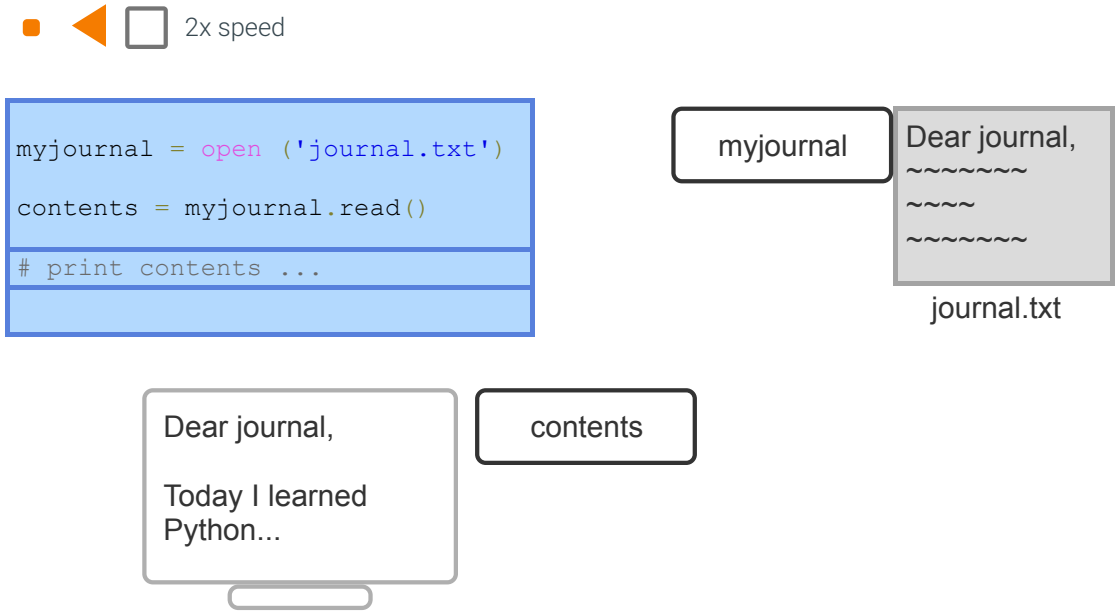
12.1 Reading files

A common programming task is to get input from a file using the built-in ***open()*** function rather than from a user typing on a keyboard.

PARTICIPATION ACTIVITY 12.1.1: Reading text from a file.



The open function creates a file object. The read function saves the content of the file as a string.

Feedback?

Assume a text file exists named "myfile.txt" with the contents shown (created, for example, using Notepad on a Windows computer or using TextEdit on a Mac computer).

Figure 12.1.1: Creating a file object and reading text.

```
print('Opening file myfile.txt.')
f = open('myfile.txt')  # create file
```

Contents of myfile.txt:

```
Because he's the hero Gotham
deserves,
```

```
object

print('Reading file myfile.txt.')
contents = f.read()  # read file text into
a string

print('Closing file myfile.txt.')
f.close()  # close the file

print('\nContents of myfile.txt:\n',
contents)
```

```
but not the one it needs right
now.
```

Program output:

```
Opening file myfile.txt.
Reading file myfile.txt.
Closing file myfile.txt.

Contents of myfile.txt:
Because he's the hero Gotham
deserves,
but not the one it needs right
now.
```

Feedback?

The open() built-in function requires a single argument that specifies the path to the file. Ex: `open('myfile.txt')` opens myfile.txt located in the same directory as the executing script. Full path names can also be specified, as in `open('C:\\Users\\BWayne\\tax_return.txt')`. The ***file.close()*** method closes the file, after which no more reads or writes to the file are allowed.

The most common methods to read text from a file are file.read() and file.readlines(). The ***file.read()*** method returns the file contents as a string. The ***file.readlines()*** method returns a list of strings, where the first element is the contents of the first line, the second element is the contents of the second line, and so on. Both methods can be given an optional argument that specifies the number of bytes to read from the file. Each method stops reading when the end-of-file (***EOF***) is detected, which indicates no more data is available.

A third method, file.readline(), returns a single line at a time, which is useful when dealing with large files where the entire file contents may not fit into the available system memory.

PARTICIPATION ACTIVITY 12.1.2: Opening files and reading text.

1) Complete the statement to open the file "readme.txt" for reading.

```
my_file =
open('readme.txt')
```

Check Show answer

Correct

open('readme.txt')

The open() function opens the specified file and returns a new file object.

Check Show answer

2) Complete the statement to read up to 500 bytes from "readme.txt" into the contents variable.

```
my_file =
open('readme.txt')

contents =
my_file.read(500)

# ...
```

Check Show answer

Correct

my_file.read(500)

The optional argument to the file.read() method limits the number of bytes read.

3) Complete the program by echoing the second line of "readme.txt"

```
my_file =
open('readme.txt')

lines =
my_file.readlines()

print(
lines[1]
)

# ...
```

Check Show answer

Correct

lines[1]

my_file.readlines() reads the contents of readme.txt and stores each line as an element in the "lines" list.

Feedback?

One of the most common programming tasks is to read data from a file and then process that data to produce a useful result. Sometimes the data is a string, like in the example above, but often the data is a numeric value. Each unique data value is often placed on its own line. Thus, a program commonly 1) reads in the contents of a file, 2) iterates over each

own line. Thus, a program commonly 1) reads in the contents of a file, 2) iterates over each line to process data values, and 3) computes some value, such as the average value.

Figure 12.1.2: Calculating the average of data values stored in a file.

The file "mydata.txt" contains 100 integers, each on its own line:

```
# Read file contents
print ('Reading in data...')
f = open('mydata.txt')
lines = f.readlines()
f.close()

# Iterate over each line
print('\nCalculating average...')
total = 0
for ln in lines:
    total += int(ln)

# Compute result
avg = total/len(lines)
print('Average value:', avg)
```

Contents of mydata.txt:

```
105
65
78
...
```

Program output:

```
Reading in data...

Calculating average...
Average value: 83
```

Feedback?

Iterating over each line of a file is so common that file objects support iteration using the `for ... in` syntax. The below example echoes the contents of a file:

Figure 12.1.3: Iterating over the lines of a file.

```
"""Echo the contents of a file."""
f = open('myfile.txt')

for line in f:
    print(line)

f.close()
```

Feedback?

12.2 Writing files

Programs often write to a file to store data permanently. The ***file.write()*** method writes a string argument to a file.

Figure 12.2.1: Writing to a file.

```
f = open('myfile.txt', 'w')  # Open file
f.write('Example string:\n  test...')  # Write string
f.close()  # Close the file
```

Final contents of myfile.txt:

```
Example string:
  test...
```

Feedback?

The write() method accepts a string argument only. Integers and floating-point values must first be converted using str(), as in `f.write(str(5.75))`.

Figure 12.2.2: Numeric values must be converted to strings.

```
num1 = 5
num2 = 7.5
num3 = num1 + num2

f = open('myfile.txt', 'w')
f.write(str(num1))
f.write(' + ')
f.write(str(num2))
f.write(' = ')
f.write(str(num3))
f.close()
```

Final contents of myfile.txt:

```
5 + 7.5 = 12.5
```

Feedback?

When writing to a file, the mode of the file must be explicitly set in the open() function call. A "mode" indicates how a file is opened, e.g., whether or not writing to the file is allowed, if

mode indicates how a file is opened, e.g., whether or not writing to the file is allowed, if existing contents of the file are overwritten or appended, etc. The most used modes are 'r' (read) and 'w' (write). The mode is specified as the second argument in a call to open(), e.g., open('myfile.txt', 'w') opens myfile.txt for writing. If mode is not specified the default is 'r'.

The below table lists common file modes:

Table 12.2.1: Modes for opening files.

Mode	Description	Allow read?	Allow write?	Create missing file?	Overwrite file?
'r'	Open the file for reading.	Yes	No	No	No
'w'	Open the file for writing. If file does not exist then the file is created. Contents of an existing file are overwritten.	No	Yes	Yes	Yes
'a'	Open the file for appending. If file does not exist then the file is created. Writes are added to end of existing file contents.	No	Yes	Yes	No

Feedback?

- Read mode 'r' opens a file for reading. If the file is missing, then an error will occur.
- Write mode 'w' opens a file for writing. If the file is missing, then a new file is created. Contents of any existing file are overwritten.
- Append mode 'a' opens a file for writing. If the file is missing, then a new file is created. Writes to the file are appended to the end of an existing file's contents.

Additionally, a programmer can add a '+' character to the end of a mode, like 'r+' and 'w+' to specify an *update* mode. Update modes allow for both reading and writing of a file at the same time.

PARTICIPATION ACTIVITY 12.2.1: File modes.

For each question, complete the statement to open myfile.txt with the appropriate mode.

1) Data will be appended to the end of existing contents.

```
f = open('myfile.txt', 'a')
```

Check Show answer

Correct

'a'

The 'a' mode opens a file for writing, and writes are appended to the end of existing contents.

2) Data will be written to a new file.

```
f = open('myfile.txt', 'w')
```

Check Show answer

Correct

'w'

The 'w' mode will open a new file for writing.

3) Existing contents will be read, and new data will be appended.

```
f = open('myfile.txt', 'a+')
```

Check Show answer

Correct

'a+'

The 'a+' mode opens the file for both reading and writing, appending new writes.

Feedback?

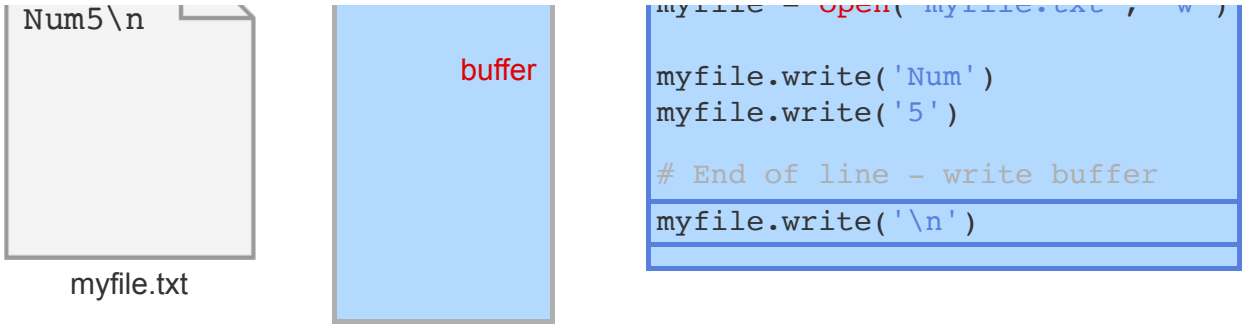
Output to a file is buffered by the interpreter before being written to the computer's hard disk. By default, data is line-buffered, e.g., data is written to disk only when a newline character is output. Thus, there may be a delay between a call of write() and that data actually being written to the disk. The following illustrates:

PARTICIPATION ACTIVITY 12.2.2: Output is buffered.

1 2 3 4 2x speed

Interpreter

```
myfile = open('myfile.txt', 'w')
```



Statement myfile.write('\n') executes. The interpreter stores '\n' in a buffer. Writing a newline causes the buffer to be written to the file, so 'Num5' is placed in myfile.txt.

Feedback?

A programmer can toggle buffering on/off or specify a buffer size with the optional *buffering* argument to the open() function. Passing 0 disables buffering (valid only for binary files, discussed in another section), passing 1 enables the default line-buffering, and a value > 1 sets a specific buffer size in bytes. For example, `f = open('myfile.txt', 'w', buffering=100)` will write the output buffer to disk every 100 bytes.

The ***flush()*** file method can be called to force the interpreter to flush the output buffer to disk. Additionally, the os.fsync() function may have to be called on some operating systems. Closing an open file also flushes the output buffer.

Figure 12.2.3: Using flush() to force an output buffer to write to disk.

```
import os

# Open a file with default line-buffering.
f = open('myfile.txt', 'w')

# No newline character, so not written to disk immediately
f.write('Write me to a file, please!')

# Force output buffer to be written to disk
f.flush()
os.fsync(f.fileno())

# ...
```

Feedback?

PARTICIPATION ACTIVITY 12.2.3: Writing output.

1) The statement `f.write(10.0)` always produces an error.
   - (•) True
   - ( ) False

Correct

The write() method requires a string argument. The programmer should write `f.write(str(10.0))` instead.

2) The write() method immediately writes data to a file.
   - ( ) True
   - (•) False

Correct

Output is buffered, thus there may be a delay between calling the write method and data being written to the disk.

3) The flush() method (and perhaps os.fsync() as well) forces the output buffer to write to disk.
   - (•) True
   - ( ) False

Correct

Data in the output buffer may not be written to disk until a newline character is encountered or if the buffer is filled.

Feedback?

12.3 Interacting with file systems

A program commonly needs to interact with the computer's file system, such as to get the size of a file or to open a file in a different directory. The computer's operating system, such as Windows or Mac OS X, controls the file system, and a program must use functions supplied by the operating system to interact with files. The Python standard library's ***os module*** provides an interface to operating system function calls and is thus a critical piece of a Python programmer's toolbox.

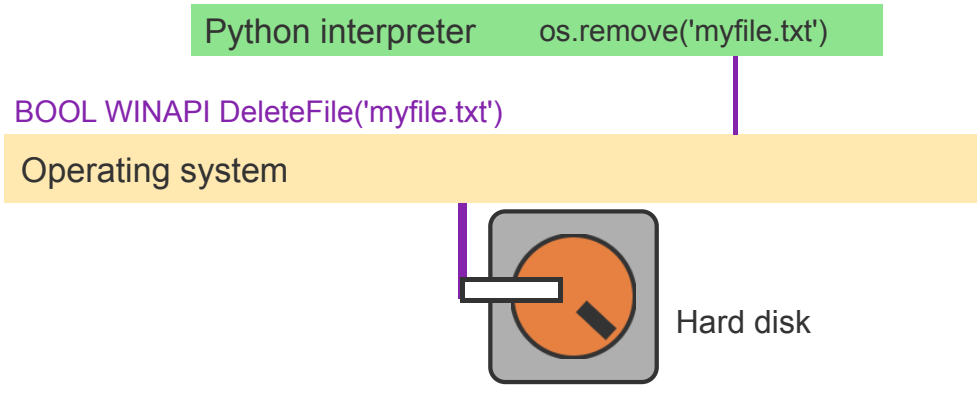
PARTICIPATION ACTIVITY 12.3.1: Using the os module to interact with the file system.

1 2 3 4 2x speed

```
import os

# ...

my_file = open('myfile.txt', 'r')
# ...
file_info = os.stat('myfile.txt')
# ...
os.remove('myfile.txt')
```



When the os remove() method executes, the interpreter calls on the operating system function DeleteFile('myfile.txt'), which removes myfile.txt from the hard disk.

Feedback?

A programmer should consider the ***portability*** of a program across different operating systems to avoid scenarios where the program behaves correctly on the programmer's computer but crashes on another. Portability must be considered when reading and writing files outside the executing program's directory since file path representations often differ between operating systems. For example, on Windows the path to a file is represented as "subdir\\bat_mobile.jpg", but on a Mac is "subdir/bat_mobile.jpg". The character between directories, e.g., "\\"or "/", is called the ***path separator***, and using the incorrect path separator may result in that file not being found.[1]

A common error is to reduce a program's portability by hardcoding file paths as string literals with operating system specific path separators. To help reduce such errors, good practice is to use the os.path module, which contains many portable functions for handling file paths. One of the most useful functions is os.path.join(), which concatenates the arguments using the correct path separator for the current operating system. Instead of writing the literal `path = "subdir\\bat_mobile.jpg"`, a programmer should write `path = os.path.join('subdir', 'bat_mobile.jpg')`, which will result in `"subdir\\bat_mobile.jpg"` on Windows and `"subdir/bat_mobile.jpg"` on Linux/Mac.

Figure 12.3.1: Using os.path.join() to create a portable file path string.

The program below echoes the contents of logs stored in a hierarchical directory structure organized by date, using the os.path module to build a file path string that is portable across operating systems.

```
import os
import datetime

curr_day = datetime.date(1997, 8, 29)

num_days = 30
for i in range(num_days):
    year = str(curr_day.year)
    month = str(curr_day.month)
    day = str(curr_day.day)

    # Build path string using current OS
path separator
    file_path = os.path.join('logs', year,
month, day, 'log.txt')

    f = open(file_path, 'r')
```

Output on Windows:

```
logs\\1997\\8\\29\\log.txt:
# ...
logs\\1997\\8\\30\\log.txt:
# ...
# ...
logs\\1997\\9\\28\\log.txt:
# ...
```

Output on Linux:

```
logs/1997/8/29/log.txt:  #
...
logs/1997/8/30/log.txt:  #
```

```
    print('%s: %s' % (file_path, f.read()))
    f.close()

    curr_day = curr_day +
datetime.timedelta(days=1)
```

```
...
# ...
logs/1997/9/28/log.txt:  #
...
```

Feedback?

On Windows systems, when using os.path.join() with a full path such that the first argument is a drive letter (e.g., 'C:' or 'D:'), the separator must be included with the drive letter. For example, `os.path.join('C:\\', 'subdir1', 'myfile.txt')` returns the string "C:\\subdir1\\myfile.txt".

The inverse operation, splitting a path into individual tokens, can be done using the str.split() method. Ex:
`tokens = 'C:\\Users\\BWayne\\tax_return.txt'.split(os.path.sep)`
returns ['C:', 'Users', 'BWayne', 'tax_return.txt']. ***os.path.sep*** stores the path separator for the current operating system.

PARTICIPATION ACTIVITY 12.3.2: Portable file paths.

1) Fill in the arguments to os.path.join to assign file_path as "subdir\\output.txt" (on Windows).

```
file_path = os.path.join(
 'subdir', 'output.txt'
)
```

Check Show answer

Correct

'subdir', 'output.txt'

On Windows, file_path would be "subdir\\output.txt". On Linux/Mac, file_path would be "subdir/output.txt".

2) What is returned by os.path.join('sounds', 'cars', 'honk.mp3') on Windows? Use quotes in the answer.

sounds\\cars\\honk.mp3

Check Show answer

Correct

"sounds\\cars\\honk.mp3"

Each argument is joined together between the Windows path separator "\\".

3) What is returned by os.path.join('sounds', 'cars', 'honk.mp3') on Mac OS X? Use quotes in the answer.

sounds/cars/honk.mp3

Check Show answer

Correct

"sounds/cars/honk.mp3"

Each argument is joined together between the Mac path separator "/".

Feedback?

The os and os.path modules contain other helpful functions, such as checking if a given path is a directory or a file, getting the size of a file, obtaining a file's extension (e.g., .txt, .doc, .pdf), creating and deleting directories, etc. Some of the most commonly used functions are listed below:

- os.path.split(path) – *Splits a path into a 2-tuple (head, tail), where tail is the last token in the path string and head is everything else.*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne',
'batsuit.jpg')
print(os.path.split(p))
```

```
('C:\\Users\\BWayne',
'batsuit.jpg')
```

- os.path.exists(path) – *Returns True if path exists, else returns False.*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne',
'batsuit.jpg')
if os.path.exists(p):
    print('Suit up...')
else:
    print('The Lamborghini then?')
```

If file exists:

```
Suit up...
```

If file does not exist:

```
The Lamborghini
then?
```

- os.path.isfile(path) – *Returns True if path is an existing file, and false otherwise (e.g., path is a directory).*

```
import os
p = os.path.join('C:\\', 'Users', 'BWayne', 'bat_chopper')
if os.path.isfile(p):
    print('Found a file...')
else:
    print('Not a file...')
```

If path is a file:

```
Found a file...
```

If path is not a file:

```
Not a file...
```

- os.path.getsize(path) – *Returns the size in bytes of path.*

```
import os
```

```
p = os.path.join('C:\\', 'Users', 'Bwayne',
'batsuit.jpg')
print('Size of file:', os.path.getsize(p), 'bytes')
```

```
Size of file: 65544
bytes
```

Explore the links at the end of the section to see all of the available functions in the os and os.path modules.

PARTICIPATION ACTIVITY 12.3.3: Path name manipulation functions from os.path.

1) What is the output of the following program?

```
import os
p =
'C:\\Programs\\Microsoft\\msword.exe'
print(os.path.split(p))
```

- ['C:\\', 'Programs', 'Microsoft', 'msword.exe']
- ('C:\\Programs\\Microsoft', 'msword.exe')
- ('C:', 'Programs', 'Microsoft', 'msword.exe')

Correct

os.path.split() returns a tuple containing the final item in the path as the second element, and everything else as the first element.

2) What does the call `os.path.isfile('C:\\Program Files\\')` return?

- True
- False

Correct

os.path.isfile() returns True only if the argument is a file.

3) What does os.path.getsize(path_str) return?

- The length of the path_str string.
- The combined size of all files in path_str directory.
- The size in bytes of

Correct

os.path.getsize() yields the size in bytes of the file at the given path argument.

the file at path_str.

Feedback?

A programmer commonly wants to check every file and/or subdirectory of a specific part of the file system. Consider the following directory structure, organized by year, month, and day:

Figure 12.3.2: Directory structure organized by date.

```
logs/
    2009/
        April/
                1/
                  log.txt
                  words.doc
        January/
               15/
                 log.txt
               21/
                 log.txt
                 temp23.pdf
               24/
                 presentation.ppt
    2010/
        March/
             3/
              log.txt
             7/
              music.mp3
```

Feedback?

The ***os.walk()*** function 'walks' a directory tree like the one above, visiting each subdirectory in the specified path. The following example walks a user-specified year of the above directory tree.

Figure 12.3.3: Walking a directory tree.

```
import os

year = input('Enter year: ')
path = os.path.join('logs', year)
print()

for dirname, subdirs, files in os.walk(path):
```

```
    print(dirname, 'contains subdirectories:', subdirs, end=' ')
    print('and the files:', files)
```

```
Enter year:2009

logs\2009 contains subdirectories: ['April', 'January'] and the files: []
logs\2009\April contains subdirectories: ['1'] and the files: []
logs\2009\April\1 contains subdirectories: [] and the files: ['log.txt',
'words.doc']
logs\2009\January contains subdirectories: ['15', '21', '24'] and the files: []
logs\2009\January\15 contains subdirectories: [] and the files: ['log.txt']
logs\2009\January\21 contains subdirectories: [] and the files: ['log.txt',
'temp23.pdf']
logs\2009\January\24 contains subdirectories: [] and the files:
['presentation.ppt']
```

Feedback?

The os.walk() function is used as the iterable object in a for loop that yields a 3-tuple for each iteration.[2] The first item *dirname* contains the path to the current directory. The second item *subdirs* is a list of all the subdirectories of the current directory. The third item *files* is a list of all the files residing in the current directory.

A programmer might use os.walk() when searching for specific files within a directory tree, and the exact path is unknown. Another common task is to filter files based on their file extensions (.pdf, .txt, etc.), which are a convention used to indicate the type of data that a file holds.

zyDE 12.3.1: Filtering files using os.walk().

Run the program and observe the above date-organized directory tree.

Modify the program to print the contents of any file called log.txt in a given year's subdirecto ignoring any other file.

Load default template..

```
1
2 import os
3
4 year = input('Enter year: ')
5 path = os.path.join('logs', year)
6 print()
7 for dirname, subdirs, files in os.walk(path):
8     print(dirname, 'contains subdirectories:', subdirs, end=' ')
```

```
    print('and the files:', files)
```

2009

Run

```
Enter year:
logs/2009 contains subdirectories: ['April', 'January'] and the
files: []
logs/2009/April contains subdirectories: ['1'] and the files: []
logs/2009/April/1 contains subdirectories: [] and the files:
['words.doc', 'log.txt']
logs/2009/January contains subdirectories: ['15', '24', '21']
and the files: []
logs/2009/January/15 contains subdirectories: [] and the files:
['log.txt']
logs/2009/January/24 contains subdirectories: [] and the files:
['presentation.ppt']
```

Feedback?

Exploring further:

- The os module: Miscellaneous operating system interfaces
- The os.path module: Common pathname manipulations

(*1) Unix-based operating systems, like Linux and Mac OS X, will not recognize paths using the windows "\\" separator. Generally, Windows recognizes both "/" and "\\".

(*2) os.walk() actually returns a special object called a generator, which is discussed elsewhere.

12.4 Binary data

Some files consist of data stored as a sequence of bytes, known as ***binary data***, that is not encoded into human-readable text using an encoding like ASCII or UTF-8. Images, videos, and PDF files are examples of the types of files commonly stored as binary data. Opening such a file with a text editor displays text that is incomprehensible to humans because the text editor attempts to encode raw byte values into readable characters.

A ***bytes object*** is used to represent a sequence of single byte values, such as binary data read from a file. Bytes objects are immutable, just like strings, meaning the value of a bytes object cannot change once created. A byte object can be created using the ***bytes()*** built-in function:

- *bytes('A text string', 'ascii')* – creates a sequence of bytes by encoding the string using ASCII.
- *bytes(100)* – creates a sequence of 100 bytes whose values are all 0.
- *bytes([12, 15, 20])* – creates a sequence of 3 bytes with values from the list.

Alternatively, a programmer can write a bytes literal, similar to a string literal, by prepending a 'b' prior to the opening quote:

Figure 12.4.1: Creating a bytes object using a bytes literal.

```
my_bytes = b'This is a bytes literal'

print(my_bytes)
print(type(my_bytes))
```

```
b'This is a bytes literal'
<class 'bytes'>
```

Feedback?

A programmer can specify raw byte values in a string or bytes literal using the \x escape character preceding the hexadecimal value that describes the value of the byte. In the example below, the raw byte values 0x31 through 0x39 are automatically converted to the corresponding ASCII encoded values 1 - 9 when printed.

Figure 12.4.2: Byte string literals

Figure 12.4.2: Byte string literals.

```
print(b'123456789 is the same as \x31\x32\x33\x34\x35\x36\x37\x38\x39')
```

```
b'123456789 is the same as 123456789'
```

Feedback?

Programs can also access files using a ***binary file mode*** by adding a "b" character to the end of the mode string in a call to open(), as in `open('myfile.txt', 'rb')`. When using binary file mode "b" on a Windows computer, newline characters "\n" in the file are *not* automatically mapped to the Windows format "\r\n". In normal text mode, i.e., when not using the "b" binary mode, Python performs this translation of line-endings as a helpful feature, easing compatibility issues between Windows and other operating systems. In binary mode, the translation is not done because inserting additional characters would corrupt the binary data. On non-Windows systems, binary mode has no effect.

When a file is opened using a binary mode, the file.read() method returns a bytes object instead of a string. Also, the file.write() method expects a bytes argument.

PARTICIPATION ACTIVITY 12.4.1: Binary Data.

1) Open "data.txt" as read-only in binary mode.

```
f = open('data.txt', 'rb')
```

Check Show answer

Correct

'rb'

Reading the contents of a file opened with binary mode will not map "\n" to "\r\n" on Windows.

2) Open "myfile.txt" as read-only in binary mode.

```
f = open('myfile.txt', 'rb')
```

Check Show answer

Correct

open('myfile.txt', 'rb')

Reading the contents of a file opened with binary mode will not map "\n" to "\r\n" on Windows. Reading from and writing to binary files requires the use of bytes objects.

3) Assign x to a bytes object with a single byte whose

Correct

a single byte whose hexadecimal value is 0x1a. Use a bytes literal.

```
x =
```

b'\x1a'

Check Show answer

b'\x1a'

The value 0x1a does not correspond to a readable ASCII character, thus the \x escape must be used to specify the value in the literal.

4) Assign x to a bytes object containing three bytes with hexadecimal values 0x05, 0x15, and 0xf2. Use a bytes literal.

```
x =
```

b'\x05\x15\xf2'

Check Show answer

Correct

b'\x05\x15\xf2'

The values do not correspond to readable ASCII characters, thus the \x escape must be used to specify the values in the literal.

Feedback?

Consider a file ball.bmp that contains the following image:

The ball.bmp file contains binary data in a format commonly called a bitmap (hence the .bmp extension at the end of the file name). Opening and reading the file with a binary mode creates a new bytes object consisting of the exact sequence of bytes found in the file's contents.

Figure 12.4.3: Inspecting the binary contents of an image file.

```
f = open('ball.bmp', 'rb')  # Open in binary mode using 'b'

# Read image binary data
contents = f.read()

print('Contents of ball.bmp\n')
print(contents)

f.close()
```

Abbreviated output:

```
Contents of ball.bmp:

b'BMb\xe6\x00\x00\x00\x00\x00\x006\x04\x00\x00(\x00\x00\x00,\x01\x00\x00
\xc1\x00\x00\x00\x01\x00\x08\x00\x00\x00\x00\x00,\xe2\x00\x00\xc4'
```

Feedback?

The `print(contents)` statement displays the value of contents, converting each byte to human-readable character if that byte's value is a readable ASCII character (less than 128). The first portion of the file's contents is shown in the output, though it is abbreviated because the image contains about 27,000 bytes. Note how the first 14 bytes of the bitmap file is "BMb\xe6\x00\x00\x00\x00\x00\x006\x04\x00\x00". This sequence constitutes the ***header*** of the binary file, which describes the bitmap's contents. The first 2 bytes "BM" indicates the type of bitmap. The following 4 bytes "b\xe6\x00\x00" indicates the size of the bitmap. The sequence "6\x04\x00\x00" indicates where in the file the RGB (red-green-blue) values for each pixel in the image are stored.

Try 12.4.1: Altering a BMP image file.

The following program reads in ball.bmp, overwrites a portion of the image with new pixel colors, and creates a new image file. Download the above image (right click the image --> save), and then run the program on your own computer, creating a new, altered version of ball.bmp. Try changing the alterations made by the program to get different colors.

```
import struct

ball_file = open('ball.bmp', 'rb')
ball_data = ball_file.read()
ball_file.close()

# BMP image file format stores location
# of pixel RGB values in bytes 10-14
```

```
# of pixel RGB values in bytes 10-14
pixel_data_loc = ball_data[10:14]

# Converts byte sequence into integer object
pixel_data_loc = struct.unpack('<L', pixel_data_loc)[0]

# Create sequence of 3000 red, green, and yellow pixels each
new_pixels = b'\x01'*3000 + b'\x02'*3000 + b'\x03'*3000

# Overwrite pixels in image with new pixels
new_ball_data = ball_data[:pixel_data_loc] + \
                new_pixels + \
                ball_data[pixel_data_loc + len(new_pixels):]

# Write new image
new_ball_file = open('new_ball.bmp', 'wb')
new_ball_file.write(new_ball_data)
new_ball_file.close()
```

Altered image

Feedback?

The ***struct*** module is a commonly used Python standard library module for *packing* values into sequences of bytes and *unpacking* sequences of bytes into values (like integers and strings). The ***struct.pack()*** function packs values such as strings and integers into sequences of bytes:

Figure 12.4.4: Packing values into byte sequences.

```
import struct

print('Result of packing 5:', end='
')
print(struct.pack('>h', 5))

print('Result of packing 256:',
end=' ')
print(struct.pack('>h', 256))

print('Result of packing 5 and
256:', end=' ')
print(struct.pack('>hh', 5, 256))
```

```
Result of packing 5: b'\x00\x05'
Result of packing 256: b'\x01\x00'
Result of packing 5 and 256:
b'\x00\x05\x01\x00'
```

Feedback?

The first argument to struct.pack() is a format string that describes how the following arguments should be converted into bytes. The "<" character indicates the ***byte-order***, or

arguments should be converted into bytes. The "<" character indicates the **byte order**, or endianness, of the conversion, which determines whether the most-significant or least-significant byte is placed first in the byte sequence. ">" places the most-significant byte first (big-endian), and "<" sets the least-significant byte first. The "h" character in the format strings above describe the type of object being converted, which most importantly determines how many bytes are used when packing the value. "h" describes the value being converted as a 2-byte integer; other common format characters are "b" for a 1-byte integer, "I" for a 4-byte integer, and "s" for a string. Explore the links at the end of the section for more information on the struct module.

The ***struct.unpack()*** module performs the reverse operation of struct.pack(), unpacking a sequence of bytes into a new object. Unpacking always returns a tuple with the results, even if only unpacking a single value:

Figure 12.4.5: Unpacking values from byte sequences.

```
import struct

print('Result of unpacking %s:' %
repr('\x00\x05'), end=' ')
print(struct.unpack('>h', b'\x00\x05'))


print('Result of unpacking %s:' %
repr('\x01\x00'), end=' ')
print(struct.unpack('>h', b'\x01\x00'))

print('Result of unpacking %s:' %
repr('\x00\x05\x01\x00'), end=' ')
print(struct.unpack('>hh',
b'\x00\x05\x01\x00'))
```

```
Result of unpacking
'\x00\x05': (5,)
Result of unpacking
'\x01\x00': (256,)
Result of unpacking
'\x00\x05\x01\x00': (5, 256)
```

Feedback?

PARTICIPATION ACTIVITY 12.4.2: The struct module.

1) Complete the statement to pack an integer variable "my_num" into a 2-byte sequence assigned to my_bytes. Use the byte ordering given by ">".

Correct

'>h', my_num

The my_num variable is packed into two bytes and assigned to my_bytes.

ordering given by ">".

```
my_bytes = struct.pack(
'>h', my_num
)
```

Check Show answer

2) Assume that variable my_bytes is b"\x00\x04\xff\x00". Complete the statement to assign my_num to the 4-byte integer obtained by unpacking my_bytes. Use the byte ordering given by ">".

```
my_num = struct.unpack(
'>I', my_bytes
)
```

Check Show answer

Correct

'>I', my_bytes

The four bytes of my_bytes are unpacked into a new integer called my_num.

Feedback?

Exploring further:

- The bytes object
- The bytearray type -- mutable sequence of bytes
- The struct module: converting strings into packed binary data

12.5 Command-line arguments and files

The location of an input file or output file may not be known before writing a program. Instead, a program can use command-line arguments to allow the user to specify the location of an input file as shown in the following program. Assume two text files exist named "myfile1.txt" and "myfile2.txt" with the contents shown. The sample output shows the results when executing the program for either input file and for an input file that does not exist.

Figure 12.5.1: Using command-line arguments to specify the name of an input file.

```
import sys
import os

if len(sys.argv) != 2:
    print('Usage: %s input_file' %
(sys.argv[0]))
    sys.exit(1)  # 1 indicates error

print('Opening file %s.' % sys.argv[1])

if not os.path.exists(sys.argv[1]):  # Make sure
file exists
    print('File does not exist.')
    sys.exit(1)  # 1 indicates error

f = open(sys.argv[1], 'r')

# Input files should contain two integers on
separate lines

print('Reading two integers.')
num1 = int(f.readline())
num2 = int(f.readline())

print('Closing file %s' % sys.argv[1])
f.close()  # Done with the file, so close it

print('\nnum1: %d' % num1)
print('num2: %d' % num2)
print('num1 + num2: %d' % (num1 + num2))
```

myfile1.txt:
```
5
10
```

myfile2.txt:
```
-34
7
```

```
>python my_script.py
myfile1.txt
Opening file myfile1.txt.
Reading two integers.
Closing file myfile1.txt

num1: 5
num2: 10
num1 + num2: 15


>python my_script.py
myfile2.txt
Opening file myfile2.txt.
Reading two integers.
Closing file myfile2.txt

num1: -34
num2: 7
num1 + num2: -27


>python my_script.py
myfile3.txt
Opening file myfile3.txt.
File does not exist.
```

Feedback?

PARTICIPATION ACTIVITY 12.5.1: Filename command line arguments.

1) A script "myscript.py" has two command line arguments, one for an input file and a second for an output file. Type a command to run the program with input file "infile.txt" and output file "out".

```
> python
```

myscript.py infile.txt out

Check Show answer

Correct

myscript.py infile.txt out

python is called first to start the Python interpreter, followed by the script name and the two arguments.

2) For a program run as "python scriptname data.txt", what is sys.argv[1]? Do not use quotes in the answer.

data.txt

Check Show answer

Correct

data.txt

sys.argv[0] is the script's name, sys.argv[1] is the next argument, which is data.txt.

Feedback?

12.6 The 'with' statement

A ***with statement*** can be used to open a file, execute a block of statements, and automatically close the file when complete.

Construct 12.6.1: The with statement.

```
with open('myfile.txt', 'r') as myfile:
    # Statement-1
    # Statement-2
    # ...
    # Statement-N
```

Feedback?

Above, the file object returned by open() is bound to myfile. When the statements in the block complete, then myfile is closed. The with statement creates a ***context manager***, which manages the usage of a resource, like a file, by performing setup and teardown operations. For files, the teardown operation is automatic closure. Other context managers exist for other resources, and new context managers can be written by a programmer, but is out of scope for this material.

Forgetting to close a file can sometimes cause problems. For example, a file opened in write mode cannot be written to by other programs. Good practice is to use a with statement when opening files, to guarantee that the file is closed when no longer needed.

Figure 12.6.1: Using the with statement to open a file.

```
print('Opening myfile.txt')

# Open a file for reading and appending
with open('myfile.txt', 'r+') as f:
    # Read in two integers
    num1 = int(f.readline())
    num2 = int(f.readline())

    product = num1 * num2
```

```
    # Write back result on own line
    f.write('\n')
    f.write(str(product))

# No need to call f.close() - f closed automatically
print('Closed myfile.txt')
```

Feedback?

PARTICIPATION ACTIVITY 12.6.1: The with statement.

1) When using a with statement to open a file, the file is automatically closed when the statements in the block finish executing.

   - (•) True
   - ( ) False

Correct

The context manager for files performs a teardown operation when the statements complete, closing the file automatically.

2) Use of a with statement is not recommended most of the time when opening files.

   - ( ) True
   - (•) False

Correct

The with statement is recommended, because closure of the file is guaranteed.

Feedback?

12.7 Comma separated values files

Text data is commonly organized in a spreadsheet format using columns and rows. A ***comma separated values*** (csv) file is a simple text-based file format that uses commas to separate data items, called ***fields***. Below is an example of a typical csv file that contains information about student scores:

Figure 12.7.1: Contents of a csv file.

```
name,hw1,hw2,midterm,final
Petr Little,9,8,85,78
Sam Tarley,10,10,99,100
Joff King,4,2,55,61
```

Feedback?

Each line in the file above represents a row, and fields between commas on each row are in the same column as fields in the same position in each line. For example, the first row contains the items "name", "hw1", "hw2", "midterm", and "final"; the second row contains "Petr Little", "9", "8", "85" and "78". The first column contains "name", "Petr Little", "Sam Tarley", and Joff King; the second column contains "hw1", "9", "10", and "4".

The Python standard library ***csv module*** can be used to help read and write files in the csv format. To read a file using the csv module, a program must first create a *reader* object, passing a file object created via *open*. The reader object is an iterable – iterating over the reader using a for loop returns each row of the csv file as a list of strings, where each item in the list is a field from the row.

Figure 12.7.2: Reading each row of a csv file.

```
import csv

with open('grades.csv', 'r') as
csvfile:
    grades_reader =
csv.reader(csvfile, delimiter=',')

    row_num = 1
```

Echoed file contents:

```
Row #1: ['name', 'hw1', 'hw2',
'midterm', 'final']
Row #2: ['Petr Little', '9', '8',
'85', '78']
Row #3: ['Sam Tarley', '10', '10'
```

```
    for row in grades_reader:
        print('Row #%d:' % row_num,
row)
        row_num += 1
```

```
Row #3: ['Sam Tarley', '10', '10',
'99', '100']
Row #4: ['Joff King', '4', '2',
'55', '61']
```

Feedback?

The optional delimiter argument in the csv.reader() function specifies the character used in the csv file to separate fields; by default a comma is used. In some cases, the field itself may contain a comma – for example if the name of a student was specified as "lastname,firstname". In such a case, the csv file might instead use semicolons or some other rare character, e.g., Little, Petr;9;8;85;78. An alternative to changing the delimiter is to use quotes around the item containing the comma, e.g., "Little, Petr",9,8,85,78.

If the contents of the fields are numeric, then a programmer may want to convert the strings to integer or floating-point values to perform calculations with the data. The example below reads each row using a reader object and calculates a student's final score in the class:

Figure 12.7.3: Using csv file contents to perform calculations.

```
import csv

# Dictionary that maps student names to a list of
scores
grades = {}

# Use with statement to guarantee file closure
with open('grades.csv', 'r') as csvfile:
    grades_reader = csv.reader(csvfile, delimiter=',')

    first_row = True
    for row in grades_reader:
        # Skip the first row with column names
        if first_row:
            first_row = False
            continue

        ## Calculate final student grade ##

        name = row[0]

        # Convert score strings into floats
        scores = [float(cell) for cell in row[1:]]

        hw1_weighted = scores[0]/10 * 0.05
        hw2_weighted = scores[1]/10 * 0.05
        mid_weighted = scores[2]/100 * 0.40
        fin_weighted = scores[3]/100 * 0.50
```

```
Sam Tarley earned
99.6%
Petr Little earned
81.5%
Joff King earned 55.5%
```

```
        grades[name] = (hw1_weighted + hw2_weighted +
                        mid_weighted + fin_weighted) *
100

for student, score in grades.items():
    print('%s earned %.1f%%' % (student, score))
```

Feedback?

A programmer can also use the csv module to write text into a csv file, using a *writer* object. The writer object's *writerow()* and *writerows* methods can be used to write a list of strings into the file as one or more rows.

Figure 12.7.4: Writing rows to a csv module.

```
import csv

row1 = ['100', '50', '29']
row2 = ['76', '32', '330']

with open('gradeswr.csv', 'w') as csvfile:
    grades_writer = csv.writer(csvfile)

    grades_writer.writerow(row1)
    grades_writer.writerow(row2)

    grades_writer.writerows([row1, row2])
```

final gradeswr.csv contents:

```
100,50,29
76,32,330
100,50,29
76,32,330
```

Feedback?

PARTICIPATION ACTIVITY 12.7.1: Comma separated values files.

The file "myfile.csv" contains the following contents:

```
Airline,Destination,Departure time,Plane
Southwest,Phoenix,615,B747
Alitalia,Milan,1545,B757
British Airways,London,1230,A380
```

1) Complete the statement to

Correct

create a csv module reader object to read myfile.csv.

```
import csv
with open('myfile.csv',
'r') as myfile:
    csv_reader =
csv.reader(myfile)
```

Check Show answer

Correct

csv.reader(myfile)

The reader object can be used to iterate over rows of the csv file.

2) Complete the statement such that the program prints the destination of each flight in myfile.csv.

```
import csv
with open('myfile.csv', 'r')
as myfile:
    csv_reader =
csv.reader(myfile)
    for row in csv_reader:
        print(
row[1]
)
```

Check Show answer

Correct

row[1]

Iterating over the reader object yields each row of the file as a list of the fields. The destination field is in position 1 of the list.

Feedback?

12.8 Files practice

Learning resources

Following is a list of recommended resources for use when completing these exercises.

- OS-Miscellaneous Operating System Interfaces in the Python Standard Library
- os.path-Common Pathname Manipulations in the Python Standard Library
- Reading and Writing Files in the Python Tutorial
- Python File Handling from W3Schools.com
- Python-Files I/O from tutorialspoint.com

Tasks

WGU is providing additional practice exercises for all students. The best method of learning Python is to practice. This will also help you prepare for the assessment, which requires you to write code.

Note: PyFiddle version 3 or higher is highly recommended for these exercises. This ensures cross-platform compatibility and removes potential file system risks from bad code.

Below is a sample exercise. You should copy the entire code snippet into your Python editor. Your code should be placed in the area that says "student code goes here" highlighted in yellow. This is inside the function. When you run this entire code snippet, there are test cases below your code. Your output should match the expected output.

```
# Complete the function to print the first X number of characters in the given string
def printFirst(mystring, x):
# Student code goes here

# expected output: WGU
printFirst('WGU College of IT', 3)

# expected output: WGU College
printFirst('WGU College of IT', 11)
```

Task 1

Complete the function to return the directory name only from the given file name

Complete the function to return the directory name only from the given file name

```
import os

# Complete the function to return the current working directory
def getCurrentDirectory():
# Student code goes here

# expected output: /tmp
# if using PyFiddle.io otherwise it varies
print(getCurrentDirectory())
```

Task 2

Complete the function to return the directory name only from the given file name

```
import os

# Complete the function to return the directory name only from the given file name
def getDirectoryName(fileName):
# Student code goes here

# expected output: /var/www
print(getDirectoryName("/var/www/test.html"))

# expected output: /var/www/apple
print(getDirectoryName("/var/www/apple/test.html"))
```

Task 3

Complete the function to return the file name part only from the given file name

```
import os

# Complete the function to return the file name part only from the given file name
def getFileName(fileName):
# Student code goes here

# expected output: test.html
print(getFileName("/var/www/test.html"))

# expected output: names.txt
print(getFileName("/var/www/apple/names.txt"))
```

Task 4

Complete the function to return the directory name only from the given file name

```
import os

# Complete the function to create the specified file and return the file name
def createFile(filename):
# Student code goes here
```

```
# expected output: True
createFile("test.txt")
print(os.path.exists("test.txt"))
```

Task 5

Complete the function to print all files in the given directory

```
import os

# Complete the function to print all files in the given directory
def printFiles(someDirectory):
# Student code goes here

# expected output: main.py
# if using PyFiddle.io otherwise it varies
printFiles(os.getcwd())
```

Task 6

Complete the function to return FILE if the given path is a file or return DIRECTORY if the given path is a directory or return NEITHER is it's not a file or directory

```
import os

# Complete the function to return FILE if the given path is a file
# or return DIRECTORY if the given path is a directory
# or return NEITHER is it's not a file or directory
def whatIsIt(somePath):
# Student code goes here

# expected output: DIRECTORY
print(whatIsIt(os.getcwd()))

# expected output: FILE
print(whatIsIt(os.listdir(os.getcwd())[0]))

# expected output: NEITHER
print(whatIsIt('apple.pie.123.txt'))
```

Task 7

Complete the function to read the contents of the specified file and print the contents

```
import os

# Complete the function to read the contents of the specified file and print the
contents
def printFileContents(filename):
# Student code goes here

# expected output: Hello
with open("test.txt", 'w') as f:
```

```
with open("test.txt", "w") as f:
f.write("Hello")
printFileContents("test.txt")
```

Task 8

Complete the function to append the given new data to the specified file then print the contents of the file

```
import os

# Complete the function to append the given new data to the specified file then print
the contents of the file
def appendAndPrint(filename, newData):
# Student code goes here

# expected output: Hello World
with open("test.txt", 'w') as f:
f.write("Hello ")
appendAndPrint("test.txt", "World")
```