Lectura y escritura de ficheros de textos

Función open()

La función open() se utiliza normalmente con dos parámetros (fichero con el que vamos a trabajar y modo de acceso) y nos devuelve un objeto de tipo fichero.

```
>>> f = open("ejemplo.txt","w")

>>> type(f)

<class '_io.TextIOWrapper'>

>>> f.close()
```

Modos de acceso

Los modos que podemos indicar son los siguientes:

Añadido en modo binario. Crea si éste no existe

Modo	Comportamiento	Puntero
r	Solo lectura	Al inicio del archivo
rb	Solo lectura en modo binario	
r+	Lectura y escritura	Al inicio del archivo
rb+	Lectura y escritura binario	Al inicio del archivo
w	Solo escritura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb	Solo escritura en modo binario. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
w+	Escritura y lectura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb+	Escritura y lectura binaria. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
a	Añadido (agregar contenido). Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al

		comienzo.
ab		Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
a+	Añadido y lectura. Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
ab+	Añadido y lectura en binario. Crea el archivo si no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.

Como podemos comprobar podemos trabajar con ficheros binarios y con ficheros de textos.

Codificación de caracteres

Si trabajamos con fichero de textos podemos indicar también el parámetro encoding que será la codificación de caracteres utilizadas al trabajar con el fichero, por defecto se usa la indicada en el sistema:

```
>>> import locale

>>> locale.getpreferredencoding()

'UTF-8'
```

Y por último también podemos indicar el parámetro errores que controla el comportamiento cuando se encuentra con algún error al codificar o decodificar caracteres.

Objeto fichero

Al abrir un fichero con un determinado modo de acceso con la función open() se nos devuelve un objeto fichero. El fichero abierto siempre hay que cerrarlo con el método close():

```
>>> f = open("ejemplo.txt","w")

>>> type(f)

<class '_io.TextIOWrapper'>

>>> f.close()
```

Se pueden acceder a las siguientes propiedades del objeto file:

- closed: retorna True si el archivo se ha cerrado. De lo contrario, False.
- mode: retorna el modo de apertura.

- name: retorna el nombre del archivo
- encoding: retorna la codificación de caracteres de un archivo de texto

Podemos abrirlo y cerrarlo en la misma instrucción con la siguiente estructura:

```
>>> with open("ejemplo.txt", "r") as archivo:
...    contenido = archivo.read()
>>> archivo.closed
True
```

Métodos principales

Métodos de lectura

```
>>> f = open("ejemplo.txt","r")
>>> f.read()
'Hola que tal\n'

>>> f = open("ejemplo.txt","r")
>>> f.read(4)
'Hola'
>>> f.read(4)
' que'
>>> f.tell()
8
>>> f.seek(0)
>>> f.read()
'Hola que tal\n'

>>> f = open("ejemplo2.txt","r")
>>> f.readline()
'Línea 1\n'
```

```
>>> f.readline()
'Línea 2\n'
>>> f.seek(0)
0
>>> f.readlines()
['Línea 1\n', 'Línea 2\n']
```

Métodos de escritura

```
>>> f = open("ejemplo3.txt","w")
>>> f.write("Prueba 1\n")
9
>>> print("Prueba 2\n",file=f)
>>> f.writelines(["Prueba 3","Prueba 4"])
>>> f.close()
>>> f = open("ejemplo3.txt","r")
>>> f.read()
'Prueba 1\nPrueba 2\n\nPrueba 3Prueba 4'
```

Recorrido de ficheros

```
>>> with open("ejemplo3.txt","r") as fichero:
...    for linea in fichero:
...        print(linea)
```

Gestionar ficheros CSV

Módulo CSV

El módulo cvs nos permite trabajar con ficheros CSV. Un fichero CSV (comma-separated values) es un tipo de documento en formato abierto sencillo para representar datos en

forma de tabla, en las que las columnas se separan por comas (o por otro carácter).

Leer ficheros CSV

Para leer un fichero CSV utilizamos la función reader():

```
>>> import csv

>>> fichero = open("ejemplo1.csv")

>>> contenido = csv.reader(fichero)

>>> list(contenido)

[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'], ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'], ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'], ['4/10/2015 2:40', 'Strawberries', '98']]

>>> list(contenido)

[]

>>> fichero.close()
```

Podemos guardar la lista obtenida en una variable y acceder a ella indicando fila y columna.

```
...

>>> datos = list(contenido)

>>> datos[0][0]

'4/5/2015 13:34'

>>> datos[1][1]

'Cherries'

>>> datos[2][2]

'14'
```

Por supuesto podemos recorrer el resultado:

```
...

>>> for row in contenido:

        print("Fila "+str(contenido.line_num)+" "+str(row))
```

Fila 1 ['4/5/2015 13:34', 'Apples', '73']

Fila 2 ['4/5/2015 3:41', 'Cherries', '85']

Fila 3 ['4/6/2015 12:46', 'Pears', '14']

Fila 4 ['4/8/2015 8:59', 'Oranges', '52']

Fila 5 ['4/10/2015 2:07', 'Apples', '152']

Fila 6 ['4/10/2015 18:10', 'Bananas', '23']

Fila 7 ['4/10/2015 2:40', 'Strawberries', '98']

Veamos otro ejemplo un poco más complejo:

```
>>> import csv
>>> fichero = open("ejemplo2.csv")
>>> contenido = csv.reader(fichero,quotechar='"')
>>> for row in contenido:
...   print(row)
...
['Año', 'Marca', 'Modelo', 'Descripción', 'Precio']
['1997', 'Ford', 'E350', 'ac, abs, moon', '3000.00']
['1999', 'Chevy', 'Venture "Extended Edition"', '', '4900.00']
['1999', 'Chevy', 'Venture "Extended Edition, Very Large"', '', '5000.00']
['1996', 'Jeep', 'Grand Cherokee', 'MUST SELL!\nair, moon roof, loaded', '4799.00']
```

Escribir ficheros CSV

```
>>> import csv
>>> fichero = open("ejemplo3.csv","w")
>>> contenido = csv.writer(fichero)
>>> contenido.writerow(['4/5/2015 13:34', 'Apples', '73'])
>>> contenido.writerows(['4/5/2015 3:41', 'Cherries', '85'],['4/6/2015 12:46', 'Pears', '14'])
>>> fichero.close()
```

$ cat ejemplo3.csv

4/5/2015 13:34,Apples,73

4/5/2015 3:41,Cherries,85

4/6/2015 12:46,Pears,14

Gestionar ficheros json

El módulo json nos permite gestionar ficheros con formato JSON (JavaScript Object Notation).

La correspondencia entre JSON y Python la podemos resumir en la siguiente tabla:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Leer ficheros json

Desde una cadena de caracteres:

```
>>> import json
>>> datos_json='{"nombre":"carlos","edad":23}'
>>> datos = json.loads(datos_json)
>>> type(datos)
```

<class 'dict'>

>>> print(datos)

{'nombre': 'carlos', 'edad': 23}

Desde un fichero:

>>> with open("ejemplo1.json") as fichero:

... datos=json.load(fichero)

>>> type(datos)

<class 'dict'>

>>> datos

{'bookstore': {'book': [{'_category': 'COOKING', 'price': '30.00', 'author': 'Giada De Laurentiis', 'title': {'__text': 'Everyday Italian', '_lang': 'en'}, 'year': '2005'}, {'_category': 'CHILDREN', 'price': '29.99', 'author': 'J K. Rowling', 'title': {'__text': 'Harry Potter', '_lang': 'en'}, 'year': '2005'}, {'_category': 'WEB', 'price': '49.99', 'author': ['James McGovern', 'Per Bothner', 'Kurt Cagle', 'James Linn', 'Vaidyanathan Nagarajan'], 'title': {'__text': 'XQuery Kick Start', '_lang': 'en'}, 'year': '2003'}, {'_category': 'WEB', 'price': '39.95', 'author': 'Erik T. Ray', 'title': {'__text': 'Learning XML', '_lang': 'en'}, 'year': '2003'}]}}

Escribir ficheros json

>>> datos = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie','felineIQ': None}

>>> fichero = open("ejemplo2.json","w")

>>> json.dump(datos,fichero)

>>> fichero.close()

cat ejemplo2.json

{"miceCaught": 0, "name": "Zophie", "felineIQ": null, "isCat": true}

Excepciones

Errores sintácticos y errores de ejecución

Veamos un ejemplo de error sintáctico:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Una excepción o un error de ejecución se produce durante la ejecución del programa. Las excepciones se pueden manejar para que no termine el programa. Veamos algunos ejemplos de excepciones:

```
>>> 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> a+4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

```
>>> "2"+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Hemos obtenido varias excepciones: ZeroDivisionError, NameError y TypeError. Puedes ver la lista de excepciones y su significado.

Manejando excepciones. try, except, else, finally

Veamos un ejemplo simple cómo podemos tratar una excepción:

```
>>> while True:
...   try:
...     x = int(input("Introduce un número:"))
...     break
...   except ValueError:
...     print ("Debes introducir un número")
```

1. Se ejecuta el bloque de instrucciones de try.
2. Si no se produce la excepción, el bloque de excepción no se ejecuta y continúa la ejecución secuencia.
3. Si se produce una excepción, el resto del bloque try no se ejecuta, si la excepción que se ha produce corresponde con la indicada en excepción se salta a ejecutar el bloque de instrucciones except.
4. Si la excepción producida no se corresponde con las indicadas en excepción se pasa a otra instrucción try, si finalmente no hay un manejador nos dará un error y el programa terminará.

Un bloque except puede manejar varios tipos de excepciones:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Si quiero controlar varios tipos de excepciones puedo poner varios bloques except. Teniendo en cuenta que en el último, si quiero no indico el tipo de excepción:

```
>>> try:
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... except:
...   print("Otro error")
```

Se puede utilizar también la cláusula else:

```
>>> try:
```

```
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... else:
...   print("Otro error")
```

Por último indicar que podemos indicar una cláusula finally para indicar un bloque de instrucciones que siempre se debe ejecutar, independientemente de la excepción se haya producido o no.

```
>>> try:
...     result = x / y
... except ZeroDivisionError:
...     print("División por cero!")
... else:
...     print("El resultado es", result)
... finally:
...     print("Terminamos el programa")
```

Obteniendo información de las excepciones

```
>>> cad = "a"
>>> try:
...   i = int(cad)
... except ValueError as error:
...   print(type(error))
...   print(error.args)
...   print(error)
...
```

<class 'ValueError'>

("invalid literal for int() with base 10: 'a'",)

invalid literal for int() with base 10: 'a'

Propagando excepciones. raise

Si construimos una función donde se maneje una excepción podemos hacer que la excepción se envíe a la función desde la que la hemos llamado. Para ello utilizamos la instrucción raise. Veamos algunos ejemplos:

```
def dividir(x,y):
    try:
        return x/y
    except ZeroDivisionError:
        raise
```

Con raise también podemos propagar una excepción en concreto:

```
def nivel(numero):
    if numero<0:
        raise ValueError("El número debe ser positivo:"+str(numero))
    else:
        return numero
```

Módulos y paquetes

- Módulo: Cada uno de los ficheros .py que nosotros creamos se llama módulo. Los elementos creados en un módulo (funciones, clases, ...) se pueden importar para ser utilizados en otro módulo. El nombre que vamos a utilizar para importar un módulo es el nombre del fichero.

- Paquete: Para estructurar nuestros módulos podemos crear paquetes. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado __init__.py. Este archivo, no necesita contener ninguna instrucción. Los paquetes, a la vez, también pueden contener otros sub-paquetes.

Ejecutando módulos como scripts

Si hemos creado un módulo, donde hemos definido dos funciones y hemos hecho un programa principal donde se utilizan dichas funciones, tenemos dos opciones: ejecutar ese módulo como un script o importar este módulo desde otro, para utilizar sus funciones. Por ejemplo, si tenemos un fichero llamado potencias.py:

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


def cuadrado(n):
    return(n**2)
def cubo(n):
    return(n**3)
if __name__ == "__main__":
    print(cuadrado(3))
    print(cubo(3))
```

En este caso, cuando lo ejecuto como un script:

```
$ python3 potencias.py
```

El nombre que tiene el módulo es __main__, por lo tanto se ejecutará el programa principal.

Además este módulo se podrá importar (como veremos en el siguiente apartado) y el programa principal no se tendrá en cuenta.

Importando módulos: import, from

Para importar un módulo completo tenemos que utilizar las instrucciones import. lo podemos importar de la siguiente manera:

```
>>> import potencias

>>> potencias.cuadrado(3)

9

>>> potencias.cubo(3)

27
```

Namespace y alias

Para acceder (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el **namespace**, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un **namespace**, es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo.

Es posible también, abreviar los **namespaces** mediante un **alias**. Para ello, durante la importación, se asigna la palabra clave as seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

```
>>> import potencias as p

>>> p.cuadrado(3)

9
```

Importando elementos de un módulo: from...import

Para no utilizar el **namespace** podemos indicar los elementos concretos que queremos importar de un módulo:

```
>>> from potencias import cubo

>>> cubo(3)

27
```

Podemos importar varios elementos separándolos con comas:

```
>>> from potencias import cubo,cuadrado
```

Podemos tener un problema al importar dos elementos de dos módulos que se llamen igual. En este caso tengo que utilizar **alias**:

```
>>> from potencias import cuadrado as pc

>>> from dibujos import cuadrado as dc

>>> pc(3)

9

>>> dc()

Esto es un cuadrado
```

Importando módulos desde paquetes

Si tenemos un módulo dentro de un paquete la importación se haría de forma similar. tenemos un paquete llamado operaciones:

```
$ cd operaciones

$ ls

__init.py__  potencias.py
```

Para importarlo:

```
>>> import operaciones.potencias

>>> operaciones.potencias.cubo(3)

27


>>> from operaciones.potencias import cubo

>>> cubo(3)

27
```

Función dir()

La función dir() nos permite averiguar los elementos definidos en un módulo:

```
>>> import potencias

>>> dir(potencias)

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'cuadrado', 'cubo']
```

¿Dónde se encuentran los módulos?

Los módulos estándar (como math o sys por motivos de eficiencia están escritos en C e incorporados en el intérprete (builtins).

Para obtener la lista de módulos builtins:

```
>>> import sys

>>> sys.builtin_module_names

('_ast', '_bisect', '_codecs', '_collections', '_datetime', '_elementtree', '_functools', '_heapq', '_imp', '_io', '_locale', '_md5', '_operator', '_pickle', '_posixsubprocess', '_random', '_sha1', '_sha256', '_sha512', '_socket', '_sre', '_stat', '_string', '_struct', '_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', 'array', 'atexit', 'binascii', 'builtins', 'errno', 'faulthandler', 'fcntl', 'gc', 'grp', 'itertools', 'marshal', 'math', 'posix', 'pwd', 'pyexpat', 'select', 'signal', 'spwd', 'sys', 'syslog', 'time', 'unicodedata', 'xxsubtype', 'zipimport', 'zlib')
```

Los demás módulos que podemos importar se encuentran guardados en ficheros, que se encuentra en los path indicados en el intérprete:

```
>>> sys.path

['', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
```

Módulos estándares: módulos de sistema

Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que también pueden ser importados. En esta unidad vamos a estudiar las funciones principales de módulos relacionados con el sistema operativo.

Módulo os

El módulo nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos

permiten manipular la estructura de directorios.

Descripción	Método
Saber si se puede acceder a un archivo o directorio	os.access(path, modo_de_acceso)
Conocer el directorio actual	os.getcwd()
Cambiar de directorio de trabajo	os.chdir(nuevo_path)
Cambiar al directorio de trabajo raíz	os.chroot()
Cambiar los permisos de un archivo o directorio	os.chmod(path, permisos)
Cambiar el propietario de un archivo o directorio	os.chown(path, permisos)
Crear un directorio	os.mkdir(path[, modo])
Crear directorios recursivamente	os.mkdirs(path[, modo])
Eliminar un archivo	os.remove(path)
Eliminar un directorio	os.rmdir(path)
Eliminar directorios recursivamente	os.removedirs(path)
Renombrar un archivo	os.rename(actual, nuevo)
Crear un enlace simbólico	os.symlink(path, nombre_destino)

```
>>> import os
>>> os.getcwd()
'/home/jose/github/curso_python3/curso/u40'
>>> os.chdir("..")
>>> os.getcwd()
'/home/jose/github/curso_python3/curso'
```

El módulo os también nos provee del submódulo path (os.path) el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios.

Descripción	Método
Ruta absoluta	os.path.abspath(path)
Directorio base	os.path.basename(path)
Saber si un directorio existe	os.path.exists(path)
Conocer último acceso a un directorio	os.path.getatime(path)
Conocer tamaño del directorio	os.path.getsize(path)
Saber si una ruta es absoluta	os.path.isabs(path)
Saber si una ruta es un archivo	os.path.isfile(path)
Saber si una ruta es un directorio	os.path.isdir(path)
Saber si una ruta es un enlace simbólico	os.path.islink(path)
Saber si una ruta es un punto de montaje	os.path.ismount(path)

Ejecutar comandos del sistema operativo. Módulo subprocess

Con la función system() del módulo nos permite ejecutar comandos del sistema operativo.

```
>>> os.system("ls")
curso  modelo.odp  README.md
0
```

La función nos devuelve un código para indicar si la instrucción se ha ejecutado con éxito.

Tenemos otra forma de ejecutar comandos del sistema operativo que nos da más funcionalidad, por ejemplo nos permite guardar la salida del comando en una variable. Para ello podemos usar el módulo subprocess

```
>>> import subprocess
>>> subprocess.call("ls")
curso  modelo.odp  README.md
0
```

```
>>> salida=subprocess.check_output("ls")
>>> print(salida.decode())
curso
modelo.odp
README.md

>>> salida=subprocess.check_output(["df","-h"])

>>> salida = subprocess.Popen(["df","-h"], stdout=subprocess.PIPE)
>>> salida.communicate()[0]
```

Módulo shutil

El módulo shutil de funciones para realizar operaciones de alto nivel con archivos y directorios. Dentro de las operaciones que se pueden realizar está copiar, mover y borrar archivos y directorios; y copiar los permisos y el estado de los archivos.

Descripción	**Método**
Copia un fichero completo o parte	shutil.copyfileobj(fsrc, fdst[, length])
Copia el contenido completo (sin metadatos) de un archivo	shutil.copyfile(src, dst, *, follow_symlinks=True)
copia los permisos de un archivo origen a uno destino	shutil.copymode(src, dst, *, follow_symlinks=True)
Copia los permisos, la fecha-hora del último acceso, la fecha-hora de la última modificación y los atributos de un archivo origen a un archivo destino	shutil.copystat(src, dst, *, follow_symlinks=True)
Copia un archivo (sólo datos y permisos)	shutil.copy(src, dst, *, follow_symlinks=True)
Copia archivos (datos, permisos y metadatos)	shutil.move(src, dst, copy_function=copy2)
Obtiene información del espacio total, usado y libre, en	shutil.disk_usage(path)

bytes	
Obtener la ruta de un archivo ejecutable	shutil.chown(path, user=None, group=None)
Saber si una ruta es un enlace simbólico	shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)

Módulos sys

El módulo sys es el encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.

Algunas variables definidas en el módulo:

Variable	Descripción
sys.argv	Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar python modulo.py arg1 arg2, retornará una lista: ['modulo.py', 'arg1', 'arg2']
sys.executable	Retorna el path absoluto del binario ejecutable del intérprete de Python
sys.platform	Retorna la plataforma sobre la cuál se está ejecutando el intérprete
sys.version	Retorna el número de versión de Python con información adicional

Y algunos métodos:

Método	Descripción
sys.exit()	Forzar la salida del intérprete
sys.getdefaultencoding()	Retorna la codificación de caracteres por defecto

Ejecución de scripts con argumentos

Podemos enviar información (argumentos) a un programa cuando se ejecuta como un script, por ejemplo:

```
#!/usr/bin/env python
import sys
print("Has introducido",len(sys.argv),"argumento")
suma=0
for i in range(1,len(sys.argv)):
        suma=suma+int(sys.argv[i])
print("La suma es ",suma)
```

```
$  python3 sumar.py 3 4 5
Has introducido 4 argumento
La suma es  12
```

Módulos estándares: módulos matemáticos

Módulo math

El módulo math nos proporciona distintas funciones y operaciones matemáticas.

```
>>> import math
>>> math.factorial(5)
120
>>> math.pow(2,3)
8.0
>>> math.sqrt(7)
2.6457513110645907
>>> math.cos(1)
0.5403023058681398
>>> math.pi
3.141592653589793
```

```
>>> math.log(10)
2.302585092994046
```

Módulo fractions

El módulo fractions nos permite trabajar con fracciones.

```
>>> from fractions import Fraction
>>> a=Fraction(2,3)
>>> b=Fraction(1.5)
>>> b
Fraction(3, 2)
>>> c=a+b
>>> c
Fraction(13, 6)
```

Módulo statistics

El módulo statistics nos proporciona funciones para hacer operaciones estadísticas.

```
>>> import statistics
>>> statistics.mean([1,4,5,2,6])
3.6

>>> statistics.median([1,4,5,2,6])
4
```

Módulo random

El módulo random nos permite generar datos pseudoaleatorios.

```
>>> import random
>>> random.randint(10,100)
```

```
>>> random.choice(["hola","que","tal"])
'que'

>>> frutas=["manzanas","platanos","naranjas"]
>>> random.shuffle(frutas)
>>> frutas
['naranjas', 'manzanas', 'platanos']

>>> lista = [1,3,5,2,7,4,9]
>>> numeros=random.sample(lista,3)
>>> numeros
[1, 2, 4]
```

Módulos estándares: módulos de hora y fechas

Módulo time

El tiempo es medido como un número real que representa los segundos transcurridos desde el 1 de enero de 1970. Por lo tanto es imposible representar fechas anteriores a esta y fechas a partir de 2038 (tamaño del float en la librería C (32 bits)).

```
>>> import time
>>> time.time()
1488619835.7858684
```

Para convertir la cantidad de segundos a la fecha y hora local:

```
>>> tiempo = time.time()
>>> time.localtime(tiempo)
```

time.struct_time(tm_year=2017, tm_mon=3, tm_mday=4, tm_hour=10, tm_min=37, tm_sec=19, tm_wday=5, tm_yday=63, tm_isdst=0)

Si queremos obtener la fecha y hora actual:

```
>>> time.localtime()
```

time.struct_time(tm_year=2017, tm_mon=3, tm_mday=4, tm_hour=10, tm_min=37, tm_sec=30, tm_wday=5, tm_yday=63, tm_isdst=0)

Nos devuelve a una estructura a la que podemos acceder a sus distintos campos.

```
>>> tiempo = time.localtime()
>>> tiempo.tm_year
```

2017

Podemos representar la fecha y hora como una cadena:

```
>>> time.asctime()
```

'Sat Mar 4 10:41:41 2017'

```
>>> time.asctime(tiempo)
```

'Sat Mar 4 10:39:21 2017'

O con un determinado formato:

```
>>> time.strftime('%d/%m/%Y %H:%M:%S')
```

'04/03/2017 10:44:52'

```
>>> time.strftime('%d/%m/%Y %H:%M:%S',tiempo)
```

'04/03/2017 10:39:21'

Módulo datetime

Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de tiempo.

```
>>> from datetime import datetime
```

```
>>> datetime.now()
datetime.datetime(2017, 3, 4, 10, 52, 12, 859564)
>>> datetime.now().day,datetime.now().month,datetime.now().year
(4, 3, 2017)
```

Para comparar fechas y horas:

```
>>> from datetime import datetime, date, time, timedelta
>>> hora1 = time(10,5,0)
>>> hora2 = time(23,15,0)
>>> hora1>hora2
False

>>> fecha1=date.today()
>>> fecha2=fecha1+timedelta(days=2)
>>> fecha1
datetime.date(2017, 3, 4)
>>> fecha2
datetime.date(2017, 3, 6)
>>> fecha1<fecha2
True
```

Podemos imprimir aplicando un formato:

```
>>> fecha1.strftime("%d/%m/%Y")
'04/03/2017'
>>> hora1.strftime("%H:%M:%S")
'10:05:00'
```

Podemos convertir una cadena a un datetime:

```
>>> tiempo = datetime.strptime("12/10/2017","%d/%m/%Y")
```

Y podemos trabajar con cantidades (segundos, minutos, horas, días, semanas,...) con timedelta:

```
>>> hoy = date.today()

>>> ayer = hoy - timedelta(days=1)

>>> diferencia=hoy -ayer

>>> diferencia

datetime.timedelta(1)
```

```
>>> fecha1=datetime.now()

>>> fecha2=datetime(1995,10,12,12,23,33)

>>> diferencia=fecha1-fecha2

>>> diferencia

datetime.timedelta(7813, 81981, 333199)
```

Módulo calendar

Podemos obtener el calendario del mes actual:

```
>>> año = date.today().year

>>> mes = date.today().month

>>> calendario_mes = calendar.month(año, mes)

>>> print(calendario_mes)

    March 2017

Mo Tu We Th Fr Sa Su

       1  2  3  4  5

 6  7  8  9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26
```

27 28 29 30 31

Y para mostrar todos los meses del año:

>>> print(calendar.TextCalendar(calendar.MONDAY).formatyear(2017,2, 1, 1, 2))

Instalación de módulos

Python posee una activa comunidad de desarrolladores y usuarios que desarrollan tanto los módulos estándar de python, como módulos y paquetes desarrollados por terceros.

PyPI y pip

- El *Python Package Index* o *PyPI*, es el repositorio de paquetes de software oficial para aplicaciones de terceros en el lenguaje de programación Python.
- pip: Sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python que se encuentran alojados en el repositorio *PyPI*.

Instalación de módulos python

Para instalar un nuevo paquete python tengo varias alternativas:

Utilizar el que esté empaquetado en la distribución que estés usando. Podemos tener problemas si necesitamos una versión determinada.

```
# apt-cache show python3-requests
...
Version: 2.4.3-6
...
```

1.

Instalar pip en nuestro equipo, y como superusuario instalar el paquete python que nos interesa. Esta solución nos puede dar muchos problemas, ya que podemos romper las dependencias entre las versiones de nuestros paquetes python instalados en el sistema y algún paquete puede dejar de funcionar.

```
# pip search requests
...
```

requests (2.13.0) - Python HTTP for Humans.

...

2.

3. Utilizar entornos virtuales: es un mecanismo que me permite gestionar programas y paquetes python sin tener permisos de administración, es decir, cualquier usuario sin privilegios puede tener uno o más "espacios aislados" (ya veremos más adelante que los entornos virtuales se guardan en directorios) donde poder instalar distintas versiones de programas y paquetes python. Para crear los entornos virtuales vamos a usar el programa virtualenv o el módulo venv.

Creando entornos virtuales con virtualenv

Podemos utilizar este software para trabajar con cualquier distribución de python, pero evidentemente es obligatorio si estamos trabajando con python 2.x o python 3.x (una versión anterior a la 3.3).

apt-get install python-virtualenv

Si queremos crear un entorno virtual con python3:

$ virtualenv -p /usr/bin/python3 entorno2

La opción -p nos permite indicar el intérprete que se va a utilizar en el entorno.

Para activar nuestro entorno virtual:

$ source entorno2/bin/activate

(entorno2)$

Y para desactivarlo:

(entorno2)$ deactivate

$

Creando entornos virtuales con venv

A partir de la versión 3.3 de python podemos utilizar el módulo venv para crear el entorno virtual.

Instalamos el siguiente paquete para instalar el módulos:

apt-get install python3-venv

Ahora ya como un usuario sin privilegio podemos crear un entorno virtual con python3:

$ python3 -m venv entorno3

La opción -m del intérprete nos permite ejecutar un módulo como si fuera un programa.

Para activar y desactivar el entono virtual:

$ source entorno3/bin/activate

(entorno3)$ deactivate

$

Instalando paquetes en nuestro entorno virtual

Independientemente del sistema utilizado para crear nuestro entorno virtual, una vez que lo tenemos activado podemos instalar paquetes python en él utilizando la herramienta pip (que la tenemos instalada automáticamente en nuestro entorno). Partiendo de un entorno activado, podemos, por ejemplo, instalar la última versión de django:

(entorno3)$ pip install django

Si queremos ver los paquetes que tenemos instalados con sus dependencias:

(entorno3)$ pip list

Django (1.10.5)

pip (1.5.6)

setuptools (5.5.1)

Si necesitamos buscar un paquete podemos utilizar la siguiente opción:

(entorno3)$ pip search requests

Si a continuación necesitamos instalar una versión determinada del paquete, que no sea la última, podemos hacerlo de la siguiente manera:

(entorno3)$ pip install requests=="2.12"

Si necesitamos borrar un paquete podemos ejecutar:

(entorno3)$ pip uninstall requests

Y, por supuesto para instalar la última versión, simplemente:

(entorno3)$ pip install requests

Para terminar de repasar la herramienta pip, vamos a explicar cómo podemos guardar en un fichero (que se suele llamar requirements.txt) la lista de paquetes instalados, que nos permite de manera sencilla crear otro entorno virtual en otra máquina con los mismos paquetes instalados. Para ello vamos a usar la siguiente opción de pip:

(entorno3)$ pip freeze

Django==1.10.5

requests==2.13.0

Y si queremos guardar esta información en un fichero que podamos distribuir:

(entorno3)$ pip freeze > requirements.txt

De tal manera que otro usuario, en otro entorno, teniendo este fichero puede reproducirlo e instalar los mismos paquetes de la siguiente manera:

(entorno4)$ pip install -r requirements.txt