Reading and Writing Data

Now we are going to learn about reading data from files and writing data to files. Today we are going to focus on files that contain text, which is generally read line by line and processed as a string. You may wish to read data in other formats, such as images or movies, in which case similar principles apply, but you will use slightly different methods to read byte by byte instead of line by line. For data science applications, "pandas" are also popular. Pandas is a library for manipulating data in a tabular (rows and columns) format. The Pandas library takes care of the row-by-row reading of the data and builds a data structure for you to work with. We won't delve into Pandas right now, but you will learn about them later.

Before we jump into reading and writing files, we want to take a bit of time to talk about some concepts that we are going to need to build on. First, we are going to talk about strings, which is the programming term for text. While you have seen string literals (like "hello" in print("hello")) before, strings are objects that we can operate on. Second, we are going to talk a little bit about how files work in general. Third, we are going to talk about exceptions, which are the way that Python (and many other languages) deal with problems, such as trying to open a file that does not exist. Finally, we'll dive into how to read and write files in Python.

Strings

A string is a sequence of characters, which is how we represent text in programs. You are familiar with *string literals*, such as "hello" where you write down exactly the text that you want inside of quotation marks (Python supports both 'single quotes' and "double quotes" for string literals). Up to this point, strings have been mostly limited to literals, and concatenation with the + operator (concatenation meaning putting two strings together: "hello" + " world" results in "hello world"). However, there is a lot more that you can do with strings. As we read data from files, we often want to *parse* that data—we want to take the string and extract some meaning from it. Parsing data can be as simple as taking regularly formatted lines of the form "field1:field2:field3" and splitting the fields apart, or as complicated as reading the code of a programming language and figuring out what the programmer wrote or reading natural language (e.g., English) and figuring out information about what was written (both of which are well beyond the scope of this course).

In Python, strings are *immutable* sequences of characters. However, there is no separate type for characters, so string operations that give back single elements return strings of length 1. Immutable means that once you create a particular string, you cannot change its contents. Instead, string operations that change strings produce a new string with the changes made. We will note that there are a lot of string methods, and we are not going to go into all of them here, but you can find more in the Python documentation as needed (https://docs.python.org/3/library/stdtypes.html#string-methods).

As a string is a sequence of characters, we can index them just like lists. For example:

```
x = "hello"

print(x[1])
```

will print e because x[1] indexes the string "hello" at index 1. As with lists, indexing starts at 0 (so x[0] would be 'h'). As with lists, you can use slicing, such as x[2:4] which would result in 'll'. As with other sequences, you can also use the "*in*" operator to check if one string appears inside of another.

File System Concepts

By this point, you have some basic practice with the Linux command line and are used to changing directories (cd), listing their contents (ls), and editing files. We are going to take just a moment to expand on those concepts here before we go into the details of exceptions and file access in Python. The data for the files is stored on a disk. Whether that disk is a "traditional" hard disk (made with spinning magnetic platters) or a solid state disk (SSD) made with non-volatile memory technology (more similar to USB memory sticks), or even if the file is stored remotely and accessed over a network filesystem does not matter from the perspective of our program (yay, abstraction!). The *operating system* handles the details of accessing the actual physical device as well as the layout of the data on the disk. The OS provides a set of *system calls* which programs can use to interact with the files via a high level interface. The OS provides abstractions for higher level concepts such as directories and file names that are easier for the programmer to work with. Each programming language then implements its support for reading and writing files in terms of these abstractions by making system calls to request the OS perform various actions. Accordingly, files work basically the same way no matter what language you are programming in. When you use a command like "cat" (which is written in C) it uses the same underlying system calls (and thus can do the same things to files) as your Python code would do. We are going to talk briefly about these concepts to give you a better idea of what is going in your Python code and any other programs you work with.

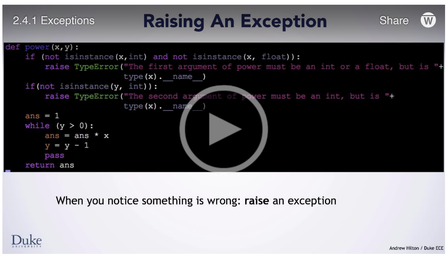
First, let us take a moment to talk about "the operating system". You are probably familiar with the idea that there are different types of operating systems, such as Windows, Mac OSX, and Linux. But what exactly is the operating system? The operating system is a combination of programs and the *kernel* that interact with the hardware and provide set of services to software. Some of these services are in the form of *system calls*, where a program makes a request to the kernel to ask it to perform some operation on its behalf. Other services include scheduling multiple processes (when you have many programs open, the OS makes them take turns using the CPU), memory management, and a variety of other things. Many of these services are provided by the kernel. The kernel is a privileged piece of software (meaning it can do things that normal programs cannot, like directly access the hard disk) which exists in the background and takes over when needed—either by request of a program, or periodically for things like changing which program is running.

When a program wants to interact with a file, the first thing it needs to do is *open* that file. To open the file, the program makes a system call, passing in the path of the file it wants to access. A path is the directory names and file name that specify the desired file. The path can either be a *relative path* (in which case it is relative the the directory the program is running in) or absolute (in which case it specifies the entire sequence of directories from the root of the file system). An absolute path starts with "/" (on Linux) and a relative path does not. For example, suppose I login to a Linux server and bash is running in my home directory (/home/drew). If I specify the relative path "code/foo.py" that path is relative to /home/drew, which is equivalent to the absolute path /home/drew/code/foo.py. We'll briefly note that absolute paths are great for specifying system resources, but we generally prefer relative paths for things like specifying input and output files for programs. If you specify such things in terms of relative paths, then it is easier to move your code to somewhere else—including if someone else needs to run it on a different system.

When the program makes the system call to open the file, the kernel checks a variety of things (such as if the file exists, and if the program has permissions to access it). If everything is ok, the kernel returns a *file descriptor* to the program. You do not generally work directly with file descriptors—they are hidden inside the details of your programming languages's file library, but they are used by your program for its future requests to work on that file (such as reading or writing it). Python remember this file descriptor inside of the file object that you get from calling "open" (which you will learn about shortly). If something goes wrong, the kernel returns an error code indicating what went wrong. Python translates this error code into an exception, which we will discuss in the next section. When the program is done with the file, it should ask the kernel to close the file descriptor.

Exceptions

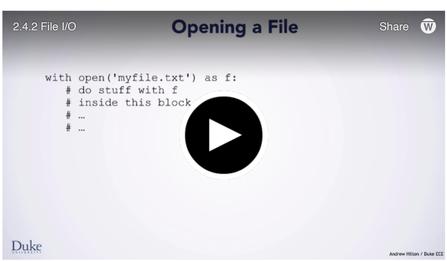
When you try to access files, a variety of things can go wrong. For example, you might try to open a file that does not exist. Or you might try to write a file when the disk is full. Python handles these (and other) problematic situations with exceptions. An exception is a special object that indicates "something has gone wrong". The exception is *raised* (or *thrown*—some languages use one term and other use the other) to indicate the problem. The exception's type indicates the general category of what went wrong, and the exception object can carry other information to indicate more specifics. An exception is *caught* (or *handled*) by code which knows how to deal with the problem. For example, if your program asks the user to input a filename to read from, the program could handle an exception that indicates the file does not exist by asking the user for another filename and trying again. The next video shows you the basic mechanics of raising and catching exceptions.



Day 9, Part 1

Now it is time to do **18_read_exn** on the Mastery Learning Platform. Login to the course server, go to **18_read_exn** in your git repository, and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.

File I/O



Day 9, Part 2

Now it is time to do **19_read_encr** on the Mastery Learning Platform. Login to the course server, go to **19_read_encr** and read the README for directions. When you have passed that assignment, complete **20_sort_file** on the Mastery Learning Platform. Login to the server, go to **20_sort_file** in your git repository, and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.

CSV Files

One common format for data files is "CSV" files (which stands for comma separated values). Often spreadsheet programs (like Microsoft Excel, Apple Numbers, or Google Sheets) will export data to a CSV file. While we are not going to do anything with CSV files here, we think they are worth mentioning because they are incredibly common. Python has a nice library for reading and writing CSV files (https://docs.python.org/3/library/csv.html). The CSV library assumes you have already opened a file as we just learned how to do, and splits the data up into rows and columns for you.