Control Flow Tools

if statement

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

```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('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.

while statement

The **while** statement is used for repeated execution as long as an expression is **true.**

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
...
4
3
2
1
0
```

The controlling expression (**n > 0**), typically involves one or more variables that are initialized prior to starting the loop and then modified somewhere in the loop body.

When a while loop is encountered, the experssion is first evaluated in **Boolean** context. If it is **true**, the loop body is executed. Then the expression is checked again, and if still **true**, the body is executed again. This continues until the expression becomes **false**, at which point program execution proceeds to the first statement beyond the loop body.

With indefinite iteration, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.

for statement

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:

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

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)
```

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))


list(range(0, 10, 3))


list(range(-10, -100, -30))
```

A strange thing happens if you just print a **range**:

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

In many ways the object returned by **range()** behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is **iterable**, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the for statement is such a construct, while an example of a function that takes an iterable is **sum()**:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

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])
```

pass statements

The **pass** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```

break and continue statements, and else clauses on loops

The **break** statement breaks out of the innermost enclosing **for** or **while** loop.

A **for** or **while** loop can include an **else** clause.

In a **for** loop, the **else** clause is executed **after** the loop reaches its final iteration.

In a **while** loop, it's executed after the loop's condition becomes **false**.

In either kind of loop, the **else** clause is not executed if the loop was terminated by a **break**.

This is exemplified in the following for loop, which searches for prime numbers:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

The **continue** statement, also borrowed from C, continues with the next iteration of the loop:

```python
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found an odd number", num)
```