CS 120 (Spring 22): Introduction to Computer Programming II

Short Project #11 - Unzip

due at 5pm, Thu 7 Apr 2022

1 Overview

In this project, you will be writing a single function; this function will (very approximately) recreate the DEFLATE algorithm - which will show you how a .zip file stores compressed data. You will be passed an array of tokens, which represent the data in the compressed file; you will return a string, which is the uncompressed data.

2 Python Hints: `type()`

Python has a handy function, which you can use to check for the type of any object: `type`. You can compare the return value to the various standard Python types, like this:

```
foo = [1,2,3]
if type(foo) == list:
    print("Yep, it's a list!")
if type(foo[0]) == int:
    print("Yep, element 0 is an integer")
```

Other common types you might want to check for:

```
str
tuple
set
dict
```

3 Python Hints: `assert`

An `assert` is a statement, somewhere in your code, which states something which must be true. Sometimes, we do this to sanity-check our own code (so that we'll notice if something goes wrong); sometimes, we use it as a way to report errors, if somebody gives us bad input.

To write an assert, you simply write the keyword `assert`, followed by some sort of Boolean expression (that is, something which resolves to `True` or `False`). For instance, if you believe that a string will never be empty, you could write

```
assert msg != ""
```

Every time that Python encounters this line of code, it will check the condition. Hopefully, it's always true; if it is, then Python keeps running your program . But if it turns out to be false, your `assert` will "throw an exception." We'll define exceptions later - but for now, you can know that exceptions kill the function they are in, and the calling function, and the one that called that - eventually killing your entire program. But it's possible to write code that will "catch" an exception - this means that we will notice the exception, handle it, and prevent it from killing the program.

In this assignment, some of the testcases will intentionally send you bad input, and we expect you to detect this with an `assert`. If you do this properly, the testcase will catch the exception, print out a success message, and you pass the testcase! But if you don't check for the error, and keep running, then you will fail the testcase.

On the other hand, if you fail an assertion when we **didn't** expect it, then this will kill the testcase - and that will be a different sort of failure.

4 Background: Zip Format

Did you ever wonder how a `.zip` file stores its data? How can they store all sorts of data, and then later decompress it, without ever losing any of the contents?

It turns out that `.zip` files encode the data in two ways: some portions of the file are simply stored directly, with no compression at all; other portions of the file are listed as **copies** of data that came earlier in the file. This doesn't work for all file formats (for instance, compressing an already-compressed file rarely accomplishes much), but for lots of different types of data (especially text), it works very well.

In fact, this format works so well that this same basic format - with only slight modifications - is also used in `.gz` files (the standard compressed file format for UNIX), and `.png` files (an image format that is widely used on the web).

We'll only be doing a tiny bit of the decoding process - and we won't even attempt to encode a data stream. But if you are curious, you can read about the details of the format online: `https://en.wikipedia.org/wiki/DEFLATE`

4.1 How it Works

In the DEFLATE format, data is made up of a sequence of items, each of which is one of the following:

- Raw, uncompressed data[1]
- A reference to previous data in the **uncompressed** stream. This reference has both a **distance** and a **length**

[1] In the real DEFLATE algorithm, each item is only a single character. But in our program, we will allow strings of any length (except zero).

For example, suppose that you wanted to encode the text

```
See Jack run.  Jack runs up the hill.  Jack and Jill roll down.
```

We won't try to write an encoder for this assignment - but let's consider how we might encode such a string; we'll do this by hand.

First, suppose that you noticed that the word "Jack" showed up in the text a number of times. You would leave the first "Jack" intact, but would replace the second with a reference to previous text, like this:

```
See Jack run.  (11,4) runs up the hill.  Jack and Jill roll down.
```

(In this example, we see that the 2nd "Jack" is 11 characters after the first one, and we wanted to copy 4 characters.)

If we were just a little smarter, we would notice that there is a lot more we can copy. In fact, we can copy the entire string `" Jack run"`, like so:

```
See Jack run. (11,9)s up the hill.  Jack and Jill roll down.
```

Next, you may notice that the string `" Jack "` shows up twice in the string. You can point the third Jack at the second one, like this:

```
See Jack run. (11,9)s up the hill.(24,7)and Jill roll down.
```

Notice that this second compression can point at data which was **expanded by the first compression!** And when it does so, the "distance" that it gives (24 characters, in this case) is measured as the distance in **uncompressed characters,** not in the number of symbols in the compressed data!

4.2 How We'll Represent the Encoding

We will represent a compressed data stream as a list of items. Each item will be one of two things: either a nonempty string, or a tuple which contains two integers. The single character represents a single character in the output; the tuple represents a backwards reference. So the compressed data above would be encoded like this:

```
["See Jack run. ",
    (11,9),
 "s up the hill.",
    (24,7),
 "and Jill roll down."]
```

4.3 In the Real World: Binary Encoding

Looking at the examples above, you may not think that this algorithm works very well; after all, it seems like the backward references take as much space as the original text - maybe even more! That's true, in this simplified picture.

But in the real encoding, the information is packed very, very tightly, with a lot of cool tricks which mean that a backward reference can be encoded in a very small number of bytes. We don't have space to discuss that here - but trust me, it works very well!

4.4 Final Detail: Overlapped Copy Ranges

The following encodes a 10-character string, but it does it in a very strange way. This is valid in the DEFLATE format - and we'll allow it in this assignment as well. What do you think would be the proper decompressed data?

```
['x',   (1,9)]
```

5 Required Function: `unzip()`

`unzip()` takes a compressed stream as its only parameter, and returns the uncompressed string.

The compressed stream is encoded as described earlier in this spec; the stream is a list, and each element is either a string (with a single character), or a tuple with two integers.

5.1 `asserts`

This function must assert that:

- Every element in the compressed stream is the proper type (string or tuple)
- Every string is at least one character long
- Every tuple has exactly two elements, both of which are positive integers
- The "offset" (the first element in the tuple) does not point back any further than the start of the uncompressed string.

6 Turning in Your Solution

You must turn in your code using GradeScope.

You must write a file named `unzip.py`. It must contain (at least) one function: `unzip()`.