Introdución a Python, Git e Github

Fran Rúa
Breixo Camiña

Grupo de Programadores e Usuarios Linux

10 de febrero de 2016

Índice

Introducción a Python

Fontes e referencias

- Documentación oficial de Python
  www.python.org/doc
- Guías de estilo PEP
  www.python.org/dev/peps/pep-0008/
- Guía para principiantes
  wiki.python.org/moin/BeginnersGuide
- Learn Python (español)
  http://www.learnpython.org/es/
- Tracducción do manual de Guido van Rossum
  http://docs.python.org.ar/tutorial/pdfs/TutorialPython2.pdf

Introdución

- Historia
- Para qué é usado.
- Características
- Ventaxas
- Inconvintes

Historia

- Creado a finais dos 80's por Guido van Rossum
- Desenvolvido para o SO Amoeba
- Toma partes de outros linguaxes de programación, como Haskell ou ABC
- A súa finalidade é programar facilmente e obrigar ó programador a realizar código entendible.
- Python 1.0 liberado en xaneiro de 1994
- Actualmente dous proxectos paralelos: Python 2.7 e Python 3.4
- Instalado por defecto na maioría das distribucións GNU/Linux

Características

- Linguaxe de programación interpretado
- Multiplataforma
- Multiparadigma (Programación imperativa, orientada a obxectos e funcional)
- Tipado dinámico
- Código aberto (Python Software Foundation License, compatible con GNU)
- Herencia
- Modo interactivo

Para qué é usado

- Inicialmente, tareas livianas ou de administración.
- Posúe unha ampla cantidade de librerías que posibilita un amplo rango de escenarios de uso
- Computación numérica
- Xeración de gráficos
- Programación web
- Interacción con bases de datos
- Programas de usuario con interface gráfico.

Funcionamento

Ó contrario que C ou Java, Python non precisa ser compilado. En realidade Python é un intérprete de comandos que acepta unha serie de instruccións que respetan unha serie de normas lóxicas e semánticas.
Por tanto, se queremos executar o noso código en múltiples plataformas, só é necesario instalar o intérprete, xa que despois será éste o encargado de executar as órdes indicadas polas instruccións.

Compilación vs. Interpretación

Figura : Compilacion de un programa

Compilación vs. Interpretación

Figura : Interpretacion de un programa

Que é o modo interactivo?

O modo interactivo permítenos iniciar unha instancia do intérprete de python, o que nos permite programar en tempo real e comprobar a sintaxis do noso programa.
Neste modo podemos cargar librerías e módulos, e executar funcións de forma rápida sen necesidade de realizar programas previamente.

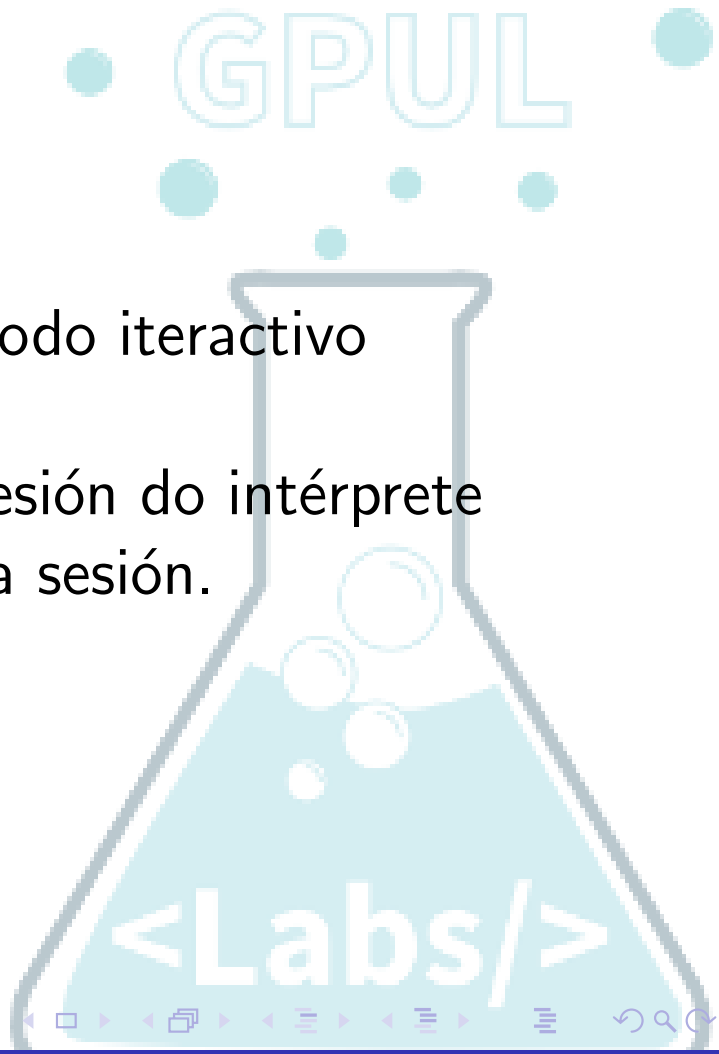
Iniciar modo interactivo

Nos sistemas GNU/Linux soamente é necesario abrir un emulador do terminal e escribir *python*.

```
[fran@izanami ~]$ python
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more i
>>>
```

Sair do modo interactivo

Unha vez que rematemos, para sair do modo iteractivo só temos que chamar á función *exit()*
As funcións e variables definidas nunha sesión do intérprete son borrados unha vez que se finaliza dita sesión.



Tipos básicos

Os tipos de datos básicos definidos por Python son os seguintes:

- Enteiros (int)
- Numeros en punto flotante (float)
- Números longos (long)
- Números complexos (complex)
- Caracteres (char)
- Cadeas de caracteres (string)
- Tuplas (tuple)

Operadores lóxicos

- and
- or
- not
- is, is not
- in, not in

Operadores matemáticos

- + (suma)
- - (resta)
- / (division)
- * (multiplicación)
- % (módulo)

Estructuras de datos

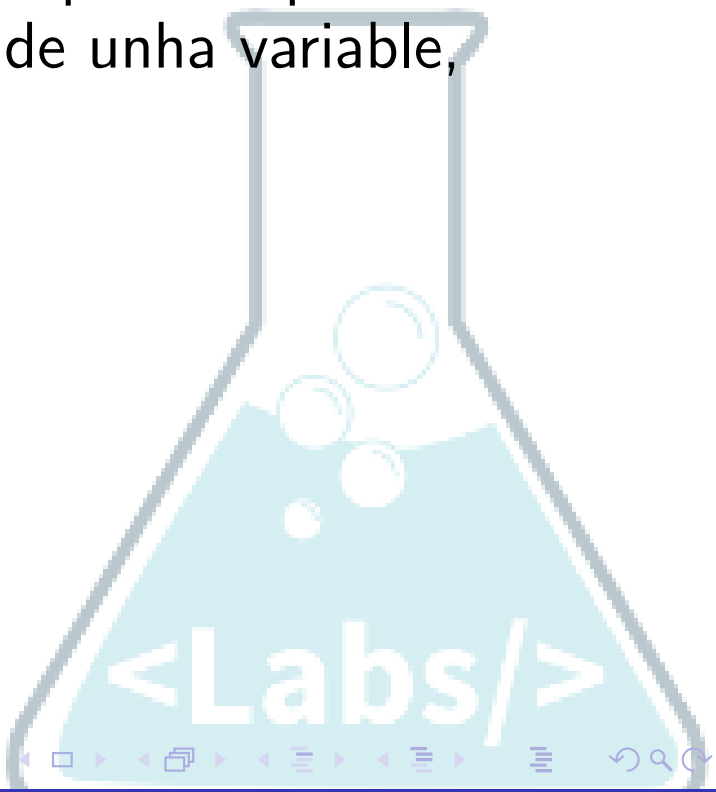
Además dos tipos de datos básicos, Python soporta de forma nativa varias estructuras de datos, das cales veremos as dúas máis empregadas.

- **Listas**
  Conxunto de tipos básicos(enteiros, números en punto flotante) ou compostos(tuplas), poden estar ordenados ou non.
  a = [1,2,3,4]
- **Diccionarios**
  Asocian un valor a unha clave para mellorar o acceso a un determinado elemento.
  alumnos = {'00001A':'Breixo','00002B':'Fran'}

Definir variables

O intérprete python ten incorporado un motor de inferencia, o que significa que cando declaramos unha variable non temos que declarar o seu tipo, xa se encarga o propio intérprete de inferilo. Se temos dúbidas acerca do tipo de unha variable, podemos sabelo coa función *type()*

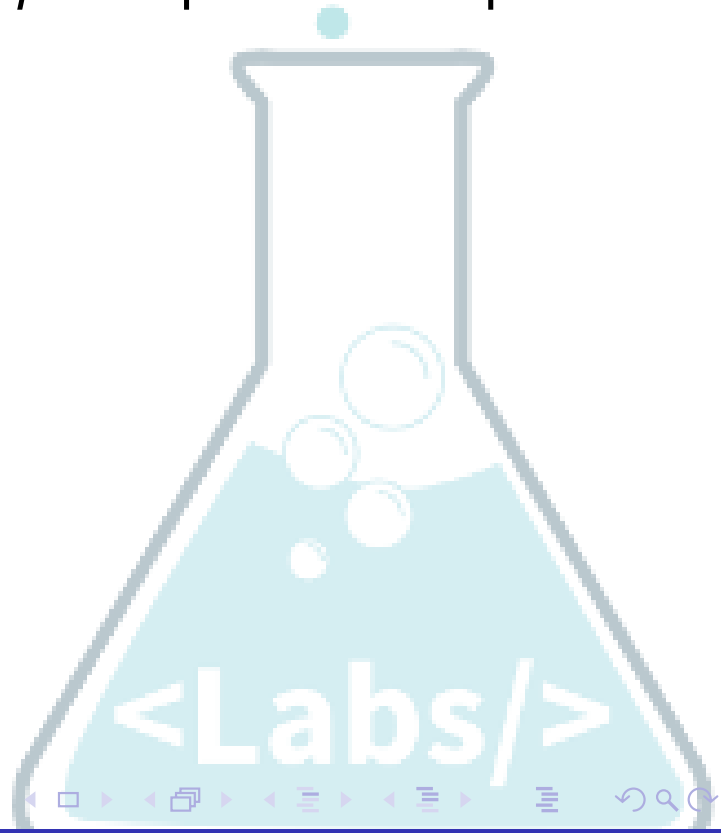
```
>>> sete = 7
>>> type(sete)
<type 'int'>
>>> a = "primeiro"
>>> type(a)
<type 'str'>
```



Funcións incorporadas (Built-in functions)

Python fai uso de librerías por defecto, que non é necesario cargar, que proveen funcionalidades básicas, coma *type*, que dado unha variable devolve o seu tipo ou *print* que mostra por saída estándar o argumento pasado.

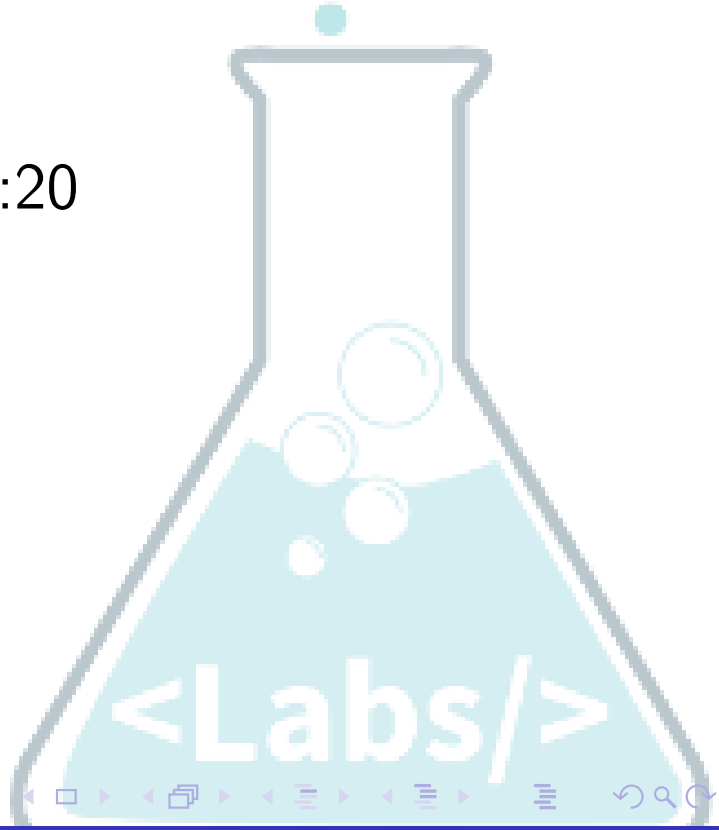
```
>>> n = "cadea"
>>> print(n)
cadea
>>> numero = 745
>>> print(numero)
745
>>> flotante = 3.14
>>> print(flotante)
3.14
```



Operacións sobre diccionarios

As operacións sobre direccionarios son proporcionados polas librerías básicas, xa que son un tipo moi empregado.

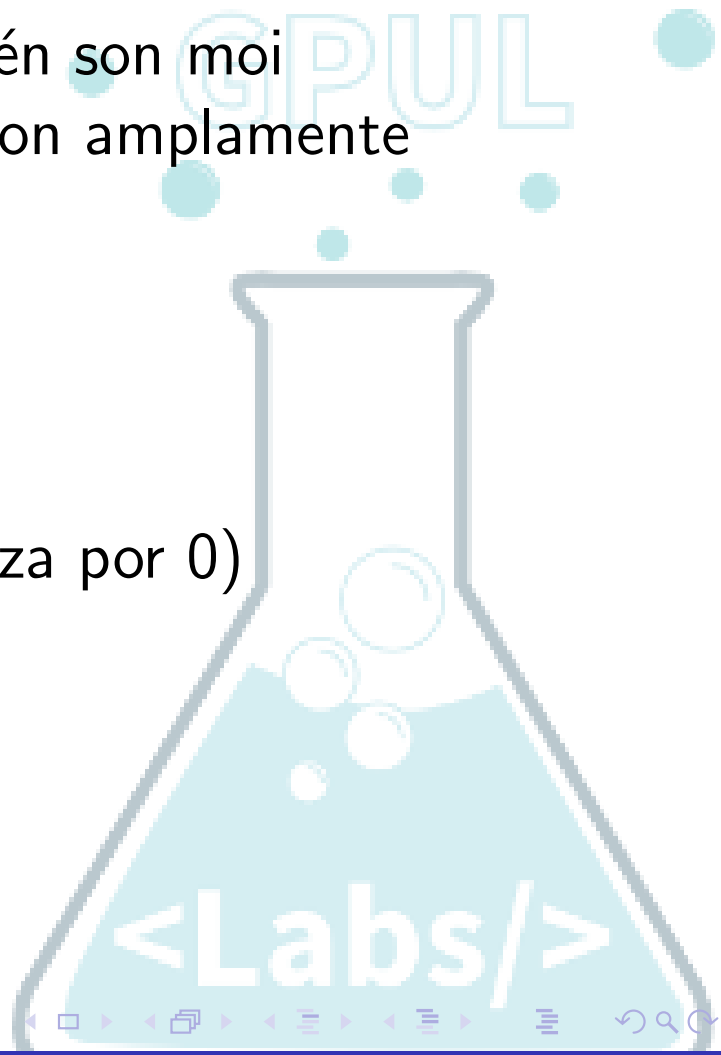
- Declaración
  d1 = {} # Diccionario valeiro
  d2 = 'Xoan':22, 'Beatriz':18, 'Anxo':20
- Obter un elemento
  d1['clave']
- Engadir un elemento
  d1.update('clave':'valor')
- Borrar un elemento
  del d1['clave]



Operacións sobre listas

Ó igual que os diccionarios, as listas tamén son moi empregadas e as operacións sobre estas son amplamente soportadas.

- Declaración
  l = [] # Lista valeira
  l = [3, 12, 15, 16]
- Obter un elemento por índice (comeza por 0)
  l[7]
- Engadir un elemento o final da lista
  l.append(n)
- Borrar un elemento i
  del l[i]



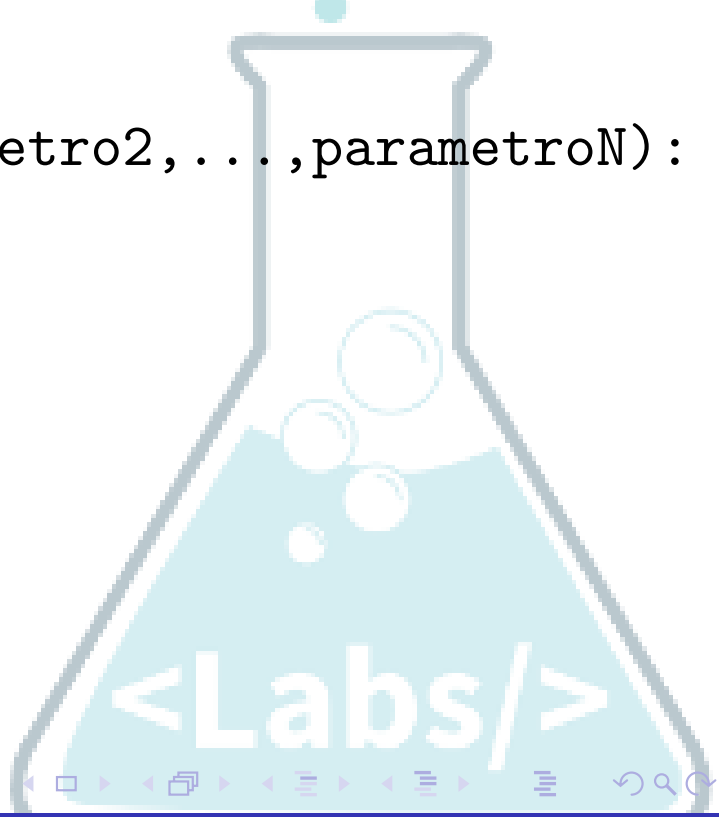
Definir funcións

Cando un fragmento de código é executado en numerosas ocasións, ou o código é extenso e complexo faise necesario estructuralo en funcións. En python as funcións declaranse da seguinte forma:

```
def <nome_funcion>(parametro1,parametro2,...,parametroN):
    sentencias dentro da función
```

Coma por exemplo esta función de suma

```
>>> def suma(a,b):
...     return a+b
...
>>> suma(3,4)
7
```



Indentación

Como vimos ó principio, un dos obxectivos de Python é conseguir un código claro e lexible, para o cal fai obligatorio o uso da indentación.
Unha mala indentación cambia o significado do programa.

```
def access_control(name, pass):
  if authentication(name,pass) == 0:
    print(''Access denied'')
    exit()
  else:
    print(''Access granted'')
    grant_access()
```

```
def access_control(name, pass):
  if authentication(name,pass) == 0:
    print(''Access denied'')
    exit()
  else:
    print(''Access granted'')
  grant_access()
```

No exemplo da dereita o acceso é otorgado a pesar de fallar a autenticación.

Estructuras de control

If

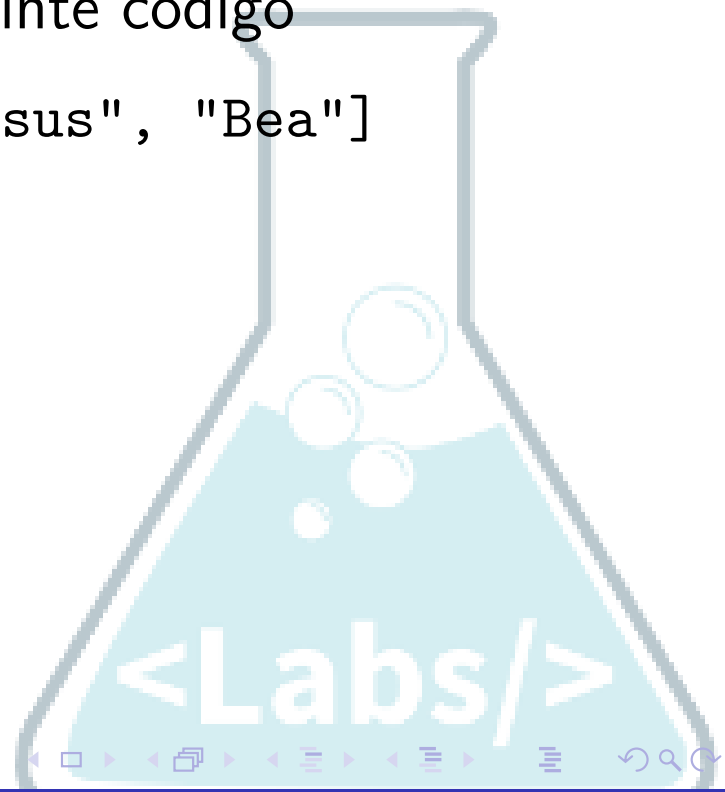
```
if test1:
   print a
elif test2 and test3:
   print b
   exit()
elif test4:
   print(‘‘nada que facer’’)
else:
   print(‘‘Saindo’’)
```

Estructuras de control

Bucle for

Os bucles for son aplicados sobre listas. Esto quere decir que se queremos percorrer unha lista e aplicar unha función a cada elemento, ésto é posible mediante o seguinte código

```
>>> alumnos = ["Xoan", "Marcos","Jesus", "Bea"]
>>> for alumno in alumnos:
...     print alumno
...
Xoan
Marcos
Jesus
Bea
>>>
```



Bucle for

Se queremos un bucle secuencial, podemos empregar a función *range*, que crea unha lista de elementos.

```
>>> for n in range(2,8):
...     print n
...
2
3
4
5
6
7
>>>
```

Estructuras de control

Bucle while

Igual que no resto de linguaxes de programación, o bucle while permítenos realizar unha acción repetitiva mentras se cumpla unha condición, usando a sentencia **break** para saír. Se non queremos saír do bucle, senón saltar as sentencias pendentes e executar o seguinte salto de bucle usamos **continue**.

```
while True:
    if test1:
        continue
    elif test2:
        print ("Hola")
        break
    else:
```

Ficheiros de código

Python é un programa de scripting, é decir, as instruccións son escritas nun ficheiro de texto plano, con extensión *.py* tal e como serían introducidas no intérprete.
A execución de un script consiste básicamente na lectura secuencial do ficheiro de texto e a execución das insctruccións correspondentes por parte do intérprete.
Existen dous tipos de programar en ficheiros: módulos e programas.

Módulos

Os módulos son definicións de funcións. Se existen funcións que usamos reiteradamente, podemos definilas nun ficheiro con extensión *.py* e importalo despois mediante a orde **import**.
Os módulos proporcionan funcionalidades, pero non realizan accións de por sí, senon que son usados polos programas para axilizar o desenvolvemento de aplicacións.
Existen dúas formas de referenciar módulos, **import** e **from**

Modulo operacions.py

```
def suma (a,b):
    return a+b

def resta (a,b):
    return a-b

def devolve_maior (a,b):
    if a>b:
        return a
    else:
        return b
```

import

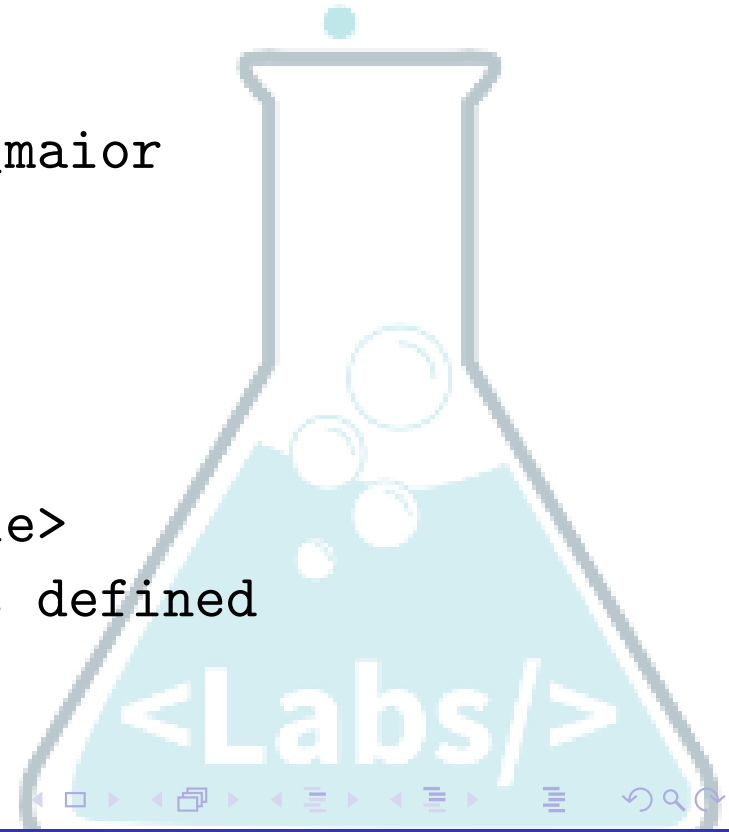
A sentencia **import** comproba se nun módulo existe o ficheiro **init.py**, que contén unha variable *all* que define as funcións que deben ser importadas cando o método é chamado. Además, engade un novo espacio de nomes, no que son definidas as funcións e variables do módulo. Debido a esto o programador debe especificar o espazo de nomes antes de chamar á función a empregar.

```
>>> import operacions
>>> suma(2,8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'suma' is not defined
>>> operacions.suma(2,8)
```

from

Mediante a orde **from**, podemos importar funcións concretas ó noso espacio de nomes, polo que para usala non é necesario especificar un espacio de nomes.

```
>>> from operacions import devolve_maior
>>> devolve_maior(15,8)
15
>>> operacions.devolve_maior(15,8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'operacions' is not defined
```



Scripts Python

Obviamente é interesante ter programas que se executen de forma que non sexa necesario o uso de unha sesión interactiva co intérprete. A estes ficheiros denomínase scripts, e, ó igual que os módulos, son leidos polo intérprete de forma secuencial. Neles podemos definir un conxunto de funcións e variables coas que construir unha secuencia de pasos a executar.
Para que se execute é necesario que exista unha función **main**, que é a última en ser declarada.
Cando o intérprete lee un arquivo de texto, se existe unha función main executa as ordes dentro dela como si se tratara dunha sesión interactiva, de forma que pode ser usada como disparador de un conxunto de accións.

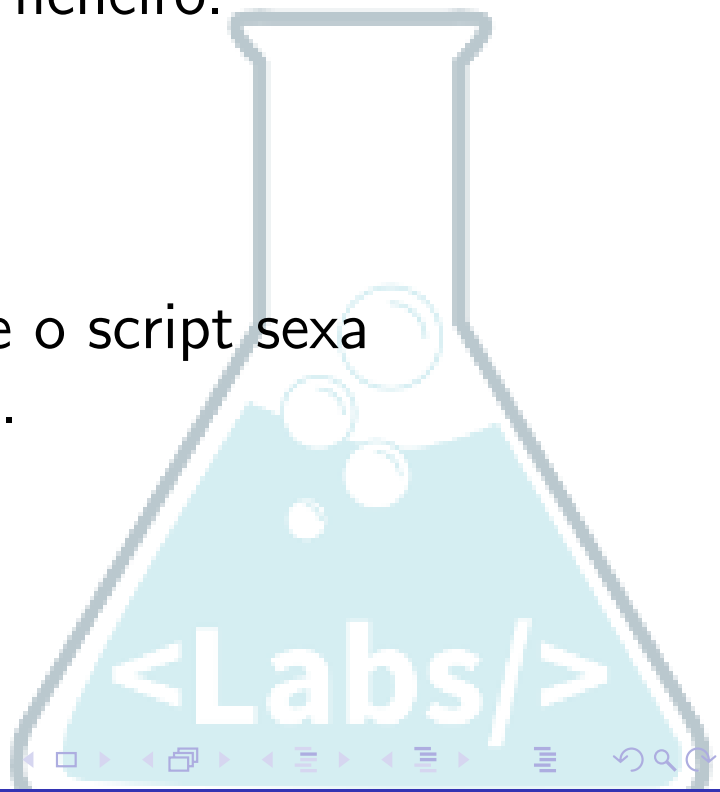
Cabeceiras

Para executar un programa en python, tan só é necesario asegurarnos de que o arquivo ten permisos de execución e a continuación chamar ó interprete sobre o ficheiro.

```
python arquivo.py
```

É posible engadir a seguinte liña para que o script sexa executado polo propio entorno de usuario.

```
#!/usr/bin/env python
```



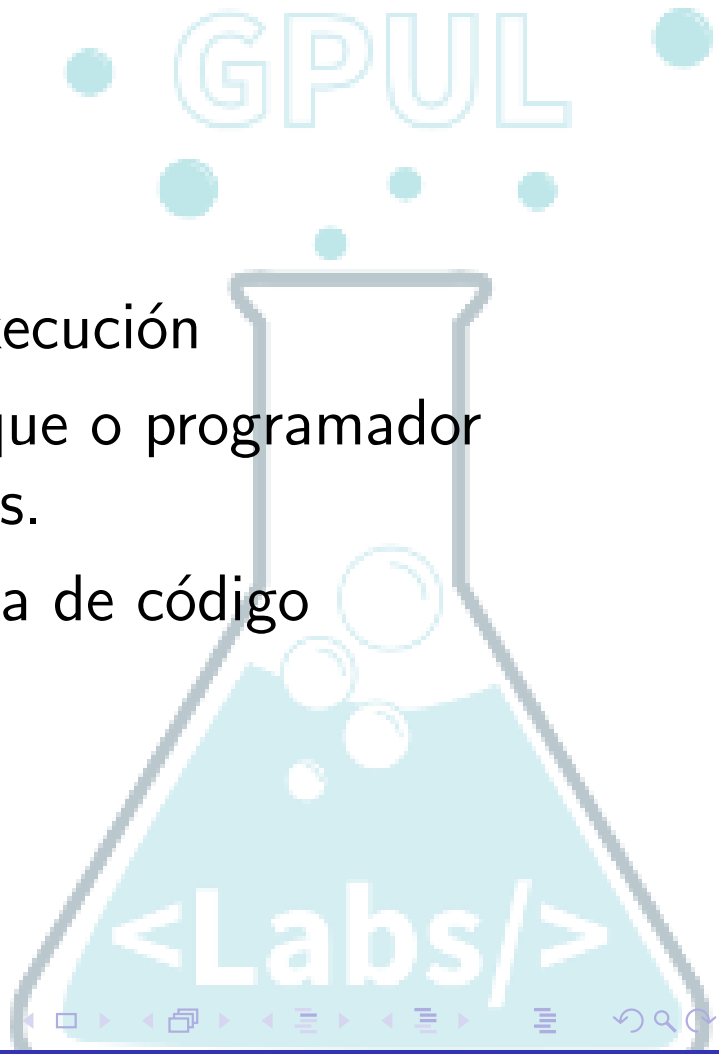
Cabeceiras

Tamén é recomendable, se o texto vai incluir carácteres que poden ser problemáticos coma 'ñ', ou tildes, especificar a codificación na que se gardou o ficheiro. O máis apropiado é facelo en UTF-8

```
# -*- coding: utf-8 -*-
```

Inconvintes

- Os erros teñen lugar en tempo de execución
- Como o tipado é dinamico, require que o programador preste maior atención a estos detalles.
- Rendemento, principalmente na carga de código



Control de versións Git

Táboa de contidos

Sistema de control de versións

¿Que é o control de versións?

É a xestión de diversos cambios que se realizan sobre os elementos de algún produto ou unha configuración do mesmo. Unha versión é o estado no que se encontra o produto nun momento do seu desenvolvemento.

¿Que é un sistema de control de versións?

Son ferramentas que facilitan a administración das distintas versións de cada produto desenvolvido.

¿Que é Git?

Definición

Git é un sistema de control de versións distribuido gratuito e de código aberto deseñado para xestionar todo dende un proxecto pequeno a un moi grande, de forma rápida e eficiente.

- É rápido, eficiente, escalable e distribuido.
- Propicia o traballo en local e o uso de ramas.
- Proporciona un nivel de historia do proxecto completísimo.
- Diferencia os commits unívocamente mediante claves xeradas con SHA-1.
- Cumpre as palabras máxicas do GPUL: é libre e gratuito.

Comparación con servizos de almacenamento na nube

Dropbox:

- Útiles para compartir ficheiros con compañeiros.
- O problema é se queres modificar o mesmo ficheiro ambos ó mesmo tempo, vai haber problemas de sincronización.

Google Drive:

- Útil para modificar ficheiros en tempo real, pero con proxectos software é imposible.

VCS

Os VCS son a solución!

Comparación con outros VCS

Subversion:

- Git é moito máis rápido e lixeiro que SVN (un repo SVN ocupa 30x o que un de Git).
- Unha rama en SVN é unha copia completa do repositorio mentres que en Git é un simple punteiro e conleva toda a historia do proxecto ata ese punto.
- Git proporciona mellor auditoría de eventos de ramificación e fusión (ramas e merge).
- SVN é usado en case todas as asignaturas da FIC.
- Por dicir algo bo de SVN, é máis fácil o seu uso para principiantes e ten máis ferramentas de integración con IDEs.

Mercurial:

- Moi parecido a Git pero é menos coñecido.

Instalación nos SO

Linux

(Proyecto Debian) apt-get install git
(RPM) yum install git

Mac

http://git-scm.com/download/mac

Windows

https://git-scm.com/download/win

Interfaces gráficas de usuario para Git

Linux

SmartGit - http://www.syntevo.com/smartgit/download

Mac e Windows

Sourcetree - https://es.atlassian.com/software/sourcetree/overview/

Inicio do uso de Git

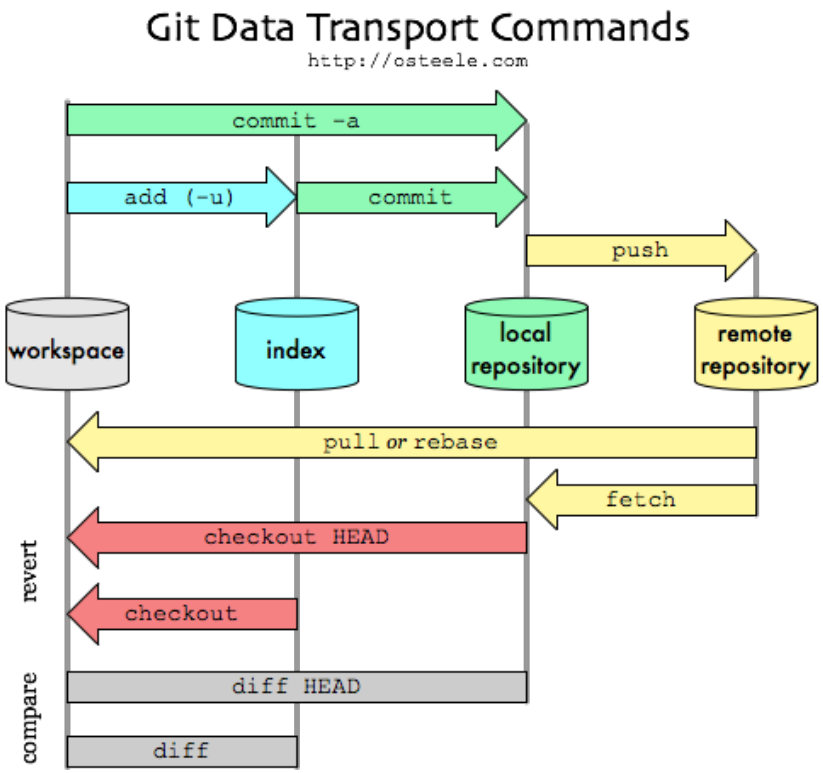
Creamos unha carpeta no escritorio que lle chamaremos 'CharlaGit'. Dentro da carpeta, executamos **git init** e debería saír:

```
breixocf@BreixoCF ~/Desktop/PruebasGit 6 $ git init
Initialized empty Git repository in /home/breixocf/Desktop/CharlaGit/.git/
```

Creouse unha carpeta .git que conterá todo o repositorio, que será a base de datos de Git. Antes de nada, imos configurar o noso repo, indicando o nome e correo do usuario con **git config** e comprobamos os cambios.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 18 $ git config --global user.name "Breixo Camiña"
breixocf@BreixoCF ~/Desktop/CharlaGit 19 $ git config --global user.email
breixocf@BreixoCF ~/Desktop/CharlaGit 21 $ git config --list
user.name=Breixo Camiña
user.email=breixo.camina@udc.es
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

Zonas de traballo de Git



- Workspace: É onde están os ficheiros que actualmente non están en seguimento, é dicir, os que están na nosa zona de traballo.
- Index: É onde están as modificacións que se engadirán ó repo local. Os ficheiros están preparados para engadir ó repositorio local, pero aínda non están no historial do proxecto.
- Local Repository: É onde están os ficheiros confirmados que conformarán unha versión nova do proxecto. Almacena os metadatos e a base de datos do proxecto. Tamén se lle chama HEAD.
- Remote Repository: Son os repositorios que estarán en internet, onde accederán varios usuarios...

Primer comando: git status

git status

Amosa o estado no que se atopan os ficheiros do repositorio.

Executando **git status** sobre o noso repositorio, debería aparecer a seguinte mensaxe:

```
breixocf@BreixoCF ~/Desktop/CharlaGit 32 $ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

Como todavía non hai ningún ficheiro en seguimento (pode haber milleiros de ficheiros na carpeta, pero non están sendo trackeados polo repositorio), dinos que usemos git add para engadir algún ficheiro a seguir.

Segundo comando: git add

git add

Engade un ficheiro da nosa zona de traballo ao índice, preparándoo para engadilo ó repositorio.

Creamos un ficheiro co comando **touch hola.txt** e executamos **git status** para ver o seu estado:

```
breixocf@BreixoCF ~/Desktop/CharlaGit 32 $ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
     hola.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Segundo comando: git add

O ficheiro non está en seguimento pese a estar na carpeta, é necesario executar o comando **git add** para iso. Así que agora executamos **git add** e indicamos o ficheiro que queremos engadir. Se queremos engadir moitos ficheiros ó mesmo tempo, executamos **git add .**

```
breixocf@BreixoCF ~/Desktop/CharlaGit 37 $ git add hola.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 38 $ git status
On branch master
Initial commit
Changes to be committed:
 (use "git rm --cached <file>..." to unstage)

new file:   hola.txt
```

Agora o ficheiro está preparado para engadir ó repo local.

Terceiro comando: git commit

git commit

Comando que serve para engadir ficheiros en seguimento (índice) ao repositorio local.

Executamos **git commit** para engadir ao repositorio o ficheiro *hola.txt*. Para iso executamos **git commit -m "mensaxe"**.

É importante que as mensaxes dos commits sexan precisos para que cando vexamos o histórico sepamos que fixemos en cada commit.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 39 $ git commit -m "Primeira subida do repositorio local"
[master (root-commit) 7e8aa0a] Primeira subida do repositorio local
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hola.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 40 $ git status
On branch master
nothing to commit, working directory clean
```

Agora o ficheiro está no repo local.

Terceiro comando: git commit

Imos crear máis ficheiros, concretamente *atalogo.txt* e *benvidos.txt*, do mesmo xeito que antes. Para comprobar os estados dos ficheiros, primeiro fago un git status, logo poño en seguimento o ficheiro atalogo.txt e volvo facer un status para que vexades como funciona:

```
breixocf@BreixoCF ~/Desktop/CharlaGit 43 $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
atalogo.txt
benvidos.txt
nothing added to commit but untracked files present (use "git add" to track)
breixocf@BreixoCF ~/Desktop/CharlaGit 44 $ git add atalogo.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 45 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
new file:   atalogo.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
benvidos.txt
```

Como podedes ver, o ficheiro *benvidos.txt* segue sen estar en seguimento pero *atalogo.txt* xa está en zona de preparación para 'commitear'.

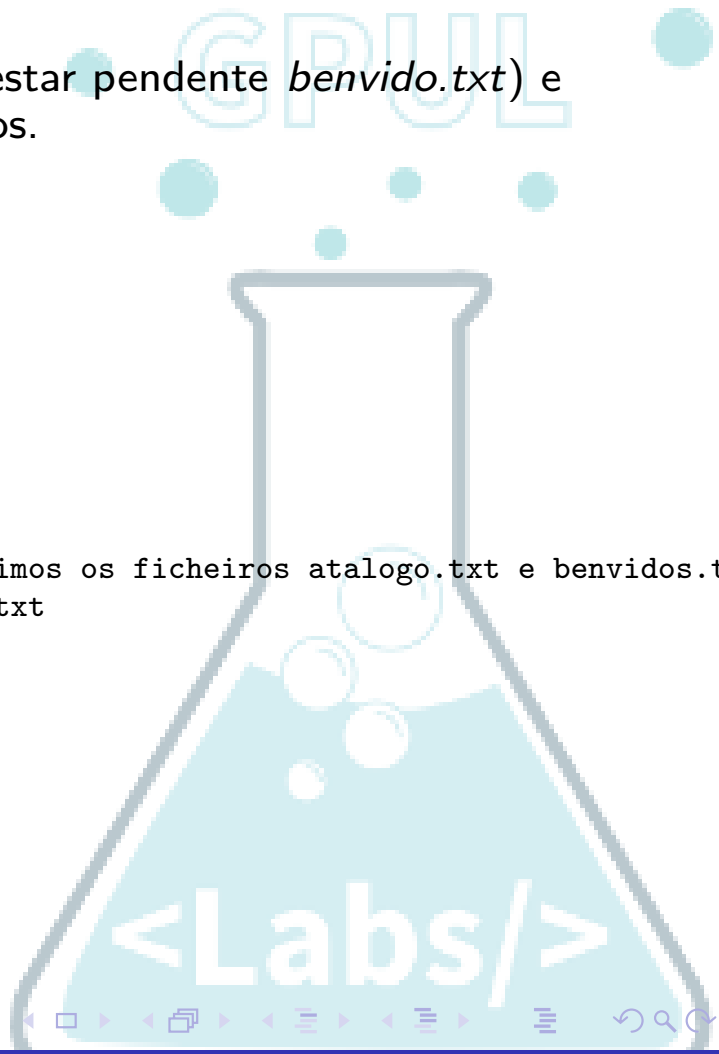
Terceiro comando: git commit

Engadimos todos os ficheiros do repositorio (so debería estar pendente *benvido.txt*) e facemos un novo commit cos dous ficheiros recién creados.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 46 $ git add .
breixocf@BreixoCF ~/Desktop/CharlaGit 47 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

new file:   atalogo.txt
new file:   benvidos.txt

breixocf@BreixoCF ~/Desktop/CharlaGit 48 $ git commit -m "Engadimos os ficheiros atalogo.txt e benvidos.tx
[master ce15277] Engadimos os ficheiros atalogo.txt e benvidos.txt
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 atalogo.txt
 create mode 100644 benvidos.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 49 $ git status
On branch master
nothing to commit, working directory clean
```



Cuarto comando: git log

git log

Este comando serve para ver o historial do repositorio.

Imos comprobar como está o noso historial de commits. Para eso executamos **git log**. Se non precisamos tanta información e queremos ver simplemente as mensaxes de cada commit executamos **git log - -oneline**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 50 $ git log
commit ce152774dcba40748cfb8f1c622484ee89e87825
Author: Breixo Camiña <breixo.camina@udc.es>
Date:   Tue Feb 9 04:41:39 2016 +0100
    Engadimos os ficheiros ...

commit 7e8aa0a63c677bc789824e494fba7549b108246d
Author: Breixo Camiña <breixo.camina@udc.es>
Date:   Tue Feb 9 04:26:02 2016 +0100
    Primeira subida do repositorio local

breixocf@BreixoCF ~ 51 $ git log --oneline
ce15277 Engadimos os ficheiros ...
7e8aa0a Primeira subida do repositorio local
```

- git log –oneline –graph –all
- git log –oneline –graph –all –decorate
- git log –oneline –since=2016-02-01
- git log –oneline –until=2016-02-14
- git log –oneline –author=”Breixo *”
- git log –oneline –grep=”hola.txt”
- git log –stat –summary

Modificando os ficheiros

Imos modificar agora os ficheiros a ver qué sucede no repositorio... Engadimos algún texto no ficheiro *hola.txt* executando **echo "Hola a todos" »hola.txt**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 52 $ echo "Hola a todos" >> hola.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 53 $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   hola.txt

no changes added to commit (use "git add" and/or "git commit -a")
breixocf@BreixoCF ~/Desktop/CharlaGit 54 $ git add .
breixocf@BreixoCF ~/Desktop/CharlaGit 55 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   hola.txt
```

Como podedes ver, o ficheiro ó estar modificado temos que volver a engadilo para pasalo da zona de traballo á zona de preparación.

Modificando os ficheiros

Engadímos o ficheiro e modificamos os outros dous do mesmo xeito. Comprobamos o estado:

```
breixocf@BreixoCF ~/Desktop/CharlaGit 56 $ echo "Benvidas/os todas/os" >> benvidos.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 57 $ echo "Marcho que teño que marchar" >> atalogo.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 58 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   hola.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   atalogo.txt
modified:   benvidos.txt
```

Agora os tres ficheiros están modificados, pero *hola.txt* está na zona de preparación listo para commitear mentres que *atalogo.txt* e *benvidos.txt* seguen na zona de traballo.

Modificando os ficheiros

Engadimos os dous ficheiros á zona de preparación, comprobamos o estado do repo, facemos commit, volvemos comprobar o estado do repo e miramos o log do historial.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 60 $ git add .
breixocf@BreixoCF ~/Desktop/CharlaGit 58 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   hola.txt
modified:   atalogo.txt
modified:   benvidos.txt

breixocf@BreixoCF ~/Desktop/CharlaGit 61 $ git commit -m "Engadimos texto nos 3 ficheiros"
[master fec23c8] Engadimos texto nos 3 ficheiros
 3 files changed, 3 insertions(+)
breixocf@BreixoCF ~/Desktop/CharlaGit 63 $ git status
On branch master
nothing to commit, working directory clean
breixocf@BreixoCF ~/Desktop/CharlaGit 62 $ git log --oneline
fec23c8 Engadimos texto nos 3 ficheiros
ce15277 Engadimos os ficheiros atalogo.txt e benvidos.txt
7e8aa0a Primeira subida do repositorio local
```

Quinto comando: git diff

git diff

Este comando serve para ver as diferencias entre dous commits, un commit e o workspace...

Se executamos **git diff** agora, non vai haber ningunha diferencia, porque o workspace contén o mesmo que o último commit. Sen embargo, se engadimos un cambio nalgún documento con **echo "Hola por segunda vez a todos" »hola.txt** e executamos **git diff** veremos o que ocorre. Se quixésemos ver os cambios dun arquivo concreto, simplemente teríamos que indicar qué arquivo sería executando **git diff hola.txt**. Para ver as diferencias entre a miña zona de preparación e o último commit, terías que executar **git diff –staged** e para ver as diferencias entre o workspace e o último commit, **git diff HEAD**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 4 $ git diff
breixocf@BreixoCF ~/Desktop/CharlaGit 5 $ echo "Hola por segunda vez a todos" >> hola.txt
breixocf@BreixoCF ~/Desktop/CharlaGit 6 $ git diff
diff --git a/hola.txt b/hola.txt
index 1218745..384efad 100644
--- a/hola.txt
+++ b/hola.txt
@@ -1 +1,2 @@
 Hola a todos, son Breixo e o animaliño con roupa que teño ó lado Fran
+Hola por segunda vez a todos
```

Sexto comando: git tag

git tag

Este comando serve etiquetar os snapshots ou releases que se quixeran facer.

Recoméndase crear etiquetas para cada versión funcional dun proxecto. Hai dous tipos de tags: simples e anotados. Un tag simple créase executando **git tag v1.0** mentres que un tag anotado crearíase executando **git tag -a v1.0 -m "Versión 1.4 do proxecto"**. Con **git tag**, amósase o listado de todos os tags do proxecto. Para mostrar info dun único tag executaríase **git show v1.0** e se quixéramos buscar por un patrón sería **git tag -l "v1.*"**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 13 $ git tag v1.0
breixocf@BreixoCF ~/Desktop/CharlaGit 14 $ git tag -a v1.1 -m "Versión 1.1"
breixocf@BreixoCF ~/Desktop/CharlaGit 15 $ git tag
v1.0
v1.1
breixocf@BreixoCF ~/Desktop/CharlaGit 17 $ git show v1.0
commit 621a67beb491ecb02da730c0fcc866d73df7d2a3
Author: Breixo Camiña <breixo.camina@udc.es>
Date:   Wed Feb 10 00:56:37 2016 +0100
    Engadida segunda liña no ficheiro hola.txt
diff --git a/hola.txt b/hola.txt
index 1218745..384efad 100644
--- a/hola.txt
+++ b/hola.txt
```

Séptimo comando: git checkout

git checkout

Este comando serve para volver a pasos anteriores en ficheiros, commits ou ramas (prox.) pero só veremos nos dous primeiros.

Executando **git checkout commit-id hola.txt**, o ficheiro hola.txt volve ó estado no que se atopaba nese commit.
Mentres que se executas **git checkout commit-id**, todos os ficheiros volverían ó estado no que estaban nese commit.

Repositorio remoto: GitHub

Existen varios repositorios remotos, como poden ser **Bitbucket de Atlassian**, **GitLab de GitLab Inc.**, ... Pero o máis extendido e o que conta coa maior comunidade é GitHub. GitHub conta con milleiros de proxectos, dende o código que estamos a subir nós agora ata o código fonte do kernel de Linux.

Repositorio remoto

Para crear o noso repositorio remoto, antes crearemos unha conta en GitHub. Logo de rexistrarnos, creamos un repositorio novo (New repository). Logo de indicarlle o nome, seguiremos os pasos que nos indican. Como xa temos o noso repositorio creado, con commits xa feitos sobre ficheiros, simplemente executaremos **git remote add origin https://github.com/BreixoCF/charlagit.git** e posteriormente faremos **git push -u origin master**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 19 $ git remote add origin
https://github.com/BreixoCF/charlagit.git
breixocf@BreixoCF ~/Desktop/CharlaGit 20 $ git push -u origin master
Username for 'https://github.com': breixocf
Password for 'https://breixocf@github.com':
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (13/13), 1.14 KiB | 0 bytes/s, done.
Total 13 (delta 2), reused 0 (delta 0)
To https://github.com/BreixoCF/charlagit.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Octavo comando: git clone

git clone

O comando git clone serve para clonar un repositorio remoto a local, executando tres pasos nun solo: git init, git remote add e git pull.

Coñecendo este comando, poderíamos executar cunha sola sentenza o clonado de repositorio da seguinte forma: **git clone https://github.com/BreixoCF/charlagit.git**.

```
~ % git clone https://github.com/BreixoCF/charlagit.git
Cloning into 'charlagit'...
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 16 (delta 3), reused 16 (delta 3), pack-reused 0
Unpacking objects: 100% (16/16), done.
Checking connectivity... done.
```

Agora copiaremos este código dunha pila en Python nun ficheiro que chamaremos **stack.py**:

```
# Implementación de unha pila facendo uso de programacion orientada a obxectos.
class Stack:
    def __init__(self):
        self.items = []

    def vacia(self):
        return self.items == []

    def apilar(self, item):
        self.items.append(item)

    def cima(self):
        if self.vacia():
            return []
        return self.items[-1]

    def desapilar(self):
        del self.items[-1]

    def tam(self):
        return len(self.items)
```

Agora imos a subir o ficheiro ao repositorio de **GitHub** que acabamos de crear. Para eso, engadimos o ficheiro *stack.py* á zona de preparación (Index) e posteriormente ao repositorio local (HEAD).

```
breixocf@BreixoCF ~/Desktop/CharlaGit 21 $ git add stack.py
breixocf@BreixoCF ~/Desktop/CharlaGit 22 $ git commit -m "Engadida a pila OO stack.py"
[master 111365e] Engadida a pila OO stack.py
 1 file changed, 21 insertions(+)
 create mode 100644 stack.py
```

¿Xa estaría subido ao repositorio remoto?

NON

Noveno comando: git push

git push

Este comando serve para subir os cambios do repositorio local (HEAD) ao repositorio remoto.

Se executamos **git push origin master**, podremos comprobar na web de github que o ficheiro está engadido correctamente.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 21 $ git add stack.py
breixocf@BreixoCF ~/Desktop/CharlaGit 22 $ git commit -m "Engadida a pila OO stack.py"
[master 111365e] Engadida a pila OO stack.py
 1 file changed, 21 insertions(+)
 create mode 100644 stack.py
breixocf@BreixoCF ~/Desktop/CharlaGit 23 $ git push origin master
Username for 'https://github.com': breixocf
Password for 'https://breixocf@github.com':
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 489 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/BreixoCF/charlagit.git
   621a67b..111365e  master -> master
```

Décimo comando: git fetch

git fetch

Este comando serve para actualizar o repositorio local respecto o repositorio remoto indicado.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 27 $ git fetch origin master
From https://github.com/BreixoCF/charlagit
 * branch            master     -> FETCH_HEAD
```

Como non hai cambios nos ficheiros non pasaría nada.

Undécimo comando: git pull

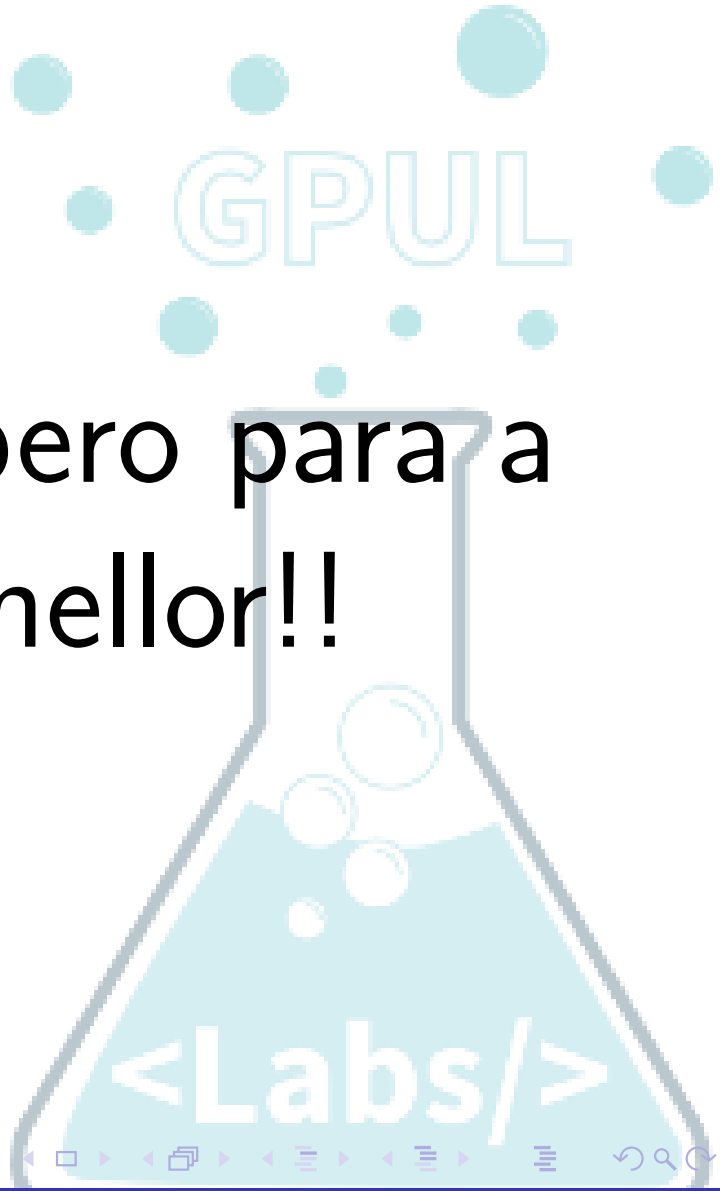
git pull

Este comando serve para descargar o commit máis novo do repositorio remoto á nosa zona de traballo (workspace).

Se estivéramos dous ou máis usuarios traballando sobre o mismo repositorio remoto, poderíamos descargarnos os cambios que fixo o outro usuario executando **git pull origin master**.

```
breixocf@BreixoCF ~/Desktop/CharlaGit 24 $ git pull origin master
From https://github.com/BreixoCF/charlagit
  * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Como non hai cambios nos ficheiros non pasaría nada.



Ata aquí por hoxe, pero para a semana máis e mellor!!