BIG DADDY'S PYTHON TOOLBOX For 3.5+

Data on Disk (commands/modules/methods)

In this document: The symbol ⇘ means "yields", "results in", or "stands for". **string** ⇘ a series of characters. **file** ⇘ your file being written or read. **path** ⇘ the path to your **file**, i.e., ***path*** + **file**.

For reading and writting data on a disk Python has **2** built-in functions: (1) **open** - opens a new or an existing file, ex: fileVariable = open(*path*, mode) and (2) **with -** which also closes files automatically. (Also note the new pathlib read/write) ex : with **open(*path*, mode)** as fileVariable . There are **5** combinable **modes**: **r, w, a, b, +** -> see table below

There are **8** file **methods**: *[On page 2, see 7 techniques for reading files.]*

file.**write**(string), write to a new file, ***add* "\n"**, ⇘ the number of characters written
file.**read**(size), get all or some data, ⇘ a string
file.**close**(), close the file, if not using **with**
file.**tell**(), gives index of location in file
file.**seek**(offset, ***from_what***) positions file pointer
file.**readline**(), gets a single line that ends in a newline ("\n"), it retains the newline except for the last file line
file.**readlines()** - same as **list**(file) - reads all lines found in a file to a list variable

from_what values (optional, defaults to 0)
text files: only allows **0** - beginning of the file
binaryfiles: 0 - beginning of the file
1 - use current position
2 - use end of the file
Note: offset can be negative

Example: (the ***with*** structure auto closes a file)
with open(*path*, 'r') as file_ref_variable:
    list_variable = list(file_ref_variable)
⇘ *whole file, and then closes the file*
in 3.5 simplified by pathlib functions - see below

modes:
'r' : read only
'r+' : read or write
'w' : write only
'w+' : write or read
'a' : append
'a+' : append or read
'+': allow read / write
'b' (binary): 'rb', 'rb+', 'wb', 'wb+', 'ab', 'ab+'
binary files are not addressed in this document

Below: An aggregation of key module functions grouped by activity. Use "import *module*" except "from pathlib import Path *as* ***p***" gets pathlib **main** class. Abbreviated descriptions below do **NOT** show most of the **options** which can alter method performance. * *Note: An absolute path is a full path, a relative path is with respect to your CWD (current working dir).*

CREATE A PATH, DIRECTORY OR FILE

Create a path/file object
mypathfileobj = p(*path*)
Create new directory
p(*path*).**mkdir*** OR os.**mkdir**(*path*)
* will use CWD if path unspecified
Create a file at path/name = (*path*)
p.(*path*)**touch**(mode=0o666)
Create an absolute path object
os.path.**abspath**() for example: os.path.abspath('.') returns a normalized string of your CWD
Create a chain of directories
os.**makedirs**(*path*)
Create a symlink
p(fullpath&file).**symlink_to**(symlink_name, target _is_directory=False)

MANIPULATE PATHS, DIRECTORIES OR FILES

Change current CWD
os.**chdir**(*path*)
*Ex: os.chdir('D:\\Users\\me') **note \ escapes itself*
*supports an open directory descriptor
Delete a file
os.**remove**(*path*)
Move file or directory ⇘ **destination *path***
shutil.**move**(src, dst)
Replace/rename unconditionally
p(' *path* _name').**replace**('new_ *path* str or *path* object')
Rename a file or directory
p('*path* _name').**rename**('new_ *path* str or path object') OR os.**rename**(src, dst)
Rename files/paths recursively
os.**renames**(old, new)
Remove an empty directory
p(somedir).**rmdir**() OR os.**rmdir**(*path*)
Remove directories recursively
os.**removedirs**(*path*) *note: *raises OSError if not empty
Remove directory tree
shutil.**rmtree**(*path*)
Remove symlinks in a path - new obj
p(some*path*).**resolve()**
Copy file contents ⇘ destination path
shutil.**copyfile**(source, destination)
Copy file from source to destination
⇘ dst *path;* ☛ strings, data & permissions
shutil.**copy**(src, dst)
Copy file from src to dst with metadata
⇘ *dst path*
shutil.**copy2**(source, destination)
Copy entire directory tree ⇘ dst directory
shutil.**copytree**(src, dst)
Concatenate Paths (smart join)
os.path.join(*path, paths*)
Split path into head and tail
os.path.**split**(path) *tail is usually file name*

GET PATH, DIRECTORY, OR FILE INFORMATION

Find Current Working Directory *(cwd)*
p.**cwd**() OR os.**getcwd**() ⇘ **Windows path**
Confirm a dir in CWD
p("\dir_name").**is_dir**()
Confirm a file in CWD
p('file_name").**is_file**()
Confirm a path exists in CWD?
p(*path*).**exists**()
Return iter of matches in CWD
iter_name = p('.').**glob**('*.ext')
Confirm path & file for equality
p(a_path_file).**samefile**(other str / *path* obj)
os.path.**samefile**() and os.path.**samestat**()
Find user's home directory
p.**home**()
Confirm a dir (given full path)
p(*path*).**is_dir**()
Return a list of entries in the CWD path
os.**listdir**(path='.')
Return a list of path names matching *path*
glob.**glob**(*path*)
Return iterator of files rendered by glob
glob.**iglob**(*path*)
Return an iterator of os.DirEntry objects
os.**scandir**(*path*)
*iterated item attributes are: **name** and **path**
Create iter of files in directory
p(*path*).**iterdir**()
Find matching files (OR use glob module)
⇘ **iter of matches in CWD**
iter_name = p('.').**glob**('*.some_ext')
⇘ **all sub dir and files**
p('.').**glob**('**/*.some_ext')
⇘ **all sub dir and files**
p.(*path*).**rglob**('*.some_ext') - *same as*
Return info about a path "x"
p.**stat**(x) OR os.**stat**(x, mode)
very extensive - beyond this toolbox scope
Get Python search strings in a list
sys.**path** - *note no parens*

WORK WITH TEXT OR BINARY FILES

new in 3.5 - the read and write functions in pathlib.Path open, execute, and close a file all in one command - no close statement, no need for a “with” structure
Write text to a file
p('somefile.txt').**write_text**('sometext')
Write a bytes file w/ binary info
p('bytes_file_name').**write_bytes**(b'Binary data')
Read text from a file
p('file_name.txt').**read_text**()
Read binary data in to a bytes object
p('bytes_file_name').**read_bytes**()
To Open a file *for low level access - ignore this as it is rarely needed, p.**open**(**)

BIG DADDY'S PYTHON TOOLBOX For 3.5+

Data on Disk - 2 (commands/modules/methods)

7 Way to Read a text file

a line == comparison must end with "\n" for the compare to succeed

(1) looping : stepping through the lines
for line in file: *(print() **adds "\n"** by default)*
(2) .readline: gets an individual line and **adds "\n"**
getaline = file.**readline()**
(3, 4, 5) .read : gets all or some of the file in a single string
3. test = file.**read()** get whole file in a string **retaining newlines**
4. test = file.**read().splitlines()** puts lines as items **in a list**, **removes newlines**
5. txtstr = **file.read(x)** gets **1st x characters** (\n counts as 1 character)
(6) list(file): read all the lines of a file **into a list**
L = **list(file) retains newlines** in list items
(7) .readlines: read all the lines of a file **into a list**
mylist = **file.readlines() retains newlines** in list items - same as list(file)

module: pickle

module: **pickle** - python specific, many object types to/from binary serilization, not human readable. **Basic** pickle uses standard "with open" structure - must be opened for **binary** operations.
import pickle - To **.dump** (save) an object/file:
pickle.dump(object-to-pickle, save-to-file, protocol=3, ...)
EX: pickle.dump (object _name, myFile)
To **.load** (retrieve) an object/file:
pickle.load(file-to-read [, fix_imports = True][, encoding="ASCII"] ...)
EX: myList = pickle.load(myFile)
.dumps creates bytes object instead of writing a file. **.loads** reads a pickled object from a bytes object. ***lambda functions cannot be pickled. pickle offers much more control with additional methods.
What can be pickled:
None, True/False, integers, floating point numbers, complex numbers, strings, bytes, bytearrays, tuples, lists, sets, and dictionaries containing only picklable objects, functions defined at the top level of a module (using def, not lambda) built-in functions defined at the top level of a module, classes that are defined at the top level of a module, instances of such classes whose __dict__ or result of calling __getstate__ is picklable.

module: shelve

module: **shelve** - **import shelve** - A "shelf" is a persistent, dictionary-like object. The shelve module provides a simple interface to **pickle** / **unpickle** objects on DBM-style database files. Not secure.
shelve.open(filename, flag='c', protocol=None, writeback=False)
Always call **Shelf.close()** explicity. (note caps and Shelf not shelve)
If writeback=True, **Shelf.sync()** writes back entries, empties cashe, syncs with object on disk. Automatic with **Shelf.close()**.
with shelve.open('spam') as db:
db['eggs'] = 'eggs'
<-see https://docs.python.org/3.6/library/shelve.html#module-shelve

module: sqlite3

module: **sqlite3** - **import sqlite3** Create **connection** object:
sq3con = sqlite3.connect ('mysqlFile.db' [,detect_types]) **or:**
sq3con = sqlite3.connect (":memory:") - to create database in RAM
A **few** key **connection object** methods: .cursor(*see below*), .close(), .iterdump(), .commit(), .rollback(),
Create **cursor** object: CurObj = sq3con.cursor() **Methods and attributes:** .fetchone(), .fetchmany(size), .fetchall(), .close(), .rowcount, .lastrowid, arraysize, description, .executemany("sql [,parameters]"), and .execute("sql [,parameters]")
EX: Curobj.execute('''CREATE TABLE table_name (col_name data_type,...)''')
Notes: sql statements are case insensitive. Multiple statements are separated by semicolons (;). SQL ignores white space. Parameters are separated by commas but a comma after the last parameter causes a error.
Create **database**: Connection creates it if it does not exist.
A few SQL commands to **.execute :** CREATE TABLE, DROP TABLE, INSERT INTO table_name VALUE(vals), ALTER TABLE, REPLACE search_str, sub_str, rep_with, UPDATE table_name, SET col_name = new_value WHERE limiting conditions, DELETE FROM *col_name* WHERE..., SELECT col_name FROM table WHERE...,
Data types (**Python**:SQL)
None:NULL **int**:INTEGER **float**:REAL **str**:TEXT **bytes**:BLOB

module: C S V

module: **C S V** - **c**omma **s**eparated **v**alues
import csv - use standard built-in **open**, then create a csv.reader or csv.writer object
If csvfile is a file object, open with newline=""
.reader(csvfile [,dialect='excel'] [,**fmtparams])
QUOTE_NONNUMERIC format converts unquoted fields to float values
.writer(csvfile [,dialect='excel'] [,**fmtparams])
None is written as "". Other data written as strings.
.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
.DictWriter(f, *fieldnames*, restval='', extrasaction ='raise', dialect='excel', *args, **kwds)
Note: fieldnames is NOT optional.
writer constants are: QUOTE_ALL, QUOTE-NONE, QUOTE_MINIMAL,QUOTE-NONNUMERIC
csvreader object methods are:
.__next__() usually call as **next(reader)**
.dialect read only value of dialect in use
.line_num number of lines (not records) read
.fieldnames if not passed, initialized on 1st access
cvswriter object **methods** are:
.writerow(row) write the row
.writerows(rows) write all rows
.dialect read only value of dialect in use
.writeheader() write a row with field names per the constructor

Basic Examples:
```
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
```

module: JSON

module: **JSON** (JavaScript Object Notation): cross platform data interchange of lists & dictionaries **.dump**(obj, fp, many opts) serialize obj to file-like obj; **.dumps**(obj,*,many opts); **.load**(fp,*,many opts); **.loads**(str [bytes],*,many opts); **.JSONDecoder** (*,many opts) **.JSONEncoder**(*,many opts)

module: filecmp

module:**filecmp** - compare files & directories
import filecmp as fc
fc.cmp(f1, f2, shallow=True) ↳ Boolean
fc.cmpfiles(dir1, dir2, common, shallow=True)
↳ three lists: match, mismatch, errors
compare directories: fc.dircmp(a,b,ignore=, hide=)
.report() - 1 of many methods/attributes

module: fileinput

module: **fileinput** **import fileinput** - **creates a recursive iterator for multiple files**
fileinput.input(files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)
for line **in** fileinput.input(files):
process(lines) *& then repeat for each file*
Methods also available after 1st line is read:
.filename() .fileno() .lineno() .filelineno()
.isfirstline() .isstdin() .nextfile() .close()

modules: tarfile, zipfile, zipapp, zipimport, zlib, gzip

modules: **tarfile, zipfile, zipapp, zipimport, zlib, gzip:** these modules provide extensive support for compression and decompression of files. **tarfile** and **zipfile** could have a whole toolbox and it would not begin to address all of their options.
tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs) *handles gzip, bz2, lzma*
ZipFile.open name mode='r', pwd=None, *, force_zip64=False) *- context manager - use the with statement (new in 3.2) will do bzip2 and lzma*