Control Flow

In [49]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

Conditional Statements

In [50]:
```
x = 6
if x < 0:
    print("It's negative")
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

```
Positive and larger than or equal to 5
```

If any of the conditions is **True**, no further **elif** or **else** blocks will be reached.

With a compound condition using **and** or **or**, conditions are evaluated left to right and will short-circuit:

In [51]:
```
a = 5; b = 7
c = 8; d = 4
```

In [52]:
```
if a < b or c > d:
    print('Made it')
```

```
Made it
```

In this example, the comparison **c > d** never gets evaluated because the first comparison was **True**.

It is also possible to chain comparisons:

In [53]:
```
4 > 3 > 2 > 1
```

Out[53]:
```
True
```

Ternary expressions

A **ternary expression** in Python allows you to combine an **if-else** block that produces a value into a single line or expression:

In [1]:
```
x = 5
'Non-negative' if x >= 0 else 'Negative'
```

Out[1]:
```
'Non-negative'
```

Loops

for loops

for loops are for iterating over a collection (like a list or tuple) or an iterater. The standard syntax for a **for** loop is:

In [55]:
```
for value in [1, 5, 2]:
    print(value)
```

```
1
5
2
```

You can advance a **for** loop to the next iteration, skipping the remainder of the block, using the **continue** keyword. Consider this code, which sums up integers in a list and skips **None** values:

In [56]:
```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

In [57]:
```
total
```

Out[57]:
```
12
```

A **for** loop can be exited altogether with the **break** keyword. This code sums elements of the list until a 5 is reached:

In [58]:
```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

In [59]:
```
total
```

Out[59]:
```
12
```

The **break** keyword only terminates the innermost **for** loop

In [60]:
```
for i in range(4):
    for j in range(4):
        if j > i:
            break
        print((i, j))
```

```
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

if the elements in the collection or iterator are sequences (tuples or lists, say), they can be conveniently unpacked into variables in the **for** loop statement:

In [61]:
```
for a, b, c in iterator:
    # do something
```

```
  File "<ipython-input-61-83e0921527b1>", line 2
    # do something
                   ^
SyntaxError: unexpected EOF while parsing
```

while loops

A **while** loop specifies a condition and a block of code that is to be executed until the condition evaluates to **False** or the loop is explicitly ended with break:

In [ ]:
```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

pass

pass is the “no-op” statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is only required because Python uses whitespace to delimit blocks:

In [ ]:
```
if x < 0:
    print('negative!')
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print('positive!')
```

range

The **range** function returns an iterator that yields a sequence of evenly spaced integers:

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

In [ ]:
```
range(0, 10)
```

In [ ]:
```
list(range(10))
```

Both a ***start***, ***end***, and ***step*** (which may be negative) can be given:

In [ ]:
```
list(range(0, 20, 2))
```

In [ ]:
```
list(range(5, 0, -1))
```

range produces integers up to but not including the endpoint.

A common use of **range** is for iterating through sequences by index:

In [ ]:
```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

In [ ]:
```
val
```

In [ ]:
```

```