A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM

To write a file, you have to open it with mode 'w' as a second parameter:

In [6]:
```
fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

open returns a file object that provides methods for working with the file. The write method puts data into the file.

In [2]:
```
line1 = "This here's the wattle,\n"
```

In [3]:
```
fout.write(line1)
```

Out[3]: 24

When you are done writing, you should close the file

In [4]:
```
fout.close()
```

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

In [7]:
```
x = 52
fout.write(str(x))
```

Out[7]: 2

An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string.
For example, the format sequence '%d' means that the second operand should be formatted as a decimal integer:

In [8]:
```
camels = 42
'%d' % camels
```

Out[8]: '42'

In [11]:
```
print("my fav number is %d" %1729)
```

```
my fav number is 1729
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

In [12]:
```
'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
```

Out[12]:
```
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

In [13]:
```
'%d %d %d' % (1, 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-4a97654d37a9> in <module>
----> 1 '%d %d %d' % (1, 2)

TypeError: not enough arguments for format string
```

In [1]:
```
'%d' %'dollars'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 '%d' %'dollars'

TypeError: %d format: a number is required, not str
```

In [2]:
```
import os
cwd = os.getcwd()
cwd
```

Out[2]:
```
'C:\\Users\\ankit19.gupta\\ankit\\ankit\\ML_Code\\Python_R_Prolog_Code\\Python_Practice\\thinkpython'
```

A simple filename, like memo.txt is also considered a path, but it is a relative path because it relates to the current directory. If the current directory is /home/dinsdale, the filename memo.txt would refer to /home/dinsdale/memo.txt.

A path that begins with / does not depend on the current directory; it is called an absolute path. To find the absolute path to a file, you can use os.path.abspath:

In [3]:
```
os.path.abspath('output.txt')
```

Out[3]:
```
'C:\\Users\\ankit19.gupta\\ankit\\ankit\\ML_Code\\Python_R_Prolog_Code\\Python_Practice\\thinkpython\\output.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

In [19]:
```
os.path.exists('output.txt')
```

Out[19]:
```
True
```

If it exists, os.path.isdir checks whether it's a directory:

In [20]:
```
os.path.isdir('output.txt')
```

Out[20]:
```
False
```

In [21]:
```
os.path.isdir('C:\\Users\\ankit19.gupta\\')
```

Out[21]:
```
True
```

Similarly, os.path.isfile checks whether it's a file.

os.listdir returns a list of the files (and other directories) in the given directory:

In [22]:
```
os.listdir(cwd)
```

Out[22]:
```
['.ipynb_checkpoints',
 'Chapter_1.ipynb',
 'Chapter_10_Lists.ipynb',
 'Chapter_11_Dictionary.ipynb',
 'Chapter_12_Tuples.ipynb',
 'Chapter_13_Files.ipynb',
 'Chapter_2.ipynb',
 'Chapter_3.ipynb',
 'Chapter_4.ipynb',
 'Chapter_5.ipynb',
 'Chapter_8.ipynb',
 'Chapter_9_Case_Study.ipynb',
 'mypolygon.py',
 'output.txt',
 'words.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

In [23]:
```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join takes a directory and a file name and joins them into a complete path.

The os module provides a function called walk that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given

directory and its subdirectories. You can download my solution from https://thinkpython.com/code/walk.py.

In [4]:
```
os.walk('C:\\Users\\ankit19.gupta')
```

Out[4]:
```
<generator object _walk at 0x0000029C3F160EB0>
```

In [7]:
```
for (root,dirs,files) in os.walk('.', topdown=True):
        print (root)
        print (dirs)
        print (files)
        print ('--------------------------------')
```

```
.
['.ipynb_checkpoints', '__pycache__']
['158-0.txt', 'captions.bak', 'captions.dat', 'captions.dir', 'Chapter_10_Lists.ipyn
b', 'Chapter_10_Lists.pdf', 'Chapter_11_Dictionary.ipynb', 'Chapter_11_Dictionary.pd
f', 'Chapter_12_Tuples.ipynb', 'Chapter_12_Tuples.pdf', 'Chapter_13_Files.ipynb', 'C
hapter_14_Classes_and_objects.ipynb', 'Chapter_15_Classes_and_functions.ipynb', 'Cha
pter_16_Classes_and_methods.ipynb', 'Chapter_17_Inheritance.ipynb', 'Chapter_1_The_w
ay_of_the_program.ipynb', 'Chapter_1_The_way_of_the_program.pdf', 'Chapter_2_Variabl
es_expressions_and_statements.ipynb', 'Chapter_2_Variables_expressions_and_statement
s.pdf', 'Chapter_3_Functions.ipynb', 'Chapter_3_Functions.pdf', 'Chapter_4_Case_stud
y_interface_design.ipynb', 'Chapter_4_Case_study_interface_design.pdf', 'Chapter_5_C
onditionals_and_recursion.ipynb', 'Chapter_5_Conditionals_and_recursion.pdf', 'Chapt
er_6_Fruitful_functions.ipynb', 'Chapter_6_Fruitful_functions.pdf', 'Chapter_7_Itera
tion.ipynb', 'Chapter_7_Iteration.pdf', 'Chapter_8_Strings.ipynb', 'Chapter_8_String
s.pdf', 'Chapter_9_Case_Study.ipynb', 'Chapter_9_Case_Study.pdf', 'mypolygon.py', 'o
utput.txt', 'wc.py', 'words.txt']
--------------------------------
.\.ipynb_checkpoints
[]
['Chapter_10_Lists-checkpoint.ipynb', 'Chapter_11_Dictionary-checkpoint.ipynb', 'Cha
pter_12_Tuples-checkpoint.ipynb', 'Chapter_13_Files-checkpoint.ipynb', 'Chapter_14_C
lasses_and_objects-checkpoint.ipynb', 'Chapter_15_Classes_and_functions-checkpoint.i
pynb', 'Chapter_16_Classes_and_methods-checkpoint.ipynb', 'Chapter_17_Inheritance-ch
eckpoint.ipynb', 'Chapter_1_The_way_of_the_program-checkpoint.ipynb', 'Chapter_2_Var
iables_expressions_and_statements-checkpoint.ipynb', 'Chapter_3_Functions-checkpoin
t.ipynb', 'Chapter_4_Case study_interface_design-checkpoint.ipynb', 'Chapter_5_Condi
tionals_and_recursion-checkpoint.ipynb', 'Chapter_6_Fruitful_functions-checkpoint.ip
ynb', 'Chapter_7_Iteration-checkpoint.ipynb', 'Chapter_8_Strings-checkpoint.ipynb',
'Chapter_9_Case_Study-checkpoint.ipynb']
--------------------------------
.\__pycache__
[]
['wc.cpython-36.pyc']
--------------------------------
```

Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an FileNotFoundError:

In [27]:
```
fin = open('bad_file')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-27-a7d7d7ad396b> in <module>
----> 1 fin = open('bad_file')

FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

In [28]:
```
fout = open('/etc/passwd', 'w')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-28-8a9adb191927> in <module>
----> 1 fout = open('/etc/passwd', 'w')

FileNotFoundError: [Errno 2] No such file or directory: '/etc/passwd'
```

And if you try to open a directory for reading, you get

In [32]:
```
fin = open('C:\\Users')
```

```
---------------------------------------------------------------------------
PermissionError                           Traceback (most recent call last)
<ipython-input-32-ae85160e3fe0> in <module>
----> 1 fin = open('C:\\Users')

PermissionError: [Errno 13] Permission denied: 'C:\\Users'
```

To avoid these errors, you could use functions like os.path.exists and os.path.isfile, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the try statement does. The syntax is similar to an if...else statement:

In [33]:
```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

```
Something went wrong.
```

Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and runs the except clause. Handling an exception with a try statement is called catching an exception. In this example, the except clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully

A database is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values.

The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module dbm provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.
Opening a database is similar to opening other files:

In [34]:
```
import dbm
db = dbm.open('captions', 'c')
```

The mode 'c' means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, dbm updates the database file.

In [35]:
```
db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, dbm reads the file:

In [36]:
```
db['cleese.png']
```

Out[36]:
```
b'Photo of John Cleese.'
```

The result is a bytes object, which is why it begins with b. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, dbm replaces the old value:

In [37]:
```
db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
db['cleese.png']
```

Out[37]:
```
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like keys and items, don't work with database objects. But iteration with a for loop works:

In [38]:
```
for key in db.keys():
    print(key, db[key])
```

```
b'cleese.png' b'Photo of John Cleese doing a silly walk.'
```

As with other files, you should close the database when you are done:

In [39]:
```
db.close()
```

Pickling

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

In [40]:
```
import pickle
t = [1, 2, 3]
pickle.dumps(t)
```

Out[40]:
```
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to interpret. pickle.loads ("load string") reconstitutes the object:

In [41]:
```
t1 = [1, 2, 3]
s = pickle.dumps(t1)
t2 = pickle.loads(s)
t2
```

Out[41]:
```
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

In [42]:
```
t1 == t2
```

Out[42]:
```
True
```

In [43]:
```
t1 is t2
```

Out[43]:
```
False
```

In other words, pickling and then unpickling has the same effect as copying the object.
You can use pickle to store non-strings in a database.
In fact, this combination is so common that it has been encapsulated in a module called shelve.

Pipes

Most operating systems provide a command-line interface, also known as a shell. Shells usually provide commands to navigate the file system and launch applications.

For example, in Unix you can change directories with cd, display the contents of a directory with ls, and launch a web browser by typing (for example) firefox.

Any program that you can launch from the shell can also be launched from Python using a pipe object, which represents a running program.

For example, the Unix command ls -l normally displays the contents of the current directory in long format. You can launch ls with os.popen:

In [8]:
```
cmd = 'ls -l'
fp = os.popen(cmd)
```

popen is deprecated now, which means we are supposed to stop using it and start using the subprocess module. But for simple cases, I find subprocess more complicated than necessary. So I am going to keep using popen until they take it away

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the ls process one line at a time with readline or get the whole thing at once with read:

In [9]:
```
res = fp.read()
```

In [10]:
```
res
```

Out[10]:
```
'total 6221\n-rw-r--r-- 1 Ankit19.Gupta 1049089  927445 Dec 25 01:10 158-0.txt\n-rw-r--r-- 1 Ankit19.Gupta 1049089   19909 Mar  6 13:52 Chapter_10_Lists.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   44071 Mar  6 13:54 Chapter_10_Lists.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   31854 Mar  6 15:12 Chapter_11_Dictionary.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   56848 Mar  6 15:13 Chapter_11_Dictionary.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   34636 Mar  6 16:05 Chapter_12_Tuples.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   57912 Mar  6 16:06 Chapter_12_Tuples.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   38575 Mar  6 16:41 Chapter_13_Files.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   39694 Dec 24 16:39 Chapter_14_Classes_and_objects.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   11364 Dec 24 17:29 Chapter_15_Classes_and_functions.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   31550 Dec 24 23:03 Chapter_16_Classes_and_methods.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   92734 Dec 25 01:48 Chapter_17_Inheritance.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089    7423 Mar  5 23:06 Chapter_1_The_way_of_the_program.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   32432 Mar  6 00:53 Chapter_1_The_way_of_the_program.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   13638 Mar  6 01:22 Chapter_2_Variables_expressions_and_statements.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   44328 Mar  6 01:23 Chapter_2_Variables_expressions_and_statements.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   18481 Mar  6 10:51 Chapter_3_Functions.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   44175 Mar  6 10:52 Chapter_3_Functions.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089    6359 Mar  6 11:10 Chapter_4_Case_study_interface_design.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   31557 Mar  6 11:18 Chapter_4_Case_study_interface_design.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089    6636 Mar  6 11:37 Chapter_5_Conditionals_and_recursion.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   30574 Mar  6 11:39 Chapter_5_Conditionals_and_recursion.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089    3035 Mar  6 11:42 Chapter_6_Fruitful_functions.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   23841 Mar  6 11:43 Chapter_6_Fruitful_functions.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089    4318 Mar  6 11:54 Chapter_7_Iteration.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   28156 Mar  6 11:55 Chapter_7_Iteration.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089   11888 Mar  6 12:09 Chapter_8_Strings.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089   36587 Mar  6 12:09 Chapter_8_Strings.pdf\n-rw-r--r-- 1 Ankit19.Gupta 1049089 2281243 Mar  6 13:25 Chapter_9_Case_Study.ipynb\n-rw-r--r-- 1 Ankit19.Gupta 1049089 1178419 Mar  6 13:30 Chapter_9_Case_Study.pdf\ndrwxr-xr-x 1 Ankit19.Gupta 1049089       0 Dec 12 13:36 __pycache__\n-rw-r--r-- 1 Ankit19.Gupta 1049089      23 Dec 12 13:12 captions.bak\n-rw-r--r-- 1 Ankit19.Gupta 1049089      40 Dec 12 13:13 captions.dat\n-rw-r--r-- 1 Ankit19.Gupta 1049089      23 Dec 12 13:15 captions.dir\n-rw-r--r-- 1 Ankit19.Gupta 1049089    2428 Nov 11 18:10 mypolygon.py\n-rw-r--r-- 1 Ankit19.Gupta 1049089       2 Dec 12 13:34 output.txt\n-rw-r--r-- 1 Ankit19.Gupta 1049089     132 Dec 12 13:35 wc.py\n-rw-r--r-- 1 Ankit19.Gupta 1049089 1130294 Nov 19 17:26 words.txt\n'
```

When you are done, you close the pipe like a file:

In [47]:
```
stat = fp.close()
print(stat)
```

```
None
```

The return value is the final status of the ls process; None means that it ended normally (with no errors).

For example, most Unix systems provide a command called md5sum that reads the contents of a file and computes a “checksum”. You can read about MD5 at http://en.wikipedia.org/wiki/Md5. This command provides an efficient way to check whether

two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run md5sum from Python and get the result:

In [50]:
```
filename = 'book.tex'
cmd = 'md5sum ' + filename
fp = os.popen(cmd)
res = fp.read()
stat = fp.close()
print(res)
```

In [51]:
```
print(stat)
```

```
1
```

Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named wc.py with the following code:

In [2]:
```
import wc
```

```
6
```

Now you have a module object wc:

In [3]:
```
wc
```

Out[3]:
```
<module 'wc' from 'C:\\Users\\ankit19.gupta\\OneDrive - Reliance Corporate IT Park Limited\\Desktop\\Practice_Code\\Python_Practice\\thinkpython\\wc.py'>
```

In [4]:
```
wc.linecount('wc.py')
```

Out[4]:
```
6
```

So that's how you write modules in Python

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

In [7]:
```
if __name__ == '__main__':
    print(wc.linecount('wc.py'))
```

```
6
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value __main__; in that case, the test code runs.

Otherwise, if the module is being imported, the test code is skipped.

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

In [8]:
```
s = '1 2\t 3\n 4'
print(s)
print(repr(s))
```

```
1 2	 3
 4
'1 2\t 3\n 4'
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences.

In [ ]:
```

```