

Other languages and C++ Writing scripts

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se

Outline

- Introduction to scripting
- Bash
 - Scripts
 - Variables: environment, binding, scope
 - Control structures
- Datasets
- Automation using scripting
 - Genesis of an algorithm
- If time allows:
 - Python
 - Variables
 - Some data structures
 - functions

Goals and non-goals of this tutorial

- Goals:
 - Being able NOT TO PANIC when somebody gives you something you've never seen before (will happen in your entire career)
 - Being able to write a bash script.
 - Understanding the concept of variable. Environment, binding, scope.
 - Being able to search for information depending on a task one wants to achieve.
 - (if time allows) Being able to understand basic python syntax.
- Non-goal:
 - Become a script-fu master. It takes long time for the black belt :)
 - Become a python coder. We cannot do this in a lecture, there's full courses out there

Scripting vs coding

- The word script is taken from a theatrical play script: a description of the environment on stage, a sequence of lines and gestures to do
- There is no practical difference between writing code in a compiled language and a scripted one.
- The main difference is that scripted languages **do not require compilation.**

Prepare for the tutorial

- Create a folder Tutorial3b somewhere in you home and cd into it
 - might be /svn/username/Tutorial3b/
- If you svn update , some code examples are in my folder floridop/Tutorial3b
- Don't work in my folder! Only work in yours, eventually copy paste my code.
- Open geany and get ready to create new files!

A bash script and its components

- A **bash script** is nothing more than a sequence of commands written in a file.
- The bash interpreter will process those in sequence, from the top line to the bottom
- Like C++, it is possible to define **variables** and **control structures** in the scripting language.
- However, the bash script language has little to share with the complexity of C++. All that it can do is to **execute commands, test conditions and store things in variables**.
- **Exercise:** Open geany, write and save the following code as `getcpuinfo.sh`

```
#!/bin/bash

# put the output of cat in the variable CPUINFO
CPUINFO=$(cat /proc/cpuinfo)

# write the content of CPUINFO to screen
echo "$CPUINFO"
```

Anatomy of a bash script

```
#!/bin/bash
```

The first line has a special syntax: **#!** tells bash which **interpreter** to use. It might be another shell!

```
# put the output of cat in the variable CPUINFO
```

Every other line starting with a hash **#** is a **comment**. The interpreter ignores everything that follows until the end of line. Useful to describe code to human readers.

```
CPUINFO=
```

```
$( cat /proc/cpuinfo | head -10 )
```

This tells bash to execute a command and return its output.

A **variable definition** is any string followed by a **=** symbol. It is a convention to use capital letters. Remember that case matters, `cpuinfo` is different from `CPUINFO`!

```
# write the content of CPUINFO to screen
```

```
echo "$CPUINFO"
```

A **variable call** is any **variable name** prefixed by the **\$** symbol. Case does matter here. The quotes affect the output, that in this case depends on how the `echo` command works. The **\$** symbol stands for “give me the value contained in that variable”

Executing a script

- The script can be **made executable** as if it was a command. Commands not in the PATH must have a directory path identified. To run those in the current directory, prefix them with `./`

```
pflorido@tjatte:~> chmod +x getcpuinfo.sh
pflorido@tjatte:~> ./getcpuinfo.sh
processor : 0
vendor_id : GenuineIntel
cpu family: 6
model      : 15
model name : Intel(R) Core(TM)2 CPU          6400 @ 2.13GHz
stepping   : 6
cpu MHz    : 2127.650
```


Functions

- One can define functions to reduce complexity and increase readability

```
#!/bin/bash

# a function that gets meminfo
getmeminfo(){
MEMINFO=$(cat /proc/meminfo)
}

# execute the function, it will change the environment
getmeninfo

# write the content of MEMINFO to screen
echo "$MEMINFO"
```

- The example above also shows that the variables are **always global** (any part of the program can access them). There is a way of scoping them, but since is not widely used, we will not cover it. Bash variables have **no type**, but most of the time is just **strings**.

Exercises

Exercise 3b.1:

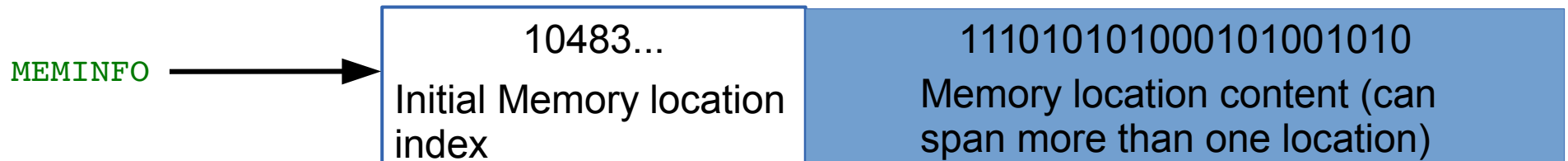
Add to the `getprocinfo.sh` script a line that outputs information about the number of lines that contain the word `cpu`. Use the pipe `|` with `echo`, `grep` and `wc` to count.

Exercise 3b.2: Debugging to debug your script, that is, see what is doing while running, modify the first line this way:

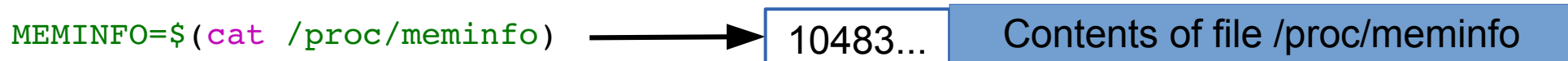
```
#!/bin/bash -x
```

Variables, types

- A **variable** is an identifier, a name, for a memory location. Its **definition** implies that the computer will find a free memory space for that variable. This space, if not **initialized**, can contain anything.



- **Assigning a value** to a variable means putting such value inside that memory location.



- A variable usually has a **type**, that is, the kind of information stored inside it. In some language one must declare it explicitly as you've seen in the previous tutorial.
- In BASH, variable have no type as it is assumed the content is a **string**, or a sequence of characters.

Environment, binding

- All the variable and function names “live” in a space called **environment**. You can think of it **as a table** in memory containing all variable names and their associations with memory chunks.
- A name is said to be **bound** to that environment when its value is associated to a memory index in that environment. In the table on the left we can see some bindings.

Environment	Variable name	Starting memory index
global	PWD	48329
global	SHELL	483985
global	PATH	3412
<code>getcpuinfo.sh</code>	CPUINFO	10289
<code>getcpuinfo.sh</code>	MEMINFO	18458
<code>getcpuinfo.sh</code>	getmeminfo()	3515

- In languages like BASH, we do not see memory indexes. In languages like C++ we can see them in the form of pointers.
- Binding can be:
 - **Static**, that is, decided at compilation time
 - **Dynamic**, that is, decided at execution time (yes one can change where in the memory that variable is pointing)

Visibility, scope

- A variable is **visible** in an environment when its binding is present in that environment, that is:
 - There exists a variable name in the environment
 - That variable name is associated to a memory location (this depends on languages)
- Usually a function has its own environment, that is, a set of variables in its own environment, and can see the variables in other environments according to some rules. These rules define the scope, or visibility, of a variable.
- In the case of BASH, functions do not have own environment. The scope or visibility of a variable in bash is **limited to a bash instance and all its children**. Let's see some examples.

The BASH environment: export

1. Run the `export` command. You'll see all the environment variables in the current bash session.

2. Create a new environment variable:

```
export MYENV1="This is a global env var"
```

3. Find the variable by running `export`, or just print its content with `echo $MYENV1`

4. Open another bash instance by issuing the command `bash`. Run `export`. Can you find the environment variable?

The environment is said to be **inherited** from the father process.

5. Open another terminal and run `export`. Can you find the environment variable? There is no inheritance.

BASH environment: scope

- Let's create a bash script `envtest.sh` with the following content:

```
#!/bin/bash

# create an environment variable
MYENV2="This is my second environment variable"

# write the content of CPUINFO to screen
echo "Content of MYENV1: $MYENV1"
echo "Content of MYENV2: $MYENV2"
```

- Make it executable
 - `chmod +x envtest.sh`
- Run it: `./envtest.sh`
- Try `echo "Content of MYENV2: $MYENV2"`
- The father environment DOES NOT inherit from children, but bash scripts executed inside it have their own environment that **inherits** from the father.

Importing an environment

- In bash, there is a command that allows you to copy the environment defined in a script to another script or bash instance. This command is **source**
- **Careful! The command also executes EVERYTHING inside the BASH script!**
- If you now try
 - `source ./envtest.sh`
 - `echo "Content of MYENV2: $MYENV2"`
You'll see that MYENV2 is now in the father bash environment.
- As a default, bash sources `/etc/profile` , `~/profile` , `~/bashrc` and some other files every time you open a terminal, so that a set of default environment variables are defined. You can `cat` these files if you're curious to see what is in them.

Predefined variables in scripts

- Prefixed by the \$ symbol, they are instantiated automatically in bash at the start of the script.
- Script arguments: \$#, \$0, \$1, \$2....
 - \$# is the number of arguments passed to the script
 - \$0 is the name of the script itself as called to be executed
 - \$1..n is each string that follows the name of the script.
- Process info and status codes:
 - \$\$: process id (PID) of the script itself
 - \$?: exit code of the last executed command (0 if it ended well, any other number otherwise)
 - \$!: PID of last command executed in background
 - ...
- Various:
 - \$PATH: list of paths where executable commands are
 - \$PS1: prompt format
 - \$SHELLOPTS: options with which the shell is run
 - \$UID: User ID of the user running the script
 - ...

Predefined variables example

```
#!/bin/bash

# predefinedvars.sh
# call with: ./predefinedvars.sh arg1 arg2 arg3
#

# print out info about arguments to this script
echo "Number of arguments: $#"
```

```
echo "Name of this script: $0"
```

```
echo "Arguments: $1 $2 $3 $4"
```

```
# print this script's PID:
```

```
echo "PID is $$"
```

Run the script. Remember to `chmod +x predefinedvars.sh` to make it executable!

Exercise: check the output of some other predefined variable, in particular `$*` and `$@`

Control structures

- Enable the machine to **decide** on actions depending on certain **conditions**.
(if..then...else..fi)
- Allow the code to **cycle until a certain condition** is met (while...do...done)
- Allow the code to **cycle** for a definite number of times or **over a list** of objects
(for...do...done)

Conditions

- Conditions are of different kinds depending on the languages. The only condition that BASH can check is whether a command execution terminates successfully.
 - An exit value of 0 is TRUE (termination successful), all other values are FALSE (termination unsuccessful).
- The way to specify conditions is as follow:
 - The square bracket [] or the test command can be used.
Documentation: `man test`
 - Example: `test -z filename` checks if a file exists
 - The double square bracket or extended test `[[some test command]]`. Use `man bash` and write: `\[\[expression`
 - Example: `[[-z filename]]`
 - The double parentheses for arithmetical expansion and logical operations `((a && b))`. `man bash` and write: `\(\(expression`

Control structures: if ... then ... else .. fi

- The BASH syntax is as follows:

```
if condition; then  
    command1; [command2;...]  
else  
    commandA; [commandB;...]  
fi
```

Control structures: if ... then ... else .. fi

- `-le` = less than or equal

```
#!/bin/bash
# testif.sh
# run with: ./testif.sh arg1 arg2 arg3
#
# test that at least two arguments are passed to the script

if [[ $# -le 2 ]]; then
    echo "Not enough arguments. Must be at least 3!";
else
    echo "More than 2 arguments. Good!";
fi
```

Control structures: for ... do ... done

- Repeat something a predefined number of times or for each element in a list.

- Syntax:

```
for i in [list]; do  
    command1; [command2; ...]  
done
```

Control structures: for ... do ... done

- Print the arguments using different condition approaches

```
#!/bin/bash
# testfor.sh
# run with: ./testfor.sh arg1 arg2 arg3 ...
#
# Print the argument values

echo "Using lists of elements"
index=1          # Reset argument counter
for arg in "$@"
do
    echo "Arg #$index = $arg"
    let "index+=1"
done              # $@ sees arguments as separate words.

echo "Using C syntax for the condition"
for ((i=1 ; i <= $# ; i++ )); do
    echo "Argument $i is ${!i}";
done
```

- `#$var` forces the content of `var` to be a number
- Parameter substitution `${!var}` Gets the **value** of a variable with the name `$var` instead of `var`

Control structures: while ... do ... done

- Keeps doing something as long as *condition* is satisfied.
- Syntax:
while *condition*; **do**
 command1; [*command2*; ...]
done

Control structures: while ... do ... done

- Ask the user to enter a variable value (using the read command) until the string end is entered

```
#!/bin/bash
# testwhile.sh
# run with: ./testwhile.sh
#
# Continue asking numbers until the user writes "end"

while [ "$var1" != "end" ]; do      # while test "$var1" != "end"
  echo "Input variable value (end to exit) "
  read var1                        # Not 'read $var1' (why?).
  echo "variable value = $var1"    # Need quotes because of "#" . . .
  # If input is 'end', echoes it here.
  # Does not test for termination condition until top of loop.
done
exit 0
```

Datasets

- A dataset is some digital collection, maybe a file or a set of files, that contains data we want to use.
- A dataset usually has his own **format**.
 - A format is a **set of rules** that define in a rigorous manner how the content of the dataset should be read, what are their meanings and the relationship among the dataset information
 - The format can be a well know data format, more or less standardized, or some custom data format that one needs to learn
 - A **description** of the format is usually provided by the community that generated the dataset. It is very rare that a dataset contains information about its format.

Sample data file

```
"imdbID","Title","Genre","Director","Country","imdbRating","imdbVotes"  
"tt0090084","Storm","Action, Comedy","David Winning","Canada","5.2","53"  
"tt0090086","Strannaya istoriya doktora Dzhekila i mistera Khayda","Mystery, Sci-  
Fi","Aleksandr Orlov","N/A","6.2","21"  
"tt0091002","Eleven Days, Eleven Nights","Drama, Romance","Joe  
D'Amato","Italy","3.3","370"  
"tt0091012","Equalizer 2000","Action, Adventure, Sci-Fi","Cirio H. Santiago","USA,  
Philippines","3.9","180"  
"tt0091017","L'escot","N/A","Antoni Verdaguer","Spain","4.8","8"  
"tt0091026","Eye of the Eagle","Action, Adventure, War","Cirio H. Santiago","USA,  
Philippines","4.5","72"  
"tt0091062","Florida Straits","Action, Adventure, Romance","Mike  
Hodges","USA","5.5","160"  
"tt0091073","Francesca","Comedy, Drama","Vérénice Rudolph","West Germany","N/A","N/A"  
"tt0091090","Fu gui bi ren","Comedy, Family, Fantasy","Clifton Ko","Hong  
Kong","6.6","97"  
"tt0091092","Fuegos","N/A","Alfredo Arias","France","4.0","8"  
"tt0091094","Funland","Comedy","Michael A. Simpson","USA","4.4","227"
```

What can we say by observing this data?

Can we guess something about the structure?

Automation and composition of languages

- Cornerstone of open source programming: if something exist that does a task, and it does it good, use it and do not rewrite code
- **Automation** of repetitive tasks
- Make use of interoperability within languages
- Technique: identify subproblems and separate tasks, increasing debuggability
- Choose the right command/language for each subtask

Automation exercise with BASH

- Description of the problem to solve:

Write a script `checkdataset.sh` that downloads a tarball from the internet and extracts it into a folder, then reads the contents of the folder and shows the content and type of each file.

- The script takes in input three arguments:
 - A URL to a file on the web.
`http://svncourse.hep.lu.se/svncourse/trunk/floridop/downloads/movies.tar.gz`
 - A name of directory where the file and the contents of the file will be stored
 - A name of file where the output will be written.

Genesis of an algorithm: a top down approach

- Write a list of each main task translating what I wrote in the description. We can brainstorm it in the class before proceeding.
- Open a new `.sh` file with `geany`
- Write down the header and start writing down as comments the steps to the algorithm. You can write that on paper first.
- An example is placed in `svn` under `floridop/Tutorial3a/solutions`

Inspecting the dataset

1. Create a folder called `Tutorial3bwip` (use `mkdir`) and `cd` into it.

2. Download the file located at:

`http://svncourse.hep.lu.se/svncourse/trunk/floridop/downloads/movies.tar.gz`

And give it the filename: `tarball.tar.gz`

(Hint: see `man wget`)

3. Extract the file with `tar`

(Hint: see `man tar` or balazs slides!)

Homework 3b.1

- Add the information requested for each file in the dataset.
 - Hints:
 - use `for` to scan a set of filenames
 - Use the operator `>>` to append text to an existing file.
 - Use the commands `ls`, `file`, `head` to gather the information requested into variables.
- Send the code to me via email with subject Homework 3b.1

References

- Bash scripting:
<http://tldp.org/LDP/abs/html/>

Additional material

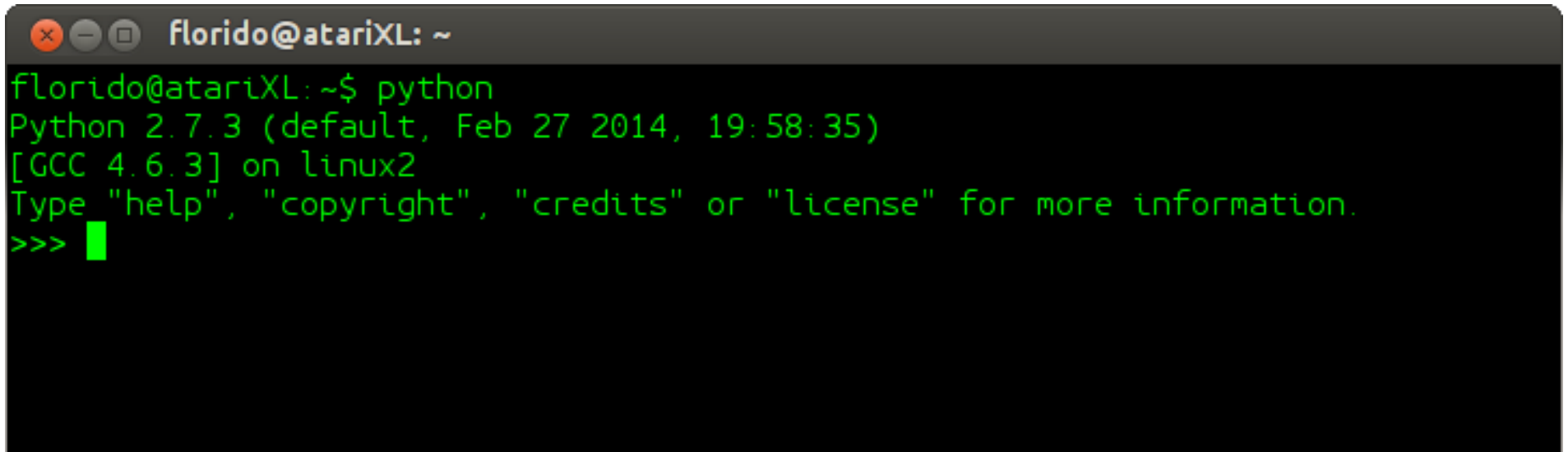
- If time allows, introduction to Python

Python

- Interpreted, code is compiled on the fly
- Widely used in the scientific community
- Easy to learn
- Good for quick proof-of-concepts, even involving complex calculations (there are a lot of nice libraries out there)

The Python interpreter

- 1) Open the terminal
- 2) Run the python interpreter:

A terminal window with a dark background and light green text. The window title is "florido@atariXL: ~". The text inside shows the command "python" being executed, followed by the output: "Python 2.7.3 (default, Feb 27 2014, 19:58:35)", "[GCC 4.6.3] on linux2", and "Type 'help', 'copyright', 'credits' or 'license' for more information." The prompt ">>>" is followed by a green cursor block.

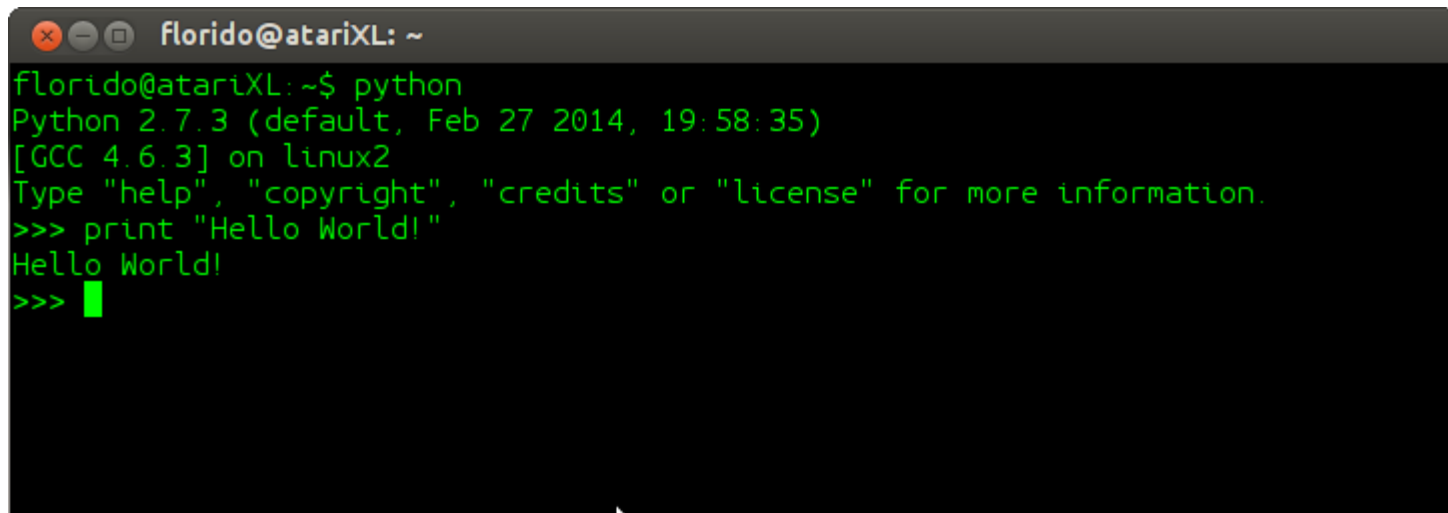
```
florido@atariXL: ~  
florido@atariXL:~$ python  
Python 2.7.3 (default, Feb 27 2014, 19:58:35)  
[GCC 4.6.3] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

Your first Python program

- As python is interpreted, you can directly write programs in the interpreter console.
- Try to write:

```
print "Hello World!"
```

and press enter.

A terminal window titled 'florido@atariXL: ~' showing the execution of a Python program. The prompt is 'florido@atariXL:~\$ python'. The output is 'Python 2.7.3 (default, Feb 27 2014, 19:58:35) [GCC 4.6.3] on linux2 Type "help", "copyright", "credits" or "license" for more information. >>> print "Hello World!" Hello World! >>>'. The text is displayed in green on a black background.

```
florido@atariXL:~$ python
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World!"
Hello World!
>>>
```

Your first python program cont.

- It is however very unpractical to write a program on the fly. It's better to save it to a file as seen for C++.
- Python code is conventionally added in a file with extension `.py`. This is not very important for the code to work, but on some systems like windows the extension matters.

Your first python program cont.

- Let's create a python *script* that prints "Hello Word".
 - 1) Open you favorite editor. In this tutorial we will use *Geany*.
 - 2) Click on the File menu → New (with template) → main.py
 - 3) Let's analyze the structure of the shown python file. Any analogy with C++?

Python program structure

1) The header

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

2) License information (optional)

```
"
# Copyright 2014 Florido Paganelli <florido@atariXL>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.
```

3) The main function

```
def main():
    return 0
```

4) The main function callback

```
if __name__ == '__main__':
    main()
```

Python program structure

1) The header

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

Required: Tells command line to use Python interpreter

Optional but recommended: Info about encoding

2) License information (optional)

```
"
# Copyright 2014 Florido Paganelli <florido@atariXL>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
```

3) The main function

Required: definition of the main function

```
def main():
```

Required: function body indentation. All you code goes before return 0

```
    return 0
```

Recommended: function return value

4) The main function callback

```
if name == '__main__':
    main()
```

Special name of a function

Function call

Special variable that asks the interpreter the predefined variable `__name__` that tells the name of the default function

Python syntax and execution

- Syntax features:
 - **Indentation** (tabs and spaces) is one of the ways to identify a *block of code* in Python. It is fundamental: the author enforced it for readability of code. Python will fail to compile and write out an error if indentation is bad.
 - **;** is the instruction **separator**, is not as important in Python as in C; it can be omitted if indentation is well done.
- Runtime features:
 - `main` will be executed as the first function by the python interpreter.
 - Therefore our `print "Hello World"` command goes right before the return statement, indented as the return statement, followed by a `;`
 - See `helloworld.py`
- Question: why is the `if` executed?

helloworld.py

```
def main():  
    print "Hello World!";  
    return 0  
  
if __name__ == '__main__':  
    main()
```

Python variables

Start the Python interpreter (command: python) and try the following:

```
Python 2.6.6 (r266:84292, Aug 12 2014, 07:57:07)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = 3
>>> b = 'hello!'
>>> print a,b
3 hello!
```

- The Python interpreter allows you to see the content of every variable by writing its name. Try writing a and b and then press enter!
- Use the builtin `len(variable_name_here)` function to see how “big” is a variable. What happens?
- More about builtin functions:
<https://docs.python.org/2/library/functions.html>

Python dict

- Start the Python interpreter (command: python) and try the following:

```
>>> dict = { 'name': 'florido', 'surname': 'paganelli' }
>>> print dict
{'surname': 'paganelli', 'name': 'florido'}
>>> print dict['name']
florido
>>> dict['name']='Rudolph'
>>> print dict['name']
Rudolph
>>> dict['Address']='unknown'
>>> print dict
{'surname': 'paganelli', 'name': 'Rudolph', 'Address': 'unknown'}
```

See:

<https://docs.python.org/2/library/stdtypes.html#mapping-types-dict>

Python list

• Start the Python interpreter (command: python) and try the following:

```
>>> list = [ 'apple', 'pear', 'banana' ]
>>> print list[1]
pear
>>> list[3]='orange'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> list[2]='orange'
>>> print list
['apple', 'pear', 'orange']
>>> list.append('peach')
>>> print list
['apple', 'pear', 'orange', 'peach']
```

See:

<https://docs.python.org/2/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange>

load data in memory

- Load the data in the files we just downloaded into a variable
 - Learn how to open a file in python
 - Learn how to use the csv library
<https://docs.python.org/2/library/csv.html>
 - Organize the movie records in a python dictionary **dict**
 - Add each record in a python **list**
 - Print the list (and learn how to PrettyPrint)
- Let's look at the code!

ExampleD6.4.py

- Let's discuss about it and then run it!

Homework 3b.2

Hacking, or learning by looking at other's code

- Based on exerciseD6.4, write some python code that loads the CSV files and prints them to screen with pretty print.

Python function

- Declaration:

```
>>> def myfunction(adictionary):  
...     TAB return adictionary.keys()  
  
...  
>>> print myfunction  
<function myfunction at 0x7ffe99b35230>
```

Remember
tabs!

- Function Call:

```
>>> myfunction(dict)  
['surname', 'name', 'Address']  
  
>>>
```

Example D6.5

Refactor code into functions

- Identify chunks of code that can be moved inside functions
- Replace blocks of code with function calls
- Try to refactor the code that opens a file and creates the db into a new function called `createdb(dirpath)`
- `dirpath` is the input argument of the function; the function should be called with a string that is the directory where the movies folder is located.

exampleD6.5.x.py

- `exerciseD6.5.first.py` shows a solution for the previous exercise
- `exerciseD6.5.better.py` shows a better refactoring. Let's have a look at it.

Example D6.6

Select subset of the dataset

- Select only movies that belong to a **genre** and write the selection to a file. We will use **Comedy**

Notable Python libraries and IDEs

- Libraries:
 - **Scipy**, for scientific computing
 - **Matplotlib**, to draw plots from scientific data
 - **Ipython**, an interactive environment like mathematica or matlab
- IDEs:
 - **Eclipse**, written in java
 - **Spyder**, specific for scientific programming
 - **Eric**

Missing but worth a look

- Regular expressions and string operations: Python is very good at it
<https://docs.python.org/2/library/re.html>
- C++ libraries compatibility
<https://docs.python.org/2/extending/extending.html>
- Python objects:
<https://docs.python.org/2/tutorial/classes.html>

References

- Python documentation:
<https://docs.python.org/>