.

Variables

Notice that Python doesn't output anything when you declare a variable as it did when you entered a math operation. But you can print them by typing:

```
[IN]: print(x)
[OUT]: 5.0
```

Variables

Variables in Python are mutable - they can be changed. An example:

Variables

Variables in Python are mutable - they can be changed. An example:

```
[IN]: x = 5
[IN]: x = 3
```

Variables

Variables in Python are mutable - they can be changed. An example:

```
[IN]: x = 5
[IN]: x = 3
```

If you output `x` you would find it is equal to 3. Another example:

```
[IN]: x = 4
[IN]: x = 2 * x + 7
```

Variables

Variables in Python are mutable - they can be changed. An example:

```
[IN]: x = 5
[IN]: x = 3
```

If you output x you would find it is equal to 3. Another example:

```
[IN]: x = 4
[IN]: x = 2 * x + 7
```

In this case, the new value for x at the end of the line would be 2 times the value of x going in, plus 7. More of this:

Variables

Variables in Python are mutable - they can be changed. An example:

```
[IN]: x = 5
[IN]: x = 3
```

If you output `x` you would find it is equal to 3. Another example:

```
[IN]: x = 4
[IN]: x = 2 * x + 7
```

In this case, the new value for `x` at the end of the line would be 2 times the value of `x` going in, plus 7. More of this:

```
[IN] : x = 4.
[IN] : y = 3.
[IN] : z = x * y
[IN] : x = z + 2
```

That is probably a bit confusing to follow, and **illustrates why typically we avoid redefining variables**, and instead come up with new variables to store the results of computations.

Data Structures

Arrays vs. Lists

We begin with lists, as these are the native data type within Python. Let's define a list:

Data Structures

Arrays vs. Lists

We begin with lists, as these are the native data type within Python. Let's define a list:

```
[IN]: my_list = [1,5,2,7,3,7,8]
```

Data Structures

Arrays vs. Lists

We begin with lists, as these are the native data type within Python. Let's define a list:

```
[IN]: my_list = [1,5,2,7,3,7,8]
```

Assume this list contains the rounded distances in parsecs of several nearby stars. We multiply by 2, as e.g. an equation has a factor of $2d$:

```
[IN]: my_list*2
[OUT]: [1,5,2,7,3,7,8,1,5,2,7,3,7,8]
```

Data Structures

Arrays vs. Lists

We begin with lists, as these are the native data type within Python. Let's define a list:

```
[IN]: my_list = [1,5,2,7,3,7,8]
```

Assume this list contains the rounded distances in parsecs of several nearby stars. We multiply by 2, as e.g. an equation has a factor of $2d$:

```
[IN]: my_list*2
[OUT]: [1,5,2,7,3,7,8,1,5,2,7,3,7,8]
```

This obviously didn't work as intended: a new list with the original list repeated N times was created.
This default behavior this actually makes sense: recall that lists can contain any data type, and indeed any combination of data types. If our list contained a mix including strings or any other non-numerical data type, this operation would fail if defined this way.

Data Structures

Arrays vs. Lists

The correct way to multiply the list elements by a factor N is the following: We have do utilize what is known as **iterating** to go through the list one by one and replace each value with N times itself. Example:

```
[IN]: [i*2 for i in my_list]
[OUT]: [2,10,4,14,6,14,16]
```

As it turns out, there is a shorter way to apply mathematical operations to every element of a collection. Numpy arrays allow to perform mathematical operations on entire arrays all at once. Example:

```
[IN]: my_array = np.array([1,2,3])
[IN]: my_array*2
[OUT]: array([2,4,6])
```

This is also faster than iteration as arrays are being treated computationally as matrices, which can be computed very efficiently.

Data Structures

From this we see: Particularly when dealing with large datasets, you almost certainly will be working with (numpy) arrays.

How to operate on those array containers? How to access data?

Data Structures

From this we see: Particularly when dealing with large datasets, you almost certainly will be working with (numpy) arrays.

How to operate on those array containers? How to access data?

The procedure of extracting subsets of a larger array/list is known as **slicing**, or **indexing**.
By convention, this index starts with 0, rather than one (this is true for most programming languages). Here is a sample list, with the indices for each entry listed below:

```
list_1 = [1, 2, 3, 4, 'star', 198]
index:    0  1  2  3  4       5
```

Data Structures

From this we see: Particularly when dealing with large datasets, you almost certainly will be working with (numpy) arrays.

How to operate on those array containers? How to access data?

The procedure of extracting subsets of a larger array/list is known as **slicing**, or **indexing**.
By convention, this index starts with 0, rather than one (this is true for most programming languages). Here is a sample list, with the indices for each entry listed below:

```
list_1 = [1, 2, 3, 4, 'star', 198]
index:    0  1  2  3  4        5
```

We now extract the first entry of the array:

```
[IN]: list_1 = [1, 2, 3, 4, 'star', 198]
[IN]: x = list_1[0]
[IN]: print x
[OUT]: 1
```

Array Manipulation

```
list_1 = [1, 2, 3, 4, 'star', 198]

index:     0  1  2  3  4       5
```

We can also extract the last entry (in a very convenient way):

```
[IN]: list_1 = [1, 2, 3, 4, 'star', 198]
[IN]: x = list_1[0]
[IN]: print x
[OUT]: 1
[IN]: y = list_1[-1]
[IN]: print y
[OUT]: 198
```

The same works for the second last entry, and so on.

Array Manipulation

There is also a way to slice through multiple indices at once. The format is as follows:

```
var = array[from:to]
```

Array Manipulation

There is also a way to slice through multiple indices at once. The format is as follows:

```
var = array[from:to]
```

An example: We have a string '123456789' and we want to extract the first four elements and convert them into an integer:

```
[IN]: x = '123456789'
[IN]: H = int(x[0:4])
[IN]: print(H)
[IN]: 1234
```

Array Manipulation

There is also a way to slice through multiple indices at once. The format is as follows:

```
var = array[from:to]
```

An example: We have a string ’123456789’ and we want to extract the first four elements and convert them into an integer:

```
[IN]: x = '123456789'
[IN]: H = int(x[0:4])
[IN]: print(H)
[IN]: 1234
```

Caution: Python slicing will not include the end index.

Array Manipulation

There is also a way to slice through multiple indices at once. The format is as follows:

```
var = array[from:to]
```

An example: We have a string '123456789' and we want to extract the first four elements and convert them into an integer:

```
[IN]: x = '123456789'
[IN]: H = int(x[0:4])
[IN]: print(H)
[IN]: 1234
```

Caution: Python slicing will not include the end index.
As a shortcut, if you are starting from the beginning, or slicing from some midpoint to the end, you can omit the 0 or the final index after, i.e.,

```
[IN]: print x[0:4]
```

is equivalent to

```
[IN]: print x[:4]
```

Array Manipulation

Often astronomical data (like images or tables) are stored in not higher-dimensional arrays, essentially matrices described by 2 indices, a row and a column.

Array Manipulation

Often astronomical data (like images or tables) are stored in not higher-dimensional arrays, essentially matrices described by 2 indices, a row and a column.

Example:

```
[IN]: print A
[OUT]: [[1 , 3, 4, 5, 6]
[ 4, 5, 9, 3, 7]
[ 9, 4, 6, 7, 1 ]]
```

We slice it with two indices, row, then column.

Array Manipulation

Often astronomical data (like images or tables) are stored in not higher-dimensional arrays, essentially matrices described by 2 indices, a row and a column.

Example:

```
[IN]: print A
[OUT]: [[1 , 3, 4, 5, 6]
[ 4, 5, 9, 3, 7]
[ 9, 4, 6, 7, 1 ]]
```

We slice it with two indices, row, then column.
Caution: row then column translates into (y,x), which is the opposite of how we are usually taught to determine ordered pairs of coordinates.

Array Manipulation

Often astronomical data (like images or tables) are stored in not higher-dimensional arrays, essentially matrices described by 2 indices, a row and a column.

Example:

```
[IN]: print A
[OUT]: [[1 , 3, 4, 5, 6]
[ 4, 5, 9, 3, 7]
[ 9, 4, 6, 7, 1 ]]
```

We slice it with two indices, row, then column.
Caution: row then column translates into (y,x), which is the opposite of how we are usually taught to determine ordered pairs of coordinates.

Example: From the above data structure, we want to get the 3 in the second row:

```
[IN]: A[1][3]
[OUT]: 3
```

Array Manipulation

Often astronomical data (like images or tables) are stored in not higher-dimensional arrays, essentially matrices described by 2 indices, a row and a column.

Example:

```
[IN]: print A
[OUT]: [[1 , 3, 4, 5, 6]
[ 4, 5, 9, 3, 7]
[ 9, 4, 6, 7, 1 ]]
```

We slice it with two indices, row, then column.
Caution: row then column translates into (y,x), which is the opposite of how we are usually taught to determine ordered pairs of coordinates.

Example: From the above data structure, we want to get the 3 in the second row:

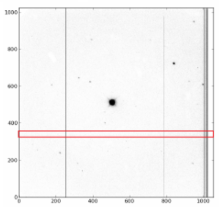
```
[IN]: A[1][3]
[OUT]: 3
```

Alternatively, you can use the comma syntax `A[1,3]` to equal effect.

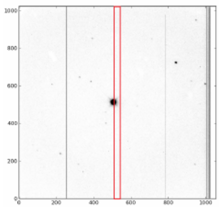
Array Manipulation

Given a 2D array, we can also illustrate this in the following way:

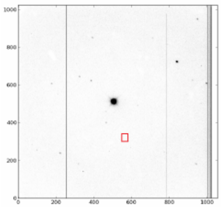
1



2



3



1. `array[350:370,:]`
this takes the full rows 350-370 in the image

2. `array[:,350:360]`
this takes the full columns 350-360 in the image

3. `array[350:370, 350:360]`
this takes the box of the region between rows 350-370 and columns 350-360 in image

Dictionaries

The final primary data container data type in Python are dictionaries.

Definition

A dictionary is a Python container, like a list or array, but which uses *keys* instead of indices to specify elements within the container. That is, the order of elements (values) in a dictionary is irrelevant, and values are retrieved by indexing for the appropriate key (which can be almost anything).

Dictionaries

Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas. Example:

```
simple_dict = {'key1':value1, 'key2':value2}
```

Dictionaries

Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas. Example:

```
simple_dict = {'key1':value1, 'key2':value2}
```

To pull a value the dictionary, the key is used. Example:

```
pulled_value = simple_dict['key1']
```

Dictionaries

Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas. Example:

```
simple_dict = {'key1':value1, 'key2':value2}
```

To pull a value the dictionary, the key is used. Example:

```
pulled_value = simple_dict['key1']
```

We can also easily change values in a dictionary, or add new key-value pairs, using this index notation. Example:

```
simple_dict['key2'] = new_val
```

Dictionaries

Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas. Example:

```
simple_dict = {'key1':value1, 'key2':value2}
```

To pull a value the dictionary, the key is used. Example:

```
pulled_value = simple_dict['key1']
```

We can also easily change values in a dictionary, or add new key-value pairs, using this index notation. Example:

```
simple_dict['key2'] = new_val
```

To add a new key-value pair:

```
simple_dict['new_key'] = new_value
```

These examples use string keys, but any other data type would be possible.

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**.

The control flow statements are similar to what we saw for `bash` scripts.

Conditional Statements

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

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

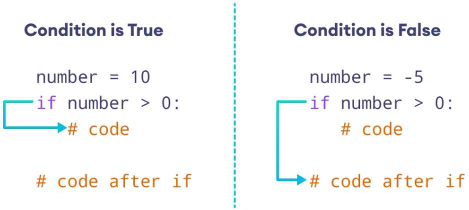
Conditional Statements

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.



```
number = 10
if number > 0:
    # code

# code after if
```

```
number = -5
if number > 0:
    # code

# code after if
```

Conditional Statements

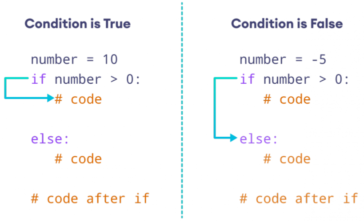
An `if` statement can have an optional **else** clause.
The syntax of the `if...else` statement is:

```
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



```
number = 10
if number > 0:
    # code

else:
    # code

# code after if
```

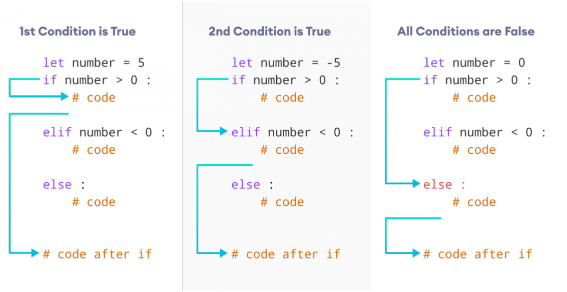
```
number = -5
if number > 0:
    # code

else:
    # code

# code after if
```

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
```



```
let number = 5
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

```
let number = -5
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

```
let number = 0
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

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:

```
arr[np.where(arr < 1500)] = 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`

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')])
```

An Outlook: Data Exploration

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

We will also see how **libraries** can assist us with data exploration by providing sophisticated algorithms.