# 1 Flow Control, Loops and Exception Handling

## 1.1 Whitespace and Flow Control

Python uses white space changes to indicate the start and end of flow control blocks, and so indention matters. For example, when using `if . . . elif . . . else` blocks, all of the control blocks must have the same indentation level and all of the statements inside the control blocks should have the same level of indentation.

Returning to the previous indentation level instructs Python that the block is complete. Best practice is to only use spaces (and not tabs), and to use 4 spaces when starting a indented level, which is a good balance between readability and wasted space.

## 1.2 `if . . . elif . . . else`

`if . . . elif . . . else` blocks always begin with an if statement immediately followed by a scalar logical expression. elif and else are optional and can always be replicated using nested if statements at the expense of more complex logic and deeper nesting.
The generic form of an `if . . . elif . . . else` block is

```
if logical_1:
Code to run if logical_1
elif logical_2:
Code to run if logical_2 and not logical_1
elif logical_3:
Code to run if logical_3 and not logical_1 or logical_2
...
...
else:
```

A few simple examples

```
>>> x = 5
>>> if x<5:
... x += 1
... else:
... x =
1
>>> x
4
```

```
>>> x = 5;
>>> if x<5:
... x = x + 1
... elif x>5:
... x = x 1
... else:
... x = x * 2
>>> x
10
```

## 1.3 for

`for` loops begin with for item in iterable:, and the generic structure of a for loop is

```
for item in iterable:
Code to run
```

item is an element fromiterable, and iterable can be anything that is iterable in Python. The mostcommon examples are xrange or range, lists, tuples, arrays or matrices.

The `for` loop will iterate across all items in iterable, beginning with item 0 and continuing until the final item. When using multidimensional arrays, only the outside dimension is directly iterable.

For example, if x is a 2-dimensional array, then the iterable elements are x[0], x[1] and so on.

**Indentation Errors in Type Setting**

```
count = 0
for i in xrange(100):
count += i

count = 0
x = linspace(0,500,50)
for i in x:
count += i

count = 0
x = list(arange(20,21))
for i in x:
count += i
```

Loops can also be nested

```
count = 0
for i in xrange(10):
for j in xrange(10):
count += j
```

or can contain flow control variables

```
returns = randn(100)
count = 0
for ret in returns:
if ret<0:
count += 1
```

This for expression can be equivalently expressed using xrange as the iterator and len to get the number of items in the iterable.

```
returns = randn(100)
count = 0
for i in xrange(len(returns)):
if returns[i]<0:
count += 1
```

Finally, these ideas can be combined to produce nested loops with flow control.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
for j in xrange(size(x,1)):
if i<j:
x[i,j]=i+j;
else:
x[i,j]=ij
```

or loops containing nested loops that are executed based on a flow control statement.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
if (i % 2) == 1:
for j in xrange(size(x,1)):
x[i,j] = i+j
else:
for j in xrange(int(i/2)):
x[i,j] = ij
```

**Whitespace**
Like if . . . elif . . . else flowcontrol blocks, for loops are whitespace sensitive. The indentation of the line immediately below the for statement determines the indentation that all statements in the block must have. **break**
A loop can be terminated early using break. break is usually used after an if statement to terminate the loop prematurely if some condition has been met.

```
x = randn(1000)
for i in x:
print(i)
if i > 2:
break
```

Since for loops iterate over an iterable with a fixed size, break is generally more useful in while loops.

**continue**
continue can be used to skip an iteration of a loop, immediately returning to the top of the loop using the next item in iterable. continue is commonly used to avoid a level of nesting, such as in the following two examples.

```
x = randn(10)
for i in x:
if i < 0:
print(i)
for i in x:
if i >= 0:
continue
print(i)
```

Avoiding excessive levels of indentation is essential in Python programming – 4 is usually considered the maximum reasonable level. continue is particularly useful since it can be used to in a for loop to avoid one level of indentation.

## 1.4 `while`

`while` loops are useful when the number of iterations needed depends on the outcome of the loop contents. `while` loops are commonly used when a loop should only stop if a certain condition is met, such as when the change in some parameter is small. The generic structure of a while loop is

```
while logical:
Code to run
Update logical
```

Two things are crucial when using a while loop: first, the logical expression should evaluate to true when the loop begins (or the loop will be ignored) and second, the inputs to the logical expression must be updated inside the loop. If they are not, the loop will continue indefinitely (hit CTRL+C to break an interminable loop in IPython). The simplest while loops are (wordy) drop-in replacements for for loops:

```
count = 0
i = 1
while i<10:
count += i
i += 1
```

which produces the same results as

```
count=0;
for i in xrange(0,10):
count += i
```

while loops should generally be avoidedwhenfor loops are sufficient. However, there are situations where no for loop equivalent exists.

```
# randn generates a standard normal random number
mu = abs(100*randn(1))
index = 1
```

```
while abs(mu) > .0001:
mu = (mu+randn(1))/index
index=index+1
```

In the block above, the number of iterations required is not knownin advance and since randn is a standard normal pseudo-random number, it may take many iterations until this criteria is met. Any finite for loop cannot be guaranteed to meet the criteria.

### `break`

`break` can be used in a while loop to immediately terminate execution. Normally, `break` should not be used in a while loop – instead the logical condition should be set to False to terminate the loop. However, break can be used to avoid running code below the break statement even if the logical condition is False.

```
condition = True
i = 0
x = randn(1000000)
while condition:
if x[i] > 3.0:
break # No printing if x[i] > 3
print(x[i])
i += 1
```

It is better to update the logical statement which determines whether the while loop should execute

```
i = 0
while x[i] <= 3:
print(x[i])
i += 1
```

### 1.4.1 continue

continue can be used in a while loop to skip any remaining code in the loop, immediately returning to the top of the loop, which then checks the while condition, and executes the loop if it still true. Using continue when the logical condition in the while loop is False is the same as using break.