Computer tools for astronomers

Thøger Rivera-Thorsen

Stockholm University,
dept. of Astronomy
trive@astro.su.se

October 23, 2013

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactive vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

DAY 1:

- Unix/Linux
- The command line
- LaTeX

DAY 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview

Day 1:

- Unix/Linux
- The command line
- LaTeX

Day 2:

- NumPy/SciPy
- Matplotlib
- Interactve vs. scripting

Overview, continued

Day 3:

- AstroPy FITS file handling
- Simple version control with `git`

Overview, continued

Day 3:

- AstroPy FITS file handling
- Simple version control with `git`

What is Ubuntu?

- A GNU/Linux-based Operating System
- Can run on most computers, alone or in parallel with e.g. Windows or Mac OS X
- Best way to learn is by doing!
- More information on `http://www.ubuntu.com`

What is Ubuntu?

- A GNU/Linux-based Operating System
- Can run on most computers, alone or in parallel with e.g. Windows or Mac OS X
- Best way to learn is by doing!
- More information on `http://www.ubuntu.com`

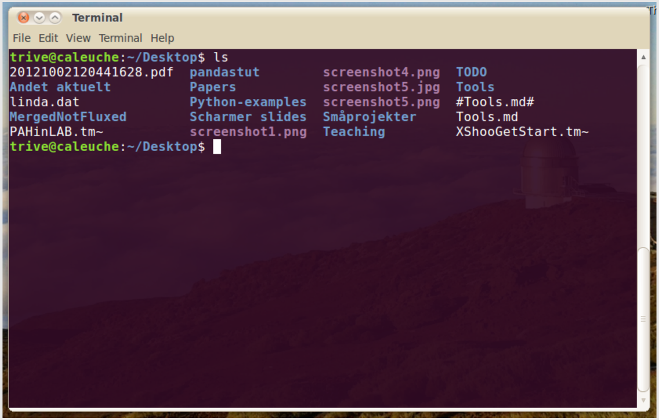
What is Ubuntu?

- A GNU/Linux-based Operating System
- Can run on most computers, alone or in parallel with e.g. Windows or Mac OS X
- Best way to learn is by doing!
- More information on http://www.ubuntu.com

What is Ubuntu?

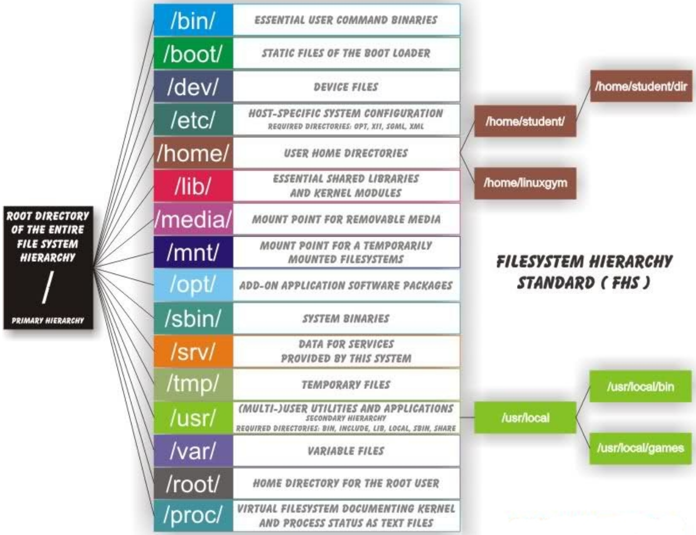
- A GNU/Linux-based Operating System
- Can run on most computers, alone or in parallel with e.g. Windows or Mac OS X
- Best way to learn is by doing!
- More information on `http://www.ubuntu.com`

The Command Line



```
trive@caleuche:~/Desktop$ ls
20121002120441628.pdf  pandastut        screenshot4.png  TODO
Andet aktuelt          Papers           screenshot5.jpg  Tools
linda.dat              Python-examples  screenshot5.png  #Tools.md#
MergedNotFluxed        Scharmer slides  Småprojekter     Tools.md
PAHinLAB.tm~           screenshot1.png  Teaching         XShooGetStart.tm~
trive@caleuche:~/Desktop$ 
```

File system structure


ROOT DIRECTORY OF THE ENTIRE FILE SYSTEM HIERARCHY
/
PRIMARY HIERARCHY
/bin/
ESSENTIAL USER COMMAND BINARIES
/boot/
STATIC FILES OF THE BOOT LOADER
/dev/
DEVICE FILES
/etc/
HOST-SPECIFIC SYSTEM CONFIGURATION
REQUIRED DIRECTORIES: OPT, X11, SGML, XML
/home/
USER HOME DIRECTORIES
/lib/
ESSENTIAL SHARED LIBRARIES AND KERNEL MODULES
/media/
MOUNT POINT FOR REMOVABLE MEDIA
/mnt/
MOUNT POINT FOR A TEMPORARILY MOUNTED FILESYSTEMS
/opt/
ADD-ON APPLICATION SOFTWARE PACKAGES
/sbin/
SYSTEM BINARIES
/srv/
DATA FOR SERVICES PROVIDED BY THIS SYSTEM
/tmp/
TEMPORARY FILES
/usr/
(MULTI-)USER UTILITIES AND APPLICATIONS
SECONDARY HIERARCHY
REQUIRED DIRECTORIES: BIN, INCLUDE, LIB, LOCAL, SBIN, SHARE
/var/
VARIABLE FILES
/root/
HOME DIRECTORY FOR THE ROOT USER
/proc/
VIRTUAL FILESYSTEM DOCUMENTING KERNEL AND PROCESS STATUS AS TEXT FILES
/home/student/
/home/student/dir
/home/linuxgym
FILESYSTEM HIERARCHY STANDARD (FHS)
/usr/local
/usr/local/bin
/usr/local/games

Short cheat sheet:

Essential commands

`cd`	*change directory*
`ls`	*list directory*
`mv`	*move file/directory*
`cp`	*copy file to location*
`rm`	*remove/delete file*
`mkdir`	*create directory*
`rmdir`	*remove directory*
`pwd`	*print working dir ("where am I?")*
`man`	*display manual for program*
`top`	*see running processes*
`ssh`	*securely log on to remote computer*
`scp`	*secure copy between machines*

Special characters

`.`	*current directory*
`..`	*one level up/parent directory*
`∼`	*home directory (=`/home/me/`)*
`/`	*root directory*
`*`	*wildcard*
`!`	*wildcard, single char.*
`$`	*evaluate (e.g.* `echo $PATH`*)*
`-, --`	*set options for command*
`&`	*run in background*

Graphical programs

```
firefox
firefox &
  firefox -new-tab ∼/myfile.html
emacs --help
gnome-open Song.mp3
```
open Song.mp3 in default player

Examples

```
cd ∼/Desktop/
mkdir testdir
cd testdir
touch file1 file2 file3
  mkdir subdir && mv file2 subdir/
cd ..  && cp -r testdir testdir2
zip -r test.zip testdir
rm testdir/*
```
create 3 empty files (for `touch file1 file2 file3`)

CAUTION! `rm -rf ./*` *(Don't do this!)*

Prepare pretty documents with LaTeX:

- A different approach to creating a document.
- Word processor: An “extended typewriter” (“wysiwyg”)
- LaTeX: An “electronic publishing house” - *semantic* editing.

Prepare pretty documents with LaTeX:

- A different approach to creating a document.
- Word processor: An “extended typewriter” (“wysiwyg”)
- LaTeX: An “electronic publishing house” - *semantic* editing.

Prepare pretty documents with LaTeX:

- A different approach to creating a document.
- Word processor: An “extended typewriter” (“wysiwyg”)
- LaTeX: An “electronic publishing house” - *semantic* editing.

Philosophy behind LaTeX:

- Separate content from layout.
- Think structure and content, let the software take care of the looks (sorta like HTML/CSS).
- LaTeX-Document is a plain text file with text and commands describing the structure. LaTeX will then make a pretty document based on the text and the instructions.

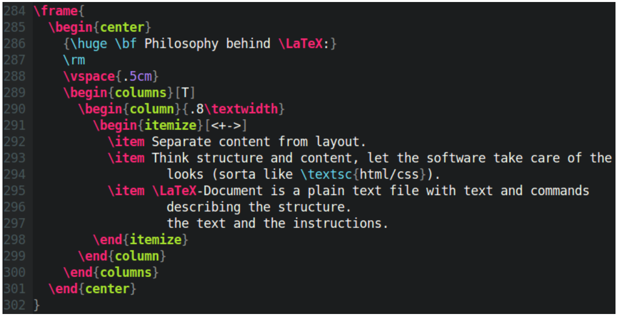
Philosophy behind LaTeX:

- Separate content from layout.
- Think structure and content, let the software take care of the looks (sorta like HTML/CSS).
- LaTeX-Document is a plain text file with text and commands describing the structure. LaTeX will then make a pretty document based on the text and the instructions.

Philosophy behind LaTeX:

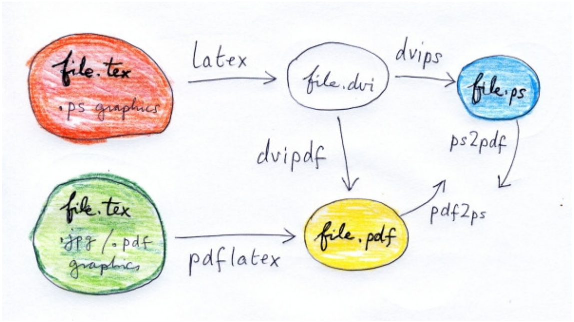
- Separate content from layout.
- Think structure and content, let the software take care of the looks (sorta like HTML/CSS).
- LaTeX-Document is a plain text file with text and commands describing the structure. LaTeX will then make a pretty document based on the text and the instructions.

Example: code behind last frame.



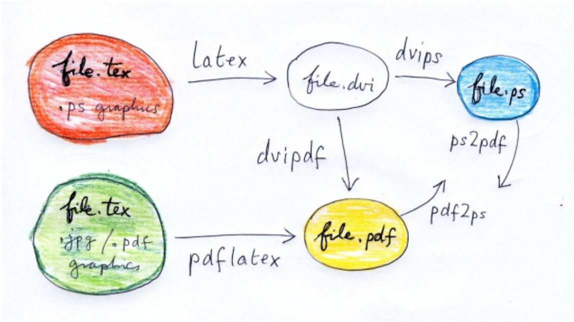
```
\frame{
  \begin{center}
    {\huge \bf Philosophy behind \LaTeX:}
    \rm
    \vspace{.5cm}
    \begin{columns}[T]
      \begin{column}{.8\textwidth}
        \begin{itemize}[<+->]
          \item Separate content from layout.
          \item Think structure and content, let the software take care of the
                looks (sorta like \textsc{html/css}).
          \item \LaTeX-Document is a plain text file with text and commands
                describing the structure.
                the text and the instructions.
        \end{itemize}
      \end{column}
    \end{columns}
  \end{center}
}
```

Workflow[1]:



[1]Figure courtesy of Andreas Sandberg

Workflow[1]:



- most people prefer `pdflatex` nowadays.

[1]Figure courtesy of Andreas Sandberg

Working environment

- Basic setup: a **text editor** (even Notepad will do) + a **terminal emulator**.
- Many more advanced working environments around, worth mentioning are *TeXnicCenter* (Windows), *TeXShop* (Mac), *Kile* (Linux), *Texmaker*, *Vim-latex*, *Emacs-AUCTeX*, *TeXWorks* (All the latter are cross-platform, TeXWorks is included in most LaTeX-downloads).
- Same functionality, the extra features are just convenience.
- For now, we **keep it simple**.

Working environment

- Basic setup: a **text editor** (even Notepad will do) + a **terminal emulator**.
- Many more advanced working environments around, worth mentioning are *TeXnicCenter* (Windows), *TeXShop* (Mac), *Kile* (Linux), *Texmaker*, *Vim-latex*, *Emacs-AUCTeX*, *TeXWorks* (All the latter are cross-platform, TeXWorks is included in most LaTeX-downloads).
- Same functionality, the extra features are just convenience.
- For now, we **keep it simple**.

Working environment

- Basic setup: a **text editor** (even Notepad will do) + a **terminal emulator**.
- Many more advanced working environments around, worth mentioning are *TeXnicCenter* (Windows), *TeXShop* (Mac), *Kile* (Linux), *Texmaker*, *Vim-latex*, *Emacs-AUCTeX*, *TeXWorks* (All the latter are cross-platform, TeXWorks is included in most LaTeX-downloads).
- Same functionality, the extra features are just convenience.
- For now, we **keep it simple**.

Working environment

- Basic setup: a **text editor** (even Notepad will do) + a **terminal emulator**.
- Many more advanced working environments around, worth mentioning are *TeXnicCenter* (Windows), *TeXShop* (Mac), *Kile* (Linux), *Texmaker*, *Vim-latex*, *Emacs-AUCTeX*, *TeXWorks* (All the latter are cross-platform, TeXWorks is included in most LaTeX-downloads).
- Same functionality, the extra features are just convenience.
- For now, we **keep it simple**.

Installing LaTeX:

Microsoft Windows: MiKTeX at http://miktex.org

GNU/Linux etc. TeXLive - get it from the software center/package manager.

Apple Mac OS X: MacTeX (Mac'ified version of TeXLive with extra OSX-specific system integration) at http://www.tug.org/mactex/

Useful guides to LaTeX:

Tobias Oetiker: The Not So Short Introduction To LaTeX 2_ε
http://tobi.oetiker.ch/lshort/lshort.pdf

Wikibooks: LaTeX
http:http://en.wikibooks.org/wiki/LaTeX

Useful guides to LaTeX:

Tobias Oetiker:	The Not So Short Introduction To LaTeX 2_ε
	http://tobi.oetiker.ch/lshort/lshort.pdf
Wikibooks:	LaTeX
	http:http://en.wikibooks.org/wiki/LaTeX

- Interpreted all-purpose programming language.
- Strongly **object oriented**, but can also be used for casual scripting [a].
- **the indentation *is* the structure** No closing brackets or end statements. A loop inside a loop[b] is simply indented by one more level.
- A number of **packages** form a scientific working environment together, in the style of MatLab, IDL etc.

[a]which is what we'll mostly do here

[b]which is generally **not a good idea**, see later.

- Interpreted all-purpose programming language.
- Strongly **object oriented**, but can also be used for casual scripting [a].
- **the indentation *is* the structure** No closing brackets or end statements. A loop inside a loop[b] is simply indented by one more level.
- A number of **packages** form a scientific working environment together, in the style of MatLab, IDL etc.

[a] which is what we'll mostly do here

[b] which is generally **not a good idea**, see later.

- Interpreted all-purpose programming language.
- Strongly **object oriented**, but can also be used for casual scripting [a].
- **the indentation *is* the structure** No closing brackets or `end` statements. A loop inside a loop[b] is simply indented by one more level.
- A number of **packages** form a scientific working environment together, in the style of MatLab, IDL etc.

[a] which is what we'll mostly do here
[b] which is generally **not a good idea**, see later.

- Interpreted all-purpose programming language.
- Strongly **object oriented**, but can also be used for casual scripting [a].
- **the indentation *is* the structure** No closing brackets or `end` statements. A loop inside a loop[b] is simply indented by one more level.
- A number of **packages** form a scientific working environment together, in the style of MatLab, IDL etc.

[a]which is what we'll mostly do here
[b]which is generally **not a good idea**, see later.

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Parts of the set-up

Python	The program that (among other things) interprets your program code and makes your commands happen.
Numpy	Module that provides N-Dimensional array handling.
SciPy	Module that provides a number of scientific subpackages for data analysis, optimization etc.
Matplotlib	Plotting functionality.
AstroPy	Library of astronomical tools, from which we'll use the FITS file handling.
IPython	Interactive Python working environment with many extra goodies for scientists[2].

[2] Seriously, *many* extra goodies

Installation:

Windows	Download Canopy Express or Anaconda (recommended) or easily set it up yourself.
Mac	Installers exist for all packages. Otherwise, install Canopy Express, Anaconda or easily install through Macports.
GNU/Linux	Install through software center/package manager or install Canopy Express or Anaconda.
All	Alternative way to keep packages up-to-date: `pip`.

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
    - `float` [floating-point number]
    - `int` [Integer]
    - `str` [Text string]
    - `list` [A list of things]
    - `tuple` [Like list, but immutable]
    - `dict` [An *associative map*: `{name: value}`].
    - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
    - `array` [N-dimensional data array]
    - `float64` [Double-precision float]
    - `recarray`, `masked_array` etc.
    - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name:   value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only array for now.*

Most important data types

- Built-in:

`float`	[floating-point number]
`int`	[Integer]
`str`	[Text string]
`list`	[A list of things]
`tuple`	[Like list, but immutable]
`dict`	[An *associative map*: `{name:   value}`].
`bool`	[Truth variable]

- *Modules can create their own data type objects.*
- NumPy:

`array`	[N-dimensional data array]
`float64`	[Double-precision float]

  - `recarray`, `masked_array` etc.
  - ***Only*** `array` ***for now.***

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - ***Only*** `array` ***for now.***

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
  - `float` [floating-point number]
  - `int` [Integer]
  - `str` [Text string]
  - `list` [A list of things]
  - `tuple` [Like list, but immutable]
  - `dict` [An *associative map*: `{name: value}`].
  - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
  - `array` [N-dimensional data array]
  - `float64` [Double-precision float]
  - `recarray`, `masked_array` etc.
  - *Only* `array` *for now.*

Most important data types

- Built-in:
    - `float` [floating-point number]
    - `int` [Integer]
    - `str` [Text string]
    - `list` [A list of things]
    - `tuple` [Like list, but immutable]
    - `dict` [An *associative map*: `{name: value}`].
    - `bool` [Truth variable]
- *Modules can create their own data type objects.*
- NumPy:
    - `array` [N-dimensional data array]
    - `float64` [Double-precision float]
    - `recarray`, `masked_array` etc.
    - ***Only*** `array` ***for now.***

OK, let’s get started with IPython:

```
import this                 # Easter egg...
import scipy                # Numpy comes automagically!
import astropy.io.fits
import matplotlib.pyplot # What’s the point mean?

a = scipy.array([1., 2., 3., 4.])
matplotlib.pyplot.plot(a)
matplotlib.pyplot.show()
```

...Too much writing! Alternatives:

```
from scipy import *   # Bad in scripts,
                      #     ok in interactive work.
import scipy as sp    # Better - keeps namespace clean.
```

OK, let’s get started with IPython:

```
import this                   # Easter egg...
import scipy                  # Numpy comes automagically!
import astropy.io.fits
import matplotlib.pyplot # What’s the point mean?

a = scipy.array([1., 2., 3., 4.])
matplotlib.pyplot.plot(a)
matplotlib.pyplot.show()
```

…Too much writing! Alternatives:

```
from scipy import *   # Bad in scripts,
                      #     ok in interactive work.
import scipy as sp    # Better - keeps namespace clean.
```

OK, let's get started with IPython:

```
import this                # Easter egg...
import scipy               # Numpy comes automagically!
import astropy.io.fits
import matplotlib.pyplot # What's the point mean?

a = scipy.array([1., 2., 3., 4.])
matplotlib.pyplot.plot(a)
matplotlib.pyplot.show()
```

…Too much writing! Alternatives:

```
from scipy import *  # Bad in scripts,
                     #     ok in interactive work.
import scipy as sp   # Better - keeps namespace clean.
```

Manipulating arrays

```python
import scipy as sp
import matplotlib.pyplot as plt
from scipy import random as ra   # Purely for laziness

a = sp.array([1., 2., 3.])       # Simplest way.
a = sp.zeros((8, 8)); print(a)   # Why double parantheses?
b = a.reshape((4, 16)); print(b) # How many rows/colums?
print(b.transpose())             # Or simply b.T

b = sp.ones_like(a)              # Same as sp.ones(a.shape)
c = ra.random(b.shape)           # Scale as you please.
d = sp.eye(b.shape[0])           # Array like unity matrix
e = sp.arange(64.).reshape((8, 8))
print(b + c)
print(b * d)
print(b ** (d*2))                # Etc. etc. ...
```

Slicing and dicing

See what you get out of the following:

```
print(e[0, 0])
print(e[:, 2:3])   # NB: in a[x:y], a[y] is *not* included.
print(e[2:4, :])
print(e[2:4])      # A 2D array is a stack of rows!
print(e[2:-1, 0])
print(e[::-1, 0:2]) # All, in reverse order.
print(e[1:6:2, :])  # Every other row between 2nd and 6th.
print(e[::2, :])    # Every other row, all columns.
# Of course, you can always assign these to new variables:
f = e[1:6:2, :]   # etc. etc.
```

Slicing and dicing

See what you get out of the following:

```
print(e[0, 0])
print(e[:, 2:3])   # NB: in a[x:y], a[y] is *not* included.
print(e[2:4, :])
print(e[2:4])      # A 2D array is a stack of rows!
print(e[2:-1, 0])
print(e[::-1, 0:2]) # All, in reverse order.
print(e[1:6:2, :])  # Every other row between 2nd and 6th.
print(e[::2, :])    # Every other row, all columns.
# Of course, you can always assign these to new variables:
f = e[1:6:2, :]   # etc. etc.
```

See `http://www.scipy.org/NumPy_for_Matlab_Users` for more info and functionality (also good if you don't know MatLab).

Loops, comparisons and conditional statements

Loops: `for`, `while`

Comparison and logical operators: <, <=, >, >=, `not`, `is`, `in`

Conditinal statements: `if`, `elseif`, `else`

Flow control: `break`, `continue`

Learn more at http://en.wikibooks.org/wiki/Python_Programming/Loops and http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements.

Loops, comparisons and conditional statements

Loops: `for`, `while`

Comparison and logical operators: `<`, `<=`, `>`, `>=`, `not`, `is`, `in`

Conditinal statements: `if`, `elseif`, `else`

Flow control: `break`, `continue`

Learn more at `http://en.wikibooks.org/wiki/Python_Programming/Loops` and `http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements`.

Loops, comparisons and conditional statements

Loops: `for`, `while`

Comparison and logical operators: `<`, `<=`, `>`, `>=`, `not`, `is`, `in`

Conditinal statements: `if`, `elseif`, `else`

Flow control: `break`, `continue`

Learn more at `http://en.wikibooks.org/wiki/Python_Programming/Loops` and `http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements`.

Loops, comparisons and conditional statements

Loops: `for`, `while`

Comparison and logical operators: `<`, `<=`, `>`, `>=`, `not`, `is`, `in`

Conditinal statements: `if`, `elseif`, `else`

Flow control: `break`, `continue`

Learn more at `http://en.wikibooks.org/wiki/Python_Programming/Loops` and `http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements`.

Loops, comparisons and conditional statements

Loops: `for`, `while`

Comparison and logical operators: `<`, `<=`, `>`, `>=`, `not`, `is`, `in`

Conditinal statements: `if`, `elseif`, `else`

Flow control: `break`, `continue`

Learn more at `http://en.wikibooks.org/wiki/Python_Programming/Loops` and `http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements`.

On loops vs. vectorization:

```python
def powerit(indata, expon):
    for i in range(indata.shape[0]):
        for j in range(indata.shape[1]):
            indata[i, j] = indata[i, j] ** expon
    return indata

%timeit e ** 2000
%timeit powerit(e, 2000)
# See the difference!
```

– We definitely want to do array-wise operations where we can!

On loops vs. vectorization:

```
def powerit(indata, expon):
    for i in range(indata.shape[0]):
        for j in range(indata.shape[1]):
            indata[i, j] = indata[i, j] ** expon
    return indata

%timeit e ** 2000
%timeit powerit(e, 2000)
# See the difference!
```

– We definitely want to do array-wise operations where we can!

Logical indexing

and the `where()` -function

```python
g = sp.arange(64.).reshape(8, 8)
evens = sp.where(g\% 2 == 0); print g[evens]
# Indexing can be based on a different array:
g[d==0] = 0; print g   # Shorthand notation for where().

g = sp.arange(64.).reshape(8, 8)  # Want the ol’ one back
# sp.where() is convenient for more elaborate indexing:
my_indices = sp.where(((g >= 10) & (g <= 20)) | (g >= 45))
h = sp.ones_like(g)
h[my_indices] = 0.; print h  # Works for all arrays with
                             #            same dimensions.
```

Stacking

Try the following:

```python
sp.hstack((b, c, d))
sp.vstack((b, c, d))
sp.dstack((b, c, d))  # Outputs 3D array
```

*Clever use of stacking, slicing and indexing can make for **much** faster and more efficient code than the use of loops (by a factor of 100)!*

Stacking

Try the following:

```
sp.hstack((b, c, d))
sp.vstack((b, c, d))
sp.dstack((b, c, d))  # Outputs 3D array
```

Clever use of stacking, slicing and indexing can make for ***much*** *faster and more efficient code than the use of loops (by a factor of 100)!*

Plotting

```python
import matplotlib.pyplot as plt
x = sp.linspace(-2*sp.pi, 2*sp.pi, 5000)
y = sp.sin(x)
z = sp.cos(x)
plt.plot(x, y, 'y-', label='sin(x)')
plt.plot(x, z, 'b-', label='cos(x)')
plt.legend()
plt.xlabel('Some text')
plt.ylabel ('Some other text')
plt.title('My very nice plot')
plt.savefig('myplot.pdf')  # Or ps, eps, png, jpg, svg, ...
```

A more thorough tutorial can be found at
http://matplotlib.org/users/pyplot_tutorial.html,
another very good tutorial at
http://cs.smith.edu/dftwiki/index.php/MatPlotLib_Tutorial_1,
and a gallery of examples with source code at
http://matplotlib.org/gallery.html

Plotting

```python
import matplotlib.pyplot as plt
x = sp.linspace(-2*sp.pi, 2*sp.pi, 5000)
y = sp.sin(x)
z = sp.cos(x)
plt.plot(x, y, 'y-', label='sin(x)')
plt.plot(x, z, 'b-', label='cos(x)')
plt.legend()
plt.xlabel('Some text')
plt.ylabel ('Some other text')
plt.title('My very nice plot')
plt.savefig('myplot.pdf')  # Or ps, eps, png, jpg, svg, ...
```

A more thorough tutorial can be found at http://matplotlib.org/users/pyplot_tutorial.html, another very good tutorial at http://cs.smith.edu/dftwiki/index.php/MatPlotLib_Tutorial_1, and a gallery of examples with source code at http://matplotlib.org/gallery.html

Interactive → script → program

Interactive session Flexible and quick for simple tasks, nice features in IPython; but not reproducible, and lots of work to change one parameter and re-run calculations.

Script A sequence of commands in a file. Called as `python myscript.py` from the command line or as `%run myscript` from inside IPython. Much easier to change one parameter and re-run calculations this way. Example: Just write the examples from these slides as a sequence in a file.

Program/module Many operations delegated to **functions**, **classes** and a `__main__` part. The program may be called from the system, run as a script and/or imported as a module, depending on its design.

Interactive → script → program

Interactive session Flexible and quick for simple tasks, nice features in IPython; but not reproducible, and lots of work to change one parameter and re-run calculations.

Script A sequence of commands in a file. Called as `python myscript.py` from the command line or as `%run myscript` from inside IPython. Much easier to change one parameter and re-run calculations this way. Example: Just write the examples from these slides as a sequence in a file.

Program/module Many operations delegated to **functions**, **classes** and a `__main__` part. The program may be called from the system, run as a script and/or imported as a module, depending on its design.

Interactive → script → program

Interactive session Flexible and quick for simple tasks, nice features in IPython; but not reproducible, and lots of work to change one parameter and re-run calculations.

Script A sequence of commands in a file. Called as `python myscript.py` from the command line or as `%run myscript` from inside IPython. Much easier to change one parameter and re-run calculations this way. Example: Just write the examples from these slides as a sequence in a file.

Program/module Many operations delegated to **functions**, **classes** and a `__main__` part. The program may be called from the system, run as a script and/or imported as a module, depending on its design.

Example: a very simple Python program.

```python
#! /usr/bin/env python
# -*- coding:utf-8 -*-

'''This is my docstring.
There are many like it, but this one is mine.'''

import scipy as sp
import matplotlib.pyplot as plt
from astropy.io import fits

def do_something(input):
    '''Individual functions should have doctrings, too!'''
    a = input + sp.random.random(input.shape) * input.max() * .3
    return a  # What does this function do?

if __name__ == '__main__':
    data = fits.getdata('example.fits')
    print do_something(data)
```

FITS files with `astropy.io.fits`

FITS Flexible Image Transport System. A file consists of one or more *Header and Data Units* (**HDU**’s). Each HDU consists of a text format **header** with keyword-value pairs; and a binary-format 1- or 2D array of numbers *or* a text format table of values.

Astropy is a library of Astronomy tools for Python, presently under heavy development. One of the submodules provides data input/output support for multiple formats, including FITS.

A comprehensive tutorial can be found here:

`http://docs.astropy.org/en/v0.2.4/io/fits/index.html`

FITS files with `astropy.io.fits`

FITS Flexible Image Transport System. A file consists of one or more *Header and Data Units* (**HDU**'s). Each HDU consists of a text format **header** with keyword-value pairs; and a binary-format 1- or 2D array of numbers *or* a text format table of values.

Astropy is a library of Astronomy tools for Python, presently under heavy development. One of the submodules provides data input/output support for multiple formats, including FITS.

A comprehensive tutorial can be found here:

`http://docs.astropy.org/en/v0.2.4/io/fits/index.html`

FITS files with `astropy.io.fits`

FITS Flexible Image Transport System. A file consists of one or more *Header and Data Units* (**HDU**'s). Each HDU consists of a text format **header** with keyword-value pairs; and a binary-format 1- or 2D array of numbers *or* a text format table of values.

Astropy is a library of Astronomy tools for Python, presently under heavy development. One of the submodules provides data input/output support for multiple formats, including FITS.

A comprehensive tutorial can be found here:

`http://docs.astropy.org/en/v0.2.4/io/fits/index.html`

Example: Working with FITS data

```python
from astropy.io import fits # Plus scipy & pyplot.

# Load data and header of first HDU by default:
data, head = fits.getdata('pix.fits', header=True)
head                                        # To see the header

# Quick and simple but inflexible:
plt.imshow(data, cmap='hot', vmin='-10', vmax='100')
colormaps?  # To see available cmap names in IPython
plt.axis((x1, x2, y1, y2)) # Insert sensible values here.

# More flexible but more work:
plt.pcolormesh(data[xmin:xmax, ymin:ymax], cmap='gray',
               vmin=0, vmax=500) # Insert values again.

# Write a fits file when you have data and header:
fits.writeto('myfile.fits', data, header=head)
```

Simple version control with Git

Because life is too short for

`project_report_final_jennyscomments_caspersupdates_ISwearThisTimeItsFinal_v2.pdf`

Version control is an incredibly powerful and potentialle incredibly complicated thing. As alwys, for now we shall **keep it simple**.

- Linear one-person workflow
- Small-group collaboration
- Simple branching

Simple version control with Git

Because life is too short for

`project_report_final_jennyscomments_caspersupdates_ISwearThisTimeItsFinal_v2.pdf`

Version control is an incredibly powerful and potentialle incredibly complicated thing. As alwys, for now we shall **keep it simple**.

- Linear one-person workflow
- Small-group collaboration
- Simple branching

Simple version control with Git

Because life is too short for

`project_report_final_jennyscomments_caspersupdates_ISwearThisTimeItsFinal_v2.pdf`

Version control is an incredibly powerful and potentialle incredibly complicated thing. As alwys, for now we shall **keep it simple**.

- Linear one-person workflow
- Small-group collaboration
- Simple branching

Simple version control with Git

Because life is too short for

`project_report_final_jennyscomments_caspersupdates_ISwearThisTimeItsFinal_v2.pdf`

Version control is an incredibly powerful and potentialle incredibly complicated thing. As alwys, for now we shall **keep it simple**.

- Linear one-person workflow
- Small-group collaboration
- Simple branching

Basic configuration and setup

```
# First, some basic configuration
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@server.net"
$ git config --global core.editor nano    # If Vi makes you sad
$ git config --global color.ui true           # Because COLORS!
# If you omit the --global option, the setup will only affect
# your current repository. Local setup overrides global.

# Now, make a project directory and initialize a git repository
$ mkdir myproject; cd myproject         # Or whatever you want
$ git init    # Also works perfecly fine in an existing folder!
$ ls .git/    # This is where git does its thing - don't touch!
```

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest useful one-person workflow

For your peace of mind

To get something useful out of `git`, you only need a few commands (but there are many, many more):

`git add`	Makes git aware of changes. Something that isn't `add`'ed yet is invisible to `git`.
`git rm`	Stop tracking, optionally remove file.
`git status`	Shows what has changed since last commit, and also shows untracked files. Use often!
`git commit`	The basic "save"-equivalent command. Each commit has a unique ID, and you add a message to each commit.
`git log`	Shows your history of commits. Try the delightful `--graph` option.
`git revert`	Undo a commit and save this as a new commit.
`git stash`	The equivalent of "Visitors! I'll throw all the mess into this box to deal with later!"

Simplest possible one-person workflow

Because you're worth it

Overview:

Most of the time: Make file → `git add` → Make changes → `git commit -a` → make more changes → etc. ad libitum.

At your convenience: `git status`, `git log`

Very occasionally: `git rm`, `git stash`, `git revert`

Getting our hands dirty

First steps

```
$ touch firstfile.txt
$ git status   # Use this often to stay in touch with your repo
$ git add firstfile.txt; git status    # The file is now STAGED
$ git commit -m 'Initial commit'; git status
$ git log --graph

# Now add a line to firstfile.txt.
$ git status
$ git add firstfile.txt
# Now add another line and save the file.
$ git status                                # Is this surprising?
$ git commit -m 'firstfile.txt now has one line'.  # Wait, not two?
# (Because the addition of the second line wasn't staged.)
# Call git commit -a to auto-stage all changes made since last time
```

Getting our hands dirty

Undoing and fixing things

```
$ git status
$ git add firstfile.txt
# We now decide we don’t want this new line in the file anyway.
# How do we get rid of it?
# Option 1:
$ git stash  # Removes but stores all changes since last commit
$ git stash apply   # To get your stashed changes reimplemented
# Option 2 (not in the command list from before)
$ git reset --hard             # Will *not* store the changes.
# Now make some change to a file, save and ‘git add‘ it.
$ touch secondfile.txt
$ git commit -am ’Changed firstfile.txt, added seconfile.txt’
# Oooooops!!
$ git add secondfile.txt
$ git commit --amend  # Only do this if commit is not published
```

More on fixing and undoing stuff at :
http://git-scm.com/book/en/Git-Basics-Undoing-Things

Branhcing & merging

Anxiety-free experimenting and more

Scenario: You have some code and calculations that kinda-sorta works. You want to try out a different approach, but don't want to risk messing up what you already have. No more:

`cp -r MyWorkDir MyWorkDir_safe_kindasortaworks_12Sept2010` !

Strategy:

- Create new branches for new **features** (this is good advice but not always easy to follow).
- Create new branch for new **ways** of doing things (e.g. splitting up code in many subfiles etc.)

New commands:

`git branch`, `git checkout`, `git merge`

Branhcing & merging

Anxiety-free experimenting and more

Scenario: You have some code and calculations that kinda-sorta works. You want to try out a different approach, but don't want to risk messing up what you already have. No more:

`cp -r MyWorkDir MyWorkDir_safe_kindasortaworks_12Sept2010` !

Strategy:

- Create new branches for new **features** (this is good advice but not always easy to follow).
- Create new branch for new **ways** of doing things (e.g. splitting up code in many subfiles etc.)

New commands:

`git branch`, `git checkout`, `git merge`

Branhcing & merging

Anxiety-free experimenting and more

Scenario: You have some code and calculations that kinda-sorta works. You want to try out a different approach, but don't want to risk messing up what you already have. No more:

`cp -r MyWorkDir MyWorkDir_safe_kindasortaworks_12Sept2010` !

Strategy:

- Create new branches for new **features** (this is good advice but not always easy to follow).
- Create new branch for new **ways** of doing things (e.g. splitting up code in many subfiles etc.)

New commands:

`git branch`, `git checkout`, `git merge`

Branhcing & merging

Anxiety-free experimenting and more

```
$ git branch experimental          # And a couple more for later
$ git branch          # The asterisk shows current working branch
$ git checkout experimental              # Now 'master' is safe
# Do your thing, make WILD and CRAZY, creative changes.
# Save, add and commit.

# Now, say we discover a bug in the ‘master‘ branch.
$ git checkout master
# Change something, save, add, commit.
$ git checkout experimental
# Make a few more changes, whatever you feel like.
# Now, we decide that our work in ‘experimental‘ is so good that
# we want to merge it into ‘master‘.
$ git checkout master
$ git merge experimental             # Does it all go smoothly?
$ git log --graph                   # It looks nice, doesn’t it?
$ git branch -d experimental  # To delete your branch (optional)
```

Small group collaboration

*- This is where the **nice** really kicks in*

Scenario: 3 students do a numerical project together. They already have a folder with files, a first LaTeX skeleton file, and the beginnings og a python script. Now they want to work on it from each their computer without messing things up.

Strategy:

- We set up a *bare* repository with no actual files in it to tamper with. This is our "server", except it's just a folder, no need to run a dedicated server.
- Each student has a normal repository on their machine set up to track and syncronize with the "server" repo.

Central commands: `git clone`, `git push`, `git pull`,`git remote`

Small group collaboration

*- This is where the **nice** really kicks in*

SCENARIO: 3 students do a numerical project together. They already have a folder with files, a first LaTeX skeleton file, and the beginnings og a python script. Now they want to work on it from each their computer without messing things up.

STRATEGY:

- We set up a *bare* repository with no actual files in it to tamper with. This is our "server", except it's just a folder, no need to run a dedicated server.
- Each student has a normal repository on their machine set up to track and syncronize with the "server" repo.

CENTRAL COMMANDS: `git clone`, `git push`, `git pull`,`git remote`

Small group collaboration

*- This is where the **nice** really kicks in*

Scenario: 3 students do a numerical project together. They already have a folder with files, a first LaTeX skeleton file, and the beginnings og a python script. Now they want to work on it from each their computer without messing things up.

Strategy:

- We set up a *bare* repository with no actual files in it to tamper with. This is our "server", except it's just a folder, no need to run a dedicated server.
- Each student has a normal repository on their machine set up to track and syncronize with the "server" repo.

Central commands: `git clone`, `git push`, `git pull`,`git remote`

Small group collaboration

Don't panic

Overview:

If we only work in the `master` branch (which often is fine for smaller projects), collaboration is particularly simple:

Setting up: `git clone`, `git remote`

Workflow: `git pull`, `git push`

At your leisure: `git remote`, `git branch`

Small group collaboration

Digging in

Let's use our existing working tree as a starting point:

```
# Be sure you are sitting in your folder from before, then:
$ cd ..
$ git clone --bare myproject [/desired/path/]myproject.git
# The trailing '.git' above is just convention.
# From now on, we want the bare repository to be the "parent":
$ cd myproject
$ git remote add origin ../myproject.git  # Or e.g. a web repo
$ git branch -u origin/master master  # Etc for other branches
# Now make changes as you wish, save & commit in local tree
# BEFORE EXPORTING ANYTHING, check for upstream changes:
$ git pull                                               # Then:
$ git push                        # ...etc. etc., rinse and repeat
# Anyone can start collaborating by running:
$ git clone [/path/to/]myproject.git
# Ready to go! Changes can be contributed by
$ git pull; git push                      # Etc. as described above
```

Remote branches

Yeah, it's that easy

Sometimes, you have a tree with more branches and want to work on it from multiple machine or with multiple contributors. This requires a little care with the setup, but once it's done, you don't have to think more about it.

Scenario 1: You have cloned a repo, but it only tracks the `master` branch, and you want to work in and contribute to a different branch.

Scenario 2: You have made a new branch in a local tree and want to push it to the remote.

Luckily, both are quite simple.

Remote branches

Yeah, it's that easy

Sometimes, you have a tree with more branches and want to work on it from multiple machine or with multiple contributors. This requires a little care with the setup, but once it's done, you don't have to think more about it.

Scenario 1: You have cloned a repo, but it only tracks the `master` branch, and you want to work in and contribute to a different branch.

Scenario 2: You have made a new branch in a local tree and want to push it to the remote.

Luckily, both are quite simple.

Remote branches

Yeah, it's that easy

Sometimes, you have a tree with more branches and want to work on it from multiple machine or with multiple contributors. This requires a little care with the setup, but once it's done, you don't have to think more about it.

Scenario 1: You have cloned a repo, but it only tracks the `master` branch, and you want to work in and contribute to a different branch.

Scenario 2: You have made a new branch in a local tree and want to push it to the remote.

Luckily, both are quite simple.

Remote branches

Yeah, it's that easy

Sometimes, you have a tree with more branches and want to work on it from multiple machine or with multiple contributors. This requires a little care with the setup, but once it's done, you don't have to think more about it.

Scenario 1: You have cloned a repo, but it only tracks the `master` branch, and you want to work in and contribute to a different branch.

Scenario 2: You have made a new branch in a local tree and want to push it to the remote.

Luckily, both are quite simple.

Remote branches

Yeah, it's that easy

Track remote branch:

```
$ git branch --remote    # To see which branches are available
# Assuming we want to track the branch origin/experimental:
$ git branch experimental --track origin/experimental
# --remote and --track can be replaced by -r and -t, resp.
```

Upload new local branch

```
# Create the branch like usual, do your work. Once it's ready
# to go upstream, do:
$ git push --set-upstream origin mynewbranch  # shorthand: -u
$ git branch -r                       # Just to see all worked well
```

Remote branches

Yeah, it's that easy

Mixed goodies

```
# Let two or more persons changeing the same file (but in
# different places!) Then push, pull etc. and then run
$ git blame                                  # Pretty neat, huh?
# Shortcuts for pushing and pulling all branches in the tree:
$ git pull --all, git push --all
# Suppose you want to see how many changes are in upstream, but
# do not want to merge it into your local tree just yet:
$ git fetch [--all]  # In fact git pull = git fetch + git merge
```

Ignoring files

There are files you don't want Git to track - like LaTeX's temporary files or log files, or Python's compiled bytecode .pyc files and similar. these can be listed in the .gitignore file in each repository. There are examples/templates at https://github.com/github/gitignore.

Remote branches

Yeah, it's that easy

Mixed goodies

```
# Let two or more persons changeing the same file (but in
# different places!) Then push, pull etc. and then run
$ git blame                                    # Pretty neat, huh?
# Shortcuts for pushing and pulling all branches in the tree:
$ git pull --all, git push --all
# Suppose you want to see how many changes are in upstream, but
# do not want to merge it into your local tree just yet:
$ git fetch [--all]  # In fact git pull = git fetch + git merge
```

Ignoring files

There are files you don't want Git to track - like LaTeX's temporary files or log files, or Python's compiled bytecode `.pyc` files and similar. these can be listed in the `.gitignore` file in each repository. There are examples/templates at https://github.com/github/gitignore.

Remote branches

Yeah, it's that easy

More resources:

A great branching tutorial - become a branching ninja with the slick, interactive tutorial at at `http://pcottle.github.io/learnGitBranching/`

A blog with git tips sorted in Beginner, Intermediate and Advanced categories and with links to other resources: `http://gitready.com`

Git's own webpage is very informative, too: `http://git-scm.com/about`