Python 3: tutorial argparse

En el tutorial anterior sobre registro de eventos con "**logging**" utilizamos a "**argparse**" para permitir a nuestros usuarios y usarias a establecer un nivel de registro de eventos en caso de ser necesario hacer seguimiento a nuestra aplicación. Prometimos allí ahondar con un tutorial completo sobre el tema y aquí lo prometido, ¡estudiemos juntos!

Introducción.

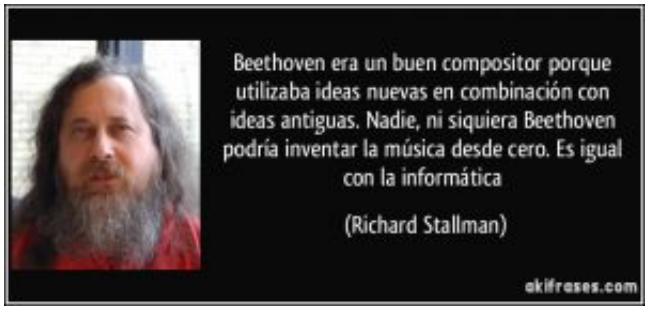
Para no caer en la redundacia, os recomendamos leer nuestro trabajo anterior, la sección "aislada" sobre **argparse**. Allí colocamos como ejemplo el comando **ls**;en realidad casi todas las aplicaciones que corren sobre la línea de comando aceptan argumentos a la derecha conformando parte de la orden de ejecución al presionar la tecla **enter** o **intro**.

Como dijimos en nuestro anterior tutorial sobre el registro de eventos en nuestras aplicaciones, **argparse** proviene de **optparse** el cual a su vez fue "descontinuada" desde la versión Python 2.7 ¿Por qué entrecomillamos? Lo hacemos porque como es software libre cualquiera puede hacer una bifurcación "fork" y continuar desarrollandolo a su gusto y conveniencia. De hecho se lleva un desarrollo en paralelo en GithHub como adelante veremos.

Lo bueno del asunto es que son bastantes similares en cuanto a su sintaxis y esto es así para facilitar a los desarrolladores que usaron **optparse** y ahora necesitan la migración a **argparse.**

Código previo a argparse.

Pero antes de entrar de lleno en **argparse** y como éste es un tutorial dedicado a dicha librería, vamos a ir un poco más allá: las bases sobre las cuales funciona **argparse**. Ya bien lo dice Richard Stallman, padre del software libre: "Nadie, ni siquiera Beethoven podría inventar la música desde cero. Es igual con la informática".



He aquí que una de las librerías básicas en el entorno de programación **Python** lo es **sys**. Para agregarlo a nuestros programas debemos enlazarlo con el comando **import sys** y podremos

comenzar a usar sus objetos, los cuales no estudiaremos completamente en este tutorial sino que vamos a centrarnos en uno de sus componentes: **sys.argv**. Por medio de éste podremos acceder a la cadena de texto completa con la que el usuario o usuaria haya invocado nuestra aplicación por medio de la línea de comando. De una vez vamos a la práctica, tras esta muy breve teoría:

```
import sys  print("Número de argumentos: ", len(sys.argv))  print("Los ar
gumentos son : ", str(sys.argv))
```

Explicación: **sys.argv** es, simplemente, una lista con cada palabra (entendiendose como palabra cualquier cadena de texto delimitada por al menos un espacio) con la que se invoca el guion "script" de nuestro, o de cualquier, programa.

- La primera línea "enlaza" con la librería **sys**, permite cargarla en memoria y nos permite acceder a sus métodos , eventos y constantes.
- La segunda línea usa la función **len()** que obtiene el largo de la lista, osea, el número de elementos -léase palabras-con la que se invocó nuestro guion "script".
- La tercera línea muestra por pantalla todos y cada uno de los elementos de la lista especial.

Lo más curioso del asunto es que podemos no solamente acceder a la lista sino que también podemos cambiar sus valores, ¡probad! Lo que si es cierto es que el primer elemento (elemento cero) será siempre el nombre del fichero que almacena el guion escrito en python, con todo y extensión (aunque si no tuviera extensión **.py** igual se ejecuta) y los demás elementos de la lista son los argumentos o parámetros ya sea que lo escriba el usuario o le sea pasado al programa por el comando tubería "pipe" o "|" o en una variable en un guion "script" BASH.

Instalando argparse en nuestros ordenadores.

Para eliminar todo tipo de dudas, usamos Python versión 3.X -ya lo hemos dicho en nuestras entradas anteriores, revisad- y probablemente ya tengáis instalado **argparse** en vuestro ordenador. Al usar el comando **import argparse** y de no estar instalado de inmediato sale el mensaje de error en Python por lo que podemos instalarlo de diferentes maneras.

Por medio de pip3.

Para instalar **argparse** por medio de **pip3** debemos escribir **pip install argparse** con los debidos derechos de administrador y así poder descargarlo de internet. Explicamos: **pip3** es un esfuerzo en reunir en un repositorio de aplicaciones oficial de muchos software hecho por terceros pero que son supervisadas de manera directa por el equipo desarrollador de Python. Para saber si tenemos instalado **pip3** simplemente escribimos **pip3 --version** y mostrará la versión instalada (ah, y de

paso mirad otro ejemplo de argumentos en una aplicación **"--version**") y dado el caso que no la tengamos instalada podremos usar:

```
sudo apt-get install python3-setuptools  sudo easy_install3 pip
```

Por medio de su código fuente.

En GitHub hallaremos el repositorio de Thomas Waldmann quien claramente advierte que el desarrollo de **argparse** es almacenado oficialmente por el equipo de desarrollo de Python *pero que él mantiene una copia para quienes tengan Python 2.X y quieran agregar **argparse** a sus aplicaciones.* De tal manera que si vosotros no lo tenéis instalado y no queréis -o no podéis- usar **pip3** pues clonad el proyecto y ejecutad **setup.py**

Para los que les gusta la "arqueología" de software en Google podéis deleitaros en el siguiente enlace (tal parece que años atrás estaba alojado por aquellos lares antes de ser migrado el código fuente de **argparse** a la Fundación Python).

Observación importante:

Si sois como nosotros que tenemos instalado tanto Python 2 como Python 3 os damos el siguiente dato: si abrís un guion o programa con Python 2 y usáis **argparse** se generará un archivo precompilado ".pyc" cuya finalidad es cargar más rápidamente nuestro programa en sucesivos llamados. Luego si abrís el mismo "script" con Python 3 obetendréis un mensaje de error más o menos indicando "**error en magic number**". Lo que debéis hacer es simplemente borrar todos los archivos "*.pyc" -por si las dudas- que en cuanto se vuelvan a ejecutar se generarán de nuevo. Advertidos quedáis ? .

Primeros pasos con argparse.

Tan solo debemos escribir nuestro guion de la siguiente manera:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.parse_args()
```

Primero llamamos a la librería y luego a un ubjeto **analizador** le asociamos con nuestra descripción de aplicación. Luego le ordenamos que muestre por pantalla los argumentos recibidos desde la línea de comandos. *Así de sencillo -y esa es la idea- con tres simples líneas tenemos acceso al pase de parámetros y opciones por la línea de comando.* Al llamar sin más el guión no

veremos ningún resultado -aparente-. Pero si ejecutamos lo siguiente:

```
python3 tutorial_argparser.py -a
```

Veremos la siguiente respuesta:

```
usage: tutorial_argparser.py [-h]  tutorial_argparser.py: error: unrecogn
ized arguments: -a
```

*Nota importante: nosotros guardamos el programa en un fichero llamado **tutorial_arparse.py** y tal vez se sientan tentados a no escribir tanto y nombrarlo simplemente **argparse.py** ¡No lo hagáis! Sucederá que al ejecutar el guion se llamará a si mismo primero antes que buscarlo en las librerías Python.* Este es el comportamiento predeterminado para nosotros cargar nuestras propias librerías: toda "importación" buscará primero en la carpeta donde está guardado el guion. Ya sabéis entonces.

Como véis ya **argparse** está trabajando para nosotros. La primera línea con el encabezado "**usage:**" indica los argumentos válidos -en este caso *opcionales* ya que está encerrados entre corchetes- y vemos que tiene la opción "-h". La segunda línea nos indica que ha sucedido un error en el archivo **tutorial_argparser.py** e indicando que es un argumento no reconocido lo que le acabamos de escribir: "**-a**".

Lo que tenemos que experimentar ahora es precisamente "correr" el programa con el argumento "**-h**" y como probablemente ya sabéis ése es precisamente la orden para solicitar ayuda, veamos:

```
python3 tutorial_argparser.py -h
```

Obtendremos el siguiente mensaje:

```
usage: tutorial_argparser.py [-h]    Tutorial sobre argparse.    optional
 arguments:    -h, --help  show this help message and exit
```

De nuevo la primera línea nos muestra los argumentos disponibles. La segunda ofrece la descripción de nuestro programa, la que le indicamos al inicializar la librería. La tercera línea

(obviamente las líneas en blanco no la numeramos por propósitos didácticos) nos indica lo que hace el argumento solicitado: muestra el mensaje de ayuda y sale sin ejecutar ningún otro código. Notad que incluso nos muestra una opción "larga" del argumento de ayuda: "**--help**". El siguiente paso es agregar nuestro primer argumento, veamos.

Agregando nuestro primer argumento a nuestro programa.

Argumento opcional:

Para que un argumento se opcional debemos antecederlo de un guion "-"; modifiquemos nuestro fichero de la siguiente manera:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument("-a", help="Detecta y confi
rma nuestro primer argumento.")  argumento = analizador.parse_args()  if
argumento.a:    print("Argumento opcional solicitado: -a")    print("Argu
mento acompañado de:"+argumento.a)  else:    print("Ningún argumento.")
```

Allí con el método "**.add_argument**" establecimos la palabra clave "**-a**" y colocamos una breve descripción que será mostrada al solicitar ayuda con la consabida "**-h**" o "**--help**" la cual de una buena vez pedimos y obtenemos:

```
usage: tutorial_argparser.py [-h] [-a A]    Tutorial sobre argparse.    o
ptional arguments:    -h, --help  show this help message and exit    -a A
        Detecta y confirma nuestro primer argumento
```

Ahora empezamos a probar el nuevo argumento, se lo pasamos a la aplicación con el comando

```
$ python3 tutorial_argparser.py -a
```

y gentilmente nos advertirá que se necesita un argumento para la opción "**-a**", es decir, será opcional, pero una vez que lo llamamos debemos acompañarlo de una cadena de texto, mirad:

```
usage: tutorial_argparser.py [-h] [-a A]  tutorial_argparser.py: error: a
rgument -a: expected one argument
```

De nuevo metemos

python3 tutorial_argparser.py -a ¡Hola!

y veremos justo lo que le ordenamos hacer:

```
Argumento opcional solicitado: -a  Argumento acompañado de:¡Hola!
```

Es hora de acompañar el argumento "**-a**" de una opción larga, nemotécnica, así que establecemos que sea "**--aviso**": ya uséis uno u otro el comportamiento será el mismo.

```
analizador.add_argument("-a", "--aviso", help="Detecta y confirma nuestro
 primer argumento.")  argumento = analizador.parse_args()  if argumento.a
viso:    print("Argumento opcional solicitado: --aviso")    print("Argume
nto acompañado de:"+argumento.aviso)  else:    print("Ningún argumento.")
```

Notad que tuvimos que cambiar el método "**.a**" por "**.aviso**". También debemos agregar un entrecomillado si la frase que queremos pasar contiene varias palabras, *de lo contrario **argparse** los interpretará como si fueran varios argumentos diferentes unos de otros:*

```
$
python3 tutorial_argparser
.py -a "¡Hola mundo!"
    Argumento opcional solicitado: --aviso  Argumento acompañado de:¡Hola
 mundo!
```

Debemos acotar que, por defecto, **argparse** espera que sean cadenas de texto los argumentos que le pasemos *a menos que le indiquemos expresamente lo contrario.* Si necesitaramos pasar algún valor numérico, y que sea interpretado como tal, debemos agregar la opción **type=int** en donde definimos el argumento. Para darle utilidad esto último, cambiamos para que muestre repetidamente tantas veces como indique el número que pasemos, mirad atentamente:

```
mport argparse  analizador = argparse.ArgumentParser(description='Tutoria
l sobre argparse.')  analizador.add_argument(    "-a",    "--aviso",    h
elp="Detecta y confirma nuestro primer argumento",    type=int  )  argume
nto = analizador.parse_args()  if argumento.aviso:    print("Argumento op
```

```
cional solicitado: --aviso")    for x in range(0, argumento.aviso):
print("Argumento acompañado de:"+str(argumento.aviso))  else:    print("N
ingún argumento solicitado")
```

Los cambios que hicimos implican usar la función **str()** que convierte la variable de tipo entero numérico a cadena de texto para poder usar el ciclo "**for():**", así imprimirá el mensaje tantas veces como sea solicitado.

> La isntrucción "type=" es poderosa, de hecho puede albergar cualquier tipo de variable, objeto ¡e incluso una función! Por ser tan avanzada por ahora **no** la estudiaremos en profundidad.

Ahora vamos a ver *argumentos necesarios* para ejecutar nuestro guion.

Argumento obligatorio.

Muchas aplicaciones precisan de un argumento obligatorio, por ejemplo, si está diseñada para analizar y trabajar con el contenido de un fichero *pues es necesario indicarle que se debe pasar un nombre de archivo.* Para ello modificaremos de nuevo de esta manera:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "archivo",    help="Ind
ica el nombre del fichero a trabajar.",  )  argumento = analizador.parse_
args()  if argumento.archivo="archivo":    print("Argumento OBLIGATORIO s
olicitado: archivo")    print("Nombre del archivo:"+argumento.archivo)
```

Si corremos sin parámetro alguno nos indicará que DEBEMOS indicar un nombre de fichero; si lo agregamos veremos esto:

```
$
python3                    tutorial_argparser.py                    list
a.txt
    Argumento OBLIGATORIO solicitado: archivo  Nombre del archivo:lista.t
```

```
xt
```

Argumento obligatorio repetido *n* veces ("nargs=*n*").

Muchas veces una aplicación necesita un archivo origen de donde sacar datos, procesarlos y verter la respuestra en otro archivo: para ello podemos utilizar el siguiente código:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "nom_arch",    nargs=2,
    help="Indica el nombre de los ficheros origen y destino a trabajar.",
  )  argumento = analizador.parse_args()  if argumento.nom_arch:    print
("Argumento OBLIGATORIO solicitado: nom_arch")    print("Nombres de los a
rchivos:")    print(argumento.nom_arch[0])    print(argumento.nom_arch[1]
)
```

Observad la línea **nargs=2**: le estamos indicando que necesita dos argumentos (o los que necesitemos) , la desventaja de este método es que al usuario colocar un solo argumento **argparse** emite un mensaje que puede ser confuso, ***no es un mensaje explícito (recordad las reglas de oro de Python: explícito es mejor que implícito)****, es decir:*

```
$ python3 tutorial_argparser.py arch1
  usage: tutorial_argparser.py [-h] nom_arch nom_arch  tutorial_argparser
.py: error: the following arguments are required: nom_arch
```

Como véis repite lo mismo *n veces cuando la cantidad de argumentos NO coincide con* ***nargs****.* La ventaja acá es que codificamos menos porque no tenemos que incluir dos parámetros con diferentes nombres *pero dejemos aparte la flojera, seamos explícitos:*

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "nom_arch_orig",    hel
p="Indica el nombre del archivo origen.",  )  analizador.add_argument(
 "nom_arch_dest",    help="Indica el nombre del archivo destino.",  )  ar
gumento = analizador.parse_args()  if argumento.nom_arch_orig:    print("
Argumentos OBLIGATORIOS solicitados: nom_arch_orig y nom_arch_dest")    p
rint("Nombres de los archivos:")    print(argumento.nom_arch_orig)    pri
nt(argumento.nom_arch_dest)
```

Así es menos confuso para nuestros usuarios y usuarias:

```
$ python3 tutorial_argparser.py
  usage: tutorial_argparser.py [-h] nom_arch_orig nom_arch_dest  tutorial
_argparser.py: error: the following arguments are required: nom_arch_orig
,  nom_arch_dest $
python3 tutorial
_argparser.py arch_orig.txt arch
_dest.txt
  Argumentos OBLIGATORIOS solicitados: nom_arch_orig y nom_arch_dest  Nom
bres de los archivos:  arch_orig.txt  arch_dest.txt
```

¿En cuales condiciones nos es útil **nargs** en modo múltiple? Ahora no viene nada a la cabeza pero alguna utilidad de seguro tendrá.

Ningún argumento, uno o más argumentos (" nargs='*' ").

Por otro lado, así como **nargs** especifica un número exacto de argumentos, también permite el caracter asterisco que funciona a modo de comodín: *puede aceptar uno, dos o más argumentos -**o ninguno***-. Escribamos este código:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(  "nombres",  nargs="*"
,  help="Recibe una lista de nombres de personas.",  )  argumento = ana
lizador.parse_args()  if argumento.nombres:  print("Argumento(s) OBLIGA
TORIO(S) solicitado(s): nombres")  print("Nombres de las personas:")
 print(argumento.nombres)
```

Y probemos su salida:

```
$ python3 tutorial_argparser.py $
python3 tutorial_argparser.py José
  Argumento(s) solicitado(s): nombres  Nombres de las personas:  ['José']
$
python3 tutorial_arg
parser.py José María Pedro Carmen
  Argumento(s) solicitado(s): nombres  Nombres de las personas:  ['José',
 'María', 'Pedro', 'Carmen']
```

En la primera línea del terminal notamos que no necesita argumento alguno para funcionar, eso sería "cero o más". Avizorad que si necesitamos *por lo menos una persona en la lista* podemos utilizar el signo de suma "+" en vez del asterisco ("uno o más"), y al sustituirlo y ejecutar el programa veremos lo siguiente:

```
$ python3 tutorial_argparser.py
  usage: tutorial_argparser.py [-h] nombres [nombres ...]  tutorial_argpa
rser.py: error: the following arguments are required: nombres
```

Así nos dice que "nombres" necesita al menos uno (no está entre corchetes, es obligatorio) y que podemos agregar otros nombres de personas, esto lo indica entre corchetes y con tres puntos suspensivos.

Un argumento no obligatorio ya que utiliza un valor por defecto (" nargs=*'?'* ").

En este caso se utiliza **nargs="?"** en combinación de un valor por defecto **default=' *cadena_de_texto'*** por lo que esta opción es un tanto extraña no es obligatoria ya que si no se le pasa un valor toma el que por defecto le pongamos, este ejemplo ilustra muy bien lo que decimos:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "nombre",    nargs="?",
    default='Jesús',    help="Recibe un nombre de persona y si no es sumi
nistrado utiliza 'Jesús'.",  )  argumento = analizador.parse_args()  if a
rgumento.nombre:    print("Argumento: solicita un nombre (por defecto uti
liza 'Jesús')")    print("Nombre:")    print(argumento.nombre)
```

Ahora bien, al ejecutarlo fijáos bien en lo que hace:

```
$ python3 tutorial_argparser.py
  Argumento: solicita un nombre (por defecto utiliza 'Jesús')  Nombre:  J
esús  $
python3 tutorial_argparser.py Pedro
  Argumento: solicita un nombre (por defecto utiliza 'Jesús')  Nombre:  P
edro
```

Esta opción es tremendamente útil si le pedimos a la usuaria que indique un archivo de origen *y, si lo desea, un archivo destino.* De no colocar un archivo destino entonces utilizará el nombre de archivo que nosotros mismo escojamos (y si ese archivo existe bien le podemos agregar datos al final o creamos un archivo nuevo con el nombre por defecto acompañado de un número que esté libre: *arch1, arch2, ... arch_n*). Colocamos el código necesario para enseñaros claramente la opción ***nargs="?"***:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "arch_orig",    help="S
olicita un nombre de archivo de origen.",  )  analizador.add_argument(
 "arch_dest",    nargs="?",    default='arch_dest.txt',    help="Solicita
 un nombre de archivo destino, si se omite utiliza 'arch_dest.txt'.",  )
 argumento = analizador.parse_args()  print("Argumentos: archivo de orige
n y destino ('arch_det.txt' si se omite destino)")  print("Nombres de arc
hivos:")  print(argumento.arch_orig)  print(argumento.arch_dest)
```

Y esta sería la salida:

```
$                      python3                      tutorial_argparser.py
  usage: tutorial_argparser.py [-h] arch_orig [arch_dest]  tutorial_argpa
rser.p
y:                                 error:                                th
e                 following                 arguments                 ar
e             required:             arch_orig                          $
python3                   tutorial_argparser.py                    list
a.txt
  Argumentos: archivo de origen y destino ('arch_det.txt' si se omite des
tino)                                      Nombres                    d
e         archivos:                  lista.txt                 arch_des
t.txt                                                                   $
python3                                                          tutoria
l_argparser.py                     lista.txt                       lista
_ordenada.txt
  Argumentos: archivo de origen y destino ('arch_det.txt' si se omite des
tino)  Nombres de archivos:  lista.txt  lista_ordenada.txt
```

Como abreboca al estudio avanzado de **argparse** colocamos el siguiente ejemplo, muy sencillo pero que ilustra hasta donde podemos llegar combinando opciones:

```
import argparse import os
  analizador = argparse.ArgumentParser(description='Tutorial sobre argpar
se.')  analizador.add_argument(    "arch_orig",    help="Solicita un nomb
re de archivo de origen.",  )  analizador.add_argument(    "arch_dest",
           nargs="?",                        default=     os.getcwd()+'/
arch_dest.txt',    help="Solicita un nombre de archivo destino, si se omi
te utiliza 'arch_dest.txt'.",  )
```

En color verde resaltamos la añadidura: primero tenemos que importar la librería **os**. Uno de los métodos es **os.getcwd()** la cual devuelve la ruta donde está almacenado nuestro guion, ¡probad vosotros! Es la manera de aprender.

Argumento opcional convertido en obligatorio.

Volviendo a nuestro ejemplo del argumento "**-a**" o"**--aviso**" (¿recordáis arriba?) lo podemos convertir en obligatorio adicionando un parámetro a la declaración del argumento, lo resaltamos en color verde:

```
analizador.add_argumen
t(    "-a",    "--aviso",    required=True,
    help="Detecta y confirma nuestro primer argumento."  )
```

*Atención: el parámetro **required NO** es compatible con* **nargs="*"** *ni con* **nargs="?"**.

A medida que avanzamos se torna compleja nuestra aplicación, *nuestra recomendación es transcribir y ejecutar, experimentar cada una de las diferentes combinaciones y una vez las tengamos comprendidas y bajo control avanzamos al siguiente nivel más complejo aún.*

Argumento opcional con valor por defecto.

Ahora veremos que un argumento opcional le podemos dar un valor por defecto y así lo invoquemos sin ningún tipo de argumento utilice dicho valor prefijado. Además, si el usuario desea introducir algún valor deberá colocar la palabra clave acompañada de un tipo de valor por nosotros especificado (texto, entero, etc.). **En este punto nos vamos acercando a la manera de como normalmente se comportan las aplicaciones más comunes, es decir, un comportamiento bastante común;** acá la codificación de ejemplo:

```
import argparse  analizador = argparse.ArgumentParser(".:|Tutorial sobre
```

```
argparser|:.")  analizador.add_argument(    '--limite',     default=3,
  type=int,     help="Especifique el número máximo de elementos, (por def
ecto 3).")  argumento = analizador.parse_args()  print("Límite: {}".forma
t(argumento.limite))
```

Si

```
$     python3     tutorial_argparser.py          Límite:     3          $
python3                    tutorial_argparser.py                     --help
  usage: .:|Tutorial sobre argparser|:. [-h] [--limite LIMITE]    optiona
l arguments:    -h, --help       show this help message and exit    --lim
ite LIMITE  Especifique el número máximo de elementos, (por defecto 3).
                                                                    3.       $
python3 tutorial_argparser.py --limite 17  Límite: 17
```

Argumento obligatorio y que exige escoger de una lista de opciones.

Muchas veces necesitamos que un usuario escoja un solo valor de una lista de opciones. Por ejemplo, solicitamos qyue escoja un mes de inicio de trimestre, el código sería el siguiente:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  analizador.add_argument(    "mes",    choices=['Ene
ro','Abril','Julio','Octubre'],    help="Permite escoger un mes de comien
zo de trimestre.",  )  argumento = analizador.parse_args()  print("Argume
nto: solicita un mes de una lista predeterminada.")  print("Mes escogido:
")  print(argumento.mes)
```

Y cuando lo ejecutamos:

```
$ python3 tutorial_argparser.py
  usage: tutorial_argparser.py [-h] {Enero,Abril,Julio,Octubre}  tutorial
_argpa
rser.py: err
or: the following argume
nts are required: mes  $
python3 tutorial_argparser.py Junio
  usage: tutorial_argparser.py [-h] {Enero,Abril,Julio,Octubre}  tutorial
```

```
_argparser.py: error: argument mes: invalid choice: 'Junio' (choose from
'Enero', 'Abril', 'Julio', 'Octu
bre')  $
python3 tutorial_argparser.py Julio
  Argumento: solicita un mes de una lista predeterminada.  Mes escogido:
 Julio
```

Como ven, ¡tremendamente útil!

Un breve receso antes de continuar con...

https://twitter.com/ossia/status/565907210497040384

Argumentos opcionales mutuamente excluyentes.

Como ya estamos prácticos con **argparse** (o deberíamos, sino retroceded y repasad) vamos a **abstraernos** un poco. Imaginemos que poseemos una impresora 3D, es decir, una 'impresora' capaz de producir objetos físicos tangibles. Nuestro programa será capaz de 'imprimir' bien sea un cubo, bien sea una esfera ***pero no ambos al mismo tiempo***. Para ello codificamos de la siguiente forma y manera:

```
import argparse  analizador = argparse.ArgumentParser(description='Tutori
al sobre argparse.')  grupo = analizador.add_mutually_exclusive_group()
grupo.add_argument(    "-c",    "--cubo",    action = "store_true",    he
lp="Imprime un cubo en tercera dimensión.",  )  grupo.add_argument(    "-
e",    "--esfera",    action = "store_true"
,    help="Imprime una esfera en tercera dimensión.",  )  argumento = ana
lizador.parse_args()  if argumento.cubo:    print("'Imprime' un cubo")  i
f argumento.esfera:    print("'Imprime' una esfera")
```

En este caso, como los parámetros son opcionales si no "pasamos" nada pues nada hace. Pero si empleamos **--cubo** o **--esfera** dará como resutlado lo correspondiente, *pero si usamos las dos al mismo tiempo nos indicará que escojamos solo una de ellas.*

Si sois avezados notando los detalles, veréis lo coloreado en verde: un parámetro nuevo llamado **action**. Por increíble que les parezca, en realidad ya lo estuvimos usando desde hace rato: lo empleamos para saber si un parámetro opcional ha sido "pasado" a la aplicación, nuestro primer ejemplo hace uso de ello. La diferencia estriba que en aquel ejemplo debíamos acompañar de una

cadena de texto *y en este caso solo nos interesa el parámetro en si.* Es decir, lo que nos interesa es si especificaron **cubo** o **esfera** y que lo represente como una variable booleana. Para que almacene un valor verdadero le asignamos **store_true** y si es un valor falso pues **store_false**. Esto último es un poco liado *¿para qué diantres necesitamos un valor falso?*

Fuentes consultadas:

En idioma inglés:

- "optparse — Parser for command line options".
- "The argparse module is now part of the Python standard library!"
- "Argparse Tutorial" by Tshepang Lekhonkhobe.
- "Python Argparse Cookbook".
- "How to use sys.argv in Python".
- "argparse – Command line option and argument parsing".
- "sys-Modul", Python Course.

YouTube:

https://www.youtube.com/watch?v=q94B9n_2nf0