



4. More Control Flow Tools

Besides the `while` statement just introduced, Python uses the usual flow control statements known from other languages, with some twists.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

>>>

There can be zero or more `elif` parts, and the `else` part is optional. The keyword ‘`elif`’ is short for ‘else if’, and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

>>>

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
```



```
if status == 'inactive':  
    del users[user]  
  
# Strategy: Create a new collection  
active_users = {}  
for user, status in users.items():  
    if status == 'active':  
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):  
...     print(i)  
...  
0  
1  
2  
3  
4
```

>>>

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> list(range(5, 10))  
[5, 6, 7, 8, 9]  
  
>>> list(range(0, 10, 3))  
[0, 3, 6, 9]  
  
>>> list(range(-10, -100, -30))  
[-10, -40, -70]
```

>>>

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']  
>>> for i in range(len(a)):  
...     print(i, a[i])  
...  
0 Mary  
1 had  
2 a  
3 little  
4 lamb
```

>>>

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```
>>> range(10)  
range(0, 10)
```

>>>