`for` loops over indices

`range`

Here is the top part of the help for `range`:

```
class range(object)
 |  range([start,] stop[, step]) -> range object
 |
 |  Returns a virtual sequence of numbers from start to stop by step.
```

The `stop` value is not included.

`range` is typically used in a `for` loop to iterate over a sequence of numbers. Here are some examples:

```
# Iterate over the numbers 0, 1, 2, 3, and 4.
for i in range(5):


# Iterate over the numbers 2, 3, and 4.
for i in range(2, 5):


# Iterate over the numbers 3, 6, 9, 12, 15, and 18.
for i in range(3, 20, 3):
```

Iterating over the indices of a list

Because `len` returns the number of items in a list, it can be used with `range` to iterate over all the indices. This loop prints all the values in a list:

```
for i in range(len(lst)):
    print(lst[i])
```

This also gives us flexibility to process only part of a list. For example, We can print only the first half of the list:

```
for i in range(len(lst) // 2):
    print(lst[i])
```

Or every other element:

```
for i in range(0, len(lst), 2):
    print(lst[i])
```

Not "What" but "Where"

Previously, we have written loops over characters in a string or items in a list. However, sometimes there are problems where knowing the value of the items in a list or the characters in a string is not enough; we need to know where it occurs (i.e. its index).

Example 1

The first example is described below:

```
def count_adjacent_repeats(s):
    ''' (str) -> int

    Return the number of occurrences of a character and an adjacent character
    being the same.

    >>> count_adjacent_repeats('abccdeffggh')
    3
    '''

    repeats = 0

    for i in range(len(s) - 1):
        if s[i] == s[i + 1]:
            repeats = repeats + 1

    return repeats
```

We want to compare a character in the string with another character in the string beside it. This is why we iterate over the indices because the location is important, and only knowing the value of the character does not provide us with enough information. This is how we are able to count repeated characters in a string. We can't execute the body of the loop if `i` is `len(s) - 1` because we compare to `s[i + 1]`, and that would produce an `IndexError`.

Example 2

The second example is described below:

```
def shift_left(L):
    ''' (list) -> NoneType

    Shift each item in L one position to the left and shift the first item to
    the last position.

    Precondition: len(L) >= 1

    >>> shift_left(['a', 'b', 'c', 'd'])
    '''

    first_item = L[0]

    for i in range(len(L) - 1):
        L[i] = L[i + 1]

    L[-1] = first_item
```

For the same reasons as above, merely knowing the value of the items in the list is not enough since we need to know where the items are located; we need to know the index (position) of the item in the list.

Jennifer Campbell • Paul Gries
University of Toronto