MATH40006: An Introduction To Computation
Module notes, Section 3

3 Loops using `for` and `while`

3.1 `for` loops

So far, we've been using Python like a kind of powerful calculator. Admittedly, it's an unusual calculator that supports named variables, data structures and integers of arbitrary size, but the point is that most of our interaction with Python so far has involved typing a line of input, getting Python to do a calculation and getting the appropriate output. But in fact, Python is a **programming language**, so let's start doing some programming.

One thing computers are brilliant at, and humans tend to find difficult and frustrating, is doing the same thing over and over again; what we call **iteration**. The simplest kind of iteration (in Python and many other languages) is probably the `for` loop. Here's an example.

Challenge 1: print "Hello World!" ten times.

```
for n in range(10):
    print('Hello World!')
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

What happened was this. We set up a variable called `n`, which began by taking the value 0. Then the string "Hello World!" was printed. Then `n` took the value 1, and "Hello World!" was printed again. Then it took the value 2, and we got another "Hello World!". Then it became 3, then 4, and so on, up to its final value, which was 9. The string "Hello World!" was printed once for each value of `n` between 0 and 9 inclusive; that is, ten times.

Now, that's fine, if a bit dull. Let's be very slightly more ambitious, by writing a program that makes use of all these values of `n`.

Challenge 2: calculate n^2 for n from 0 to 9.

```
for n in range(10):
    print(n**2)
```

```
0
1
4
9
16
25
36
49
64
81
```

We might want to use string formatting here:

```
for n in range(10):
    print(f'The square of {n} is {n**2}')
```

```
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
```

Now let's write something that might actually be useful. It turns out that if you evaluate the cosine of, say, 1.0, and then the cosine of that, and then the cosine of that, and so on, you get closer and closer to an angle in radians that is equal to its own cosine; that is, to a solution of the equation $x = \cos x$.

Challenge 3: find a value of x approximately equal to its own cosine.

First we better import a cosine function. Then we should set an initial value for x; we'll use 1.0. Then we're going to repeat the step `x = cos(x)` a set number of times (let's use 20). This step means "Let the new value of the variable `x` be equal to the cosine of the old value." Then, each time, let's print the current value of `x`.

```
from math import cos
x = 1.0
for n in range(20):
    x = cos(x)
    print(f'Iteration {n+1}: {x}')
```

```
Iteration 1: 0.5403023058681398
Iteration 2: 0.8575532158463933
Iteration 3: 0.6542897904977792
Iteration 4: 0.7934803587425655
Iteration 5: 0.7013687736227566
Iteration 6: 0.7639596829006542
Iteration 7: 0.7221024250267077
Iteration 8: 0.7504177617637605
Iteration 9: 0.7314040424225098
Iteration 10: 0.7442373549005569
Iteration 11: 0.7356047404363473
Iteration 12: 0.7414250866101093
Iteration 13: 0.7375068905132428
Iteration 14: 0.7401473355678757
Iteration 15: 0.7383692041223232
Iteration 16: 0.739567202212256
Iteration 17: 0.7387603198742114
Iteration 18: 0.7393038923969057
Iteration 19: 0.7389377567153446
Iteration 20: 0.7391843997714936
```

Our fourth challenge now.

Challenge 4: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1}.$$

Here's a first go at this. We set up a variable called `total`, with initial value 0.0 (a `float`, notice). We then want to use the values of `n` between 0 and 100, meaning our `range` object needs to be `range(101)`. We're going to calculate the value of the term for each of those values of `n`, then add it to `total`.

```
total = 0.0
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
    print(total)
```

However, if you run this, you get a massive printout of all 101 partial sums for values of `n` between 0 and 100. That seems excessive. How can we tweak this code so that we only see the final value of `total`?

In programming terms, what we're trying to do is move the command `print(total)` **outside the loop**, so that it only executes once, when 101 iterations have taken place. Different computer languages have different ways of marking where a block of code begins and ends; Python uses **indentation**. To move `print(total)` outside the loop, we simply remove its indentation, so that it begins at the start of the line.

```
total = 0.0
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
print(total)
```

```
3.1514934010709914
```

(Notice this is not far from π. That's no accident: this summation does converge to π, though really rather slowly.)

Now for our final challenge.

Challenge 5: iterate the two-dimensional Hénon map

$$\begin{aligned} x_{n+1} &= 1 - 1.4\,{x_n}^2 + y_n, \\ y_{n+1} &= 0.3\,x_n \end{aligned}$$

20 times, starting with $x_0 = y_0 = 0.5$.

For this, we're going to make use of a neat trick, which not every programming language allows, which lets us assign values to more than one variable at the same time. To initialise the values of `x` and `y`, for example, we simply need to type

```
x, y = 0.5, 0.5
```

Here's the full code.

```
x, y = 0.5, 0.5
for n in range(20):
    x, y = 1 - 1.4*x**2 + y, 0.3*x
    print(f'({x}, {y})')
```

```
(1.15, 0.15)
(-0.7014999999999995, 0.345)
(0.656056850000001, -0.21044999999999983)
(0.18697517339530675, 0.1968170550000003)
(1.1478734533473132, 0.05609255201859203)
```

```
(-0.7885662988406887, 0.34436203600419396)
(0.47379050526997063, -0.2365698896522066)
(0.4491616903102298, 0.1421371515809912)
(0.8596924379217113, 0.13474850709306893)
(0.1000489841453833, 0.2579077313765134)
(1.243894012456581, 0.030014695243614987)
(-1.1361665446718507, 0.3731682037369743)
(-0.4340559803872273, -0.3408499634015552)
(0.39538360484456087, -0.13021679411616818)
(0.650923732912, 0.11861508145336826)
(0.5254326929580385, 0.1952771198736)
(0.8087657991128091, 0.15762980788741154)
(0.24188684294699858, 0.24262973973384272)
(1.1607167970266303, 0.07256605288409958)
(-0.813602823175564, 0.34821503910798907)
```

(There's not much apparent order in these numbers, and that's no accident: there's strong evidence that this map is what we call **chaotic**.)

3.2 Using append

The `for` loops in the last section are all very well, but all they do is print values. Often, we'd prefer it if instead our programs built a **data structure** containing all the output; that way, it's available for us to do calculations with. Let's start with a slightly silly example.

Challenge 1: create a list containing 10 copies of the string "Hello World!".

This is exactly like the challenge in the last section, except that we don't want a printout of ten "Hello World!" strings, but a list containing them. Our tactic will be to set up a variable that begins its life as an empty list, and then to use the `append` method to lengthen this list by one on each turn of the loop.

```
hw_list = []
for n in range(10):
    hw_list.append('Hello World!')
    print(hw_list)
```

```
['Hello World!']
['Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
```

```
'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

That shows quite nicely what happens: at each turn of the loop, the list gets another copy of the string appended to it. However, in practice, we're unlikely to want to see all those intermediate values of the list, with one, two, three elements etc; let's move the `print` command outside the loop, so we only see the final version, with ten elements.

```
hw_list = []
for n in range(10):
    hw_list.append('Hello World!')
print(hw_list)
```

```
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

Challenge 2: create a list of all the squares of the integers between 0 and 9.

```
sq_list = []
for n in range(10):
    sq_list.append(n**2)
print(sq_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note: this probably isn't the best way to do this task; later in the module we'll look at **comprehensions**, which give us a slicker method.

One nice thing about having all these numbers in a list like that is that we can do calculations with them. For example, we could plot them. More about plotting, in systematic detail, later in the module, but for now, here's how we could create an appropriate plot. We need to tell

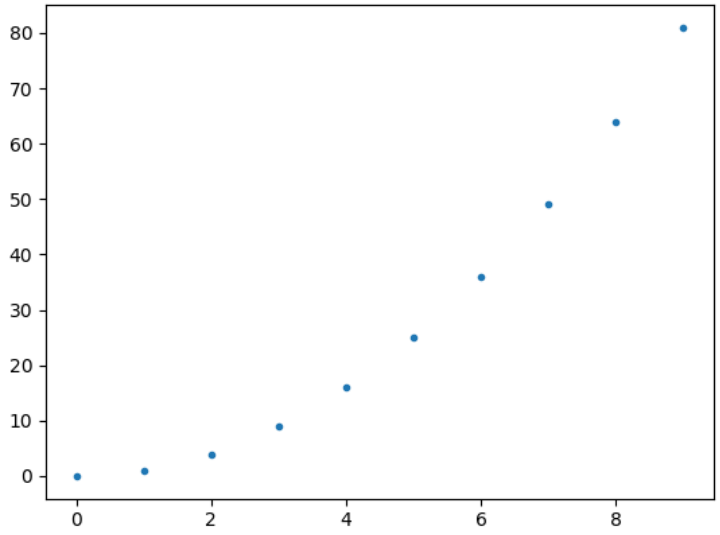

Figure 1: Plot of n^2 against n for $n = 0, 1, 2, \ldots, 9$

Jupyter that we want graphics to appear in the notebook rather than a separate window; we then want to import the `pyplot` submodule of the `matplotlib` module; finally, we want to create a **point plot**, using the integers from 0 to 9 on the horizontal axis, and these squares we've just calculated on the vertical one.

Here's the code:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(10), sq_list, '.')
```

The image is shown in Figure 1.

Notice that if we'd simply printed those squares, we wouldn't have been able to do that; using `append` like this gives us our values in a form we can use for calculations. All programming languages allow programs to create data structures in broadly this way, though the precise details of how it works vary rather a lot from language to language.

Challenge 3: iterate $x_{n+1} = \cos x_n$ 20 times, starting with $x = 1.0$, this time creating a list of all the iterates

One slight difference here is that we'll set up our list of values not as an empty list but as a list containing only the initial value, 1.0. This is a *design decision*; we could leave out this value if we liked. However, I think it makes more sense to include it.

```
from math import cos
x = 1.0
x_list = [x]
for n in range(20):
    x = cos(x)
    x_list.append(x)
print(x_list)
```

```
[1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792,
0.7934803587425655, 0.7013687736227566, 0.7639596829006542,
0.7221024250267077, 0.7504177617637605, 0.7314040424225098,
0.7442373549005569, 0.7356047404363473, 0.7414250866101093,
0.7375068905132428, 0.7401473355678757, 0.7383692041223232,
0.739567202212256, 0.7387603198742114, 0.7393038923969057,
0.7389377567153446, 0.7391843997714936]
```

What about the plotting code? Let's create a line plot this time (we simply leave out the `'.'`). Also, we needn't include the first two lines, in which we tell Juypter to put the image in the notebook and import the relevant submodule, because we've already done that this session; that's unless you've either broken session, moved to a new notebook or restarted the Kernel since creating the last plot (if you have, simply include those two lines).

```
plt.plot(range(21), x_list)
```

Notice that it's `range(21)` and not `range(20)`; this is a consequence of our decision to include the initial value, which appears as "iterate zero". The image is shown in Figure 2.

Challenge 4: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1},$$

this time creating a list of all the partial sums

```
total = 0.0
partial_sums = []
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
    partial_sums.append(total)
print(partial_sums)
```

This time, I won't include the printout of the list, because it's massive. Here's the plotting code:

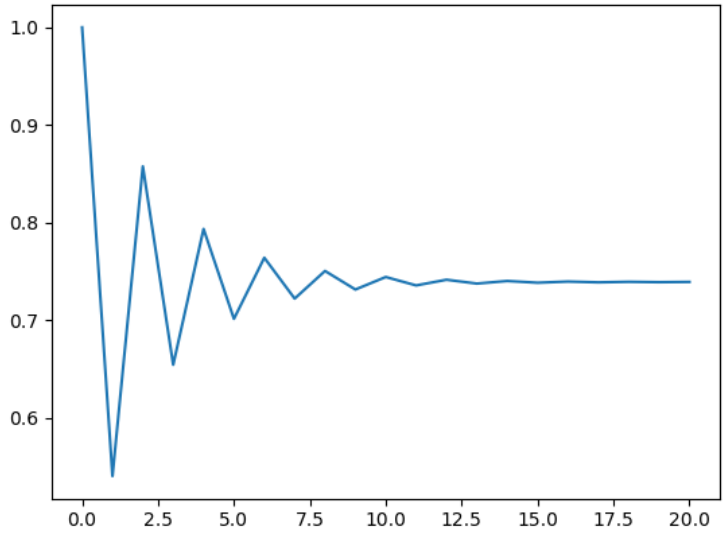

Figure 2: Plot of iterates of $x_{n+1} = \cos x_n$

```
plt.plot(range(101), partial_sums)
```

Again, it's easy to put `range(100)` instead of `range(101)`; this "off by one" type of coding error is one of the easiest to make in computing! The plot appears in Figure 3; you can see clearly how slow the convergence is.

Challenge 5: iterate the two-dimensional Hénon map

$$\begin{aligned} x_{n+1} &= 1 - 1.4\,{x_n}^2 + y_n, \\ y_{n+1} &= 0.3\,x_n \end{aligned}$$

10000 times, this time creating lists of coordinate pairs.

The idea this time is to create two separate lists, `x_list` and `y_list`, containing, respectively, the iterates $x_0, x_1, x_2, \ldots$ and $y_0, y_1, y_2, \ldots$. We'll initialise these two lists as "list containing x_0" and "list containing y_0" respectively.

```
x, y = 0.5, 0.5
x_list, y_list = [x], [y]
for n in range(10000):
    x, y = 1 - 1.4*x**2 + y, 0.3*x
    x_list.append(x)
```

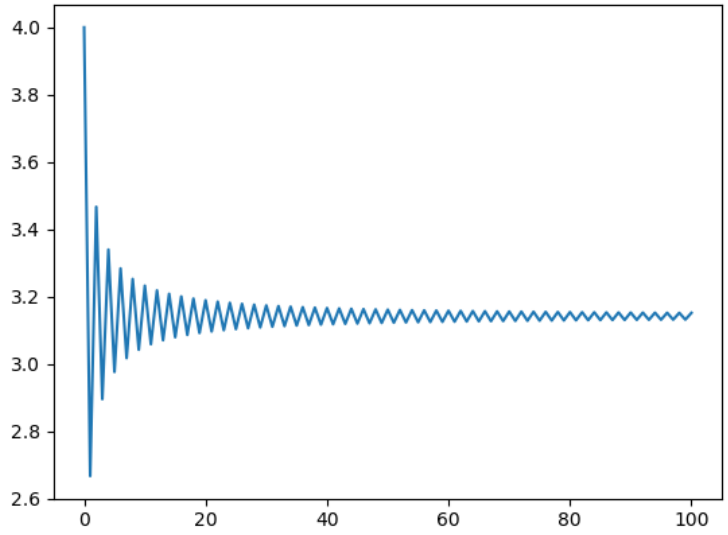

Figure 3: Plot of partial sums in series that converges slowly to π

```
    y_list.append(y)
```

I won't bother printing these enormous lists, each of which has 10001 elements. Instead, let's do a point plot of `x_list` (horizontal axis) against the corresponding values of `y_list` (vertical axis), using the optional **keyword argument** `markersize` to make the points as small as we can.

```
plt.plot(x_list, y_list, '.', markersize=0.1)
```

The image appears as Figure 4. This shape, which is called a **strange attractor**, is extremely intricate, and has some fascinating mathematical properties.

3.3 `while` loops

The `for` loop is a wonderful thing: it allows us to get a computer to do the same thing over and over again, which they're great at and we hate. But they only work if we know in advance exactly how many times we want the thing, whatever it is, to be done. And sometimes we don't; sometimes we want to say to the computer "Do this thing over and over again until the job is done, but I don't know how many times that will be yet." Here's an example.

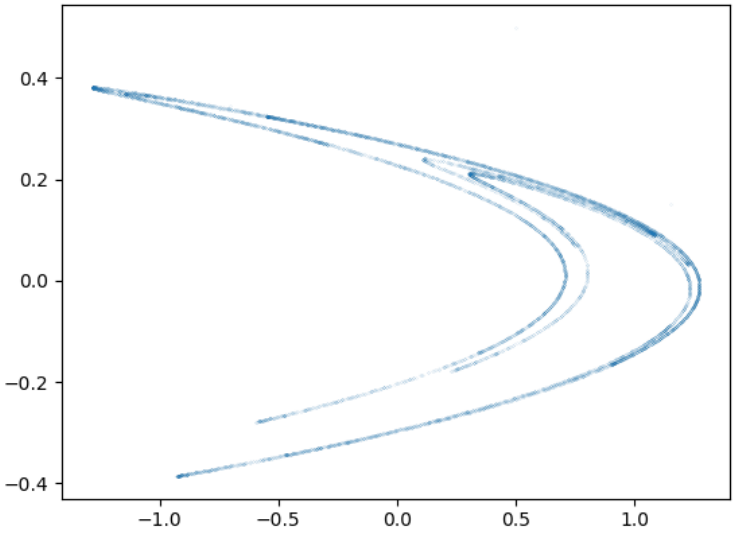

Figure 4: Strange attractor of the Hénon map

Challenge 1: the iteration

$$x_{n+1} = \frac{x_n + 2}{{x_n}^2 + 1}$$

converges fairly slowly to $\sqrt[3]{2}$. Run this iteration 20 times, starting with $x = 1.0$, printing the successive values of x.

Then run it, instead, not for a fixed number of times but until x^3 is within 0.00005 of 2.

First the fixed number of times; a `for` loop is ideal here.

```
x = 1.0
for n in range(20):
    x = (x + 2.0)/(x**2 + 1.0)
    print(x)
```

```
1.5
1.0769230769230769
1.4246575342465755
1.1303809228863424
1.3743236803695795
1.1680849791244732
1.339897989633841
```

```
1.1948149323951562
1.3160478127293076
1.2137883779591594
1.2994022047163227
1.2272525074812513
1.2877338385533594
1.2367999252342665
1.2795324040899612
1.2435645601831735
1.2737579666268228
1.2483541288095155
1.269687822783779
1.2517433678387995
```

For the other part of the task, where we want to run it until x^3 gets close to 2, we replace the `for` loop with what's called a `while` loop:

```
x = 1.0
while abs(x**3 - 2) > 0.00005:
    x = (x + 2.0)/(x**2 + 1.0)
    print(x)
```

```
1.5
1.0769230769230769
1.4246575342465755
1.1303809228863424

...

1.259930405269743
```

(There are many values here, and I've cut most of them!)

What this means is "Do this thing for as long as the absolute value of $(x^3 - 2)$ is greater than 0.00005".

Note: A while loop always includes a **condition**, which should consist of a **Boolean expression**. This expression starts off with the value `True`, otherwise the loop won't run at all. And it ends up with the value `False`, otherwise the loop will run forever.

OK, strictly, neither of those things is quite right: sometimes, we want a loop that fails to run in certain circumstances, and sometimes, surprisingly, we want one that runs forever, or at any rate until we force a break.

But those are the exceptions: almost always we want a while loop that executes at least once, but only a finite number of times, which means the condition must start off true, and

change its value during the execution of the loop in such a way that it ends up false. This is a good example: we start with a value of x whose cube is a long way from 2, making the condition `abs(x**3 - 2) > 0.00005` true, and we have an iteration we know converges to $\sqrt[3]{2}$, meaning that eventually the condition will be false.

Note: If the condition in a while loop goes from being true to being false, it must depend on the value of at least one **variable**, and that value must **change** during the execution of the loop.

This seems almost obvious, right? But you'd be surprised by how many attempted `while` loops we see that don't obey these rules! So do check, when you write a `while` loop: does its condition depend on the value of a variable, and does that value change when the loop is executed? Your `while` loop *may* not work even if both those things are true, but it definitely won't work if they're not. (Except for those special cases I talked about; but let's worry about those later.)

Challenge 2: run the iteration

$$x_{n+1} = \cos x_n$$

starting with $x = 1.0$, until successive iterates lie within 0.00005 of each other.

This one is a little different. It's one thing to check whether we're near the cube root of 2 by comparing x^3 with 2, but how do we check we're near a solution of $x = \cos x$? The only way, really, is to carry on iterating until successive iterates don't change very much; we call this a **heuristic convergence criterion**.

Here's a first crack at the code:

```
from math import cos
x = 1.0
while abs(x - cos(x)) > 0.00005:
    x = cos(x)
    print(x)
```

```
0.5403023058681398
0.8575532158463933
0.6542897904977792
0.7934803587425655
0.7013687736227566
0.7639596829006542
0.7221024250267077
0.7504177617637605
0.7314040424225098
0.7442373549005569
0.7356047404363473
```

```
0.7414250866101093
0.7375068905132428
0.7401473355678757
0.7383692041223232
0.739567202212256
0.7387603198742114
0.7393038923969057
0.7389377567153446
0.7391843997714936
0.7390182624274122
0.7391301765296711
0.7390547907469175
0.7391055719265361
```

This is fine, except for one thing: it's not very efficient. That's because the cosine function is evaluated twice at each turn of the loop: once when the condition

```
abs(x - cos(x)) > 0.0005
```

is checked, and once when the update step,

```
x = cos(x)
```

is executed inside the loop.

This is a very minor problem for a small, fast program like this, but it's good to get into good habits early on. How do we solve it? One way is to have not one value of `x` but two on the go; let's call them `oldx` and `newx`, where at any one time, `newx = cos(oldx)`. The condition will then be

```
abs(oldx - newx) > 0.00005
```

which doesn't involve the evaluation of a cosine. Here's the complete program:

```
from math import cos
oldx = 1.0
newx = cos(oldx)
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = cos(oldx)
    print(oldx)
```

```
0.5403023058681398
0.8575532158463933
0.6542897904977792
0.7934803587425655
0.7013687736227566
0.7639596829006542
0.7221024250267077
0.7504177617637605
0.7314040424225098
```

```
0.7442373549005569
0.7356047404363473
0.7414250866101093
0.7375068905132428
0.7401473355678757
0.7383692041223232
0.739567202212256
0.7387603198742114
0.7393038923969057
0.7389377567153446
0.7391843997714936
0.7390182624274122
0.7391301765296711
0.7390547907469175
0.7391055719265361
```

Actually, we can improve it slightly by printing the value of `newx` instead of that of `oldx`; that way, we squeeze out one more iterate, and get what's probably a more accurate result.

> **Note**: The **heuristic convergence criterion** is impossible to justify in general, and indeed, in general it's false. The fact that iterates aren't changing very much tells us nothing rigorous about whether or not we're close to a solution. But it's extremely widely used anyway, because exceptions are rare.

Here's an example of an exception. If you run the following code

```
oldx = 0.9
newx = -oldx**4 + 4*oldx**3 -5*oldx**2+3*oldx+0.00001
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = -oldx**4 + 4*oldx**3 -5*oldx**2+3*oldx+0.00001
    print(newx)
```

you get quite a long printout, ending in

```
0.993773880037735
```

And yet there's no solution of $x = -x^4 + 4\,x^3 - 5\,x^2 + 3\,x + 0.00001$ anywhere near 0.99377; the real solutions are about $x = -0.00005$ and $x = 2.00005$. So be a bit wary of the heuristic convergence criterion!

> **Challenge 3**: run the `while` loop iteration from Challenge 1 again, this time putting the iterates into a list.

```
x = 1.0
x_list = [x]
while abs(x**3 - 2) > 0.00005:
    x = (x + 2.0)/(x**2 + 1.0)
    x_list.append(x)
print(x_list)
```

I won't bother showing the values!

Challenge 4: run the `while` loop iteration from Challenge 2 again, this time putting the iterates into a list.

```
from math import cos
oldx = 1.0
newx = cos(oldx)
x_list = [oldx, newx]
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = cos(oldx)
    x_list.append(newx)
print(x_list)
```

Again, let's not bother with the values.