Lab 6: Multi-Part File Downloader

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

The following file contains code and other resources as a starting point for this lab: `lab06.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.

You can also see and participate in online discussion about this lab in the "Labs" Category in the forum.

This lab is worth a total of 4 points.

Note that passing all of the tests on the server will require that your code runs reasonably efficiently.

The questions on this page (including your code submission) are due after the spring break at 5pm Eastern on Friday, 1 April (no joke!). The checkoff is due after the spring break, at 10pm Eastern on Wednesday, 6 April.

Your Checkoff This Week

You are not logged in. Please log in to see information about your checkoff for this lab.

2) Introduction

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 unbounded and never end, 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.

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

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. There are many different status codes. For this lab, though, we will need to know that `301`, `302`, and `307` represent redirects; that `404` means that the specified file was not found; that `500` means that an error occurred on the server; and that `200` corresponds to a successful request.
- `r.getheader(name)` returns the HTTP response header associated with the given name. In the case of a redirect, `r.getheader('location')` will provide the location to which we were redirected. In the case of a successful request, `r.getheader('content-type')` will tell you the media type of

the response.

- `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 this page 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.

2.1.1) Using the HTTPResponse Object

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 use this info to answer the questions below.

Note for Mac Users!

Some Mac users may encounter an issue along the lines of the following when using `http_response`:

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: unable to get local issuer
certificate (_ssl.c:1091)
```

If you are encountering this issue, you should be able to fix it by installing the `certifi` package using `pip`:

```
pip install --upgrade --force-reinstall certifi
```

Which of the following URLs return a redirect status code?

- ☐ `'https://py.mit.edu'`
- ☐ `'https://py.mit.edu/spring22/labs/lab11'`
- ☐ `'https://py.mit.edu/spring22/info/basics'`
- ☐ `'https://discourse6009.mit.edu/'`
- ☐ `'http://nonexistent.mit.edu/some_file.jpg'`

How many bytes are in the body returned by the following url:

`'https://py.mit.edu/spring22/info/basics'`

What are the first 10 bytes returned from the following URL? Enter your answer as a python bytestring literal.

`'https://py.mit.edu/_static/spring22/logo.gif'`

What is the status code if we make a request for the following URL?

`'http://py.mit.edu'`

What is the `'location'` header associated with a request for the following URL?

`'http://py.mit.edu'`

Enter your answer as a Python string:

2.2) File Manifests

In addition to being able to download regular files, we include a definition of a *file manifest*, which is a way of specifying how a particular file is broken down into parts. Your program can assume that a given URL represents a file manifest if it ends with `.parts`, or if the `Content-Type` header associated with the request is `text/parts-manifest`.

The manifest contains several parts, each of which is separated by a line containing only 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 a file specified by a manifest, you would need to download each of the individual parts, and then to concatenate them together to determine the contents of the entire file.

In addition, the manifest can provide *alternatives* for each section (so that a part may be redundantly stored in multiple locations):

```
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`.

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).

Note that caching is not a good idea if a file changes frequently (that is, frequently enough that a part would be changed within a single download). To handle this, we allow caching for a part only 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, we should first check to see if any of the URLs has been cached. If so, we should simply use that result (without making any HTTP requests). If not, we should follow the normal procedure, but we should also store the result in the cache before continuing. 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).

2.2.2) Example Manifest

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 section 4 of the binary data reference page), 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 39
```

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 39 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.

Suppose a file sequence has the following bytestring body. How many bytes are in the third file?

```
b'\x00\x00\x00\x09\x44\x65\x62\x75\x67\x67\x69\x6E\x67\x00\x00\x00\x02\x69\x73\x00\x00\x00\x04\x68\x61\x72\x64\x00\x00\x00\x03\x62\x75\x74\x00\x00\x00\x04\x66\x75\x6
```

3) Structure

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. "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. This style of download makes sense in a lot of contexts, most notably in situations where we want to download a large file but not need to wait until the whole download has finished in order to make use of some of it (for example, streaming audio and video applications).

`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, thoughout 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 an `HTTPFileNotFoundError`. If a request gives a `500` status code, or if the connection was not made because of a network error, your downloader should raise an `HTTPRuntimeError`. A status code of `200` indicates a successful request.

Note that you can raise exceptions using Python's `raise` keyword, and you can also specify an associated wrror message if you wish, for example:

```python
raise ValueError()


raise ValueError('here is an error message')
```

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

We would like to provide two interfaces to the programs we have just written. We have already provided one such 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).

In addition, 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.

From inside of Python, these arguments are available as part of the `sys.argv` list (note that, if you haven't already, you should add `import sys` near the top of your file at this point). For the example above, `sys.argv` will contain:

```
['lab.py', 'URL', 'FILENAME']
```

Note that you can open a file for writing binary data with `open(filename, 'wb')` and you can write data to the file by calling the `write` method of the resulting file-like object.

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 `test_streaming` should pass), then
- add the command line interface for downloading a file, then
- introduce support for handling redirects and/or errors (after which `test_behaviors` should pass), then
- add support for handling file manifests (after which `test_manifest` should pass), then
- add support for caching (after which `test_cache` 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://py.mit.edu/_static/spring22/logo.gif` (single image)
- `http://hz.mit.edu/009_lab6/bird.jpg` (single image, with single redirect)
- `http://scripts.mit.edu/~6.009/lab6/redir.py/0/wug.jpg` (single image, with many redirects)
- `http://mit.edu/6.009/www/lab6_examples/yellowsub.txt.parts` (manifest file representing a piece of text)
- `http://mit.edu/6.009/www/lab6_examples/fugue.wav.parts` (manifest file representing a piece of music)
- `http://mit.edu/6.009/www/lab6_examples/cornsnake.jpg.parts` (manifest file representing an image, with pieces spread across multiple machines)
- `http://mit.edu/6.009/www/lab6_examples/cached_yellowsub.txt.parts` (same as `yellowsub.txt.parts`, but with caching)
- `http://mit.edu/6.009/www/lab6_examples/kafka.txt-seq` (sequence of many small files, best viewed in GUI)
- `http://scripts.mit.edu/~6.009/lab6/fly.py/fly.txt-seq` (infinitely long sequence)
- `http://scripts.mit.edu/~6.009/lab6/redir.py/0/smallcat.png-seq` (sequence of images representing a short film, many redirects)
- `http://scripts.mit.edu/~6.009/lab6/cats.png-seq.py` (infinitely long manifest of an animation, with simulation of a slow connection)
- `http://scripts.mit.edu/~6.009/lab6/cats.png-seq.cached.py` (like the above, but with caching)
- `http://mit.edu/6.009/www/lab6_examples/skeletons.png-seq.parts` (recursive manifest, sequence representing an infinitely looping animation)
- `http://mit.edu/6.009/www/lab6_examples/skeletons.cached.png-seq.parts` (like the above, but with caching)

Use your downloader to download the file represented by `http://mit.edu/6.009/www/lab6_examples/cornsnake.jpg.parts`. This corresponds to an image, which you can view in an image editor of your choice. Take a look, and then upload the resulting file below:
Select File `No file selected`

Use your downloader to download the animation at `http://scripts.mit.edu/~6.009/lab6/redir.py/0/smallcat.png-seq`. Upload the 53rd PNG file in the resulting file sequence:
Select File `No file selected`

7) 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. In particular, we want to think about kinds of inputs (valid or invalid) that could break the either of the functions we've implemented above (`download_file` and `files_from_sequence`), and adjust the code to account for these errors. **You should be prepared to discuss this during your checkoff conversation.**

8) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `submit-009-lab` script. The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ submit-009-lab -a lab06 /path/to/lab.py
```

Running that script should submit your file to be checked, and it should also provide some information about how and where to get feedback about your submission. Reloading this page after submitting will also show some additional information:

> You have not yet made any submissions to this assignment.

9) Checkoff

Once you are finished with the code, please come to office hours and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases 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:

- Briefly describe the high-level design of your code for parsing manifest files. How do you handle the case of potentially-infinite files? Recursive manifests?
- Briefly describe the high-level design of your implementation of caching, including how you made the cache persist within a single top-level call to `download_file` but not across multiple calls.
- Briefly describe the high-level design of your code for parsing file sequences into their constituent files. How does your code handle the fact that we don't have the complete file sequence available to work with?
- Use the GUI to watch the animation represented by `http://mit.edu/6.009/www/lab6_examples/skeletons.cached.png-seq.parts`. This animation loops indefinitely, but there is a difference in all loops after the first one. Why is this?
- Section 7 asks you to consider other errors that might occur, beyond what the lab currently specifies or what `test.py` tests for. Briefly describe some of the errors you thought about, as well as how you accounted for those possible errors in your code.

You have not yet received this checkoff.

10) (Optional) Additional Extensions

10.1) 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/YOUR_ATHENA_USERNAME/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 IST for information about how to transfer files to your Athena account.

Then try downloading these files.

10.2) 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[1]. 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).

10.3) Avoiding the Recursion Limit

At this point, a fairly common solution to the lab will not be able to play the stream in `skeleton.png-seq.parts` indefinitely, as it will typically error out due to hitting Python's recursion limit. If your program has this issue, try to find a way to fix it!

Footnotes

[1] 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.