Data on Disk (storing data permanently)

For reading and writing data on a disk, Python has **2 built-in functions**:
(1) open - opens a new or an existing file, ex: *fileVariable = **open(path, mode)***
(2) with - which also closes files automatically. ex: ***with open(path, mode) as fileVariable***
There are **8** file **methods**:

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, <u>if not using **with**</u>
file.**readline**([#chars]), 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

file.**tell**(), gives index of location in file
file.**seek**(offset, *from_what [value— default is 0])*, positions file pointer
   text files allow 0 only
   binaryfiles: 0 - beginning of the file
               1 - use current position
               2 - use end of the file

*Note: **offset** can be negative*
Note: addition functionality using IO module

There are **5** combinable **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 explained in detail here

---

7 Ways to Read a text file [Caveat: a line == comparison must end with "\n" for the compare to succeed.]

(1) looping : stepping through the lines - **for** line in file: *(note: 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 or a list:
    3. test_string = file.**read()** get whole file in a **string**, *retaining* newlines (\n)
    4. test_string2 = file.**read(x)** gets **1st x characters** (\n counts as 1 character)
    5. test_list = file.**read().splitlines()** gets lines as items in a **list**, *removing* newlines (\n)
(6) list(file): read all the lines of a file **into a list** - l2 = **list(file)** retains newlines in list items
(7) .readlines: read all lines of a file **into a list** - mylist = file.**readlines()** retains newlines in list items - like **list(file)**

```
# assumes mary.txt is a 4 line nursery rhyme
count=1
with open(mary_path, "r") as mary:
    for line in mary:
        print(count, end=" ")
        print(line, end="")
        if "snow" in line:
            print("  **Found snow")
        count+=1
```

Pathlib   Since [3.4] the **pathlib** module has simplified access functions: use "from pathlib import **P**ath *[as p]*" to get pathlib's **main** class which creates **Concrete** paths (as opposed to **Pure** paths) for the platform on which code is running. The new limited read/write simplifies the most basic disk transactions. ***p**(pathsegs)* adapts to the type system in which it finds itself running. *Note: These abreviated descriptions do not show many available options! * An absolute path is a full path, a relative path is with respect to your CWD (**C**urrent **W**orking **D**ir).   **'.'** is a shortcut for your CWD*
The **with** structure is a bit different when using Pathlib: see wikipython;   with pathlib_path.open('w+') as file_alias

---

PATHLIB Path FUNCTIONS
Working with Text or Binary Files

new in 3.5 -The read/write methods in pathlib.Path open, execute, and close a file in one command though they have limited utility; no close statement, no need for "with", but first you must use:
p(pathsegments) creates concrete path based on the current op sys (a PosixPath or WindowsPath)
Write text   wholepath.**write_text**('sometext') one string only, there is no append mode, *replaces any existing file! Writes ONE string as a whole file.*
mypath is path to directory, myfile is name of file:
    wholepath = **p**(mypath / myfile) **# join** parts
    wholepath. write_text(*"some_string"*)
Read text from a file (whole file read to a string)
wholepath.read_text()
Read binary data in to a bytes object
wholepath.read_bytes()
Write a bytes file w/ binary info
wholepath.write_bytes(b'Binary data')
To Open a file *for low level access* - mostly ignore this as it is <u>rarely</u> needed, **wholepath.open**(**)
For more **read/write control** use a with/open structure: **with** wholepath.**open**('w+') as *file_alias*:
Path, Directory, or File Information
Find Current **W**orking **D**irectory   **p.cwd**()
Find user's **home** directory   **p.home**()
Confirm dir in CWD   wholepath.**is_dir**()
Confirm a dir (given full path) my**path.is_dir**()
Confirm a file
wholepath.is_file()   or in cwd   **myfile.isfile**()
Confirm a path exists to a directory or file
wholepath.exists()
Confirm path & file for equality
wholepath.samefile(other str / *path* obj)
Create iterator of files in directory

wholepath.iterdir() ↳ path objects of dir contents
Find matching files (OR use glob module) - only works inside a sorted() structure : * - all char, ? - a single char, [ ] - literal match like[?], ** - recursive
   iter of matches in CWD
     iter_name = **sorted(p**('.').**glob**(*'pattern'***)**
   all sub dir and files
     sorted(p('.').**glob**('**/*.some_ext')**)**
   all sub dir and files
     sorted(wholepath.**rglob**('*.some_ext')**)**
Return info about a path "x"
wholepath.stat(x)[.st_*information];* including:
_mode, _dev, _gid, _uid, _size, _mtime, more…

Create a Path, Directory or File
Create new directory
wholepath.mkdir(*parents=False*) if parents=True those segs will be created
Create a file at path/name = (*path***)**
wholepath.touch(mode=*0o666*)
Create a symlink
wholepath.symlink_to(symlink_name, target _is_directory=False)

Manipulate Paths, Directories or Files
Replace/rename unconditionally
wholepath.replace('new_ *path* str or *path* object')
Rename a file or directory
wholepath.rename('new_ *path* str or path object')
Remove an empty directory
wholepath.rmdir()
Remove symlinks in a path - new obj
wholepath.resolve() - make path absolute
Other methods include: chmod(mode), group(), ismount(), is_symlink(), is_socket(), is_fifo(), is_block_device(), is_char_device(), lchmod (mode), lstat(). owner(), readlink(), rglob(pattern) - like adding **/, symlink_to(), unlink, link_to

Useful os/os.path Functions
Change current CWD - oddly Pathlib does not support this critical function, must use:
os.**chdir**(*path*) *poor practice to change cwd*
*Ex: os.chdir('D:\\Users\\me') **note \ escapes itself*
*supports an open **directory** descriptor
Return a list of entries in the CWD path
os.**listdir**(path='.')
Return an iterator of os.DirEntry objects
os.**scandir**(*path*)
*iterated item attributes are: **name** and **path**
Create a chain of directories
os.**makedirs**(*path*)
Delete a file
os.**remove**(*path*)   same as os.**unlink**(*path*)
Delete a directory
os.**rmdir**(*path*)
Concatenate Paths (smart join)
os.path.**join**(*path, paths*)
Split path into head and tail
os.path.**split**(path)   *tail is usually file name*
Rename files/paths recursively
os.**renames**(old, new) or os.**rmdir**(*path*)
Get Name of user logged in
os.**getlogin()**

Useful shutil Functions
Move file or directory ↳ destination *path*
shutil.**move**(src, dst)
Copy file contents   ↳ destination path
shutil.**copyfile**(source, destination)
Copy file source to destination ↳ dst *path;*
shutil.**copy**(src, dst)
shutil.**copy2**(src, dst) sab but save metadata

BIG DADDY'S
2021A p2 © 2021 John A. Oakey
V020621a
PYTHON ② TOOLBOX For 3.6+
Data on Disk - (commands/modules/methods)
compliments: Big Daddy & www.wikipython.com

module: C S V - **c**omma **s**eparated **v**alues - **import csv**
with open(cvsfile_*file_path*, [mode], newline='') **as** *alias_name*: code to read/write/etcetera
Note: If csvfile is a file object, open with newline=""
Next create a .reader or .writer object or call another **function**:
* csv.**register_dialect**(name[, dialect[, format_parameters]])
* csv.**unregister_dialect**(name)
* csv.**get_dialect**(name)
* csv.**list_dialects**()
* csv.**field_size_limit**([new_limit])
* csv.**reader**(*file_alias* [,**dialect**='excel'][,**format_parameters**])
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
* csv.**writer**(csvfile [,**dialect**='excel'] [,**format_parameters**])
Note: None is written as "". Other data written as strings.
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:
    print(row)
csv.DictReader and csv.DictWriter classes
The module provides classes that operate like regular read/write functions but map rows into a dict(**kwarg) class mapping object with keys given by the fieldnames parameters. If fieldnames is omitted, the values in the first row of file f will be used as the fieldnames. As of **[3.8]** returned rows are of type *dict*.
csv.**DictReader**(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
csv.**DictWriter**(f, *fieldnames*, restval='', extrasaction ='raise', dialect='excel', *args, **kwds)
Note: fieldnames is NOT optional.
DictReader and DictWriter Basic Examples:
with open ('names.csv', newline='') as csvfile:
  reader = csv.DictReader(csvfile)
    for row in reader:
      print(row['first name'], row['last name'])
with open('names.csv', 'w', newline='') as csvfile:
  fieldnames = ['first_name', 'last_name']
  writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

  writer.writeheader()
  writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
  writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
  writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})

CSV dialects and format parameters
A **dialect** is a shortcut notation for a **group** of **format_parameters** common to a specific "type" of comma separated files. **'excel',** the granddaddy of all dialects, occupies its own class automatically registered as a dialect as is **'excel_tab'** for tab delimited files and **'unix_dialect'** which quotes all fields and line terminates with '\n'. Individual format parameters can override the group defaults. Constants are provided to specify how quoting is written or read:
format_parameters for dialects are:
.delimiter - defaults to ','
.doublequote - defaults to True, quotechar doubled if false, escapechar is prefix to quotechar
.escapechar - default to None which disables escaping
.lineterminator - string writer uses to terminate, defaults to '\r\n' which is standard windows format
.quotechar - defaults to a standard quote: "
.quoting - controls when quotes are generated or recog-nized using constants. QUOTE_MINIMAL is default.
  Reader Constants:
  csv.QUOTE_NONNUMERIC - converts unquoted fields to float values
  csv.QUOTE_NONE - perform no processing of quote characters
  Writer Constants:
  csv.QUOTE_ALL - quote all fields
  csv.QUOTE_MINIMAL - quote fields with delimiter, quotechar, or lineterminator characters
  csv.QUOTE_NONNUMERIC - quote all non-numeric fields
  csv.QUOTE_NONE - never quote fields
.skipinitialspace - default is False
.strict - default is Fault, True raised Error on bad input

module: **pickle** - not secure, **Python specific**, many object types to/from binary serilization, not human readable. **Basic** pickle uses standard "with open" structure - must open for **binary** ops.
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, does not write a file
.loads reads a pickled object from a bytes object.
*lambda functions cannot pickle. pickle offers more control with additional methods.
What can be pickled: None, True/False, integers, float-ing 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, 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.

modules: **tarfile, zipfile, zipapp, zipimport, zlib, gzip:** these modules provide extensive support for compression and decom-pression 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)

Example from: (demo of context manager application)
https://docs.python.org/3/library/zipfile.html#module-zipfile
with ZipFile('spam.zip') as myzip:
  with myzip.open('eggs.txt') as myfile:
    print(myfile.read())

module: **fileinput import fileinput** - a **recursive iterator** for **multiple files.** Methods:
.filename(), .fileno(), .lineno(), filelineno(), isfirstline(), .isstdin(), .nextfile(), .close
for line **in** fileinput.input(files):
  process(lines)   *& then repeat for each file*

(JavaScript Object Notation) – import json

JSON is a minimal data interchange format replacing XML for server-to-web-app communication. All json data **objects** are **key (or name) : value pairs** (*similar* to a Python dictionary) with elements separated by commas.

json **objects** are defined by braces **{ }**. A **value** can be any json data type (see table) including an array or nested array. **All objects, keys and values are coerced into strings.** Some nested objects are in **arrays** defined by brackets **[ ]**. Examples: **{"name": value} or {"name":[{"key":"value", "key":[value, value,[...]]}]}**. The table shows Data type conversions. json encodes and decodes its data objects using, or producing, a **'''json string'''** or a json **file object.** json code is valid JavaScript. json files end with ".json"

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

Placing and Retrieving json Data on Disk

json.load (fp; *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None,object_pairs_hook=None, **kw**) [Load json data from a .json file on disk to a Python dictionary]
A json **file object** can be retrieved and converted to Python with the **json.load()** method. <u>Example:</u>
with open(file_path_to_file.json, 'r') as *file_ref_name*:
  new_dict = json.load(*file_ref_name*)

json.dump (obj, **fp**, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw**) [Save data to disk]
A Python dictionary **obj**, even nested, can be encoded and saved to a json **file** with the **json.dump()** method. You can NOT make repeated calls to dump using the same fp.   <u>Example:</u>
with open(file_path_to_file.json, 'w') as *file_ref_name*:
  json.dump(*dictionary_name*, *file_ref_name*, *[indent=some_int][sort_keys=True/False]*)

Interchanging/Converting json Strings and Python Objects

json.dumps (obj, *, skipkeys=false, ensure_ascii=True, check_circular=True, allow_nan= True, cls=None, **indent**=None,separators=None, default=None, **sort_keys**=False, **kw**) [Used to convert a Python dictionary to a string for output, or to be used as a json object.]
A Python **dictionary** can be converted to a Python/json **string** for review or storage with **json.dumps (**name_ of_object**)**. Particularly note the indent and sort_keys options - see Notes below.

json.loads (**s**, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None,object_pairs_hook=None, **kw**)   [Used to convert json strings to Python object(s)]
json strings can convert to one or more Python **dictionaries and nested objects** with the **json.loads (***name_ of_*string**)** method. *Notes: (1) json.loads is pronounced like "Jason load ess", where "ess" is for string (2) true / false / null are converted to True / False / None.*

Notes on Option Terms: s—a string holding valid json code; **fp**—file name, derived from "file pointer" used in C; **skipkeys**—if set to true, a nonstandard dict key is skipped (false is default); **ensure_ascii** (default is true) escapes non-ASCII characters - to allow Unicode characters, set this to False and when opening a file to be written set **encoding="utf-8"**; **check_circular** False results in an OverflowError (at best); **allow_nan** assigns an out of range float to *Nan, Infinity, -Infinity*; **indent**—pretty-print with this spacing; **separators**—if used, must be a tuple of (item_separator, key_separator); **default**—if specified, is a function called for objects that cannot serialize; **sort_keys** if set to true, yields dictionaries sorted by key. It may be helpful to review workflow on the JSON page under Other Modules on www.wikipython.com

An example of a valid Python/json string object: *(see wikipython.com for workflow of json objects)*

```
zoostr = '''{"cageA":{"monkey1":
[{"type":"howler","age":5,"name":"Bigmout
h"}], "monkey2":[{"type":null, "age":4,
"name":"Webster"}]}, "cageB":{"ape1":
[{"type":"gorilla","age":20,"name":"Mr.
Big"}],"ape2": [{"type":"orangutan",
"age":8,"name":"Longarm"}]}}'''
```

mydata = json.loads(zoostr) / pp.pprint(**mydata**) #modified pretty print

```
{ 'cageA':{ 'monkey1': [{'age': 5, 'name': 'Bigmouth', 'type': 'howler'}],
       'monkey2': [{'age': 4, 'name': 'Webster', 'type': None}]},
  'cageB':{ 'ape1': [{'age': 20, 'name': 'Mr. Big', 'type': 'gorilla'}],
       'ape2': [{'age': 8, 'name': 'Longarm', 'type': 'orangutan'}]}}
}
```

USING SQLITE3 - *this is a high level abstraction of the process of creating and using SQLite3*

① **Import the module:** import sqlite3 [as **sq**] ➢ *(creates an abbreviated alias for sqlite3)*

Module Level Functions:
sq.PARSE_DECLTYPES - a constant used with detect_types parameter of connect to force parsing 1st word of declared type to assign proper conversion; **sq.sqlite_version** - ↳ a str; **sq.PARSE_COLNAME** - like above, parses column name looking for [mytype] and excludes that as part of column name. **sq.complete_statement(str)** - ↳ True if the string contains one or more complete SQL statements terminated by semicolons. (Allows construction of a shell.) **sq.connect(**db[, timeout, detect_types, isoloation_level, check_same_ thread, factory, cached_statements, uri]**)** opens connection, returns **connection obj.**

② **Create a connection object** to database (disk or RAM)
sq3con = sq.**connect** ('some-db-including-path' [,detect_types]**)** Note: *special name **":memory:"** can be used to CREATE a database in RAM;* Impermanent, but very fast. See module doc for options: https://docs.python.org/3/library/sqlite3.html#module-sqlite3 Examples:
dbpath = r"D:\Temp\Testdb" | or | **sq3con = sq.connect(":memory:")**
sq3con = sq.connect(dbpath)
The connection object exposes most of the TCL components and some non-standard shortcuts.

Methods of the connection object once created: *sq3con*.**cursor**(factory=Cursor) – underline step 3 below.
sq3con.**commit()** - save changes and makes them visible; *sq3con*.**row_factory**;
sq3con.**rollback()** – reverses any changes to the database since the last commit();
sq3con.**close()** – closes the database connection – does NOT call commit() before closing;
sq3con.**create_function(name, num_params, func)** - Creates a user-defined function, see online doc;
sq3con.**create_aggregate**(name, num_params, aggregate_class); *sq3con*.**total_changes**;
sq3con.**iterdump()** - Returns an iterator to dump the database in an SQL text format.
Note sq3con.***execute()*** *and sq3con*.***executemany()*** *connection objects are non-standard SQLite3* **shortcuts** *that look like the cursor objects, both use the .execute and .executemany keywords, but these shortcuts also return a cursor.*

③ **Create a cursor object** using the **connection object.** The cusor object exposes methods and attributes necessary to use the database) – a cursor object is essentially an active session of the database. ***CurObj = sq3con*.cursor()**
④ **Use DDL** (Data Definition Language) commands to CREATE, DROP or ALTER a database.

Python Implementation of SQL DDL Commands: **CurObj.execute ("sql [,parameters]")** - this is the underline method called to execute SQL commands; The 3 most common DDL commands are: **CREATE TABLE** (table name(col name, data type […])), **ALTER TABLE**(table name ([ADD *parameters*][DROP *parameters*]), and **DROP TABLE** - CurObj.**execute (**"DROP TABLE *tablename***")** *general form*: *CurObj*.***execute* (**"CREATE TABLE underline table name (underline col name data type , …)").

⑤ **Use DML/DQL** commands to maintain and retrieve. change and manipulate information.
   INSERT, UPDATE, DELETE, MERGE, plus **DQL's SELECT**
⑥ **Use Cursor Object methods** to access data: **curobj.execute('SELECT * FROM zoo_data WHERE type = "mammal")**

Cursor Object Methods and Attributes: **.fetchone()** - Fetches the next row of a query result set, returning a single sequence (a tuple of col values), or **None** when no more data is available. **.fetchmany(size=cursor, arraysize)** - Fetches the next set of rows of a query result, returning a list of tuples. An empty list is returned when no more rows are available. **.fetchall()** - Fetches all (remaining) rows of a query result, returning a list. **.close()** - Close the cursor session
.rowcount - Note the determination of "rows affected"/"rows selected" is quirky. **.lastrowid** - This read-only attribute provides the rowid of the last modified row. Set this only if you issued an INSERT or a REPLACE statement using the **execute()** method. In addition to fetchone() and fetchall() to retrieve data, the **cursor** can be used as an **iterator**.

⑦ **Close connection and/or cursor.** Use a module level command to destroy the database, if desired.

sqlite3.**Row** is used as a row factory, accessed by both index and case insensitive name. It returns rows as tuples. Initialize with something like: **con.row_factory = sq.Row**

Python—SQLite3 Data Types

Python	SQLite
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

SQLite3 Components & Terms
DDL: Data Definition Language - commands to create a db
DML: Data Manipulation Language - maintenance commands
DQL: Data Query Language
DCL: SQL security components
TCL: Transaction Contol Commands
sqlite3 non-standard methods:

SQLite Keywords/Operators

		EXCLUDE	INSTEAD	PLAN	TRANSACTION
		EXCLUSIVE	**INTERSECT**	PRAGMA	TRIGGER
		EXISTS	**INTO**	PRECEDING	UNBOUNDED
		EXPLAIN	**IS**	PRIMARY	**UNION**
ABORT	**COMMIT**	FAIL	**ISNULL**	QUERY	**UNIQUE**
ACTION	CONFLICT	**FILTER**	**JOIN**	RAISE	**UPDATE**
ADD	CONSTRAINT	**FIRST**	**KEY**	**RANGE**	USING
AFTER	**CREATE**	**FOLLOWING**	**LAST**	RECURSIVE	VACUUM
ALL	CROSS	FOR	LEFT	REFERENCES	**VALUES**
ALTER	CURRENT	FOREIGN	LIKE	REGEXP	VIEW
ALWAYS	CURRENT_DATE	**FROM**	LIMIT	REINDEX	VIRTUAL
ANALYZE	CURRENT_TIME	FULL	MATCH	RELEASE	**WHEN**
AND	CURRENT_TIMESTAMP	GENERATED	NATURAL	**RENAME**	**WHERE**
AS	**DATABASE**	**GLOB**	**NO**	**REPLACE**	WINDOW
ASC	DEFAULT	GROUP	**NOT**	RESTRICT	**WITH**
ATTACH	DEFERRABLE	GROUPS	**NOTHING**	RIGHT	**WITHOUT**
AUTOINCREMENT	DEFERRED	**HAVING**	**NOTNULL**	**ROLLBACK**	
BEFORE	**DELETE** FROM table WHER	**IF**	**NULL**	**ROW**	
BEGIN	DESC	IGNORE	NULLS	**ROWS**	
BETWEEN	DETACH	IMMEDIATE	**OF**	**SAVEPOINT**	
BY	**DISTINCT**	**IN**	OFFSET	**SELECT**	
CASCADE	DO	**INDEX**	**ON**	**SET**	
CASE	**DROP** object, object name	INDEXED	**OR**	**TABLE**	
CAST	**EACH**	INITIALLY	**ORDER**	TEMP	
CHECK	**ELSE**	**INNER**	OTHERS	TEMPORARY	
COLLATE	ESCAPE	**INSERT**	**OUTER**	THEN	
COLUMN	**EXCEPT**	INTO table (col1, col2)	OVER	TIES	
		INTO table values (v1,v2)	PARTITION	**TO**	

www.wikipython.com