3.1 *If* Statement

The most well-known statement type is `if` statement.

In [7]:
```
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')
```

```
Please enter an integer: 42
More
```

Those `if` , `elif` , `else` sequence is a substitue of `switch` or `case`

3.2 *for* Statement

The `for` statement iterates over the items of any sequence (a list or a string), in order that they appear in the sequence. For example:

In [9]:
```
# Measure some strings:
words = ['cat', 'window', 'defenestate']
for w in words:
    print(w,len(w))
```

```
cat 3
window 6
defenestate 11
```

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:

In [19]:
```
# Strategy: Iterate over a copy
for user, status in user.copy().items():
    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
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-c8836006d039> in <module>
      1 # Strategy: Iterate over a copy
----> 2 for user, status in user.copy().items():
      3     if status == 'inactive':
      4         del users[user]
      5

NameError: name 'user' is not defined
```

3.3 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:

In [20]:
```
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 starting from 0. It is also possible to count numbers from specified point, or even negative values; this is called 'step'

In [33]:
```
for i in range(5, 10):
    print(i)
```

```
5
6
7
8
9
```

In [34]:
```
for i in range(0,10,3):
    print(i)
```

```
0
3
6
9
```

In [35]:
```
for i in range(-10,-100,-30):
    print(i)
```

```
-10
-40
-70
```

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

In [36]:
```
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
```

With the combination of `range()` and `len()`, it is possible to get all list members, even though we don't know the numbers of list's total member quantity. However, in most cases, it's pretty much easier to use `enumerate()` function than using the combination of `range()` and `len()` function

In [39]:
```
print(range(10))
```

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

`range()` function seems like list, but actually 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*

In [41]:
```
sum(range(4)) # 0 + 1 + 2 + 3
```

Out[41]:
```
6
```

Those `range()` function can be switched into `list` as below:

In [42]:
```
list(range(4))
```

Out[42]:
```
[0, 1, 2, 3]
```

3.4 *break* and *continue* Statements and *else* Clauses on Loops

The `break` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop

In [1]:
```
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 priome number')
```

```
2 is a priome number
3 is a priome number
4 equals 2 * 2
5 is a priome number
6 equals 2 * 3
7 is a priome number
8 equals 2 * 4
9 equals 3 * 3
```

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

In [2]:
```
for num in range(2,10):
    if num % 2 ==0:
        print("Found and even number", num)
        continue
    print("Found and oddn number", num)
```

```
Found and even number 2
Found and oddn number 3
Found and even number 4
Found and oddn number 5
Found and even number 6
Found and oddn number 7
Found and even number 8
Found and oddn number 9
```

3.5 *pass* Statements

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

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

This is commonly used for creating minimal `classes`

In [ ]:
```
class MyEmptyClass:
    pass
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored.

In [ ]:
```
def initlog(*args):
    pass # Remeber to implement this!
```

In [ ]:
```

```