Lecture #8: with and Files

Presented by Jamal Bouajjaj

2023–10–02

For University of New Haven's Fall 2023 CSCIxx51 Course

with

with statement

The `with` statement is a special statement designed to wrap execution with a block of a *Context Manager*.

So `with` will handle the context manager it's given, and is guaranteed to exit it properly no matter what.

with statement

This is the syntax of a `with` block. The `as` is optional:

```
with CONTEXT:
  <-- stuff -->
with CONTEXT as f:
  <-- stuff -->
```

Context Manager

A *Context Manager* is an object that defines how it's resource is used when the `with` block is entered and exited.

The context manager object must define two methods:
`__enter__(self)` and
`__exit__(self, exc_type, exc_value, traceback)`.

the enter method executes when the with block is entered, and can optionally return a variable for the block to use.

The exit method gets called when the with block is exited, no matter what happens. The input arguments are optionally given by the with function if there was an exception raised within the block.

example

```
class WithExampleClass:
    def __enter__(self):
        print("-> We are entering the object")
        return "something"
    def __exit__(self, exc_type, exc_value, traceback):
        print("-> We are exiting the with block")

o = WithExampleClass()
with o as a:
    print(f"we are in the with block with '{a}'")
    #raise UserWarning("oh no a problem")
    print("doing stuff")
```

Files

stream

Files can be though of, and are treated as, as stream of data. The stream can be indexed.

Think of it as a river, where naturally you go down until it ends, but magically can teleport where you are in the river.

open sesame

Files can be read with the

```
open(file, mode='r', buffering=-1, encoding=None,
↪  errors=None, newline=None, closefd=True, opener=None)
```

built in Python function. The primary arguments to be given is `file` and `mode`: the file name and the opening mode.

Buffering is whether to buffer the file, and by how many bytes. 0 is for no buffering, and -1 is for default buffer size.

The rest of the arguments arent't as important, but do read up on it on the docs.

mode

There are several mode to open the file with, some are mutually exlusive. They are denoted by characters that are concatenated for the mode, which are:

- 'r' $\leftarrow$ open for reading (default)
- 'w' $\leftarrow$ open for writting
- 'x' $\leftarrow$ open for exclusive creation
- 'a' $\leftarrow$ open for append writting if file exist
- 'b' $\leftarrow$ binary mode
- 't' $\leftarrow$ text mode (default)
- '+' $\leftarrow$ open for reading and writting (updating)

So 'rb' is to open a file for read in binary mode for example.

open return

The object returned by `open` depends on the mode the file was opened with, and the buffering argument. The following are the abstract base classes for the class object that is returned:

- `RawIOBase` $\leftarrow$ For stream of bytes
- `BufferedIOBase` $\leftarrow$ For buffering on a `RawIOBase`
- `TextIOBase` $\leftarrow$ For stream of bytes representing text.

They all inherit `IOBase`.

Basic Callbacks

Here are some basic functions of the classes above you should know about:

- `read(x)` Reads the entire file, or x bytes if specified
- `readline()` Reads until a new line character, returning the line
- `write(x)` Writes x data into the file
- `tell()` Returns the current stream position
- `seek(x)` Sets the stream position to x
- `close()` Closes the I/O

If the current stream position is EOF (End-Of-File), then read returns an empty string or byte.

Generally, it's BAD to just call `read()` without any arguments, especially for a large file.

Example

```
f = open('text.txt', 'r')    # default, read-only as text
print(f.readline())          # read a single line
print(f.tell())          # where we are in the stream
f.seek(0)                # go back to start of file
print(f.read(1))           # read a single char
print(f.read())          # read until EOL
f.close()                # close file
```

Itterable

What if you want to read the file line by line in a for loop?

Itterable

What if you want to read the file line by line in a for loop?

Thankfully, `IOBase` implements the `__iter__()` and `__next__()`

```
f = open('text.txt')    # default, read-only as text
for line in f:
  print("line read: ", line)
f.close()               # close file
```

Closing Files

For good practice, you must close the file when you are done with it. This allows other programs to access the file.

But what if you are only accessing a file in a single block. Wouldn't it be nice if there was a block operation that automatically closed the file for you?

with file

As it turns out, a `IOBase` does implement the `__enter__()` and `__exit__()` callbacks!

```
with open('text.txt', 'r') as f:
  for line in f:
    print("line read: ", line)
```

End

The end