RANGE

SPECIALIZED WAYS:

There are situations where you will need to iterate in more specialized ways.
For example:

- What if you need to visit every second or third item in a list, or change the list along the way?

- How about traversing more than one sequence in parallel, in the same for loop?

- What if you need indexes too?

You can always code such unique iterations with a **while** loop and manual indexing, but Python provides a set of **built-ins** that allow you to **specialize the iteration in a for**:

- The **built-in range function** (available since Python 0.X) produces **a series of successively higher integers,** which can be used as indexes in a **for** loop.

Because **for** loops may run quicker than **while**-based counter loops, though, it's to your advantage to use tools like these that allow you to use for whenever possible.

Let's look at each of these built-ins in turn, in the context of common use cases. As we'll see, their usage may differ slightly between 2.X and 3.X, and some of their applications are more valid than others.

Enhance Skill

RANGE

Counter Loops: Our first loop-related function, **range**, is really a general tool that can be used in a variety of contexts. Although it's used most often to generate ***indexes in a* for**, you can use it anywhere you need a series of integers.

In Python 2.X range creates a physical list; in 3.X, range is an iterable that generates items on demand, so we need to wrap it in a list call to display its results all at once in 3.X only:

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
>>>
```

With one argument, **range** *generates a list of integers* from zero up to but not including the argument's value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a step; if it is used, Python adds the step to each successive integer in the result (the step defaults to +1). Ranges can also be non-positive and non-ascending, if you want them to be:

```
>>> list(range(−5, 5))
[−5, −4, −3, −2, −1, 0, 1, 2, 3, 4]
>>>
>>> list(range(5, −5, −1))
[5, 4, 3, 2, 1, 0, −1, −2, −3, −4]
>>>
```

Although such range results may be useful all by themselves, they tend to come in most handy within for loops. For one thing, they provide a simple way to repeat an action a specific number of times.

To print three lines, for example, use a range to generate the appropriate number of integers:

```
>>> for i in range(3):
        print(i, 'Pythons')
0 Pythons
1 Pythons
2 Pythons
>>>
```

Note: that **for** loops force results from range automatically in 3.X, so we don't need to use a list wrapper here in 3.X.

Enhance Skill

Sequence Scans: while and range versus for

The **range** call is also sometimes used to iterate over a sequence indirectly, though it's often not the best approach in this role. The easiest and generally fastest way to step through a sequence exhaustively is always with a simple **for**, as Python handles most of the details for you:

```
>>> X = 'spam'
>>> for item in X:
        print(item, end=' ')          # Simple iteration
s p a m
```

Internally, the **for** loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly, you can do it with a while loop:

```
>>> i = 0
>>> while i < len(X):               # while loop iteration
        print(X[i], end=' ')
        i += 1
s p a m
```

You can also do **manual indexing** with a **for**, though, if you use **range** to generate a list of indexes to iterate through. It's a multistep process, but it's sufficient to generate **offsets**, rather than the items at those offsets:

```
>>> X
'spam'
>>> len(X)                          # Length of string
4
>>> list(range(len(X)))             # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)):
        print(X[i], end=' ')          # Manual range/len iteration
s p a m
```

Although the **range/len** combination suffices in this role, it's probably not the best option. It may run slower, and it's also more work than we need to do. Unless you have a special indexing requirement, you're better off using the simple for loop form in Python:

```
>>> for item in X:
        print(item, end=' ')        # Use simple iteration if you can
```

Enhance Skill

RANGE

As a general rule, use for instead of while whenever possible, and don’t use range calls in for loops except as a last resort. This simpler solution is almost always better. Like every good rule, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: range and len

Though not ideal for simple sequence scans, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals when required. For example, some algorithms can make use of sequence reordering—to generate alternatives in searches, to test the effect of different value orderings, and so on. Such cases may require offsets in order to pull sequences apart and put them back together, as in the following:

The range’s integers provide a repeat count in the first, and a position for slicing in the second:

```
>>>
>>> S = 'spam'
>>> for i in range(len(S)):             # For repeat counts 0..3
        S = S[1:] + S[:1]               # Move front item to end
        print(S, end=' ')
pams amsp mspa spam
>>>
>>> S
'spam'
>>> for i in range(len(S)):             # For positions 0..3
        X = S[i:] + S[:i]               # Rear part + front part
        print(X, end=' ')
spam pams amsp mspa
>>>
```

The second creates the same results as the first, though in a different order, and doesn’t change the original variable as it goes.

Because both slice to obtain parts to concatenate, they also work on any type of sequence, and return sequences of the same type as that being shuffled — if you shuffle a list, you create reordered lists:

```
>>>
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
        X = L[i:] + L[:i] # Works on any sequence type
        print(X, end=' ')
[1, 2, 3] [2, 3, 1] [3, 1, 2]
>>>
```

Enhance Skill

RANGE

<u>Non-exhaustive Traversals:</u> range Versus Slices

Cases like that of the prior section are valid applications for the **range/len** combination. We might also use this technique to skip items as we go:

```
>>>
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]
>>>
>>> for i in range(0, len(S), 2):
        print(S[i], end=' ')
a c e g i k
>>>
```

- Here, we visit every second item in the string S by stepping over the generated range list.
- To visit every third item, change the third range argument to be 3, and so on.
- In effect, using range this way lets you skip items in loops while still retaining the simplicity of the for loop construct.
- In most cases, though, this is also probably not the "best practice" technique in Python today.
- If you really mean to skip items in a sequence, the extended three-limit form of the slice expression, provides a simpler route to the same goal.

To visit every second character in S, for example, slice with a stride of 2:

```
>>>
>>> S = 'abcdefghijk'
>>> for c in S[::2]:
        print(c, end=' ')
a c e g i k
>>>
```

The result is the same, but substantially easier for you to write and for others to read. The potential advantage to using range here instead is space: slicing makes a copy of the string in both 2.X and 3.X, while range in 3.X and xrange in 2.X do not create a list; ***for very large strings, they may save memory***.

Enhance Skill

Changing Lists: Range

Another common place where you may use the **range/len** combination with **for** is in loops that change a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list.

```
>>> L = [1, 2, 3, 4, 5]
>>> for x in L:
        x += 1                    # Changes x, not L
        x
2
3
4
5
6
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

This doesn't quite work—**it changes the loop variable x, not the list L**.

- In the first iteration, for example, **x** is integer 1.
- In the next iteration, the loop body sets **x** to a different object, integer 2, but it does not update the list where 1 originally came from; it's a piece of memory separate from the list.
- ***To really change the list as we march across it***, we need to use indexes so we can assign an updated value to each position as we go.
- The **range/len** combination can produce the required indexes for us:

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):          # Add one to each item in L
        L[i] += 1                     # Or L[i] = L[i] + 1
>>> L
[2, 3, 4, 5, 6]
```

When coded this way, ***the list is changed*** as we proceed through the loop. There is no way to do the same with a simple for x in L:–style loop, because such a loop iterates through actual items, not list positions. But what about the equivalent **while** loop?

```
>>> i = 0
>>> while i < len(L):
        L[i] += 1 ; i += 1
>>> L
[3, 4, 5, 6, 7]
```

Enhance Skill