Strings

Strings

Three ways to define string literals:

- with single quotes: 'Ni!'
- double quotes: "Ni!"
- Or with triples of either single or double quotes, which creates a multi-line string:

```
>>> """I do HTML for them all,
... even made a home page for my dog."""
'I do HTML for them all,\neven made a home page for my dog.'
```

Strings

Note that the REPL echoes the value with a \n to represent the newline character. Use the print function to get your intended output:

```
>>> nerdy = """I do HTML for them all,
... even made a home page for my dog."""
>>> nerdy
'I do HTML for them all,\neven made a home page for my dog.'
>>> print(nerdy)
I do HTML for them all,
even made a home page for my dog.
```

That's pretty nerdy.

Strings

Choice of quote character is usually a matter of taste, but the choice can sometimes buy convenience. If your string contains a quote character you can either escape it:

```
>>> journey = 'Don\'t stop believing.'
```

or use the other quote character:

```
>>> journey = "Don't stop believing."
```

Active Review

- How does Python represent the value of the variable `journey`, that is, how is it echoed by the REPL?

String Operations

Because strings are sequences we can get a string's length with `len()`:

```
>>> i = "team"
>>> len(i)
4
```

and access characters in the string by index (offset from beginning – first index is 0) using `[]`:

```
>>> i[1]
'e'
```

Note that the result of an index access is a string:

```
>>> type(i[1])
<class 'str'>
>>> i[3] + i[1]
'me'
>>> i[-1] + i[1] # Note that a negative index goes from the end
'me'
```

Active Review

- What is the index of the first character of a string?
- What is the index of the last character of a string?

String Slicing

`[:end]` gets the first characters up to but not including `end`

```
>>> al_gore = "manbearpig"
>>> al_gore[:3]
'man'
```

`[begin:end]` gets the characters from `begin` up to but not including end

```
>>> al_gore[3:7]
'bear'
```

`[begin:]` gets the characters from `begin` to the end of the string

```
>>> al_gore[7:]
'pig'
>>>
```

Active Review

- What is the relationship between the ending index of a slice and the beginning index of a slice beginning right after the first slice?

String Methods

`str` is a class (you'll learn about classes later) with many methods (a method is a function that is part of an object). Invoke a method on a string using the dot operator.

`str.find(substr)` returns the index of the first occurence of `substr` in `str`

```
>>> 'foobar'.find('o')
1
```

Active Review

- Write a string slice expression that returns the username from an email address, e.g., for 'bob@aol.com' it returns 'bob'.
- Write a string slice expression that returns the host name from an email address, e.g., for 'bob@aol.com' it returns 'aol.com'.

String Interpolation with %

The old-style (2.X) string format operator, %, takes a string with format specifiers on the left, and a single value or tuple of values on the right, and substitutes the values into the string according to the conversion rules in the format specifiers. For example:

```
>>> "%d %s %s %s %f" % (6, 'Easy', 'Pieces', 'of', 3.14)
'6 Easy Pieces of 3.140000'
```

Here are the conversion rules:

- %s string
- %d decimal integer
- %x hex integer
- %o octal integer
- %f decimal float
- %e exponential float
- %g decimal or exponential float
- %% a literal

String Formatting with %

Specify field widths with a number between % and conversion rule:

```
>>> sunbowl2012 = [('Georgia Tech', 21), ('USC', 7)]
>>> for team in sunbowl2012:
...     print('%14s %2d' % team)
...
Georgia Tech 21
USC           7
```

Fields right-aligned by default. Left-align with - in front of field width:

```
>>> for team in sunbowl2012:
...     print('%-14s %2d' % team)
...
Georgia Tech 21
USC           7
```

Specify n significant digits for floats with a .n after the field width:

```
>>> '%5.2f' % math.pi
' 3.14'
```

Notice that the field width indludes the decimal point and output is left-padded with spaces

String Interpolation with `str.format()`

Python 3.0 - 3.5 interpolation was done with the string method `format`:

```
>>> "{} {} {} {} {}".format(6, 'Easy', 'Pieces', 'of', 3.14)
'6 Easy Pieces of 3.14'
```

Old-style formats only resolve arguments by position. New-style formats can take values from any position by putting the position number in the {} (positions start with 0):

```
>>> "{4} {3} {2} {1} {0}".format(6, 'Easy', 'Pieces', 'of', 3.14)
'3.14 of Pieces Easy 6'
```

Can also use named arguments, like functions:

```
>>> "{count} pieces of {kind} pie".format(kind='punkin', count=3)
'3 pieces of punkin pie'
```

Or dictionaries (note that there's one dict argument, number 0):

```
>>> "{0[count]} pieces of {0[kind]} pie".format({'kind':'punkin',
'count':3})
'3 pieces of punkin pie'
```

String Formatting with `str.format()`

Conversion types appear after a colon:

```
>>> "{:d} {} {} {} {:f}".format(6, 'Easy', 'Pieces', 'of', 3.14)
'6 Easy Pieces of 3.140000'
```

Argument names can appear before the :, and field formatters appear between the : and the conversion specifier (note the < and > for left and right alignment):

```
>>> for team in sunbowl2012:
...     print('{:<14s} {:>2d}'.format(team[0], team[1]))
...
Georgia Tech 21
USC           7
```

You can also unpack the tuple to supply its elements as individual arguments to format (or any function) by prepending tuple with *:

```
>>> for team in sunbowl2012:
...     print('{:<14s} {:>2d}'.format(*team))
...
Georgia Tech 21
USC           7
```

f-Strings

Python 3.6 introduced a much more convenient inline string interpolator. Prepend `f` to the opening quote, enclose arbitrary Python expressions in culy braces (`{}`), and put formatters similar to `str.format()` after colons.

```
>>> for team, score in sunbowl2012:       # Tuple-unpacking assignment
...     print(f'{team:<14s} {score:>2d}')
...
Georgia Tech   21
USC             7
```

Conclusion

- Strings are a kind of `Sequence`
- Unlike some other languages, it's not a `Sequence[char]` – single characters are also `str` s
- Strings are immutable, so operations that "modify" strings actually return new strings containing the modificaitons