Lecture 1.2 : Strings

Introduction

- A string is simply a *sequence* of characters. The string type is a *collection type* meaning it contains a number of objects (characters in this case) that can be treated as a single object. The string type is a particular type of collection type called a *sequence type*. This means each constituent object (here character) in the collection occupies a specific numbered location within it i.e. elements are *ordered*.

- Python strings are enclosed in either single or double quotes. (Pick a style and be consistent.) Python strings can contain *non-printing* characters (such as a `\n` which causes a new line to be emitted when printing the string).

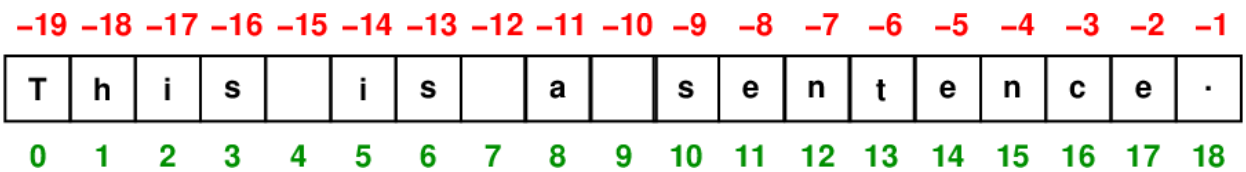
```
>>> name1 = "Connie Smith"
>>> name2 = "Timmy O'Brien"
>>> name3 = 'Jenny Murphy'
>>> name4 = 'Tommy O\'Neill'
>>> two_lines = 'This line is above\nthis line.'
>>> print(name1)
Connie Smith
>>> print(name2)
Timmy O'Brien
>>> print(name3)
Jenny Murphy
>>> print(name4)
Tommy O'Neill
>>> print(two_lines)
This line is above
this line.
```

- Note if we use single quotes then any apostrophes in the string must be *escaped* with a backslash in order to prevent them signifying the end of the string to Python as in `'Tommy O\'Neill'`.

String representation

- As mentioned, a string is a *sequence* type. This means each member object (character in this case) occupies a numbered position in the collection and can be accessed by *indexing* the sequence at that index.

- For example, the characters of the string `'This is a sentence.'` reside at the indices indicated here:



- We can extract individual characters by *indexing* the string at a given location. The first character in the string is located at index zero. Thus if the length of the string is N (i.e. it contains N characters), because indices begin at zero, the final character in the string is located at index N-1. Indexing beyond N-1 is an error.

```
>>> s = 'This is a sentence.'
>>> s[0]
'T'
>>> s[1]
'h'
>>> s[2]
'i'
>>> s[3]
's'
>>> s[4]
' '
>>> len(s)
19
>>> s[18]
'.'
>>> s[19]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

- In Python it is possible to index relative to the end of the string using negative indices: the last character is at index -1, the second last at index -2, the third last at index -3, etc.

```
>>> s = 'This is a sentence.'
>>> s[-1]
'.'
>>> s[-2]
'e'
>>> s[-3]
'c'
```

String slicing

- We can extract more than a single character from a string. We can extract subsequences or *slices* by specifying a range of indices separated by a colon. Writing `s[start:end]` will return a new string composed of the characters `s[start]`, `s[start+1]`, `s[start+2]`, `...`, `s[end-3]`, `s[end-2]`, `s[end-1]`. Note that the character located at `s[end]` is *not* returned.

```
>>> s = 'This is a sentence.'
>>> s[0:4]
'This'
>>> s[5:7]
'is'
>>> s[8:9]
'a'
>>> s[10:18]
'sentence'
```

- If either the starting or ending indices are omitted their values default to the beginning and end of the string.

```
>>> s = 'This is a sentence.'
>>> s[0:4]
'This'
>>> s[:4]
'This'
>>> s[10:19]
```

```
'sentence.'
>>> s[10:]
'sentence.'
>>> s[:]
'This is a sentence.'
```

- As usual, negative indices can be used to specify locations relative to the end of the string.

```
>>> s = 'This is a sentence.'
>>> s[:-10]
'This is a'
```

Extended slicing

- It is also possible to specify a third parameter when slicing sequences. It indicates the *step size* to take along the sequence when extracting its elements. Writing `s[start:end:step]` will return a new string composed of `s[start]`, `s[start+step]`, `s[start+2*step]`, `s[start+3*step]`, etc. Extraction continues for as long as `start+i*step < end` where i = 0, 1, 2, 3, etc.

```
>>> s = 'This is a sentence.'
>>> s[::1]
'This is a sentence.'
>>> s[::2]
'Ti sasnec.'
>>> s[::3]
'Tss nn.'
>>> s[0:3:3]
'T'
>>> s[0:4:3]
'Ts'
```

- A negative step size is interpreted as a step backwards through the sequence. This is handy for reversing a string. (If the starting and ending indices are omitted, for negative step sizes, their values default to the end and beginning of the string respectively.)

```
>>> s = 'This is a sentence.'
>>> s[-1:-20:-1]
'.ecnetnes a si sihT'
>>> s[::-1]
'.ecnetnes a si sihT'
```

String concatenation and replication

- We can use the + and * operators to concatenate and replicate strings:

```
>>> s = 'This is a sentence.'
>>> t = 'This is yet another sentence.'
>>> s + ' This is another sentence. ' + t
'This is a sentence. This is another sentence. This is yet another sentence.'
>>> s * 3
'This is a sentence.This is a sentence.This is a sentence.'
>>> s + ' ' * 3
'This is a sentence.   '
```

```
>>> (s + ' ') * 3
'This is a sentence. This is a sentence. This is a sentence. '
```

String testing

- Strings can be tested for equality with the == operator:

```
>>> 'cat' == 'cat'
True
>>> 'cat' == 'dog'
False
```

- The empty string `''` is interpreted as `False`. Any non-empty string is interpreted as `True`:

```
>>> if '':
...     print('Empty string is True')
... else:
...     print('Empty string is False')
...
Empty string is False
>>> if 'a':
...     print('Non-empty string is True')
... else:
...     print('Non-empty string is False')
...
Non-empty string is True
```

Strings are iterable

- Because a string is an *iterable* sequence we can use a `for` loop to examine each of its characters in turn:

```
>>> s = 'This is a sentence.'
>>> for c in s:
...     print(c)
...
T
h
i
s

i
s

a

s
e
n
t
e
n
c
e
.
>>> for c in s:
...     print(c, end='')
```

```
...
This is a sentence.>>>
```

Strings are immutable

- Strings are *immutable*. This means they cannot be modified. If we try to modify a string we get an error:

```
>>> s = 'This is a sentence.'
>>> s[0] = 't'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- If we want a "modify" a string we have to build a whole new version from the original:

```
>>> s = 'This is a sentence.'
>>> s = 't' + s[1:]
>>> s
'this is a sentence.'
```

String methods

- Python comes with built-in support for a set of common string operations. These operations are called `methods` and they define the things we can do with strings. Calling `help(str)` in the Python shell or `pydoc str` in a Linux terminal outputs a list of these methods. We see the methods we can invoke on a string `s` include `capitalize()` (returns a capitalized version of `s`), `isdecimal()` (returns `True` if `s` contains only decimal characters), `lower()` (returns a new copy of `s` with all characters converted to lowercase), etc.

```
>>> s = 'This is a sentence.'
>>> n = '123'
>>> t = s.lower()
>>> t
'this is a sentence.'
>>> s
'This is a sentence.'
>>> n.isdecimal()
True
>>> s.isdecimal()
False
>>> t.capitalize()
'This is a sentence.'
```

- Note that, because strings are immutable, calling a method on a string will *not* alter the string itself.

- Whenever you find it necessary to carry out some string processing, first look up the list of built-in string methods. There may be one that will help you with your task. There is no point writing your own code that duplicates what a built-in string method can do for you already.