DSC 510

Week 9 File Operations

Files

Processing files is a fundamental activity in any programming language. File progressing happens in a vast majority of programs used every day including the text editor you've been using throughout the course to develop and save your Python programs. It is common for programs to open files for reading, write data to files, and append data to files.

Reading Files

Reading text from a file requires three steps:

1. Open the file for reading using the `open()` method. open() returns a file object, and is most commonly used with two arguments: open(filename, mode).
2. Read from the file (usually into a string variable).
3. Close the file `close()` method.

In order to read from a file you must create a file handle. When you're done with the file you must close the file handle.

It is best practice to open the files using the **with** keyword. The **with** keyword ensures that the file handle is closed even if an exception occurs. If you do not open a file handle using with you must explicitly close the file handle using the close() method.

Reading File Example

Opening and reading a file using “with” will create a file object. In this case the file object is called fileHandle

```
filePath = ‘testFile.txt’
with open(filePath, 'r') as fileHandle:# r for reading
    data = fileHandle.read() # read into a variable
print(data)
```

Opening and Reading a File without “with”

```
fileHandle = open(filePath, 'r') # r for reading
data = fileHandle.read() # read into a variable
fileHandle.close() #if you don’t open using ‘with’ you must close the file!
print(data)
```

Methods to Read Files

Read a single line from a file

```
fileHandle.readline()
```

Iterate through a file one line at a time using a for loop

```
for line in fileHandle:
  print(line)
```

Read all of the lines of a file and store in a list. This allows you to work with the file contents outside of the “with” block

```
fileHandle.readlines()
```

File Modes

- '***r***' – Read mode which is used when the file is only being read
- '***w***' – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)
- '***a***' – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end
- '***r+***' – Special read and write mode, which is used to handle both actions when working with a file

Useful File Object Function

fileHandle.closed returns true if the file is closed. If the file is open false is returned.

fileHandle.mode returns the mode used to open the file.

fileHandle.name returns the name of the file.

fileHandle.softspace returns a Boolean that indicates whether a space character needs to be printed before another value when using the print statement.

Writing Files

Writing text to a file requires three similar steps:

1. Open the file for writing `open()` method.
2. Write a string (usually from a string variable) to the file.
3. Close the file using the `close()` method.

And here is the code needed to write text to a file:

```
# text to be written is contained in the variable textToWrite
With open(filePath, 'w') as fileHandle: # w for writing
    fileHandle.write(textToWrite) # write out text from a variable
```

Writing to File Example

```
filename = 'programming.txt’

with open(filename, 'w') as fileHandle:
    fileHandle.write("I love programming.")
```

If you’d like to append to a file you can use the ‘a’ mode instead of ’w’

```
filename = 'programming.txt’

with open(filename, ’a') as fileHandle:
    fileHandle.write("I love programming and Python.")
```

Writing to File (Example #2)

```
filename = input("What's the filename you wish to write to? ")

# Below is a standard print message
print("I love Programming")

#If you wanted to modify that print statement to write to a file instead this would be the syntax
with open(filename, 'w') as fileHandle:
    fileHandle.write("I love programming.")


#If you'd like to append to a file you can use the 'a' mode instead of 'w'
with open(filename, 'a') as fileHandle:
    fileHandle.write("I love programming and Python.")
```

Python OS Library

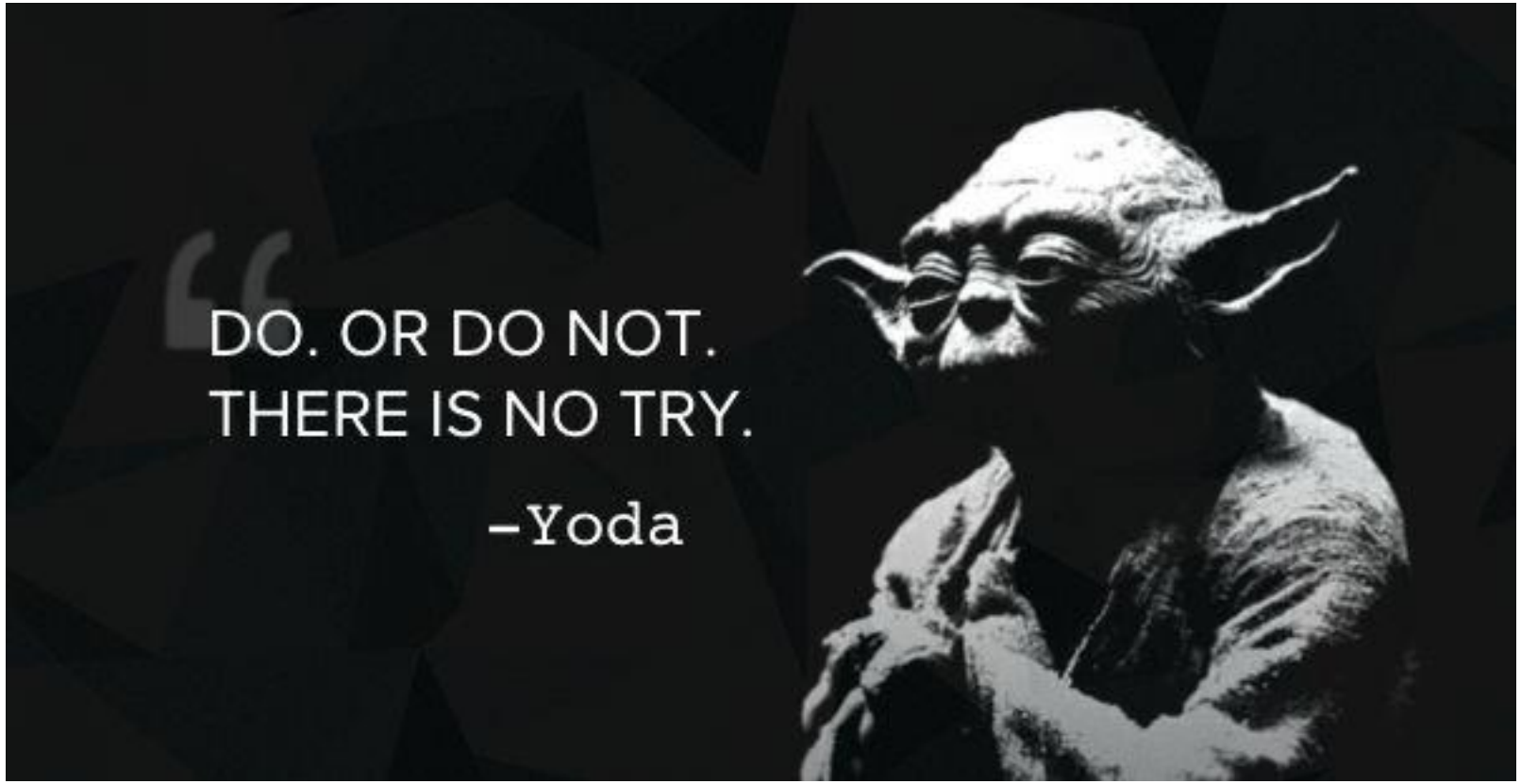
The Python OS library has useful features for dealing with files and directories. In many cases you may wish to validate if a file or directory exists before attempting to work with the file/directory. In order to work with the filesystem path you should import the OS library.

```
import os
os.path.isfile('/file.txt') #will return true if the file exists
os.path.isdir('dir') # will return true if the directory exists
os.path.exists('file.txt') #will return true if the file or directory passed to the
    function exists
```

OS Path and File Processing Example

```
import os #import the OS library
filePath = '/users/eller/Documents/Python/'
fileName = 'testFile.txt'
completePath = filePath+fileName
if os.path.isfile(fileName): #check if file exists
    print('File Exists')
if os.path.isdir(filePath): #check if file path exists
    print('Directory Exists')
if os.path.exists(completePath): #check if complete path exists
    print('Complete path exists')
print('Complete Path',completePath)
with open(completePath, 'w') as fileHandle: #open file for writing
    fileHandle.write("I love programming and Python.") #write data to file
with open(completePath, 'r') as fileHandle: #open same file for reading
    data = fileHandle.read() #read data from the file
    print(data)
```

Except in Programming…



Try Except

In Python we have the ability to use a Try Except block to catch exceptions in our programs. The try statement works as follows:

- *First, the try clause (the statement(s) between the try and except keywords) is executed.*
- *If no exception occurs, the except clause is skipped and execution of the try statement is finished.*
- *If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.*
- *If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.*
- *A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.*

Try Except Syntax

```
try:
    <statement(s) that may cause an error>
except:
    <statement(s) to execute IF an error occurs>
else:    #optional, often not needed
    <statement(s) to execute if NO error occurs>
finally:
   <statement(s) to execute in all cases>
```

Try/Except Example

```
# Check for potential error
# if the userInput isn’t an Integer, catch the exception and print a message

userInput = input('Please enter an integer: ')
try:
    userInput = int(userInput)
except ValueError:
    print 'The number you entered was not an integer'
    # Code here to alter execution because we do not want to keep going
```

Try Except Example With Files

```
filePath = ‘testFile.txt’

try:
    with open(filePath, 'r') as fileHandle:# r for reading
     data = fileHandle.read() # read into a variable
except FileNotFoundError:
    print(‘The file could not be located’)
else:
    print(data)
```

BaseException Class

The BaseException class is the base class of all the exceptions. It has four sub-classes.

- **Exception** – this is the base class for all non-exit exceptions.
- **GeneratorExit** – Request that a generator exit.
- **KeyboardInterrupt** – Program interrupted by the user.
- **SystemExit** – Request to exit from the interpreter.

- The Class Hierarchy of Python Exceptions can be found here:
  - https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Built-in exception classes in Python

- **ArithmeticError** – this is the base class for arithmetic errors.
- **AssertionError** – raised when an assertion fails.
- **AttributeError** – when the attribute is not found.
- **BufferError**
- **EOFError** – reading after end of file
- **ImportError** – when the imported module is not found.
- **LookupError** – base exception for lookup errors.
- **MemoryError** – when out of memory occurs
- **NameError** – when a name is not found globally.
- **OSError** – base class for I/O errors
- **ReferenceError**
- **RuntimeError**
- **StopIteration**, **StopAsyncIteration**
- **SyntaxError** – invalid syntax
- **SystemError** – internal error in the Python Interpreter.
- **TypeError** – invalid argument type
- **ValueError** – invalid argument value