8. for loops (definite interations)

1 Overwiev of definite iteration in programming

Historically, programming languages have a few assorted flavours of for-loop:

- **(1) Numeric Range Loops:** the most basic for-loop is a simple numeric range statement with **start** and **end** values. Here, the body of the loop is executed ten times:

  ```
  for i = 1 to 10
      <loop body>
  ```

- **(2) Three-Expression Loops:** this type of has the following form:

  ```
  for (i = 1; i <= 10; i++)
      <loop body>
  ```

  This loop is interpreted as follows:

  – Initialize, `i = 1`;
  – Continuation condition, `i <= 10`;
  – Increment `i` by 1 after each loop iteration.

  These types loops are popular because the expressions specified for the three parts has more flexibility than the simpler numeric range form. These for loops are also featured in the C++, Java, PHP.

- **(3) Collection(Iterator)-Based Loops:**

  ```
  for i in <collection>
     <loop body>
  ```

  Each time through the loop, the variable i takes on the value of the next object in <`collection`>. This type of for-loop is arguably **the most generalized and abstract.**

2 The Python for-loop

Of the loop types listed above, Python implements the **collection-based iteration**:

```
for <var> in <iterable>:
    <statement(s)>
```

where:
<`iterable`> is an collection of objects e.g. a list, tuple, etc.
<`statement(s)`> in the loop body are denoted by indentation, as with all Python control structures, and executed once for each <`iterable`>.
<`var`> is the loop variable that takes on the value of the next element in <`iterable`> each time through the loop.

Example:

```
a = ['foo', 'bar', 'baz']
for i in a:
print(i)
>> foo >> bar >> baz
```

Before examining for loops further, it will be beneficial to delve more deeply into what **iterables** are in Python.

2.1 Iterables, iter()

Iterable: is an **object that can be used in iteration**.

If an object is iterable, it can be passed to the built-in function `iter()`, which returns an **iterator**:

```
iter('foobar')                          # String
<str_iterator object at 0x036E2750>

iter(['foo', 'bar', 'baz'])             # List
<list_iterator object at 0x036E27D0>

iter(('foo', 'bar', 'baz'))             # Tuple
<tuple_iterator object at 0x036E27F0>

iter({'foo', 'bar', 'baz'})             # Set/frozenset
<set_iterator object at 0x036DEA08>

iter({'foo': 1, 'bar': 2, 'baz': 3})    # Dict
<dict_keyiterator object at 0x036DD990>
```

But **these are not only types that we can iterate over**. E.g. **open files in Python are iterable**, i.e. iterating over an open file object reads data from the file.

2.2 Iterators, next()

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function `next()` is used **to obtain the next value from iterator**:

```
a = ['foo', 'bar', 'baz']
itr = iter(a)
b = a.copy()

while b:
    print(next(itr))
    b.pop()
>> foo >> bar >> baz
```

Note, that iterator is associated with the iterable i.e. if we change an iterable, the iterator will be affected either.

What happens when the iterator runs out of values? - it will return a `StopIteration` **exception**.

```
a = ['foo', 'bar', 'baz']
itr = iter(a)

print(itr)

while a:
    print(next(itr))
    a.pop()
>>
<list_iterator object at 0x0000027465E1FA20>
foo
bar
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-69-c91a07ce6a7a> in <module>
5
6 while a:
----> 7     print(next(itr))
8     a.pop()
StopIteration:
```

If you **want to grab all the values from an iterator at once**, you can use the built-in `list()` function. Among other possible uses, `list()` takes an iterator as its argument, and returns a list consisting of all the values that the iterator yielded:

```
a = ['foo', 'bar', 'baz']
itr = iter(a)
list(itr)
>> ['foo', 'bar', 'baz']
```

Similarly, the built-in `tuple()` and `set()` functions return a tuple and a set, respectively, from all the values an iterator yields:

```
itr = iter(a)
tuple(itr)
>> ('foo', 'bar', 'baz')
```

```
itr = iter(a)
set(itr)
>> {'bar', 'baz', 'foo'}
```

Iterators are "lazy" i.e. when you create an iterator, it doesn't generate all the items it can yield just then. It waits until you ask for them with `next()`. Items are not created until they are requested.

3 The guts of the Python for-loop

Overview of the terminology:

– **Iteration:** process of looping through the objects or items in a collection;
– **Iterable:** object (or the adjective used to describe an object) that can be iterated over;
– **Iterator:** object that produces successive items or values from its associated iterable;
– **`iter()`:** built-in function used to obtain an iterator from an iterable.

Now, consider again the simple for loop:

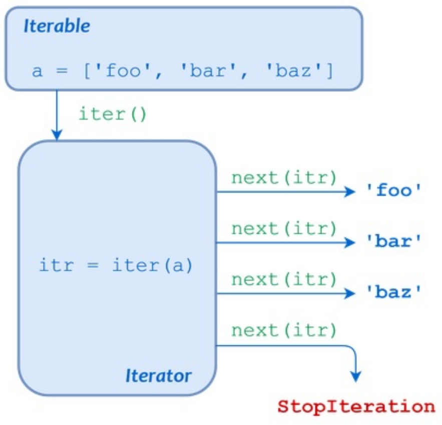
```
a = ['foo', 'bar', 'baz']
for i in a:
    print(i)
>> foo >> bar >> baz
```

This loop can be described entirely in terms we have just learned about i.e. Python does the following:

1. Calls `iter()` to obtain an iterator for **a**;
2. Calls `next()` repeatedly to obtain each item from the iterator in turn;
3. Terminates the loop when `next()` raises the `StopIteration` exception.

The loop body is executed once for each item `next()` returns, with loop variable `i` set to the given item for each iteration.

This sequence of events is summarized in the following diagram:



Advantages of this approach:

(1) There is a std. library module `itertools` with many **useful functions that return iterables**;

(2) Many built-in and library objects are iterable;

(3) User-defined objects created with Python's object-oriented capability can be made to be iterable;

(4) Python has a so called **generator** allowing to create your own iterator in a straightforward way.

4 Iterating through a dictionary or parallel lists

Dictionaries must be iterable. What happens when you loop through a dictionary?

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
for k in d:
    print(k)
>> foo >> bar >> baz
```

So, when a `for` loop iterates through a dict., the **loop variable is assigned to the dictionary's keys**.

To access the dictionary values within the loop, you can make a dictionary reference using the key as usual:

```
for k in d:
    print(d[k]) # use returned keys to access values
>> 1 >> 2 >> 3
```

You can also iterate through a dictionary's values directly by method `.values()`, **but it is not a Pythonic way**:

```
for i in d.values():
    print(i)
>> 1 >> 2 >> 3
```

In fact, you can iterate through both the keys and values of a dictionary simultaneously. That is because the **loop variable of a for loop isn't limited to just a single variable – it can also be a tuple**, in which case the assignments are made from the items in the iterable using **packing and unpacking**:

```
for i, j in [(1, 2), (3, 4), (5, 6)]:
    print(i, j)
>> 1 2 >> 3 4 >> 5 6
```

So, the dict method `.items()` effectively returns a list of key/value pairs as tuples:

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
for k in d.items():
    print(k)
>> ('foo', 1) >> ('bar', 2) >> ('baz', 3)
```

You can also **iterate over two or more lists in parallel** using `zip(*iterables)` function. The `zip()` function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and **returns an iterator of tuples**:

```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

for i, j in zip(a, b): # unpacking the returned tuple: a, b = (a, b)
    print('k = ', i, ', v = ', j)
>> k =  1 , v =  5
>> k =  2 , v =  4
>> k =  3 , v =  3
>> k =  4 , v =  2
>> k =  5 , v =  1
```

5 The range() function

In the beginning of this tutorial we saw a **numeric range for-loop**. Although this form of for loop isn't directly built into Python, it is easily arrived at.

The built-in `range([start,] stop[, step])` function **returns an immutable iterable that yields a sequence of integers.** `range()` **takes a small amount of memory, since it does not stores the whole list.** We can then iterate over it:

```
x = range(5)

for i in x:
    print(i)

>> 0 >> 1 >> 2 >> 3 >> 4
```

All the parameters specified to `range()` must be integers. For floats, use NumPy.
`range()` also allows to impement a **C-style loop**:

```
for (int i = a; i < n; i+=s) {
        // ...
}
```

Is equivalent to:

```
for i in range(a, n, s):
    # ...
```

6 Taking the item index, enumerate() function

What if you need the **item index?** The `enumerate()` built-in is helpful in this case.
It returns the tuple of the value together with its index:

```
lst = range(10)
for i in enumerate(lst):
    print(i)
>> (0, 0)
>> (1, 1)
>> ...
```

7 Altering for loop behavior

7.1 The break and continue statements

`break` and `continue` work the **same way as with while-loops**:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
        break
    print(i)
>> foo
```

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
       continue
    print(i)
>> foo >> qux
```

7.2 The else clause

A `for` loop can have an `else` clause as well. The interpretation is **analogous to that of a while-loop** i.e. **the `else` clause won't be executed if the list is broken out of with a `break` statement**:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if i == 'baz':
        break
    print(i)
else:
    print('Done')  # Will not execute
>> foo >> bar
```