general info:

- seminar on: bash code

**LECTURE 1 – INTRODUCTION**

**Info on the exam** = realize a project (also a very easy project) by yourself or consider the project of another exam. You have to show the features and skills acquired during the course in order to realize a program that can be easly understood, modified and replicated.

- you will be evaluated on a programming project

- this project needs to be hosted on a public repository

- any programming leanguage

- accepted control version systems are git and fossil

- there is a list of suggested projects or you can propose your own project

**Exam evaluation**:

- clarity of the repository commit history (6 points)

- clarity and completeness of the documentation and source code (12 points)

- presence and executability of test routines (12 points)

Advanced:

- Usage of innovative technologies and libraries (3 points)

- contribution to an open source project (3 points)

**Objective:** optimize and be productive while programming – collaborate with other people – write programs that can be easly understood, modified and replicated by others.

**Programm:**

I module

- bash command line

- introduction to python

------------------------------------

- version control

- testing

- documentation and collaborative working

------------------------------------- main evaluation points for applied physics exam!

- object oriented programming

- functional programming

II module

- numerical calculus and vectorization (montecarlo calculation, differential calculus etc…)

- scientific libraries and symbolic algebra

- visualization

- databases and dataframes

- data pipeline and data formats

- introduction to machine learning

**Python** is one of the most useful leanguage for a scientist cause is an **high level dynamic leanguage**, which means is a leanguage that easly communicate with the human interface and that, in its execution phase, casts a lot of instruction that are typically considered in the compiling phase.

The usage of the shell is useful to obtain info and modify data fastly and to automize process 🡪 is an alternative to the user friendly OS interface. We can interact with the shell through commands or recalling premade scripts. From the shell we can move between directories using the command “dc name\_of\_the\_directory”.

CLI = Command Line Interface = is the alternative to the program leanguage editor such as DevC++ or Spyder

Some basic instruction that can be used from the shell are:

- ls (list of the file in the current directory) - [LINUX] - ls \*.txt (will find only txt files – NOTE: the space after “ls” is needed) – ls file?\*.txt (will find only txt files with the name “file” + another arbitrary character or number)

- ls -la [LINUX] = shows all (and only) hidden directories

- cd = change directory [LINUX and WINDOWS]

- mkdir [WINDOWS] = make a directory (then write it’s name)

- echo (stampa) Ciao a tutti!(riga di testo)>(e inseriscilo)file1.txt(nel file di testo file1.txt generato appositamente) [WINDOWS]

- dir (list of the file in the current directory) [WINDOWS]

- dir -ah = shows all and only hidden directories [WINDOWS]

- del <nomefile> = delate the file with that name (in the correct directory) [WINDOWS]

- notepad = open the windows text editor from terminal [WINDOWS]

- notepad <nomefile.txt> = open the txt file with that name in the correct directory and allows for modification [WINDOWS]

- type nul > nomefile.txt = crea un file txt vuoto [WINDOWS]

- copy con > nomefile.txt = sovrasciriv ([Sì\No]) contenuto del file nomefile.txt con ciò che digiti nella riga del terminale [WINDOWS]

- mkdir foldername = create a folder in the current repository [WINDOWS]

The **file system** is the way our OS orginize files in our disks (Os disk, data disk, external disk or device).

Each file is owned by an user or a group, each time an user want to interact with a file there are three types of permissione we he can have: **reading**, **writing**, **executing**.

Other users can achieve the content of the file through a soft or an hard link (one points to the file name that ponts to the memory location, the other points directly to the file content a.k.a. the memory location).

Typically file names specify **extension**, which gives info about the file type and content and on the suggested program with which we can open that file. Change the extention of a file does not change the memory location of a file.

**LECTURE 2 – PROGRAMMING AND CODING**

Programming = think how to formalized a certain sequence of operations

Coding = writing a computer program that actually perform the operations we have planned

There are some main programming concepts which are common to each programming leanguage:

- storing and retrieving things

- conditional execution

- functions and routines

- iteration

- data structure(s)

1) **Storing an retrieving things**

Declaration in Python has no sense (different from C) cause it is a dynamical leanguage, defining the value of a variable means inizialize it. With python we can, as in C, open files, modify them and save changes. From the terminal we are able to manage files in our computer.

2) **Conditional execution**

All the conditional instruction firstly evaluate an expression and then, depending on the result, evaluate something.

3) **Functions and Routines**

Function = use the input and return an output

Routines = usually they have no return and they simply create, move, destroy something

Functions can be defined in the script, just avaiable for Python programmation or can be imported from a library. To do hat we have to import that library.

example

> import math

> a = math.sqrt(2)

or

> from math import \*

> a = sqrt(2)

When we import the library we have always to recall the name of the library dot the name of the function, importing \* from all the library we will import all the functions (from math import \*)

4) **Iterations**

Can be bounded (for cylce) or unbounded (while), depending on wheter we want to perform the first instruction

5) **Data structures**

We basically can work with numbers (int, float, double and arrays), character (characters and strings) and boolean espression (True and False). Lists are arrays whose element can be of different type. We can work with lists in different ways. Another useful structure is the tuple, which store heterogeneous set of elements giving them an order. In the end, dictionaries are association map which encodes a relationship between two different groups of value (How does this character/number/etc… translates into another character/number/etc… according to the dictionary we have chosen?).

**LECTURE 3 – GIT and VERSION CONTROL**

Git is a program to take under version control your work (repository).

How to make sure that your program is doing what it should do? 🡪 focus on programming

Four main parameters we care of:

- **correctness** = be sure your code does exactly what you think it does, that any modification you perform does not introduce an errors (or if it does, you have to be able to find these mistakes and rollback to a previous version 🡪 version control contribution)

- **replicability** = repeat the simulation and have always the same results, specifiy how you have to use the code (prerequisites, data and paramters etc..) – allow someone to run your code on a different machine without needing you in the same room.

- **reproducibilty** = allows other to reuse the same analysis and your ideas on other similar cases

- **auditing** (revision) = keep the history of your project! It’s important to know what have been done, when and why. People can come and check your code time by time. This is necessary both to maintain the knowledge you obtain and to allow some external reviewer to verify what you did. Typically we use some repository or program to take track of everything 🡪 version control contribution!

Doinging research is not writing code, research is manage them in order to make them avaiable for everyone. Writing a code means solving a request.

What are the main elements that are necessary to have an “healty” code?

- **version** **control** 🡪 keep track of the history in order to better understand the modification of you code but also to recover previous version in case of errors/mistakes

- **documentation**

- **test procedure**

- **procedure automation**

- **well designed anlysis pipeline**

General idea = correctly managing your metadata (info about the data) 🡪 brief example: name of the variables can help

**Distribuited Version Control and Code Source Management**

Code source management = set of tools

Everybody does a rude version control when working on a project.

Using a proper version control system we can:

- **keep track** of everything that happends

- **roll back** to a previous version at any moment

- **visualize** the difference bwetween two versions of it

- keep track of **parrallel** world **version of our work**

This work in tandem with the source code management tools (responsable of high level coordination) that allow to keep track of bugs, documentation, feature requests etc…

Software we are gonna use: **Git** = it’s a version control system that run on your system and allow a distribuited version control

**Github** = hosting website, give space to host your repository (semipermanent hosting of your date), useful to collaborate with people. In this space a version control system is avaiable (info on previous version, etc..) if installed on your pc.

So: we will use git ad a control version system and the webite GitHub to do the general code management and as a central repository of our work.

**Idea behind git**

Simple first version of git

1. everytime you modify a file, save it as a nwe version with the date of edit attached to it

example: thesis\_2021-03-01T15:42, thesis\_2021-06-15T19:12

NOTE: ISO date representation = year – month – dayThour : minute

if you have a lot of files it’s a problem

1. save a snapshot of the whole directory everytime you modify something, with the date in the name (waste of file)
2. let’s keep all the version of all the files in order by date + we specify in a small document (the manifest) with written what is the current state of the directory

Anyway, we need to know the state in previous moments in time…

1. keep several manifests, each one referencing the one before it and what changed between them, in order to keep the track of version in time. example: the version of today is based on the version of that previous day

Anyway, we need to manage a collaboration…

1. it’s important to use an unique identifier for each version different from the date – identify each file and manifest with a unique hash, store the various version of the various files in a hiddend subdirectory togheter with the manifest, build the directory based on those, exchange the files and manifest with other people….

So… on github:

- you create your directory on which you want to work with and you create your files

- at a certain point you want to start to follow the version control system = you are saying to git: “this is the version control directory”

- git will start keeping track on what happend in that directory = it will not automatically save every edit you do, it will only record a new snapshot when you tell that (you have to chose when you want to fix a version, automatic save it will be a problem)

- git also manage the coordination with your local pc and the remote server

Git is based on the idea of subcommands, we always say: “git, do these things with these parameters…” 🡪 the actions are commit, push, pull etc…

In order to **configure git** 🡪 define your user and your email

> git config -- global user.name “Giovanni Olivetti”

> git config --global user.email "giovanni.olivetti@studio.unibo.it" 🡪 in order to define the email that appear to label you as an author

In order to change author and email in a second moment the command is:

> git commit --amend --author="Author Name <email@address.com>" --no-edit

where the < > are needed to insert the email address.

In order to create a repository controlled by git we have to **inizialze it inside the repository** we want to track.

Let’s create an empty repository and access at it

> mkdir first\_repo

> cd first\_repo

> git init 🡪 command you run ONCE, the current directory is now under git version control

You should recieve a message of confirmation (presence of a terminal \.git)

After having done that, in the directory the same previous file are visible. Anyway, a new (hidden) directory is generated and it’s called .git 🡪 all the activity that I’m gonna do will be avaiable in this .git directory!

A command with which we should be familiar is:

> **git status** 🡪 tells the current status of your repository

If there are no subfolders (so there isn’t any three structure) the “git status” command will give an error!

First of all, git tells in which branch you are 🡪 if you run “git status” in the main folder you will be in the master branch = main branch. Furter branches can be generated.

The second information regards the committment you have done: “no commits yet” means you had not realized any changes since the initialization of git.

Moreover it advise you for what regard the tracked and untracked file. If there is one untracked file (= file that is in the directory but is not tracked anywhere by git), we can put it under git control with the command:

> **git add** file1.txt (general: git add <file>

Also when we modify a file you have to add it again in order to succesfully save it with the next commit

If now we cast:

> git status 🡪 now git is telling me that I can remove the cached file to unstage (🡪 concept of stages)??

If I added i file that I don’t want to track anymore I can remove it using the command:

- git rm –cached file2.txt (general: git rm –cached <file>)

If I check the git status there will be both tracked and untracked files with both suggested actions.

The git output basically explain everything and suggest a lot of possible actions! Always read it and learn all possible information it gives.

Now that we have our files that we want to track and the files that we don’t want to track (untracked files), we can committ one of the files.

**Commit** = where you want to take a snapshot

In this way we take a snapshot to the history of the repository. Typically we take a snapshot after a significant change in our programm, so it’s of our interest to write also a comment that describes/identifies this variation.

Commands:

> git commit 🡪 git simply takes a snapshot

> git commit -m “message” 🡪 git takes a snapshot and label that snapshot with the message you want

The output of this command tells us that we have written something to this commit, writing also a unique code that identifies the commit. It notifies also that one file is changed and there are some data that are inserted. Each commit is associated to a **unique alphanumeric code** that label that snapshot and that can be used to roll back to taht version.

If you enter in the \.git directory and in the hidden \object directory we can see that the elements inside it are just binary objects. These are the zip version of the files that we have added here. It’s clear that the directory \.git contains all the history of my directory.

If we check the git status of the directory there is written that there is nothing added to commit, cause we have already committed the tracked file (there are only untracked files). The untrucked files are still present and git notify us of that.

If we track all the files and we commit them, the git status output tells us that there is nothing to commit and that the work three is clean.

Now it is possible to check the history of the repository, the command we can use is:

> **git log** 🡪 shows all the commits done in that repository

Further paramter we can insert after the command git are documented in the following link: https://git-scm.com/book/it/v2/Git-Basics-Viewing-the-Commit-History

Note: on anaconda prompt on windows OS the command “git log” does not show any output on screen 🡪 the command to do that is “git --no-pager log”. Don’t know precisly why “git log” command does not function, but It may be a pager issue, which blocks the output on the console, but does not apply when redirecting to a file. We can in fact save the git log info on a file, using for example:

> git log > \Desktop\Unibo…\first\_repo\file1.txt

or, if we are inside \first\_repo repository, we can simply write:

> git log file1.txt

The output of the git log command consist in a series of info (user, email etc..) and in a description of the commits realized. For each commit is written an alphanumeric code that represent the unique fingerprint that uniquely represent that commit. The last commit on which you have worked on is called HEAD, and its branch is specified. The git log command also report the label you have select for each commit.

Sometimes an error message tells you that the the pc cannot understand of which history the head is referred (?) – the head is not attached to any other commit (detatched head). This means that the snapshot that was the head of the previous version now is no ore the head… in order to solve the problem of the detacted head we can build a new branch and start, from the actual snapshot, a parrallel path

We can also see the commit reffered to a precise file. The command is:

> git --no-pager log file4.txt

If you edit a file and then check for the git status you’ll se two options:

🡪 use git add <file> to update what wull be commited

🡪 use git restore <file> to discard changes in working directory

Another useful thing is **checking the differences** at which is subjected a file between the first and the last version, the command is:

> git diff file4.txt

In this way we can see which are the first and the last modifications applied(in red and with a minus before it the first, in green with a plus before it the last). This command gives an output only if the final version of the file considered hasn’t been added yet to git version control! The ouptu consist in the modification step by step with respect to the initial file. If the file has been added, git does not see any differences.

Actually we can also check differential difference between the last and the previous version with the command:

> git --no-pager diff HEAD~1 file8.txt

in this way we can check differences also of file that has been just added t git version control.

If you are working with a bigger file typically the terminal will show just individual row of text that are interested in the modification.

Another operation typically considered is to **revert back the history** = replace the current version of the file with the previous one from an older commit. Pay attention because you can revert the whole repository if you forgot to specify that you want to revert the single file. The command is:

> git checkout <commit SHA1> myfile.txt

where <commit SHA1> is the unique alphanumeric code that characterize the commit number “x” (that commit is refferred to the whole modification, so if you want to focus on a file be sure that you precise it!)

After having enter that command the whole directory will be converted into its selected previous version. In order to avoid the detached head error it’s convenient to switch to another branch.

Using the windows terminal you can easily manage the file in your repository (create files, delate files, edit them etc…). Each time you operate a modification on a file you have to add it again under the git control, in order to commit it and fix the changes. Also when you delete a file you have to add it again to the git repository in order to commit this change (and save, in this way, the situation in which the file is deletad from your repository)

It’s possible to rename a file using the command:

> git mv <oldfile> <newfile>

It’s possible to delate a file using

> git rm <file>

Watch out because git will do the modification and keeptrack of it, anyway, you can always revert the changes by doing:

> git checkout HEAD <file>

git will explain precisely what to do to revert your changes, just remember to always check the status of the repo before committing!

NOTE - ATTENTION: if we have file5.txt and file4.txt and we write: git mv file5.txt file4.txt the output of the shell is an error, we cannot rename one with the other cause there will be two files with the same name… BUT if we delate file4.txt (git rm file4.txt) and then we write git mv file5.txt file4.txt again, the delated file will be recognized by git status output as file5.txt!! It seems that git has exchanged the name of the two file considering that one of the two isn’t anymore physically in the folder.

A very important functionality is the **syncronization with the main server**.

The big advantage of distributed version control systems such as git is that you can store your repository on a remote server that multiple people can collaborate on. first thing you have to set the remote server location (such as github), with one of two version depending if it is a public location (http://) or behind an ssh login (remote server). The commands are:

> git remote add <origin> ssh://login@IP/path/to/repository

> git remote add <origin> <http://IP/path/to/repository>

You can have multiple remotes, and each one will have to be synced independently. **origin** is the most common name.

You can also duplicate an existing repository in a clean folder with

> git clone http://IP/path/to/repository

this will automatically also set up the remote for you.

To see the list of currente remotes, you can use:

> git remote -v

**note - updating the origin**

If you want to change the location for a remote, such as origin, of your repository, you have to update it, don't try to use again remote add

> git remote set-url origin <link to repo>

Once you have the remote set up, everytime you want to modify the project, you should

> git pull origin master

to get the latest updates from the other authors. After you do your commits, you can

> **git push origin master**

to share it with them.

After having shared your repository with the remote server on github will be possible to see all the commits with their respective data (the data of the commit, not the data of the push)

In this way you can use github as a sort of drive on which will be avaiable the history of your work!

Note for collaboration: if you and another author modify the same file, git will try to be smart and merge by itself the edits, as long as they involve different part of the file.

If you are editing the same positions, it will complain, **stop you from committing** and ask you to **solve it by hand**, giving you a modified version of the file that highlight the stuff that you need to merge by hand.

To limit this, try to keep each commit nice and well focused, so that you don't risk modifying random stuff in a file and risk a merge conflict

Sometimes you will have some files you don't want to be notified about, for example chaches, temporary files, etc...

For these files, you can silence all the notifications by creating a .gitignore file, that is a list of filters that will be used to not show the files.

for example, a .gitignore file could be:

\*.temp

\*.cache

you have to leep the .gitignore under version control like any other file!

**Branching** is the process by which you can create alternative realities in your repository to experiment without messing up other people's work.

it can lead to some real mess, so use them sparingly...

With the command

> git switch -c slave1

you create a new branch called slave1.

with the command

> git switch nameofthebranch

you can move into the branch specified.

In this way you can progressively take care of parrallel project (maybe you are working on the same project in three different ways!)

LECTURE 4 – TESTING

Fundamental block to improve corretness of your code 🡪 a correct code is a code without bugs and errors.

Bugs = difference between what the code say that it does and what actually does. Precisely, there is a bug in the code when the code behave differently from what it’s written in its documentation.

Documentation 🡪 manual, docstrings (what each function does), name of the variables, comments, etc…

Also a comment can have a bug.

It’s clear that the documentation is fundamental…

ZOOM:

You can install numpt from anaconda shell using:

> conda install numpy

and then import it in the editor you are using to programm in python (ex: spyder)

> import numpy

then is sufficient to write in the correct section (right and above) the name of the function of which you want the documentation to recieve the appropriate help. Alternatively, write:

> help(name\_of\_the\_function)

to reciev help directly from spyder’s terminal output.

Another way is to write the name of the function in the spyder terminal and click cntrl+I.

numpy is a good example of well designed documetation.

Main info:

- definition (structure of the function, ex: linalg(inp1, inp2, par1, par2, etc…))

- type (ex: function..)

- parameters possibles (and their functionality)

- returns

- what type of error can be raised

- reference of other function related to the considered one and that can be useful.

testing function is hard, but there are simpler and harder function to test.

We keep our analysis on pure function (the easiest one, they do not change the state of the program)

A function is called pure if:

1. with the same input, it returns the same output (it is deterministic)
2. it does not change the state of the rest of the program.

This lets you:

1. know that once you've proven that function is correct, it will always be correct
2. if you change the implementation, as long as the input-output relationship is the same, the functions that depend on it are going to be fine

pure functions are very useful, in particular because they are the easier to test and prove correct

Using global variables (or non local ones) inside a function stops it from being pure! It’s also more complicated to be tested.

Most of your code must be as much as possible composed by pure function.

if a function is hard to test, it will be also hard to use…

Note: plotting does not need testing.

**Type of test:**

* **advancement test**: edits to the function introduce the new features I desider without giving errors and bugs
* **regressions test**: edits to the function do not lose functionality that other code relies on
* **positive tests**: the code does the things I expect it to do when I give the right parameters
* **negative tests**: the code fail the way I expect it to when I give the wrong parameters

Positive test are much more difficoult then negative tests (and much more important)… there are much more combination to be tested. If something fail, can be linked in the documentation in order to give instruction on how you can improve that…

There are three main different strategies to test the correctness of a single function:

* informal testing
* unit testing (anecdotal testing)
* property testing

>> examples on the testlesson2.py

As a general rule, I need at least one example of a typical case use, plus one for any limit case.

Imagine to be writing a function that sort a list. You would need to test at least:

1. that an out of order list, such as [1, 3, 2], get sorted [1, 2, 3]
2. an empty list gives back an empty list
3. an already sorted list [1, 2, 3] gives back the same list as output

So, first of all, it’s fundamental to list some simple and possible cases.

A good library to start with **unit testing** is [**pytest**](http://doc.pytest.org/).

**pytest** is a command line program that:

1. search all the current directory for each \*.py files
2. in each one of those, search all the functions named test\_<something>
3. execute all of them and keep track of the results (check if they end succesfully)
4. prints a summary of the tests’ result

>> example on testlesson3.py

LECTURE 5 – LOGGING AND DEBUGGING

Last lessons about general programming – idea of logging and debugging

This topic deal with how to be sure that our code is doing what is supposed to be doing

This is clearly related to testing and it’s complementary to it

While tests check that the program logic is correct, these tools helps you check that the program is implementing the operations you think it's doing. We can say also that testing deal mainly with the possibile behaviour of the result of the execution of our code, while debugging deal with what the program actually do step by step.

We try to understand why our code is doing something (typically non expected)

Test = working correctly

Logging and debugging = be sure the program is implementing the operations you think it’s doing step by step

Debugger = program that attach to yours and monitor it state allowing you to control the execution in real time. Concretly we can execute instructions one by one in order to undertand which is the uncorrect instruction. Python has a basic debugger for the command line, blue buttons on the top represent a user friendly interface for the debugger of spyder.

Two main operations cab be carried out by our debugger

- managing the execution

- examone and modify the program state

List of commands:

- list 🡪 list the code at the current position

- where 🡪 which is the exact postion in the call stack

- next 🡪 execute the next line (does not go down in new function)

- step 🡪 execute the statement (gos down in new function)

- return 🡪 return out of a subroutine

- bt 🡪 print the call stack

- print 🡪 print the result of the expression

- a 🡪 print the local variables

- !command 🡪 Execute the given Python command (instead of pdb commands)

**-** up 🡪 Walk up the call stack

**-** down 🡪 Walk down the call stack

- help 🡪 Show a list of commands, or find help on a specific command

- quit 🡪 Quit the debugger and the program

- continue 🡪 Quit the debugger, continue in the program

if we write in our terminal

%run -d name\_of\_the\_programm\_py

the programm is executed step by step and stops itself befor the uncorrect statement

There are some command that help us to understand which is the problem.

Using ! we can execute the given Python command instead of pdb commands.

Each time we go on executing the next line(short command “n”) we can check the value of variables (“a”), print results of expression(“p”), and other information

In presence of a cylce, each time we command “n” we go to the next step of the cylce.

In presence of cylcing parameter that runs over 4 values, we will have four steps for the cylce.

Command “breakpoint()” = point where we think the error is located

using > breakpoint() inside our code, during the execution the program stops at that point, allowing us to check the status of our variables and functions. If we use the command !next and we put a breakpoint inside a function which is constituited by a loop, the execution will be stopped at the line where breakpoint is located, giving us the possibility to further investigate.

*example*

num\_list = [500, 600, 700] # list of numbers

alpha\_list = ['x', 'y', 'z'] # list of characters

def nested\_loop():

for number in num\_list: # first cycle

print(number)

for letter in alpha\_list: # second cycle, fully executed at each execution

print(letter) # of the main cycle

if \_\_name\_\_ == '\_\_main\_\_': # construction that will call the nested\_loop()

nested\_loop()

Using:

> %run -d debuggin\_example.py

we run the program in the debugging environment, we can notice on the left of the new line in our terminal the word: “IPdb”, which notify you we are in the debugging environment.

Here you can start to give instruction from the terminal in order to recieve information about the program state and to move through each line (executing it).

What we will obtain at screen is:

NOTE: Enter 'c' at the ipdb> prompt to continue execution.

> c:\users\giovanni olivetti\desktop\unibo\software and computing for applied physics\looping.py(2)<module>()

1 # -\*- coding: utf-8 -\*-

----> 2 """

3 Created on Mon Mar 15 19:31:02 2021

4

5 @author: Giovanni Olivetti

We can see some information about the program: the directory path where we can find it, the date, the author and so on…

When working with programs in the Python debugger, you’re likely to use the list, step, and next commands to move through your code. Each command in the shell have to be preceded by an esclamation point (“!”) if the option is disabled.

The command: > !list 🡪 will notify you which is the line the debugger is considering, identified through a ->

To move through the program line by line, we can use !step or !next: The difference between step and next is that step will stop within a called function, while next executes called functions to only stop at the next line of the current function.

The step command will iterate through the loops once it gets to the running of the function, showing exactly what the loop is doing, as it will first print a number with print(number) then go through to print the letters with print(letter), return to the number, etc:

(Pdb) step

> /Users/sammy/looping.py(5)<module>()

-> def nested\_loop():

(Pdb) step

> /Users/sammy/looping.py(11)<module>()

-> if \_\_name\_\_ == '\_\_main\_\_':

(Pdb) step

> /Users/sammy/looping.py(12)<module>()

-> nested\_loop()

(Pdb) step

--Call--

> /Users/sammy/looping.py(5)nested\_loop()

-> def nested\_loop():

(Pdb) step

> /Users/sammy/looping.py(6)nested\_loop()

-> for number in num\_list:

(Pdb) step

> /Users/sammy/looping.py(7)nested\_loop()

-> print(number)

(Pdb) step

500

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb) step

> /Users/sammy/looping.py(9)nested\_loop()

-> print(letter)

(Pdb) step

x

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb) step

> /Users/sammy/looping.py(9)nested\_loop()

-> print(letter)

(Pdb) step

y

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb)

The next command, instead, will execute the entire function without showing the step-by-step process.

While going through your code, you may want to examine the value passed to a variable, which you can do with the pp command, which will pretty-print the value of the expression.

**Assertion**

> **assert** 1 == 0

So: **assert** espression

After assert we have to write an espression that can be True or False

They are useful to express your expectation about the code.

They can be removed from the execution using the falg -O (Optimize).

They are typically used only during testing activity and in normal code they have only one real usage: express invariant of our code.

Assertion are a way to formalize the fact that the programmer can rely on a certain propriety.

example:

>def my\_smart\_sort(sequence)  
>sorted\_sequence = something(partial\_result)  
> **assert** len(sorted\_sequence) == len(sequence)  
> return sorted\_sequence

In this way we always know wether the sorting activity somhow forgot of an element giving you as a result a different lenght for the sequence

Anyway assert in general does have a terrible sintax

We could use a better assertion library that is grappa

Logging = a rudemntal form of logging is when we print the internal state of the program to check that it’s able to work properly. This is something that works since the program is small.

Python presents a logging system that is a sofisticated way of printing and allow you to work with long and complex codes.

The loggin activity is a process of record of the events of a program with the precise aim of check and understand the programm execution and recognize problems.

Five level of logging: debug, info, warning, error, critical

import logging

logging.basicConfig(level=logging.WARNING)

logging.debug('This is a debug message')

🡪 debug: detailed diagnosis of the problem

logging.info('This is an info message')

🡪 info: gives a feedback on the correct functioning of the system

logging.warning('This is a warning message')

🡪 warning: the application function properly, but there is a warning regarding a future problem

logging.error('This is an error message')

🡪 error: it was not possible to execute the function due to a problem

logging.critical('This is a critical message')

🡪 critical problem, could be necessary to rewrite the whole code.

A good choice is to use a library called eliot that allows a more structured logging instead of just rinting to stderr

Warning system = way to communicate directly with the user and let them know that there is something fishy going on.

import warnings

warnings.warn("this is an old script, use a new one!")

Warning should appear only in some specific situations

A typical example is when the waring message want to advice the user that he is using a function whose future implementation will be changed in next versions.

The warning usually prints on the terminal the information. But you can also set a way for which an error is rised when there is a warning message, in this way you cannot see them and you are forced to fix them. Another option is to silence warnings (when the final user will use different libraries from yours)

Linters = kind of program that check your code for possible mistakes and errors before the execution. Typically these errors are easly to be recognized, is less easy for long codes, some examples:

- variable defined but not used

- overloading of existing functions

- syntax errors

Spyder do that automatically on the left of each row of code (as in overleaf)

Static type checking = another form of linter

Python is a dynamic language, but is still strongly typed. One just don't have to declare the typed beforehand.

Can be used before the run time in order to check

LECTURE 6 – NUMPY AND PLOTTING

First topic: Precision, accuracy and speed for scientific computations

- Floating point computation and bheaviour

- Vectorization

Floating point = used to represent real numbers (alternative to int). There should be no finite representation for real numbers, but for necessity (finite memory) we have to approximate our number. the approximation is done by the calculator.

To represent a real number we can keep some bits for the integer part and some other for the decimal

part. This will limits the range of number that can be described.

The other option is to exploit the scientific notation: one bit represents the sign of the value, one bits is the

exponent of the scientific notation and the other for the number we want to plot approximated in the

corret way.

Keep in mind some concepts:

- as soon as you start working on a computer, you have to forgot about the idea of continuos

- any number that a computer can manipulate and output is, by necessity, with finite precision and only an

approximation of a real number.

Approximation made in order to represents real number can lead to numerical imprecision. We cannot store 2/3, but something that is a bit bigger than 2/3 (0.66666667). The scientific notation reduce the impact of this approximation, but cannot completly delete it. Main problem regards irrational number and real number with an infinite number of digits.

The main source of errors is that the simple fractions in base ten might not have a finite representation in base 2.

We can have a case for which each value is overstimated, the final result can be strongly overstimated.

With the command

> print(“{:.18f}”.format(0.1))

we print at screen: 0.1000000000000056

we have at screen the number with 17 digits and we can see the value 3 as last digit.

while having

> print(“{:.18f}”.format(0.2))

we obtain: 0.20000000000000111

Their sum is

> print(“{:.18f}”.format(0.1+0.2))

we obtain: 0.3000000000000444

So its clar ther is something that happen….

For the computer prospective everything is normal and ther is a reason for which these number have these strange values. Python works with 53 bits of precision, so the values that is working internally are not the same as the ones that it shows using the simple fnction print!!

Anyway floating points arithmetic is not equivalent to the one you are familiar with using the real numbers.

The problem rises when we are working with numbers having a very different magnitude..

Writing:

> print(“{:.18f}”.format(1e32))

we get: 100000000000000000000000000000536662204393472.0000

while

> print(“{:.18f}”.format(1e17))

we get: 1000000000000536662204393472.0000

The final part is identical…

For each value there is a precision limit (a precise digit of the number) below which the digits has no univoque meaning (summing to a big number a unity, for example, could give an unchanged result cause the first non decimal digit is below the precision that characterize the bigger number).

*example*

Editor:

> a = 10\*\*32

> b = a +1

> if b == a

> print(“b under the precision threshold of a”)

Terminal:

> b under the precision threshold of a

Also some simple proprieties don’t stand in some cases, such as the commutative propriety of addition.

SO: take home message = don’t made operations between number/values which has a very different magnitude, pay attention in case of multiple overstimated or understimated approximation.

Alongside precision errors and round offs there are several ossible exceptions that can be rised while operating with floating point number, which are, according to the IEEE standard:

- Underflow: the result of an operation is too smal to be represented with a number

- Overflow: the result of an operation is too big to be represented with a number

- Divide-by-zero: when one tries to divide bu 0

- Invalid: not well defined operations (ex. 0/0)

- Inexact: happens when the result of the operation is strongly approximated.

We can also test if two floating point are **close enough** to be for all intent and purposes considered equal, you can use the function inside math that is math.isclose

Definition: math.isclose(a, b, rel\_tol, abs\_tol)

where a and b are the values to check for closeness, rel\_tol is the relative tolerance = maximum allowed trelative difference between value a and b to be considered equal (default: 10\*\*-9), abs\_tol is the minimum absolute tolerance, if the difference between two values is bigger than the abs\_tol, they are not close enough.

*example*

> a = 05

> b = 0,499998

> math.isclose(a, b, abs\_tol=0.00003)

True

> math.isclose(a, b, abs\_tol=0.00001)

False

> math.isclose(a, b, rel\_tol=0.00005)

True

> math.isclose(a, b, rel\_tol=0.000006)

False

If we don’t care about numerical performances we can also wotk with decimal and fractions functions of matlab.

So:

> Fraction(‘7e-6’)

will give us: Fraction(7, 1000000)

In this way it’s possible to perform operations workng with two integers number, the numerator and the denominator, which in most of cases help a lot. Anyway the final result in a numerical form will have the same problem since now listed.

Algorithm performances

Given the proprieties and the issues of the floating point another aspect we have to consider is the performance of the algorithm in terms of accuracy, time and memory necessities.

Example: operations that are mathematically equivalent can anyway be very different from a computational point of view, also divide per N or multiply by 1/N is a different operation.

Division is 4 time slower then the multiplication, even if the result is the same. There are a lot of specification which can help us to optimize the time execution of our code.

example: computation of 1/sqrt(x)

We can simply: print(1/sqrt(x))

There is anyway a Python implementation (script) that can be quite faster than the standar operation, even if this algorithm is more complicated and less precise. Sometime a faster and less precise algorithm is preferred for time optimization issues.

Using the command: “%timeit” followed by an operation you have performed, you’ll obtain as a return the time in which that operations have been performed.

*example*

> import numpy as np

> a = np.randn(1000)

> %timeit a/2 🡪 approx 40 us

or

> %timeit a\*0.5 🡪 approx 10 us

There are several ways to increase the speed of our code

- memory access optimization

- code.parallelization (such as pipeline)

- **code vectorization 🡪** easy and effective method thorugh the compression of data in order to help the compuyter calculation (we reduce to operations on entire data structures)

Instead of summing 1000 of number one by one, it’s better to sum the element of a list for example.

We can use vectorization when

- we have a loop over a data structure

- each step of the loop can be executed in parallelo over the elemetns of the structure

Vectorization is one of the easiest methods to implement and that have the best gain for effort... especially in high level programming languages

Numpy library = Numerical Python can help us a lot in these things.

The library provide an object, the **array**, that support vectorized and parallel computation in order to make Python compet with C in terms of velocity.

The library also provide most common manipulation algorithm (for which there is an extension with scipy), all implemented in a vectorized fashion, but also read and write capabilities, tabular data manipulation, linear algerba and other…

References

- Book = From Python to Numpy = introduction to the usage of numpy

- Scipy Lectures Notes = introduction to Python and its libraries

*example*

Editor:

> import numpy as np

> a = [1, 2, 3, 4]

> b = [[1, 2, 3, 4], [2, 3, 4, 5]]

> # I need firs of all to define them as arrays with numpy

> a\_array = np.array(a)

> b\_array = np.array(b)

> # let's print at screen some info about these arrays

> print(a\_array.shape)

> print(len(a\_array))

> print(a\_array.size)

> print(b\_array.shape)

> print(len(b\_array))

> print(b\_array.size)

Terminal:

(4,) / 4 / 4 / (2, 4) / 2 / 8

Modyfing arrays

- array.shape = return the dimensions of the array (number of elements, number of elements of each element)

- array.dtype = returns the type of data inside the array

- len(array) = returns the size of the rows (first dimension of the arry = number of elements in it) of an array or a multidimensional array

- array.size = how many object I have in the array

- array.itemsize = bitys occupied by each element in the array

- a.nbytes = the total number of bytes required for the array (for a.size = 4: using integers we use 4 byte in total, using floating number we need 32). So different data types can massively change the array size in terms of memory.

We can also use short command to define arrays whose element have identical value

> a = np.ones(10) = arrays of 10 elements which are all 1

> b = np.empty(20) = arrays with 20 elements with null values (different from 0, they are basically random numbers)

Some useful functions are np.ones\_like(other vector), np.emty\_like(other vector) which take as reference array as input and generates a nwe one with the right shape and data type, with number according on the command selected.

Other useful way to inizialize a vectir is

- a = np.arange(1, 10, 2) = initial value, final value, step size

- a=np.linspace(0, 10, 5) = initial value, final value, number of steps

- a = np.logspace(0, 2 ,5, base = ?) = inital value (in log scale with base ?), final value(in log scale wiht base ?), number of steps

- a = np.geomspace(1, 100, 5) = initial value, final value, number of steps

**Operations with arrays**

Numpy's array are not only simplifying the generation and management of data, but they have a huge number of operations that are already vectorized, such as **array-array** operations and **array-scalar**

a = np.array( [ 1, 2, 3, 4] )

a \* 10; a+1; a\*\* 2 perform the operation over each value of the array

For matrix and vector operations we can use the special command @ in order to replace the dot product.

For a matrix using: a[0] we get the first row, with: a[:, 0] we get the first column. Using: a[:2, :2] we get the elements (in a form of a smaller matrix) that belong at the same time to the first and second column and first and second row.

We can also modify the element of an array:

> a = np.array ( [ ] )

> a[a > 4] = 0 puts each element bigger then 4 in the array to zero

Some useful functions

- x\_mean = np.mean(x)

this also allow us to use a much smaller time to perform this operation cause the algorithm which defines the np.mean() function is optimized in term of memory and time.

- np.sin

- np.cos

- np.exp

- np.var = varians

- np.std = standard deviation

- np.sum

All these functions in general improve the performances of our code in terms of time and memory.

When you slice an array, you don't get a copy unless you specifically ask for it with the array.copy() method. So if you modify an element, the original array will be modified as well!

*example:*

a = np.arange(10)

print(a)

arr\_slice = a[2:5] # view of a

arr\_slice[1] = 10000 # change the second element of the slice

print(a) # the original array is changed!

a = np.arange(10)

print(a)

arr\_slice = a[2:5] # view of a

arr\_slice[1] = 10000 # change the second element of the slice

print(a) # the original array is changed!

Random number generation

Inside numpy there is a sublibrary called random

- rando.rand() = uniform distribution

- rand.randn() = normally distribuited numbers

- rand.exponential()

-random.rand(2,3) = matrix with random numbers and dimension 2x3. we can clearly specify how the random number are distribuited (ex: random.randn(2,3) for normally distribuited

Random number for simulations are a bless and a course! The problem is that you can never repeat in the same way how number are generated, so if there is an error in a simulation you cannot precisly know what caused it. To improve the replicability of a random simulations is possible to fix the random seed. This will ensure that the random number generated are repeated exactly in the same way 🡪 fundamental for debugging and testing cause we can repeat border case that break the code.

> random.seed(0)

> etc… generation of random numbers

> random.seed(0)

> etc.. generation of same random numbers

So, after a borderline results, just add random.seed(0) and tghe next random number will be equale to the previous one.

With numpy we can also reproduce logical operations

- np.logical\_and

- np.logial\_or

- np.greater

- np.equal

These can be used as boolean espression, is it true that a is bigger than b?

Some other useful functions are the sorting, argsortinf and where function

- np.where(a > 2, 1, 0) condition, if(condition satisfied), else 🡪 so where the element of the array a are bigger than 2 substituite them with 1, instead with 0.

- np.where(1 , 2, 3) where the element is equal to 1 put there the value of 2, contrary overwrite 3.

- np.sort(a) = return an array with the element of a sorted

- np.argsort(a) will give us the index of each element which say to us where they should be in the array

- p.argsort([1,3,2,4] will return array[0,2,1,3]

Why do I have to use np.argsort? I might want to sort an array depending on the order of another array or depending on its values. If i sort an array I can change the order of a second array in oder to keep the correspondance between arrays.

Broadcasting

Numpy can combine arrays of different sizes in a smart way, as long as they have a similar number of dimensions (or this can be padded out with sizes 1).

Main command:

- np.reshape

Directional application of functions

**numpy** supports also the application of function along certain axis and not only on the whole array.

The command:

> data = np.random(4, 2)

> np.mean(data, axis = 0) # mean of the first axis

> np.mean(data, axis = -1) # mean of the last axis

Memory recycling

One option to increase the memory performances is to use an array memory space to store the result of an operation, without requiring the creation of partial result.

Numpy for scientific data

Numpy arrays are well adapted to store and manipulate scientific data 🡪 structurally homogenous one!

Problem: a flourescence microscope might return a 6 dimensional array with dimoension for

- x,y and z axis

- colors (light vivid, various flourescence)

- etc…

In order to represents these type of data we can use an alternative library, such as xarray

In order to work with complex data we can consider Structured arrays in order to store arbitrary tuples as items inside an array. A good library for this is pandas. These are called Dataframe and will be discussed later.

Numpy for eigenvalue equation

To compute the eigenvalues and eigenvector of a matrix we can use the np.linalg.eig function.

*example*

> a = np.eye(4) + random.rand(4,4)

🡪 np.eye(4) gives an array whose elements are 4 arrays with four elements (another way to represent a matrix) with 1 on the diagonal and 0 elsewhere, the random.rand() function sum some rando values in order to have a matrix different from an identity

> eigenvals, eigenvecs = np.linalg.eig(a)

the function np.linalg.eig() returns a vector with two elements, each of these elemets is a vectors with 4 elements (4 eigenvalues and 4 eigenvectors).

Often one does not need all the eigenvectors or eigenvalues, usually one only needs the biggest or the smallest. There are allgorithms that allow the calculation of them, achieving the necessary result in a faster way.

Using:

> np.linalg.eig(a, which = ‘ ? ’)

where after “which” there will be:

- ‘LM’ to look for the largest magnitude (default)

- ‘SM’ to look for the smallest magnitude

- ‘LR’ to look for the largest real part

- ‘SR’ to look for the smallest real part

- ‘LI’ to look for the largest immaginary part

- ‘SI’ to look for the smallest immaginary part

The python scientific kit:

- NumPy: the array data structure and linear algebra

- SciPy: extensive scientific computational library

- Matplotlib: data visualization

With these three we can do 99% of our computational actions.

SciPy is a collection of library where each of them provides certain features:

* Clustering package (**scipy.cluster**)
* Constants (**scipy.constants**)
* Discrete Fourier transforms (**scipy.fftpack**)
* Integration and ODEs (**scipy.integrate**)
* Interpolation (**scipy.interpolate**)
* Input and output (**scipy.io**)
* Linear algebra (**scipy.linalg**)
* Miscellaneous routines (**scipy.misc**)
* $\rightarrow$
* Multi-dimensional image processing (**scipy.ndimage**)
* Orthogonal distance regression (**scipy.odr**)
* Optimization and Root Finding (**scipy.optimize**)
* Signal processing (**scipy.signal**)
* Sparse matrices (**scipy.sparse**)
* linear algebra (**scipy.sparse.linalg**)
* graphs routines (**scipy.sparse.csgraph**)
* Spatial algorithms and data structures (**scipy.spatial**)
* Special functions (**scipy.special**)
* Statistical functions (**scipy.stats**)

Differential equation with scipy

There are different algorithm with which we can integrate a differential equation, scipy provides us with a

unified interface for this under the scipy.integrate module, precisily with the function odeint.

odeint(espression to be integrated, initial point, variable of integration)

With Matplotlib we can plot and characterize our graphs.

Seaborn is a library which takes tabular data and visualize them with very convenient functions.

In Scipy we can also work with quantum concepts and eigenvalues equations.

NOTE out of context

in order to use the prefix np instead of numpy we can code:

> import numpy as np

The same for matplotlib

> import matplotlim as mpl

Metropolis algorithm = can be used to estimate the integral of a function using a Monte Carlo evaluation. In this case we will use it to estimate the value of pi.

LECTURE 7 – PANDAS DATAFRAME AND ADVANCED PLOTTING ACTIVITY

Dataframe = data structure equivalent to individual tables in a database, indicized with rows and column.

Python implement dataframe with Pandas DataFrames, Matlab implement dataframe with Table.

> import pandas as pd

Typically the column are used for measured quantity regardin the objects in the rows.

Dataframes are strongly used in data management, a special role is played by the data that are orginized as TIDY DATA. Tidy data is a way of structuring your data that makes analysis, visualization and data maitenance particulary easy.

Def: following proprieties:

- a table represent a single set of measurements

- every measured variable is represented by a column (each element present in that column is one of the values of that measurement referred to the respective observed unit)

- every observed unit is represented as a row

Cleary the observed unit can be reapeted on more rows in the case the other variables presents possible different values.

Name – year – type of home - via

Andrea – 2015 – residenza – via Roma

Andrea – 2015 – domicilia – via Milano

Andrea – 2016 – residenza – via Roma

etc…

Let’s have a look to the following code with which we can create the table above

> import pandas as pd

> data = pd.DataFrame (

> [

> (‘Andrea’, 2015, ‘residenza’, ‘via Roma’)

> (‘Andrea, 2015, ‘domicilio’, ‘via Milano’)

> (‘Andrea’, 2016, ‘residenza’, ‘via Roma’)

> ],

> columns = [‘name’, ‘year’, ‘type of home’, ‘via’]

> )

They are not always the best way to orginize data for your experiment. Tidy data is used for long term storage, but some analysis need the data to be transformed in non-tidy format. In fact all the libraries that manage dataframe have strong focus on data transformation.

Introduction to Pandas

Main library to manage dataframe, same levelo of importance of Numpy.

Pandas introduces both **dataframe** and the **series** = colum in the dataframe, all sharing the same group of indices.

Pandas write and read almost any tabular format! Let’s check if can read also txt files with few rows of strings. Pandas can also extrac all tables on a web page.

Command:

> import pandas as pd

in order to use pd.function while programming

example:

> wikitables = pd.read\_html(page)

where page is a link (> page = ‘link’)

> len(wikitables) 🡪 number of tables contained in the web page

> wikitable[0] 🡪 first of all the tables extracted

> wikitable[0].head[] 🡪 this function returns the first n rows for the object based on position. It’s useful for quickly testing if your object has the right type of data in it.

Each table readed should have a specific and different class, we can also sort tables depending on their class.

We can also modify the style of our table using a series of appropiate commands (style).

We can now store tables

> dataframe = wikitables[0].copy() 🡪 we want to store a copy because we want to modify and manage our data, if I for error damage a table, instead of re-downloidng the original data I can go back o the orginal one, deleting the copy

Using:

> dataframe[‘name of one of the column– the header’].head()

we will obtain a series, so the column whose header was “name of one of the column – the header”

Printing that we will obtain info on the name of the column and its type or dimension.

Series are a relative of the numpy array, the difference is that series is indicized by specific indices, that can also defined by the user using letters.

The indexing element of a series is a key concpet, we can creat a list [1, 2, 3] where the index are [a, b, c].

Defining another series assigning to the value a column of indices which has a different order with respect to [a, b, c] (let’s use [c, b, a]). Summing the two series, element with the same index are summed. The order is the one you decide, not the natural order the values you are considering has. If you don’t specify any order the index will run from 0 to d-1, where d is the size of the column.

Each series behave as a dictionary, each index of a series is a different value.

We can also remove specific column we don’t want.

> del dataframe[‘name\_of\_the\_column\_that\_have\_to\_be\_deleted’]

Using

> dataframe.info()

We obtain some info about the dataframe: number of columns, order, name and so on….

using:

> dataframe.select\_dtypes(include= np.number).head()

will return all the series which contains numbers.

Dataframe manipulation

str.split 🡪 Split strings around given separator/delimiter.

str.replace 🡪 used to replace section of a string with another element

str.astype 🡪 Cast a pandas object to a specified dtype.

are the most used commands for datframe manipulation.

Sometimes Pandas does not recognize an integer if in the original table (such as wikipedia) the integer number has a reference (as an exponent). We can do an element by element substitution, there are no specific function which help us in manage this problem.

Given that there is no simple transformation, we need to use **regular** **expressions** shich are a way to espress text structure and to specify which part to extract from it. They ara a high level instrument.

Using:

>dataframe.describe()

we will have numerical info about our data

Dataframe visualization

We can use matplotlib to visualize data instead of pandas.

> plt.scatter(‘name1’, ‘name2’, data = dataframe)

or

> plt.plot(‘name1’, ‘name2’, data = dataframe)

where name1 and name2 are the name of two series of a dataframe.

Pandas dataframes (and in general the dataframe structure) is designed to represent data stored in table form, with columns and rows, containing scalar data such as height, weight, age and so on...

For more complicated data, where different kind of data structure need to be related to each other while keeping most of the goodies that pandas provides, an alternative is [**XArray**](http://xarray.pydata.org/en/stable/).

This library includes the equivalent of the pandas dataframe and series (**DataArray** and **DataSet**), that provide an efficient and comfortable ways to manipulate multidimensional arrays, where several of them share one of more of their indexes

Manipulating Dataframes

Three operations

- groupby

- join and merge

-pivot, melt and stacking

**Groupby**:

- divide data in groups

- transform some of these data (mapping elements to other elements or aggregation to a single value – typical example of aggregation is the calculus of global variable susch as means, standard deviation etc…)

- merge the results from the previous step

So: mapping means transform elements in other elements, graphical representation of functions.

Aggregation means use all the elements, or a lot of them, to compute a global information. SO the results of the agregation is a value for group, in case of mapping the result will be another dataframe with the transformed values.

Using:

> name.groupby(‘name of the series’).mean()

to compute the mean value

Join and Merge

Two tables which share at least partially their index can be joined into one. In general this is used to obtain a clean and tidy format for storage which merge all the information I need for my analysis.

There are several way to join two table; left join, right join, inner join, outer join.

In the **left join** the left dataframe is kept completely.

The right dataframe is selected based on the index of the first one:

* if the index does not appear in the first one, the item is not selected
* if it appears multiple times, it is selected multiple times

this is the default join in pandas and most databases. The **right join** is exactly the opposite but using the index of the right dataframe to select on the left one.

In the **inner join** only the elements that are common to both dataframes indexes are kept (it is akin to an intersection of the two).

In the **outer join** all the elements are kept, with all the possible combinations repeated (it is akin to a combination of cartesian products of the elements)

Pivoting, Melting and stacking

Pivoting is a family of operations that allows to create aggregation tables starting from a tidy dataset.

esemple:

for each person for each mounth for each currency there is how much it has spent

we can orginize a table where for each person we have the annula spent.

We typically work n tidy data set, taht have to be previously put in a wide format (melting operation)

To perform a pivot one chooses:

* one column whose unique values are going to be the new index
* one column whose unique values are going to be the new column names
* one column whose values are going to be fill the new tabls

if there is more than one value that correspond to each pair of values (index and column) then one needs to specify some aggregation function to summarize those values, such as sum, average, counts, standard deviation, etc...

Using

> pd.pivot.table() what open the table in order to pivoting

> fill\_value = 0, will fill each section of the table that has NaN with 0.

Plotting

Inside matplotlib the sublibrary Seaborn wil provide a very good user interface.

> import seaborn

Seaborn by default changes the standard configuration of matplotlib for visualization

You can set however you like using the style module of matplotlib.

> from matplotlib import style

etc..

> seaborn.lmplot(

> data1,

> data2,

> hue = ‘group’, different color for each data belongin to the group

>)

Using instead of hue:

> col = ‘group’ 🡪 will realize a plot for each data, on near the other (in a row)

> row = ‘group’ 🡪 will realize a plot for each data, on near the other (in a column)

LECTURE 8 – OBJECT ORIENTED PROGRAMMING

The goal is the idea of libraries development and script development, when we want to implement some sort of API.

Code categorization:

- prompt: typing command, see what happend at screen or at your variables

- scripts: collection of some line of code, run them to do the operation you are intersted in

- libraries: organized collection of

- programs: high level interaction with the user, we don’t have to write a code, it usually has a some type of user friendly interface that define the type of program we are considering.

- frameworks: programs with which the user interact by writing small piece of code. As the program call libraries to run the code, framework call your code to run everything.

Everything in Python is an object, we’ll discuss the idea of OOP in Python.

The advantage is that allows you to write clean and nice program, but require a specific way of thinking about your program.

We typically use OOP to create a new library or to do wrapping: taking another library (old, or written in another leanguage) and create a nice interface (such a shell) that can be used by the user.

Object = black box with whome we can communicate, giving commands to perform and asking questions (as for the routine and functions). We just care about the communication, the answer we can have from the object characterize it.

Let’s consider the object “list” in Python. We can manipulate it in lot of way, but we don’t know how the list work under the hood 🡪 but we are not interested in it…

A good object should be self sufficient (having a high cohesion) and not depend on specific other object (have low coupling). We cannot have perfect cohesion and no coupling, it’s not possible in real world.

In OOP we can talk about instaces of classes, which implements several interfaces (in Python called protocol). Interfaces is a set of logiacally connected ways I have to communicate with the object.

example:

- my phone (my = instance - specification) (phone = classes - general)

- touch screen, buttons, voice etc… (interfaces)

The instance is the object with which I interact, the class describe the kind of object I’m interacting with, telling me how I can interact with it. The interfaces are the various ways I can interact with it. The interface should not specificy how the class works internally, only how I can interact with it.

Mutability = I can change the values inside the object once it has been created.

Indexable = I can extract items from inside the object with the [ ] syntax.

Methods are used to communicate with objects, methods have sam logic of normal functions 🡪 they can behave as routin (change the state of the objects) or as function (having a return from the object).

We can have methods that are routins which change the state of the object and return the object itself.

How can we use a method?

Given an instance of a specific class we use:

> <instance>.<method>(<parameters>)

for example

> a = [1, 2, 3]

> a.append(4) # add a new element to the list

> a.index(3) # return the index of the element without changin it.

Each objext can have an internal state inside it, called attribute.

Working in the frame of OOP we are interested in how something behaves and not in what objects are.

A functions in which we sum a and b through a+b, assumes a and b know how to be added togheter, no assumptions are made on what they are…

So when i define a function, I’m defining the interface I expect from my objects. We can clearly fix the class that the function accept, but it’s generally incorrect… we shoudl be fixing the interface we want our object to provide!

You have to leave the user do what he wants with your function!

How to use objects for library

Don’t start writing the code to implement the features, write the code you want to be able to write

- write how the features should look like in real code

- then try to implement it

This is useful to understand which are the most important feature + allow you to put testing in the header to be able to validate immediatly your newborn code.

In creating a class remember that:

- one class should have one job only

- the class should be as small as you can get away with to do the job

- it should not make assumption on who or which other object is going to communicate with it (most complex requirment)

Remember that code is work: have to be tested, mantained, the debugger has to go through it… Let’s keep everything simple and short.

example of class Timer.

Scikit - learn = library of machine learning with Python.

It has a very simple interface.

Nowdays the majority of the machine learning taht is done follow the Scikit-learn, useful to correctly write the interface for working with classes.

The whole sklearn is base on:

- classifiers, regressors and transformers (simple interface)

- pipelines and unions

Every class that is a transformer should implement a fit and transform method, while a regressior or a classifier implement the method fit and predict.

With fit the object should learn from the data.

fit(X, y) should take

- X (rows for the observations, colomn for features)

- y (rows for observations, multiple column for values)

- return = the predictor itself (the object is allowed to return itself)

predict(X) should take:

- array X (same)

- return: an array like (row for obs, one column per features)

transform(X) should take:

- array X (same)

- return: an array like (row for obs, one column per features)

Unions = take series of transformer and combine all their outputs in a single transformed dataset: it behaves as a transformer itself

Pipeline = take a list of transformers and pipe the data from one to the other: it behaves like a tranformer or estimator itself.

They can be be used inside other unions and pipelines in order to create a full data flow structure that behaves correctly.

LECTURE 9 – RANDOM NUMBER AND MONTE-CARLO METHODS

Random Numbers

While generating random numbers w want the randomness, but, at the same time, for certain purpose we want that this randomness is repeatable.

> import random

> random.random() 🡪 randum number as output

Every time we run the script we’ll se different number. All random number allow you seeding random number that means: freezing a random sequence.

> random.seed(42)

> random.random() for i in range(3)

For each run of this script we obtain the same result.

the value in the parenthesis is optional and is used to generate the random numbers (if it’s an integer is used directly, if not is converted into an integral, we can specify how to make this conversion putting a second integer as a parameter)

If we run a simulation and we see that for a certain random number we obtaina strange result (bug, error, unexpected result), usign seed we are able to repeat the sequence in order to study in a detailed way the case of interest.

NEVER DO ANY SIMULATION (which involve random numbers) WITHOUT random.seed()

🡪 how can we actually obtain random number?

The main method exploit the linear congruential generator, starting from a number and using a precise rule we can generate a random series of number using three constants as parameters: the result is a pseudo-randomized series of number.

The generator is in fact ricorsively defined by:

X\_(n+1) = (a\*X\_n + C) \* mod(m)

X\_n is the n-th value of a sequence of pseudorandom values, a is the multiplier and c the increment.

The parameter m gives the total number from which we can take the ranomic number we will use (?) and its typically 2^32 or 2^64..

We typically divide the final value of the rule by m, in order to keep small value between 0 and 1 (also a and c has typically quite high value 🡪 typically: a = 1664525, c = 1013904223).

Generating enough number and plotting their pdf, we will se each value has the same probability (constand pdf) except 0 and 1, the probability of obtain each of them is very very low (1 over all the possible generated value).

For all intend whetever we generate random number we can basically assume that neither perfect 0 or perfect 1 will be generated.

Between the standard libraries and the scientific stack, we have access to 3 set of random functions generation:

- random

- numoy.random

- scipy.stats

Using them we have different type of distributions: normal, exponential, Poisson, gamma etc…

Many times is not sufficient to consider a constand or normal pdf. It manly depends on the object we want to randomize: think about a random angle in a 3D system.

Each object has some preferred pdf which can generate it.

scipy.stats 🡪 huge library of pdf

All these distributions are decribed using a location and scale paramter (and if necessary also one or more shape parameters) 🡪 the firsts can be: mean and standard deviation for example (gaussian).

The location is always included, also for pdf that might not use its value.

These distribuition can be used in frozen or unfrozen state. A frozen distribution has its parameters fixed, while and unfrozen one requires them every time we call it.

We can use st.gamma.rvs command: rvs means random variates.

Using

> dist = st.gamma(2.0, loc = 0, scale = 1)

we can have our customized distibution

Then writing

> dist.rvs(5)

we will have 5 elements from the distribution “dist”

other commands can be: dist.str(), dist.mean() to compute the main parameter of a distribution.

We can also plot a distribution to understand its shape: we can generate through linspace a series of numbers in which we can evaluate the distribuition.

The location is substantailly a shift of the distribution taht, for certain distribution is not apropriate (like for the gamma)

We can aslo fit a distribution and specify some parameter to be fixed, such as we can fix the location at zero

> st.gamma.fit(data, floc = 0)

Precisly using:

> random\_values = dist.rvs(100)

> st.norm.fit(random\_values)

will give us two parameters that represents the mean and the standard deviations (cause we are fitting using a normal distribution as reference).

Same for others, the output will be a series of parameters useful to describe a theoretical curve using the distribution which define the fit as a reference.

Increasing the number of element that compose a distribution we will obtain better results.

Empirical cumulative distribution function

In plotting the cumulative pdf we can evidence the data using sns.rugplot function.

A way to visualize a cumulative pdf is using a step pdf. Before generating a cumulative pdf is always better if we sort the elements we have.

Writing:

> plot\_ecdf(plt.randn(100), color = ‘r’)

we plot a cumulative distribution of a normal distribution made of 100 values.

With a cumulative distribution we can see where the median is, looking where the curve has its higher slope, other characteristic can be deduced simply looking to the comulative plot.

There are a lot of ways to work with distributions.

We can also generate multiple distributions having same paramaters but slightly different number.

We can extract info about our distributions. The cumulative distribution is also used to define different momentum from mean and variance.

being abel to describe the ECDF, even as an approximation, allows us to generate random numbers with an arbitray complicated ditribution.

Monte Carlo method

From a formal prosepctive is a way to estimate integral. We substantially generates random numbers to estimates the average value of some specific function, often integrals. So we can sum the function eveluated in each of these numbers, dividing by the number or this values. We can sometimes approximate an integral with a summation made over a domain where lives numbers randomly generated.

the rate of convergence using an unifomr sampling of N numbers is O(1/N).

What is the advantages of using random numbers?

We can do a lot of estimates on different series of random numbers, also enlarging a random number sequence, and, in the end, combine the information togheter in order to reach a better precision.

Sampling using random numbers converge more slowly, but can be performed for as lons as we want until we reach the convergence up to a certain precision. In fact, with a uniformely sampled numbers, if the chosen sample wasn’t enoigh, we have to start from the scratch and redo everything.

There are some good method for exploring a function, using for example low-discrepancy sequence. These are sequence that looks quite random, but do not tend to cluster as random numbers do, covering the space in a more uniform way. The advantage is that they keep having this propriety (of uniform covering) even if we continue forward with the sequence of sampling. Methods that use these low-discrepancy sequence are called quasi-Monte Carlo methods.

The simplest low discrepancy sequence is the Van der Corput sequence and is quite easy to implement. It’s a sequence made of fraction where the denominators are increasing powers of the base, the numerators are all the values in that base (we have sorted base in the last digit).

It’s clear the difference of using VdC sequence or not, simply comparing two constant pdf function generated in differente ways. The VdC sequence based pdf is much more similar to a constant pdf even if in both cases we used 1000 number (quite high) 🡪 more uniform generation, absence of clustering.

SO:

If we know the dimension the sample should have we can set the number of points initially.

If we don’t know the number of points our sample has to have we have to keep generating them!

In conclusion (TAKE HOME MESSAGE)

If you want to have high precision numerical integration the cumulative density function is what allows you to generate these random values with very high precision, by generating either an **uniform sampling** (with linspace function) or, if we want to **keep generating more and more data**, using a low descrepancy sequences that garantees that you can generate them indefinitivly without clustering effects.

We can clearly generalize the VdC sequence for high dimensions

An interesting alternatove is a low-discrepancy series based on the golden ratio numbers. It’s easy to implement and very similar to the linear generator method. Using this sequence we can generated a sort of real lattice (quite periodic, but with some defects) on a 2D plane.

LECTURE 10 – INTEGRATION, SENSITITVITY AND FITTING

Cauchy problem using scipy **odeint** function, that needs:

- function that describe the derivative of the system

- the starting state of the system (initianl conditions)

- the time points over which to integrate (the domain)

- some parameters that describes the differential equation, that will be routed to the derivatove function

odeint use the time points it needs to integrate, the time points you give are used to ?

from scipy.integrate import odeint

logistic equation = describe growth of biological organism (ex: bacteria) and depends on two competing effect:

- classical exponential growth

- limiting process

at a certain point we start to see a competition/limiting effect and we no more see an exp growth.

Typically it’s described by two different parameters:

- prop to N

- prop to N^2

alpha, beta prop. coefficient (alpha >> beta)

When N increase a lot the protoN^2 term start to be more important.

Input parameter of logistic function

- N = initial state

- time = time domain in which you want to integrate

- alpha

- beta

The result that you obtain from logistic function is 100% determined by the inputs.

deltaN = alpha \* N - beta \* N^2

N0 = inital state = 0.1 (10% of maximum popolation – saturation value)

alpha = 10

beta = 1

time = linspace(\*\*\*)

using odeint(function, time) you’ll obtain the integral evaluation

res = odeint(parameters)

Then we can plot the result

ax.plot(time, res, label =”population”)

A convenient function is to transform the result of the integration in a pandas dataframe.

This allows us to use it for better visualization using seaborn

\* and \*\* notation

python allows to define some special arguments in the form of \* and \*\*

\*args = arbitrary number of parameters 🡪 it’s a sort of arbitrary string

\*\*kwargs = arbitrary number of name 🡪 it’s a sort of arbitrary dictionary