Looping with `for` and `while`

Very often, one wants to repeat some action. This can be achieved with `for` and `while` statements.

`for` loops

A `for` loop is typically used when we want to repeat an action a given number of times.

```
for i in range(5):
    print(i**2, end=', ')
print() # print a new line at the end
```

Here, `for i in range(n):` will execute the loop body `n` times with `i = 0, 1, 2, ..., n - 1` in succession.

Note on Python syntax

Python uses syntatic indenting. This means that indenting code has a meaning in the programming language. In languages like C, C++, and Java, loop bodies are enclosed in braces, but good coding style suggests that statements in a loop or conditional body are indented:

```
for (int i = 0; i < 10; i++) {
    printf("i = %d\n",i);
}
```

Python takes this a step further and requires the indenting of loop and conditional bodies. We recommend that you use 4 spaces to indent python code (so does the python community]). Please tell your text editors to insert spaces instead of tab characters when you hit the tab key on the keyboard.

The `range()` function

The `range()` function can be used in a few different ways. We can convert a range object to a python list with the `list()` function:

```
# get range 0,...,6
print(list(range(7)))
# get range 4,...,10
print(list(range(4,11)))
# get range [4,16) with step of 3
print(list(range(4,16,3)))
```

See `help(range)` for more info.

Note that `range` differs in Python 2 and 3. In Python 2, `range()` returns a list. In Python 3, `range()` returns an object that produces a sequence of integers in the context of a `for` loop, which is more efficient, because memory for a new list need not be allocated.

`for` and lists

We can use a `for` loop to iterate over items in a list:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]]
for item in my_list:
    print(item)
```

It is often handy to get access to both the list item and index in a `for` loop. This can be achieved with the `enumerate()` function:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]]
for index, item in enumerate(my_list):
    print("{}: {}".format(index, item))
```

Example adding numbers

```
summation = 0
for n in range(1,101):
    summation += n
print(summation)
```

Also achievable in Python via `sum`:

```
sum(range(1,101))
```

`while` loops

When we do not know how many iterations are needed, we can use `while`.

```
i = 2
while i < 100:
    # loop body only execute if conditional statement is True
    print(i**2,end=", ")
    i = i**2
print() # print a new line at the end
```

Infinite loops

```
while True:
    print("hah!")
```

- In Jupyter Notebook, select "Interrupt" from the Kernel menu
- Use `ctrl-c` to interrupt the interpreter

Nesting loops

A *nested loop* is a loop in the body of a loop.

```
for i in range(8):
    for j in range(i):
        print(j, end=' ')
    print()
```

`continue`

`continue` continues with the next iteration of the smallest enclosing loop:

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

Here, `num in range(2,10)` sets up a loop where `num = 2, 3, ..., 9`.

`break`

The `break` statement allows us to jump out of the smallest enclosing `for` or `while` loop.

Finding prime numbers:

```
max_n = 10
for n in range(2, max_n):
    for x in range(2, n):
        if n % x == 0: # n divisible by x
            print(n, 'equals', x, '*', n/x)
            break
    else: # executed if no break in for loop
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

`pass`

The `pass` statement does nothing, which can come in handy when you are working on something and want to implement some part of your code later.

```
traffic_light = 'green'
if traffic_light == 'green':
    pass # to implement later
else:
    print('whatever you do, stop the car!')
```

Loop `else`

- An `else` can be used with a `for` or `while` loop
- The `else` is only executed if the loop runs to completion, not when a `break` statement is executed

```
for i in range(4):
    print(i)
else:
    print("all done")
```

```
for i in range(7):
    print(i)
    if i > 3:
        break
else:
    print("all done")
```

A note on Python variables

It is bad practice to define a variable inside of a conditional or loop body and then reference it outside:

```
name = "Nick"
if name == "Nick":
    age = 45 # newly created variable

print("Nick's age is {}".format(age))
```

Here is what happens when a variable is not created:

```
name = "Bob"
if name == "Nick":
    id_number = 45 # also newly created variable

print("{}'s id number is {}".format(name, id_number))
```

Good practice to define/initialize variables at the same level they will be used:

```
name = "Bob"
age = 55
if name == "Nick":
    age = 45

print("{}'s age is {}".format(name,age))
```