Python Strings

Python has a built-in string class named "str" with many handy features (there is an older module named "string" which you should not use). String literals can be enclosed by either double or single quotes, although single quotes are more commonly used. Backslash escapes work the usual way within both single and double quoted literals -- e.g. \n \' \". A double quoted string literal can contain single quotes without any fuss (e.g. "I didn't do it") and likewise single quoted string can contain double quotes. A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline. String literals inside triple quotes, """ or ''', can multiple lines of text.

Python strings are "immutable" which means they cannot be changed after they are created (Java strings also use this immutable style). Since strings can't be changed, we construct *new* strings as we go to represent computed values. So for example the expression ('hello' + 'there') takes in the 2 strings 'hello' and 'there' and builds a new string 'hellothere'.

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing, so if str is 'hello' str[1] is 'e'. If the index is out of bounds for the string, Python raises an error. The Python style (unlike Perl) is to halt if it can't tell what to do, rather than just make up a default value. The handy "slice" syntax (below) also works to extract any substring from a string. The len(string) function returns the length of a string. The [ ] syntax and the len() function actually work on any sequence type -- strings, lists, etc.. Python tries to make its operations work consistently across different types. Python newbie gotcha: don't use "len" as a variable name to avoid blocking out the len() function. The '+' operator can concatenate two strings. Notice in the code below that variables are not pre-declared -- just assign to them and go.

```
s = 'hi'
print s[1]          ## i
print len(s)        ## 2
print s + ' there'  ## hi there
```

Unlike Java, the '+' does not automatically convert numbers or other types to string form. The str() function converts values to a string form so they can be combined with other strings.

```
pi = 3.14
##text = 'The value of pi is ' + pi      ## NO, does not work
```

```
text = 'The value of pi is '  + str(pi)  ## yes
```

For numbers, the standard operators, +, /, * work in the usual way. There is no ++ operator, but +=, -=, etc. work. If you want integer division, it is most correct to use 2 slashes -- e.g. 6 // 5 is 1 (previous to python 3000, a single / does int division with ints anyway, but moving forward // is the preferred way to indicate that you want int division.)

The "print" operator prints out one or more python items followed by a newline (leave a trailing comma at the end of the items to inhibit the newline). A "raw" string literal is prefixed by an 'r' and passes all the chars through without special treatment of backslashes, so r'x\nx' evaluates to the length-4 string 'x\nx'. A 'u' prefix allows you to write a unicode string literal (Python has lots of other unicode support features -- see the docs below).

```
raw = r'this\t\n and that'
print raw     ## this\t\n and that

multi = """It was the best of times.
It was the worst of times."""
```

String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common string methods:

- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced

by 'new'

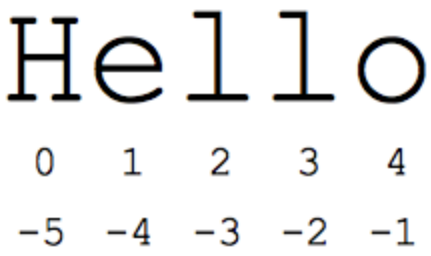
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

A google search for "python str" should lead you to the official python.org string methods which lists all the str methods.

Python does not have a separate character type. Instead an expression like s[8] returns a string-length-1 containing the character. With that string-length-1, the operators ==, <=, ... all work as you would expect, so mostly you don't need to know that Python does not have a separate scalar "char" type.

String Slices

The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice s[start:end] is the elements beginning at start and extending up to but not including end. the Suppose we have s = "Hello"



- s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4
- s[1:] is 'ello' -- omitting either index defaults to the start or end of the string
- s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)
- s[1:100] is 'ello' -- an index that is too big is truncated down to the string length

The standard zero-based index numbers give easy access to chars near the start of the string. As an alternative, Python uses negative numbers to give easy access to the chars at the end of

the string: s[-1] is the last char 'o', s[-2] is 'l' the next-to-last char, and so on. Negative index numbers count back from the end of the string:

- s[-1] is 'o' -- last char (1st from the end)
- s[-4] is 'e' -- 4th from the end
- s[:-3] is 'He' -- going up to but not including the last 3 chars.
- s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.

It is a neat truism of slices that for any index n, `s[:n] + s[n:] == s`. This works even for n negative or out of bounds. Or put another way s[:n] and s[n:] always partition the string into two string parts, conserving all the characters. As we'll see in the list section later, slices work with lists too.

String %

Python has a printf()-like facility to put together a string. The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parenthesis):

```
# % operator
text = "%d little pigs come out or I'll %s and %s and %s" % (3, 'huff', 'puff'
```

The above line is kind of long -- suppose you want to break it into separate lines. You cannot just split the line after the '%' as you might in other languages, since by default Python treats each line as a separate statement (on the plus side, this is why we don't need to type semi-colons on each line). To fix this, enclose the whole expression in an outer set of parenthesis -- then the expression is allowed to span multiple lines. This code-across-lines technique works with the various grouping constructs detailed below: ( ), [ ], { }.

```
# add parens to make the long-line work:
text = ("%d little pigs come out or I'll %s and %s and %s" %
  (3, 'huff', 'puff', 'blow down'))
```

i18n Strings (Unicode)

Regular Python strings are *not* unicode, they are just plain bytes. To create a unicode string, use the 'u' prefix on the string literal:

```
> ustring = u'A unicode \u018e string \xf1'
> ustring
u'A unicode \u018e string \xf1'
```

A unicode string is a different type of object from regular "str" string, but the unicode string is compatible (they share the common superclass "basestring"), and the various libraries such as regular expressions work correctly if passed a unicode string instead of a regular string.

To convert a unicode string to bytes with an encoding such as 'utf-8', call the ustring.encode('utf-8') method on the unicode string. Going the other direction, the unicode(s, encoding) function converts encoded plain bytes to a unicode string:

```
## (ustring from above contains a unicode string)
> s = ustring.encode('utf-8')
> s
'A unicode \xc6\x8e string \xc3\xb1'  ## bytes of utf-8 encoding
> t = unicode(s, 'utf-8')             ## Convert bytes back to a unicode string
> t == ustring                      ## It's the same as the original, yay!

True
```

The built-in print does not work fully with unicode strings. You can encode() first to print in utf-8 or whatever. In the file-reading section, there's an example that shows how to open a text file with some encoding and read out unicode strings. Note that unicode handling is one area where Python 3000 is significantly cleaned up vs. Python 2.x behavior described here.

If Statement

Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements. The boolean test for an if does not need to be in parenthesis (big difference from C++/Java), and it can have *elif* and *else* clauses (mnemonic: the word "elif" is the same length as the word "else").

Any value can be used as an if-test. The "zero" values all count as false: None, 0, empty string, empty list, empty dictionary. There is also a Boolean type with two values: True and False (converted to an int, these are 1 and 0). Python has the usual comparison operations: ==, !=, <, <=, >, >=. Unlike Java and C, == is overloaded to work correctly with strings. The boolean operators are the spelled out words *and*, *or*, *not* (Python does not use the C-style && || !). Here's what the code might look like for a policeman pulling over a speeder -- notice how each block of then/else statements starts with a : and the statements are grouped by their indentation:

```
if speed >= 80:
  print 'License and registration please'
  if mood == 'terrible' or speed >= 100:
    print 'You have the right to remain silent.'
  elif mood == 'bad' or speed >= 90:
    print "I'm going to have to write you a ticket."
    write_ticket()
  else:
    print "Let's try to keep it under 80 ok?"
```

I find that omitting the ":" is my most common syntax mistake when typing in the above sort of code, probably since that's an additional thing to type vs. my C++/Java habits. Also, don't put the boolean test in parens -- that's a C/Java habit. If the code is short, you can put the code on the same line after ":", like this (this applies to functions, loops, etc. also), although some people feel it's more readable to space things out on separate lines.

```
if speed >= 80: print 'You are so busted'
else: print 'Have a nice day'
```

Exercise: string1.py

To practice the material in this section, try the **string1.py** exercise in the Basic Exercises.

Last updated February 5, 2015.