Austin Z. Henley
Assistant Professor

azh@utk.edu
@austinzhenley
github/AZHenley

Python strings are immutable, but only sometimes

2/15/2021

Update 2/16/2021: See the discussion of this post on Hacker News.

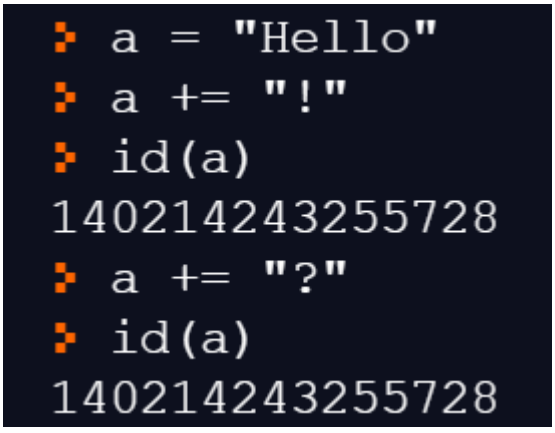
The standard wisdom is that Python strings are immutable. You can't change a string's value, only the reference to the string. Like so:

```
x = "hello"
x = "goodbye"  # New string!
```

Which implies that each time you make a change to a string variable, you are actually producing a brand new string. Because of this, tutorials out there warn you to avoid string concatenation inside a loop and advise using *join* instead for performance reasons. Even the official documentation says so!

This is wrong. *Sort of*.

There is a common case for when strings in Python are actually mutable. I will show you an example by inspecting the string object's unique ID using the builtin *id()* function, which is just the memory address. The number is different for each object. (Objects can be shared though, such as with interning.)



I concatenated two strings but the memory address did not change!

For an extra sanity check, let's make our own "pointer" and see if it points to the original or modified string.

```
> a = "abcd"
> a += "e"
> b = id(a)
> a += "f"
> a
'abcdef'
> _ctypes.PyObj_FromPtr(b)
'abcdef'
```

If strings were truly immutable, then the address we stored in *b* would point to the original string. However, we see that the strings printed are equivalent.

We can try another test to see how often we get a new string object by doing 10,000 small concatenations.

Only 46 of the 10,000 concatenations allocated a new string!

So what is going on here?

CPython is clever. If there are no other references to the string, then it will attempt to mutate the string instead of allocating a new one. Though it will sometimes need to resize the buffer if the string grows too big, much like C++'s *vector* or C#'s *List*.

We can see the first clue in the code for this in CPython's Python/ceval.c file. It tries to reduce the reference count to 1 if possible when doing a string concatenation.

```
5945  static PyObject *
5946  unicode_concatenate(PyThreadState *tstate, PyObject *v, PyObject *w,
5947                      PyFrameObject *f, const _Py_CODEUNIT *next_instr)
5948  {
5949      PyObject *res;
5950      if (Py_REFCNT(v) == 2) {
5951          /* In the common case, there are 2 references to the value
5952           * stored in 'variable' when the += is performed: one on the
5953           * value stack (in 'v') and one still stored in the
5954           * 'variable'.  We try to delete the variable now to reduce
5955           * the refcnt to 1.
5956           */
5957          int opcode, oparg;
5958          NEXTOPARG();
5959          switch (opcode) {
5960          case STORE_FAST:
5961          {…
5966          }
5967          case STORE_DEREF:
5968          {…
5977          }
5978          case STORE_NAME:
5979          {…
5993          }
5994          }
5995      }
5996      res = v;
5997      PyUnicode_Append(&res, w);
5998      return res;
5999  }
```

Now take a look at *PyUnicode_Append()* in Objects/unicodeobject.c. If the string is modifiable (has only 1 reference and is not interned), then it will try to append in place! *unicode_resize()* will see if it will fit or if a new allocation needs to be made.

```
11737     if (unicode_modifiable(left)
11738         && PyUnicode_CheckExact(right)
11739         && PyUnicode_KIND(right) <= PyUnicode_KIND(left)
11740         /* Don't resize for ascii += latin1. Convert ascii to latin1 requires
11741            to change the structure size, but characters are stored just after
11742            the structure, and so it requires to move all characters which is
11743            not so different than duplicating the string. */
11744         && !(PyUnicode_IS_ASCII(left) && !PyUnicode_IS_ASCII(right)))
11745     {
11746         /* append inplace */
11747         if (unicode_resize(p_left, new_len) != 0)
11748             goto error;
11749
11750         /* copy 'right' into the newly allocated area of 'left' */
11751         _PyUnicode_FastCopyCharacters(*p_left, left_len, right, 0, right_len);
11752     }
11753     else {
11754         maxchar = PyUnicode_MAX_CHAR_VALUE(left);
11755         maxchar2 = PyUnicode_MAX_CHAR_VALUE(right);
```

There we have it. Evidence that you *can* mutate a string in Python.

Does this *really* count as mutability though? Not really. The string would be thrown away anyway so this is just an optimization to reuse the memory. The important takeaway here is that you are not allocating a new string every single time like the internet says.