Math 246 Unit 3B. Iteration with `for` and `while`

Professor Brenton LeMesurier, September 8, 2015 (slightly update October 3)

The two last main fundamental bulding blocks we need are for *iteration:* tasks that repeat the same sequence of actions repeatedly, with possible variation due to the values of variables being changed during some repetitions.

There are two versions, depending on whether the list of iterations is determinined before we start, or "on the fly".

I. Repeating a predetermined number of times, with `for` loops

We can apply the same sequence of commands for each of a list of values by using a `for` statement, flowed by an ndented list of statements.

For example, we can compute the square roots of several numbers. Here I give those numbers in a tuple (i. e., in parentheses). They could just as well be in a list (i. e., in brackets.)

In [22]:

```
import math
```

In [23]:

```
for x in (1, 2, 4, 9, 20, 1000000, 3141592653589793):
    square_root_of_x = math.sqrt(x)
    print('The square root of %g is %g' % (x, square_root_of_x))
```

```
The square root of 1 is 1
The square root of 2 is 1.41421
The square root of 4 is 2
The square root of 9 is 3
The square root of 20 is 4.47214
The square root of 1e+06 is 1000
The square root of 3.14159e+15 is 5.60499e+07
```

Aside 1, on using `import`.

I use another option for importing selecte items froma module: the `import` statement just specifies that the module is wanted, and then I refer to itesm in it by their "full name", prefixed by the module name.

This is often the recommended way to do things in larger program files, because it keeps clear where the item (here "sqrt") come from, without having to search above for the import statrment.

Aside 2, on formatting printed output.

Note also the new way that I inserted the values of the variables x and square_root_of_x into the string of text to be printed. Each "%g" in the text string takes in turn a value from the tuple after the "%" that is after the string, and displays that value in the string, using the general ("g") format for a real number. In this format, large enough values (like 1000000) get displayed with an exponent, to save space.

Repeating for a range of equally spaced integers

One very common choice for the values to use in a `for` loop is a range of consecutive integers, or more generally, equally spaced integers. The first n natural numbers are given by `range(n)` -- remember that for Python, the natural numbers start with 0, so this range is $\{i : 0 \leq i < n\}$ and n is the first value *not* in the range!

Let's combine this with an example from Unit 3A:

In [24]:

```
for n in range(8):
    if n % 4 == 0:
        print('%i is a multiple of 4' % n)
    elif  n % 2 == 0:
        # A multiple of 2 but not of 4, or we would have stopped with the above
"hit".
        print('%i is an odd multiple of 2' % n)
    else:
        print('%i is odd' % n)
```

```
0 is a multiple of 4
1 is odd
2 is an odd multiple of 2
3 is odd
4 is a multiple of 4
5 is odd
6 is an odd multiple of 2
7 is odd
```

Aside 3: more on formatting printed output, ths time with an integer. Here I use "%i" in place of the "%g" above. This specifies that the value to insert is an integer.

Also, with only a single item to insert into the text string, we do not need to put it in parentheses as with the tuple above -- but you may still use parentheses if you wish!

Ranges that start elsewhere

To specify a range of integers starting at at integer m instead of at zero, use

```
range(m, n)
```

Again, the terminal value n is the first value *not* in the range: this gives $\{i : m \leq i < n\}$

In [25]:

```
for i in range(10, 15):
        print('%i cubed is %i' % (i, i**3))
```

```
10 cubed is 1000
11 cubed is 1331
12 cubed is 1728
13 cubed is 2197
14 cubed is 2744
```

Aside 4: yet more on formatting printed output. Note that in the tuple of stuff to insert into the text strong for printing, each item can be an *expression* (a formula), which gets evaluated to produce the value to be printed.

Ranges of equally-spaced integers

The final, most general use of the function `range` is generating integers with equal spacing other than 1, by giving it three arguments:

```
range(start, stop, increment)
```

So to generate just even integers, specify a spacing of two:

In [26]:

```
for even in range(0, 10, 2):
        print('2^%i is %i' % (even, 2**even))
```

```
2^0 is 1
2^2 is 4
2^4 is 16
2^6 is 64
2^8 is 256
```

Note that even though the first value is the "default" of zero, I could not omit it here. What would happen if I did so?

Decreasing sequences of integers

We sometimes want to count down, which is done by using a negative value for the increment in function `range`.

Before running the following code, work out what it will do -- this might be a bit surprising at first.

I also put parentheses around the single value to be inserted into the text string, just to show that this option is allowed.

In [27]:

```
for n in range(10, 0, -1):
        print('%i seconds' % (n))
```

```
10 seconds
9 seconds
8 seconds
7 seconds
6 seconds
5 seconds
4 seconds
3 seconds
2 seconds
1 seconds
```

II. Repeating an initially unknown number of times, with `while` loops

Often, calculating numerical approximate solutions follows a pattern of iterative improvement, like

1. Get an initial approximation.
2. Use the best current approximation to compute a new, even better one.
3. Check the accuracy of this new approximation.
4. If the new approximation is good enough, stop -- othewise, repeat from step 2.

For this, a `while` loop can be used. Its general meaning is:

```
while "some logical statement is True"
   repeat this
   and this
   and so on
   and when done with the last indented line, go back to the "while" line
and check the logical statement again
This less indented line is where we continue after the "while" iteration i
s finished.
```

For a change, I will illustrate this with something with more mathemaical substance: computing cube roots using only a modest amount of basic arithmetic. For now this is just offered as an example of programming methods, and the rapid success might be mysterious, but this will soon be explained in Math 245.

In [28]:

```
# We are going to approximate the cube root of a:
a = 8

# A first very rough approximation:
root = 1

# I will tolerate an error of this much:
error_tolerance = 1e-8
# Math 245 students: this is a _backward error_ specification.

# The answer "root" should satisfy root**3 - a = 0, so check how close we are:
while abs(root**3 - a) > error_tolerance:
    root = (2 * root + a/root**2)/3
    print('The new approximation is %22.16g, with estimated error of %.1e' % (ro
ot, abs(root**3 - a)))

print('Done!')
print('The cube root of %g is approximately %-22.16g' % (a, root))
print('The "backward error" in this approximation is %e' % (abs(root**3 - a)))
```

```
The new approximation is      3.333333333333333, with estimated erro
r of 2.9e+01
The new approximation is      2.462222222222222, with estimated erro
r of 6.9e+00
The new approximation is      2.081341247671579, with estimated erro
r of 1.0e+00
The new approximation is      2.003137499141287, with estimated erro
r of 3.8e-02
The new approximation is      2.000004911675504, with estimated erro
r of 5.9e-05
The new approximation is      2.000000000012062, with estimated erro
r of 1.4e-10
Done!
The cube root of 8 is approximately 2.000000000012062
The "backward error" in this approximation is 1.447429e-10
```

Aside. I have thrown in some more refinements of output format control, "%22.16g", "%-22.16g", "%e and "%.1e". If you are curious, you could try to work out what they do from these examples, or read up on this – but that is not essential, at least for now.

Exercise 3.2

Write a Python file that inputs a natural number n, and computes and prints the first n Fibonacci numbers.

Hint: Use a `for` loop.

Exercise 3.3

Write a Python file that inputs a natural number n, and computes and prints all the Fibonacci numbers less than or equal to n.

Hint: Use a `while` loop.