Lab 10: Multi-Part File Downloader

Table of Contents

1) Preparation

This lab assumes you have Python 3.5 or later installed on your machine. Like Lab 1, this lab also requires the `pillow` module to be installed.

The following file contains code and other resources as a starting point for this lab: `lab10.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- passing the test cases from `test.py` under the time limit (2 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2 points).

For this lab, you will only receive credit for a test case if it runs to completion under the time limit on the server.

Both the code submission and the checkoff for this lab come due at the last possible time for submissions to the course, Wednesday, May 15th, at 2:55 PM. Because of Institute rules, this is the last time at which assignments come due. As such, neither late days nor automatic extensions can be used on Lab 10.

2) Introduction

We suggest reading the entire lab before writing any code, to get a sense of the program you are being asked to write.

One way to handle the problem of reliably storing large files is to break them into pieces that are placed on multiple disks, or on multiple machines. This way, if something goes wrong with a particular file (and/or with a particular machine as a whole), the file is still available.

In this lab, you will build a *multi-part* downloader that assembles a data stream from multiple parts streaming individually from multiple machines. It allows the same part to be stored redundantly in multiple locations so it can be resilient to failures. Since the multipart streams you will be downloading may be nonterminating, your program will assemble the stream incrementally from its parts as they are downloaded, displaying the file or streamed sequence of files (an animated sequence of images, for example) as the download progresses.

This page describes the specification for the lab, but it also aims to provide some review on a few aspects of Python that were discussed in lecture but that have not been used explicitly in a lab yet.

2.1) Downloads

Throughout this lab, we will be interested in downloading files from various web servers. We have provided a function called `http_response`, which is defined in `http009.py`, to help you with this task. `http_response` is a thin wrapper around functionality from Python's built-in `http.client` module.

That function takes a single argument (a string representing the URL of a resource we wish to look up), and it returns an `HTTPResponse` object.

In particular, we are interested in three or four attributes/methods of these objects. If `r` is the result of calling `http_response`, then:

- `r.status` gives the HTTP status code that the server returned. For this lab, we will need to know that
  - `301`, `302`, and `307` represent redirects.
  - `404` means that the specified file was not found.
  - `500` means that an error occurred on the server.
- `r.getheader(name)` returns the HTTP response header associated with the given name.
  - `r.getheader('location')` will provide the location if we were redirected.
  - `r.getheader('content-type')` will tell you the media type of the response of a successful request.
- `r.read(num_bytes)` returns a bytestring containing at most `num_bytes` number of bytes read from the body of the response. If no more bytes can be read, returns an empty bytestring `b''`. If `num_bytes` is not given, return the entire page contents. **See section 10 if you are unfamiliar with Python bytestrings.**
- `r.readline()` returns a bytestring containing the next line of the body of the response. It returns an empty bytestring `b''` if no more bytes can be read.
  - to decode a bytestring into a string run `b''.decode("utf-8")`

> **Check Yourself:**
>
> Try calling `http_response` from the Python REPL for a few different URLs, to get a sense of how it behaves (you can get to a Python REPL with that function defined by running `python3 -i http009.py`).
>
> Inspect the objects returned from this function, and try to extract the status code, headers, and response body for the following:
>
> - `'https://py.mit.edu'` should be a redirect.
> - `'https://6009.cat-soop.org/spring19/labs/lab11'` is a nonexistent file.
> - `'https://6009.cat-soop.org/spring19/python'` exists, and it should have a body that is 6822 bytes long.
> - `'http://nonexistent.mit.edu/some_file.jpg'` is trying to access a file on a *computer* that doesn't exist.

> **NOTE FOR MAC USERS**
>
> There is a known issue with the default installer for Python 3.6 and 3.7 on Mac OSX. If you are having trouble accessing files at HTTPS addresses, it is likely because there is still some action required on your part. In particular, you may need to run one of the following commands (while in the Users/your_username directory) to enable support for SSL communication:
>
> ```
> /Applications/Python\ 3.6/Install\ Certificates.command
> /Applications/Python\ 3.7/Install\ Certificates.command
> ```

2.2) File Manifests

In addition to being able to download regular files, there are also *file manifests*, which is a way of specifying how a particular file is broken down into parts. File manifests in our implementation will either end with `.parts` or have the `Content-Type` header equal to `'text/parts-manifest'`.

The manifest contains several parts, each adjacent pair of which is separated by a line containing exactly two hyphens `--`. For example, a file `picture.jpg` might be broken down into three parts, which could be specified with:

```
http://mymachine.mit.edu/picture.jpg-part1
--
http://mymachine.mit.edu/picture.jpg-part2
--
http://mymachine.mit.edu/picture.jpg-part3
```

To download the contents of this file specified by the manifest, you would need to download each of the individual parts and concatenate them together.

In addition, the manifest can provide *alternatives* for each section (so that if one URL doesn't work we can use the other one):

```
http://mymachine.mit.edu/picture.jpg-part1
http://yourmachine.mit.edu/picture.jpg-part1
--
http://mymachine.mit.edu/picture.jpg-part2
http://yourmachine.mit.edu/picture.jpg-part2
--
http://mymachine.mit.edu/picture.jpg-part3
http://yourmachine.mit.edu/picture.jpg-part3
```

In this example, each of the three parts is stored both on `mymachine.mit.edu` and on `yourmachine.mit.edu`. For each part, if the first option is not successfully downloaded, we should try the second option (and so on until we find one that works). In general, a manifest is not limited to 2 options; there may be arbitrarily many options for each part, and each part might have a different number of options.

Manifest streams can be recursive, which means that a part might itself be specified by a URL ending in `.parts` (which represents a manifest stream). In the case where a part refers to a manifest stream, the entire contents of the file referred to by that manifest stream should be used.

For example, if `http://mymachine.mit.edu/endless.txt.parts` contains the following lines:

```
http://mymachine.mit.edu/verse.txt
http://yourmachine.mit.edu/verse.txt
http://hermachine.mit.edu/verse.txt
--
http://yourmachine.mit.edu/chorus.txt
--
http://mymachine.mit.edu/endless.txt.parts
```

Then someone downloading the file should receive an endless file alternating between the contents of `verse.txt` and `chorus.txt`.

> **Check Yourself:**
>
> To summarize:
>
> - A manifest has multiple parts.
> - Parts are separated by lines of `--`.
> - Parts can have multiple URLs.
> - Parts can point to URLs that are manifests themselves.

2.2.1) Caching

Manifest files may have parts that are replicated multiple times in the same file. This may be obvious from the previous manifest file, which is an endless stream of two replicated text parts. Thus, we will also add the capability to cache parts of files within a single download (though we will explicitly **not** cache these results across multiple downloads).

Caching is a bad idea if a file changes too frequently. Therefore we are not going to cache all the files we download. We will only cache for a part if `(*)` is listed as a possible URL for that part in the manifest file. This indicates that the manifest file deems a particular part to be largely fixed content, so it can be safely cached within a single download.

When downloading, if we reach a part that is marked as cacheable (contains `(*)`), we should first check to see if any of the URLs has been cached. If the file has already been cached, we should simply use that result (without making any HTTP requests). If not, we should send a HTTP request to download the first URL in the part and store the result in the cache. Importantly, caching should not interrupt the streaming (i.e., we should still be able to stream from files as we download them, rather than having to wait for the whole file to be downloaded first).

Parts in the manifest that are not marked with an `(*)` **should never be cached**. This requires that the content for these links should not be read from the cache or stored to the cache. Thus, a sample Manifest file that supports caching could look like the following:

```
(*)
http://mymachine.mit.edu/verse.txt
http://yourmachine.mit.edu/verse.txt
http://hermachine.mit.edu/verse.txt
--
http://yourmachine.mit.edu/chorus.txt
(*)
--
http://mymachine.mit.edu/endless.txt.parts
```

For simplicitly, it is not necessary to support caching manifest files themselves, and **it is not necessary to support caching infinite files** (i.e., you can assume that any file you are being asked to cache is finite in length).

> **Check Yourself:**
>
> To summarize:
>
> - Caching should be "off" by default and should reset at the beginning of each `download_file` call.
> - Only cache and use cached files when `(*)` appears in the part of file manifest.
> - Do not support infinite files. (All cached files are finite.)

2.3) File Sequences

In addition to manifests, we would also like to handle *file sequences*. In general, a file sequence is a single stream of bits that represents a sequence of multiple files. We'll use a format of our own invention, but it's not too different from, for example, a movie file (which can be thought of as a sequence of frames, each of which is an image).

If a URL contains the string `-seq`, the resulting stream represents a sequence of files. Each sub-file in a sequence is represented as:

- 4 bytes containing the length of the file (in bytes, represented as a single big-endian unsigned integer (see subsection 10.4)), followed by
- the raw bytes contained in the file (there should be as many bytes as were specified in the length field).

As an example, consider a file consisting of the following 5 bytes (shown in hexadecimal):

```
36 2E 30 30 35
```

and another file consisting of the following 6 bytes:

```
72 75 6C 65 73 21
```

We could encode a *file sequence* consisting of these two files with the following stream of 19 bytes (4-byte length + 5-byte file + 4-byte length + 6-byte file):

```
00 00 00 05 36 2E 30 30 35 00 00 00 06 72 75 6C 65 73 21
```

If a URL contains `-seq` and is also a manifest (the URL ends with `.parts` or the content type is `text/parts-manifest`), then it represents a manifest file that, when all parts are concatenated, represents a file sequence.

3) Implementation

We have provided a very small skeleton in `lab.py`, indicating the functions that must be defined for this lab. Beyond those, you are welcome to define whatever helper functions / classes you deem necessary.

3.1) Basic Downloads

We'll start by discussing basic downloads. You should fill in the body of the `download_file` function to implement a basic "streaming" downloader[1].

`download_file` should be a **generator** that yields the result of the response as bytestrings of at most `CHUNK_SIZE` length (it should yield bytestrings of this size until it is no longer able to do so, at which point it should yield one bytestring of smaller size if the file contains additional content).

Note that, throughout this lab, we will assume that any URL refers to a potentially infinite stream (for example, a stream from a webcam). As such, your `download_file` function **should not download the entire contents of the file and yield values from that result; rather, it should yield chunks of the file as they are downloaded.**

We will also assume that any URL refers to arbitrary binary data (it could be an image, or text, or random bytes, etc.). Aside from checking whether the URL refers to a file manifest, your code should not need to change its behavior based on the type of the data (or the `content-type` header).

There are a couple of additional complications:

- If a request results in a redirect (status codes `301`, `302`, or `307` for our purposes), your downloader should follow that redirect and try the location to which you are being redirected.
- If a request gives a `404` status code, your downloader should raise a `FileNotFoundError`.
- If a request gives a `500` status code, or if the connection was not made because of a network error, your downloader should raise a `RuntimeError`.
- A status code of `200` indicates a successful request.

3.2) Handling Manifests

If the location given at initialization time ends with `.parts` **or** if the `Content-Type` header associated with a request is `'text/parts-manifest'`, then it represents a file manifest. In this case, rather than simply yielding the contents of that `.parts` file, `download_file` should instead yield the bytes from each of the files specified in the manifest, in the order they are specified.

In order to keep things as smooth as possible, your function should start streaming the first part as soon as it is available, rather than waiting for all parts to be available before streaming.

3.2.1) Caching

You should also make sure that your program properly caches files that were indicated with `(*)` in any file manifests that were given (so that, if we have already retrieved their contents once in the process of downloading the file, we do not do so a second time). Your program should not cache other files.

If a file is cached, it is OK to yield the entire contents of the cached file as one bytestring (rather than splitting it up into chunks and yielding those separately).

Importantly, the cache should not persist between calls to `download_file`. That is, starred files that we try to download multiple times *within a single top-level call to* `download_file` should be cached, but that cache should be invalidated with every new top-level call to `download_file`.

3.3) Handling File Sequences

The description above should be enough for *downloading* file sequences, but it is not yet enough for interpreting them. You should not need to change any behavior in your downloader to account for file sequences. But the GUI needs to know something about how to handle file sequences in order to properly display them, so we'll add a small piece of functionality to allow us to grab individual files from the sequence.

Write a new generator called `files_from_sequence(stream)`. Given a generator of the form described for `download_file`, `files_from_sequence` should yield the contents of each file contained in the sequence, in the order they are specified. Note that each of the chunks yielded from `download_file` might contain multiple files, or it might not contain an entire file. Your function will need to account for both of these cases.

4) Interfaces

In order to test your code, we have already provided an interface (for displaying downloaded images and/or text files) for you in `gui.py` in the lab distribution.

You can run the GUI from the command line by giving it a URL as argument:

```
$ python3 gui.py URL
```

The GUI will then use your code to stream the file, and it will attempt to display it. You can also optionally specify an additional integer argument that specifies how long the GUI should wait between switching files in a file sequence when displaying them (this value is specified in milliseconds).

5) Implementation Advice

This is a fairly complicated task, and so trying to write it all in one go might make the task of organizing your code (and of debugging) more difficult. You may find it easier to implement the lab in small pieces first. For example, you might:

- start by writing something that downloads a single file that you know exists (after which `Test1_Streaming` should pass), then
- introduce support for handling redirects and/or errors (after which `Test2_Behaviors` should pass), then
- add support for handling file manifests (after which `Test3_Manifest` should pass), then
- add support for caching (after which `Test5_Caching` should pass), then
- add support for file streams (after which all tests should pass).

You might find a different order preferable, but starting small and then adding functionality piece-by-piece is likely to make the debugging process more straightforward.

6) Test Files

The files below provide some data for testing (roughly in order of increasing complexity):

- `https://6009.cat-soop.org/_static/spring19/helloworld.txt` (line of text)
- `https://6009.cat-soop.org/_static/spring19/logo.gif` (single image)
- `http://hz.mit.edu/009_lab9/bird.jpg` (single image, with single redirect)
- `http://scripts.mit.edu/~6.009/spring19/lab10/redir.py/0/wug.jpg` (single image, with many redirects)
- `http://mit.edu/6.009/www/spring19/lab10_examples/yellowsub.txt.parts` (manifest file representing a piece of text)
- `http://mit.edu/6.009/www/spring19/lab10_examples/fugue.wav.parts` (manifest file representing a piece of music)
- `http://mit.edu/6.009/www/spring19/lab10_examples/cornsnake.jpg.parts` (manifest file representing an image, with pieces spread across multiple machines)
- `http://mit.edu/6.009/www/spring19/lab10_examples/cached_yellowsub.txt.parts` (same as `yellowsub.txt.parts`, but with caching)
- `http://mit.edu/6.009/www/spring19/lab10_examples/kafka.txt-seq` (sequence of many small files, best viewed in GUI)
- `http://scripts.mit.edu/~6.009/spring19/lab10/fly.py/fly.txt-seq` (infinitely long sequence)
- `http://scripts.mit.edu/~6.009/spring19/lab10/redir.py/0/smallcat.png-seq` (sequence of images representing a short film, many redirects)
- `http://mit.edu/6.009/www/spring19/lab10_examples/skeletons.png-seq.parts` (recursive manifest, sequence representing an infinitely looping animation)
- `http://mit.edu/6.009/www/spring19/lab10_examples/skeletons.cached.png-seq.parts` (like the above, but with caching)

7) Code Submission

When you have tested locally and are satisfied that your code is working, you can use the box below to submit your code to the server for testing.

Submissions have closed.

8) Checkoff

Once you are finished with the code, please come to a tutorial, lab session, or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular, be prepared to discuss:

- a demonstration of your code working on `http://scripts.mit.edu/~6.009/spring19/lab10/redir.py/0/smallcat.png-seq` using the GUI
- takes a while: a demonstration of your code working on `http://mit.edu/6.009/www/spring19/lab10_examples/skeletons.cached.png-seq.parts` using the GUI
- a *very* quick overview of any helper functions and/or classes you wrote, and why they were helpful
- aren't you glad you're done with 009?

8.1) Grade

Grading:

- Concept questions (0 points): 0
- Tests (2 points): 2

- Checkoff (2 points): 2

Total: 4 Points (of 4)

Comments from Grader:
Good job!

9) (Optional) Additional Extensions

9.1) Interfaces

In addition to the `gui.py` interface, we would like to provide an interface for saving the results of downloading a file and saving its contents to disk.

Modify your `lab.py` so that it takes two arguments when it is run from the command line:

```
$ python3 lab.py URL FILENAME
```

When invoked in this way, your program should download a file from `URL`, and save its contents at a location given by `FILENAME`. If the given URL refers to a parts manifest, you should not directly save the contents of the manifest; rather, you should save the contents of the file to which that manifest refers. If the given URL refers to a sequence, then instead of saving a single file, you should save multiple files of the form `FILENAME-file1`, `FILENAME-file2`, ...; each of these files should contain one file from the sequence.

Recall that you can access command-line arguments via the `sys.argv` variable.

9.2) Files of Your Own

It would be cool to use your downloader on some files of your own. Try making some files:

- an image or audio file of at least 100kB
- an image or text file of at least 100kB, split into at least 3 distinct pieces and represented as a `.parts` file
- a file sequence of at least 100kB, consisting of at least 3 distinct files

Note that you can make a file accessible at `http://mit.edu/griffinl/www/FILENAME` by creating a file called `FILENAME` in your Athena account under the `www` directory. You can use this to put your files on the web so that your downloader can reach them. See this page from IS&T for information about how to transfer files to your Athena account.

Then try downloading these files.

9.3) Infinitely Long Manifests

You are not required to deal with infinitely long manifest, however, for fun you can try.

- `http://scripts.mit.edu/~6.009/spring19/lab10/cats.png-seq.py` (infinitely long manifest of an animation, with simulation of a slow connection)
- `http://scripts.mit.edu/~6.009/spring19/lab10/cats.png-seq.cached.py` (like the above, but with caching)

9.4) Parallel Downloads

So far, our focus has been on the increased *reliability* that can come from distrtibuting files across multiple parts/machines. It turns out that doing this can also be used to increase download *speeds*. The basic idea is that a way to download a file more quickly is to break it into parts, and to download the different parts from different machines[2]. By downloading files in multiple small parts, the chance of the entire download failing is reduced; by downloading those parts in parallel, we can often achieve faster download speeds; and load is spread more evenly across the network. As such, it is often faster to download files in this way, compared to downloading the entire file as one piece.

One nice extension for this lab would be modifying your code so that, when it encounters a manifest file, it downloads all of the pieces in parallel (rather than sequentially).

9.5) Other Errors

What we have built in this lab is a fairly realistic system that interacts with other computers via the Internet. In such systems, there is the potential for unforeseen errors to arise. When using this system, a variety of errors could happen that the description above has not explicitly accounted for.

Think through a few possible kinds of errors that could occur but that aren't explicitly described above, and modify your code so that it handles them.

10) Appendix: Working with Raw Binary Data in Python

The strings in Python that we are familiar with (the `str` type) consist of a useful abstraction for representing text. In particular, Python strings represent sequences of characters (where each character represents a Unicode character).

These are a useful abstraction for displaying text, but the fact that they exist as an abstraction for the text means that they do not work as a representation for other kinds of data, for example images or audio (since text has a particular struture in its binary data that images or audio won't generally have). Ultimately, what can be stored on disk (or sent over the network) is a series of binary values (1's and 0's). Saving character strings to disk, for example, requires first converting them to raw binary data first.

In this lab, we will be working with binary data in a variety of formats (including not only text, but also images and audio), so we will need a different, lower-level abstraction that lets us work directly with binary data. Conveniently, Python offers just such an abstraction, through *bytestrings* and *bytearrays*.

10.1) Byte Strings

Python offers a built-in type called `bytes` that represents a sequence of raw binary data. While this does represent a sequence of 0's and 1's, the `bytes` type exposes those values to us not as individual *bits*, but rather as *bytes*, each of which contains 8 bits. As such, we can think of each byte as representing an integer between 0 and 255 (inclusive).

One way to construct bytestrings is directly from arrays of integers, for example:

```
>>> x = bytes([207, 128])
>>> y = bytes([207, 132, 32, 62, 32])
```

If we print the resulting value, we see a different Python syntax for bytestrings (which looks like a regular string, but with a `b` in front:

```
>>> print(x)
b'\xcf\x80'
>>> print(y)
b'\xcf\x84 > '
```

Note that this is not human-readable like a regular Python character string. However, it is in a format that makes it easy to write the data to disk directly, or to send it across a network (which will be relevant for this lab!).

Byte strings support indexing like regular strings. When we index into a bytestring, we get the integer value associated with the byte at that index (this will be an integer in the range 0 to 255, inclusive). Similarly, looping over a bytestring gives us integer values.

```
>>> print(x[0])
207
>>> for i in y:
...     print(i)
...
207
132
32
62
32
```

10.2) Converting Between Character Strings and Bytestrings

As we have mentioned above, Python has two separate internal representations for strings: character strings (`str`) and byte strings (`bytes`). The conversion between these is described by an *encoding* (which specifies how each character is represented as a sequence of bytes).

In order to store a character string on disk, it must first be *encoded*; and to view a human-readable form of a string that has been read from a file on disk, it must be *decoded*.

There are many different possible encodings for string data (that is, given a sequence of abstract characters, there are a number of different ways we could choose to represent those as raw binary data). Two of the most common encodings are `ASCII`, `UTF-8`, and `UTF-16`. As we can see, encoding the same string with two different encodings gives us two different sequences of bytes:

```
>>> x = "Fußgänger"
>>> u8 = x.encode('utf-8')
>>> u8
```

```
b'Fu\xc3\x9fg\xc3\xa4nger'
>>> u16 = x.encode('utf-16')
>>> u16
b'\xff\xfeF\x00u\x00\xdf\x00g\x00\xe4\x00n\x00g\x00e\x00r\x00'
```

When decoding, we also need to know which character encoding we are using. Trying to use the wrong encoding will either produce an unexpected answer or an error:

```
>>> u8.decode('utf-8')
'Fußgänger'
>>> u8.decode('utf-16')
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    u8.decode('utf-16')
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x72 in position 10: truncated data
```

10.3) Reading and Writing Binary Data From Files

When we dealt with files in previous labs, we were making the (unstated) assumption that the files we were working with contained textual data (specifically, textual data encoded using the UTF-8 encoding, which is a broadly used standard).

Let's see what happens when we try to open a file that does not contain UTF-8-encoded text:

```
>>> with open('Havana.mp3', 'r') as f:
...     x = f.read()
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/usr/local/lib/python3.7/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 45: invalid start byte
```

Indeed, we get an exception saying that Python does not know how to convert the data in that file into a human-readable character string! In order to grab the data from the file, we need to tell Python to open the file in *binary read* mode by specifying `'rb'` when opening the file:

```
>>> with open('Havana.mp3', 'rb') as f:
...     x = f.read()
...
```

Having done this, we have the raw binary data available to us in the variable `x` (as a bytestring).

```
>>> len(x)
3471822
>>> x[:100]
b'ID3\x04\x00\x00\x00\x00\x00#TSSE\x00\x00\x00\x0f\x00\x00\x03Lavf57.56.101\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xfbP\x00\x00\x00\x00\x00\x00\
 r\x004\xf9\xa1\x00\x02\x05'
>>> for i in x[:10]:
...     print(i)
...
73
68
51
4
0
0
0
0
0
35
```

10.4) Representing Integers with Bytestrings

As we mentioned above, each byte in a bytestring has a representation as an integer in the range 0-255 (inclusive). A few different representations for each possible byte are shown in the table on this page for reference, though it is certainly not necessary to commit that table to memory.

The question might arise, though: what do we do if we want to represent a number bigger than 255? Clearly, one byte isn't enough in that case! So we need a different convention for representing larger integers.

In general, we can use the concatenation of multiple bytes to represent larger integers. In this lab, we choose a "big-endian" representation, where the most-significant byte comes first. By this, we mean that, if we have a byte string containing bytes b_0, b_1, b_2, and b_3 (in that order), this represents an integer:

$$(256^3 \times b_0) + (256^2 \times b_1) + (256 \times b_2) + b_3$$

In theory, we could extend this to an arbitrary number of bytes to represent arbitrarily large integers. In this lab, though, we chose to represent lengths using 4 bytes only (though this still allows us to represent integers up to 4,294,967,295).

Since we can index into bytestrings to get integer values, it is possible to compute this value directly without a complicated algorithm. Alternatively, you can look up the documentation for the `int.to_bytes` and `int.from_bytes` functions, which will handle this conversion (in either direction).

10.5) Byte Arrays

It is worth noting also that the `bytes` type, like regular Python strings, is *immutable*. On occasion (depending on the operation you want to perform), you may wish to work with a *mutable* array of bytes (such that you can mutate it by changing a character in place, by appending/extending values, etc.).

To this end, Python gives us the `bytearray` type. You can create a `bytearray` from a bytestring with `bytearray(x)`, or you can create an empty `bytearray` with `bytearray()`.

```
>>> ba = bytearray()
>>> ba.extend(u8)
>>> ba.extend("übergänge".encode('utf-8'))
>>> ba
bytearray(b'Fu\xc3\x9fg\xc3\xa4nger\xc3\xbcberg\xc3\xa4nge')
```

Note that the elements in the `bytearray` need to be valid bytes (integers in the range 0 to 255, inclusive). So you can `extend` a `bytearray` by an iterable of integers. But if you want to append values (or modify values), those values need to be given as integers.

It is worth noting, also, that `bytearray` objects can, in many ways, be treated like `bytes` objects. For example, they provide a `decode` method:

```
>>> ba.decode('utf-8')
'Fußgängerübergänge'
```

But if you need to, you can always convert back to a `bytes` object:

```
>>> x2 = bytes(ba)
```

Just like lists vs. tuples, the question of whether you want a byte string or a byte array is heavily dependent on the application (certain operations might only be possible on one type or the other, or certain operations might be more efficient with one type than with the other).

Footnotes

[1] In this case, "streaming" refers to the fact that we want to provide bytes from the file as they are being received, rather than downloading the entire file first and then returning its contents.

[2] You may be familiar with the peer-to-peer file-sharing protocol BitTorrent, which uses this idea. It is also used in a variety of other contexts.