Code Golfing for Characters (not Bytes)

Russell Schwartz

March 15, 2023

Abstract

Code golf challenges are usually judged by measuring the length of the source code in bytes. However, certain venues instead count the number of characters. If a variable-length character encoding like UTF-8 is used, and non-ASCII-range characters are allowed, it becomes possible to pack more information into the same number of characters by using high-value code points. We propose a general method for Python code golf that involves compressing source code into a dense string of unicode characters which, when packaged along with a decoder, is (hopefully) shorter than the original.

Introduction

In regular golf, players try to complete a hole in as few strokes of the club as possible. In code golf, programmers try to complete a coding challenge (also called a "hole") in as few *key* strokes as possible. In other words, the challenge is the write the shortest possible computer program that performs some task in an agreed upon language. Holes are usually judged by measuring the length of the source code in bytes. However, certain venues (e.g. `https://code.golf/`) instead count the number of characters. If a variable-length character encoding like UTF-8 is used, and non-ASCII-range characters are allowed, it becomes possible to pack more information into the same number of characters by using high-value code points.

Source code generally consists only of ASCII characters, whose values range from 0 to 127 (giving 8 bits of information per character). Unicode ordinals on the other hand range all the way from 0 to 1,112,063 (giving 20 bits of information per character). Thus, a unicode string has the potential to be over twice as information-dense as an ASCII string. We present a method for losslessly compressing python source code into this denser format while remaining executable.

Methodology

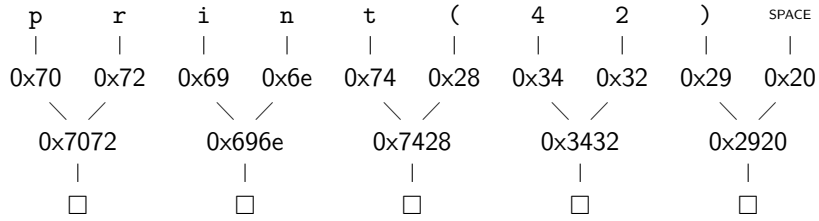
We convert an ASCII string into a unicode string by merging pairs of consecutive ASCII chars into a single unicode char. Let a and b represent adjacent ASCII chars. The resulting unicode char's ordinal is given by

$$\mathsf{c} = (\mathsf{a} >> 8) \mid \mathsf{b}$$

effectively setting the high byte to be a and the low byte to be b. The maximum possible value is

$$(\mathsf{0x7f} >> 8) \mid \mathsf{0x7f} = \mathsf{0x7f7f} = 32639 \leq 1{,}112{,}063$$

so we are comfortably within the bounds of unicode. We are effectively using 16 of the available 20 bits. If our original string has an odd number of characters, we pad it with a space. For example, we have the following conversion for the string `print(42)`. Convincing latex to actually render the resulting characters, which mostly fall in the range of *CJK Unified Ideographs*, is left as an exercise to the reader.



Decompression is straightforward. For each unicode character, split the bytes and convert each back to ASCII. In fact, the decompression algorithm is so simple that it can be written in python using only ~ 50 characters. Finally, we pass the decompressed result to `exec()`, which invokes the interpreter.

Results

The results are quite beautiful. Converting to unicode results in a compression ratio of 2, however we have to include the decoder and the call to `exec()`, so there is a break-even point below which this method actually makes the code longer. The overhead incurred by decoding is 58 chars, so we break even at 168 chars. Beyond that point, it's all gravy.

Example 1: Print all palindromic primes under 1000

```
# -------- ORIGINAL (236 chars) --------
def is_prime(n):
    for d in range(2, n):
        if n % d == 0:
            return False
    return True

def is_palindrome(n):
    return str(n) == str(n)[::-1]

for n in range(2, 1000):
    if is_prime(n) and is_palindrome(n):
        print(n)
```

```
# ------- COMPRESSED (124 chars) -------
exec(''.join(chr(ord(c)>>8&255)+chr(ord(c)&255)for c
in"摥映楳彰物浥⡮⤺ਠ†潲渠牡湧攨㈬⤺ਠ†††映渠┠搽㴰㨊††††††††牥瑵
牮⁆慬獥ਠ†牥瑵牮⁔牵攊੤敦⁩獟灡汩湤牯浥⡮⤺ਠ†牥瑵牮⁳瑲⡮⤠㴽⁳瑲⡮⥛
㨺ⴱ崊੦潲渠楮⁲慮来⠲Ⱐ㄰〰⤺ਠ†楦⁩獟灲業攨温⁡湤⁩獟灡汩湤牯浥⡮⤺ਊ
⡮⤊††††灲楮琨温⤊"))
```

Example 2: Roman to Arabic Numerals Converter

```
# -------- ORIGINAL (156 chars) --------
import sys
d=dict(zip('IVXLCDM',[1,5,10,50,100,500,1000]))
for a in sys.argv[1:]:print(sum(d[x]*(i<len(a)-1and-
(d[x]<d[a[i+1]])or 1)for i,x in enumerate(a)))
```

```
# ------- COMPRESSED (138 chars) -------
exec("".join(chr(ord(x)>>8&255)+chr(ord(x)&255)for x
in"業灯牴⁳祳੤㵤楣琨穩瀨❉噘䱃䑍✬嬱ⰵⰱ〬㔰Ⱡ〰ⰵ〰Ⱡ〰〩⤊景爠愠楮⁳祳⹡牧
癛ㄺ崺灲楮琨獵洨摛硝⨨椼汥渨愩ⴱ慮搭⡤學崼摛慛椫ㅝ崩潲‱⥦潲⁩ⱸ⁩渠敮畭敲慴
攨愩⤩⤊"))
```

Example 3: Brainf*ck Interpreter

```
# -------- ORIGINAL (205 chars) --------
import sys
for P in sys.argv:
 o="A=[0]*99;p=0";l=0
 for c in P:o+="\n"+" "*l+"p+=1|p-=1|A[p]+=1|A[p]-
=1|print(end=chr(A[p]))|while A[p]:|".split("|")[">
<+-.[]".index(c)];l+=(c in"[]")*(92-ord(c))
 exec(o)
```

```
# ------- COMPRESSED (162 chars) -------
exec(''.join(chr(ord(c)>>8&255)+chr(ord(c)&255)for c
in"業灯牴⁳祳੦潲⁐⁩渠獹献慲杶㨊⁯㴢䄽嬰崪㤹㭰㴰∻氽《⁦潲⁣⁩渠债漫㴢屮∫∠∪氫≰
⬽ㅼ瀭㴱籁孰崫㴱籁孰崭㴱籰物湴⡥湤㵣桲⡁孰崩⥼睨楬攠䅛灝㩼∮獰汩琨≼∩嬢
㸼⬭⸮嬢崢⹩湤數⡣⥝㭬⬽⡣⁩渢嬢崢⤪⠹㈭潲搨挩⤊數散⡯⤊"))
```

Conclusion

The results really speak for themselves. Not only is the compressed code shorter, but it is also only a single line! Imagine telling your project manager that you just refactored your 10 KLOC codebase into a single line. You might want to also mention that you just doubled or even tripled the total size on disk, but that's the price we pay for greatness.