MATH40006: An Introduction To Computation
Course notes, Section 5

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

5 User-defined Functions

5.1 The basics

Python comes with a number of **functions**, such as `pow` and `ord`; modules like `math` and `cmath` contain many more. A function takes Python data as its input, and **returns** Python data as its output, as in

```
ord('M')
```

```
77
```

or

```
pow(2, 7, 7)
```

```
2
```

The next step in Python programming is *writing your own functions*.

Challenge 1: write and test a function called `absolute_difference`, which takes as its arguments two numbers `x` and `y`, and returns the value of `|x - y|`.

Notice that this challenge is different from any you've done before, in that you're not writing a one-off program that will run only once. The aim here is to *teach Python a new function*. That means that the process has at least two stages: write the function, and then use it.

The syntax for functions in Python is this:

```
def <function_name>:
    <code>
    <code>
    <code>
    return <value>
```

In this case, that looks like this:

```python
def absolute_difference(x, y):
    return abs(x - y)
```

Python has now “learned” a new function, called `absolute_difference`. Let’s test it:

```python
absolute_difference(5, 7)
```

```
2
```

```python
absolute_difference(5.3, 2.9)
```

```
2.4
```

```python
absolute_difference((8 + 1j), (4 - 2j))
```

```
5.0
```

Unlike all the other programs you’ve written on this course, this one is available for multiple re-use. Python will forget it when you close the session, but you’ve kept the code, so you can always “Shift-Return” on it again.

The bulk of your Python programming from now on will consist of writing and using Python functions.

Challenge 2: write and test a function called `square_list`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns a list of the squares between 0 and `(n-1)` inclusive.

```python
def square_list(n):
    return [r**2 for r in range(n)]
```

Let’s test this function:

```python
square_list(12)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

```python
square_list(1)
```

```
[0]
```

```
square_list(0)
```

```
[]
```

5.2 Testing

It's always important to **test** your functions, as above. But how much testing is the right amount of testing?

That's a large question. For complex software, the testing process needs to be exhaustive, painstaking and mainly automated, and that includes sophisticated Python functions; the cleverer and more intricate the code, the more there is to go wrong. The last function is right at the simple end, though, so all I've done is manually test one typical case, `n=12`, and two **edge cases**, `n=1` and `n=0`. The function has performed adequately on all three tests.

5.3 The importance of `return`

Challenge 3: write and test a function called `pi_sum`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns the value of

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

```
def pi_sum(n):
    total = 0.0
    for r in range(n+1):
        total += (4*(-1)**r)/(2*r+1)
    return total
```

A couple of things to notice here. One is the tricksy use of

```
    total += (4*(-1)**r)/(2*r+1)
```

to mean

```
    total = total + (4*(-1)**r)/(2*r+1)
```

This is called **augmented assignment**, and we'll make fairly free use of it from now on. (There are some subtleties associated with it, which we'll look at.) Not all languages offer augmented assignment, though many do.

The second, which is more general, concerns how we've made sure that the output from the function is our desired `total`, using the line

```
    return total
```

Nearly every function you'll write will have a line, at or near the end, beginning with the word `return`, telling Python what it is you'd like the function to output. In this case, it was simply the final value of the variable called `total`.

All languages that allow you to write your own functions like this use a mechanism along these lines allowing you to specify the function's output, though how it works in detail can vary a lot from language to language.

Now for some testing:

```
pi_sum(100)
```

```
3.1514934010709914
```

```
pi_sum(10000)
```

```
3.1416926435905346
```

```
pi_sum(0)
```

```
4.0
```

```
pi_sum(1)
```

```
2.666666666666667
```

Here, I've tested the case n=100, for which I already know what the answer should be; a larger value of n, to check the convergence to π, and a couple of small values of n, which serve as edge cases and also allow me to check by hand-calculation.

5.4 Returning vs printing

Challenge 4: write and test a function called cos_nest, which takes as its arguments a number x0, assumed to be a float or an int, and a second number n, assumed to be a non-negative integer, and returns the iterates x_0, x_1, x_2, $\ldots$, x_n of the iteration

$$x_{r+1} = \cos x_r,$$

starting with x=x0.

```
def cos_nest(x0, n):
    from math import cos
    x = x0
    x_list = [x]
    for r in range(n):
        x = cos(x)
        x_list.append(x)
    return x_list
```

Here, notice, what we want to be returned is the final value of x_list. Notice too that we've imported the cos function inside our function definition; that means that we don't have to remember to do that every time we use the function. It's good practice, within function code, to import anything you know you're going to need (but only that).

Testing:

```
cos_nest(1.0, 20)
```

```
[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]
```

```
cos_nest(0.0, 0)
```

```
[0.0]
```

Certainly, the function *seems* to be working. But we need to tread slightly carefully. One very frequent error that people often make when they're new to functions is replacing the return line with a call to print, as in the following piece of code:

```
def cos_nest_with_bug(x0, n):
    from math import cos
    x = x0
    x_list = [x]
    for r in range(n):
        x = cos(x)
        x_list.append(x)
    print(x_list)
```

The behaviours of our two functions look superficially similar:

```
cos_nest(1.0, 5)
```

```
[1.0,
 0.5403023058681398,
 0.8575532158463933,
 0.6542897904977792,
 0.7934803587425655,
 0.7013687736227566]
```

```
cos_nest_with_bug(1.0, 5)
```

```
[1.0, 0.5403023058681398, 0.8575532158463933,
0.6542897904977792, 0.7934803587425655,
0.7013687736227566]
```

The problems arise if we try to perform calculations using the "output" from the second function. With the first function, for example, plotting the iterates works fine:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(21), cos_nest(1.0, 20))
```

produces an image identical to Figure 2 from Section 3 of the notes. However, if we try

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(21), cos_nest_with_bug(1.0, 20))
```

we get a printout of our list of iterates, and then a very scary-looking error message:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-2bc00b1d27d2> in <module>()
      1 get_ipython().run_line_magic('matplotlib', 'qt')
      2 import matplotlib.pyplot as plt
----> 3 plt.plot(range(21), cos_nest_with_bug(1.0, 20))

C:\Anaconda3\lib\site-packages\matplotlib\pyplot.py in plot(*args, **kwargs)
   3315                       mplDeprecation)
   3316     try:
```

```
-> 3317         ret = ax.plot(*args, **kwargs)
   3318     finally:
   3319         ax._hold = washold

C:\Anaconda3\lib\site-packages\matplotlib\__init__.py in inner(ax, *args, **kwargs)
   1895                     warnings.warn(msg % (label_namer, func.__name__),
   1896                                   RuntimeWarning, stacklevel=2)
-> 1897             return func(ax, *args, **kwargs)
   1898         pre_doc = inner.__doc__
   1899         if pre_doc is None:

C:\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in plot(self, *args, **kwargs)
   1404         kwargs = cbook.normalize_kwargs(kwargs, _alias_map)
   1405
-> 1406         for line in self._get_lines(*args, **kwargs):
   1407             self.add_line(line)
   1408             lines.append(line)

C:\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _grab_next_args(self, *args,
    405                 return
    406             if len(remaining) <= 3:
--> 407                 for seg in self._plot_args(remaining, kwargs):
    408                     yield seg
    409                 return

C:\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _plot_args(self, tup, kwargs)
    366         # downstream.
    367         if any(v is None for v in tup):
--> 368             raise ValueError("x and y must not be None")
    369
    370         kw = {}

ValueError: x and y must not be None
```

Wow. What's happened here and why?

The main lesson is that **printing is not the same as returning**. The values that a function *returns* are available for calculation; the values that it *prints* can only be read by humans like us.

Printing is what we call a **side-effect** of using the function. It's actually very useful that functions can have side-effects, and it's even useful that printing can be one of them (as you may find out when you start to **debug** your functions), but what we typically want is for a value to be *returned*. Functions should have inputs and outputs!

(Actually, like many of my sweeping generalisations, this isn't strictly true; some functions don't have inputs, and some functions don't have outputs. But those are the exceptions; for now, assume that every function you write ought to have a line that begins with the word `return`.)

Challenge 5: write and test a function called `cos_fixedpoint`, which takes as its arguments a number `x0`, assumed to be a float or an int, and a second number `tolerance`, assumed to be a positive float, and returns the iterates of the iteration

$$x_{r+1} = \cos x_r,$$

starting with `x=x0`, until successive iterates are within `tolerance` of one another.

```
def cos_fixedpoint(x0, tolerance):
    from math import cos
    oldx = x0
    newx = cos(x0)
    x_list = [oldx, newx]
    while abs(oldx-newx) > tolerance:
        oldx = newx
        newx = cos(oldx)
        x_list.append(newx)
    return x_list
```

Testing:

```
cos_fixedpoint(1.0, 0.00005)
```

```
[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,
 0.7390182624274122,
 0.7391301765296711,
 0.7390547907469175,
 0.7391055719265361,
 0.739071365298945]
```

```
cos_fixedpoint(1.0, 0.005)
```

```
[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]
```

```
cos_fixedpoint(0.0, 5.0)
```

```
[0.0, 1.0]
```

This function is behaving as we would wish.

Challenge 6: write and test a function called `henon_iterates`, which takes as its arguments `a` and `b`, assumed to be floats or ints, `x0` and `y0`, assumed to be floats, and `n`, assumed to be a non-negative int, and returns the iterates up to x_n, y_n of the iteration

$$\begin{aligned} x_{r+1} &= 1 - a\,{x_r}^2 + y_r, \\ y_{r+1} &= b\,x_r \end{aligned}$$

starting with `x=x0`, `y=y0`.

Your function should return a **tuple** containing a list of x-values and a list of y-

values.

```
def henon_iterates(a, b, x0, y0, n):
    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)
    return (x_list, y_list)
```

Our test strategy here, I think, should be to try, first, one with a small value of `n`, which we can check by hand if we like and, secondly, one with the values we used when we generated Figure 4 from Section 3 of the notes. First, then:

```
henon_iterates(2.0, 0.5, 1.0, 1.0, 5)
```

```
([1.0, 0.0, 1.5, -3.5, -22.75, -1035.875],
 [1.0, 0.5, 0.0, 0.75, -1.75, -11.375])
```

And then, for the plot:

```
xy_values = henon_iterates(1.4, 0.3, 0.5, 0.5, 10000)
plt.plot(xy_values[0], xy_values[1], '.', markersize=0.1)
```

This does indeed produce a copy of Figure 4 from Section 3.

Note that the line

```
return (x_list, y_list)
```

could simply have been written

```
return x_list, y_list
```

Note too that we can use this function to assign values to two variables simultaneously, as follows:

```
x_values, y_values = henon_iterates(1.4, 0.3, 0.5, 0.5, 10000)
plt.plot(x_values, y_values, '.', markersize=0.1)
```

This is a neat feature, which not all computer languages offer.

5.5 And one more

Lastly, a nice simple one that I hope you'll find useful:

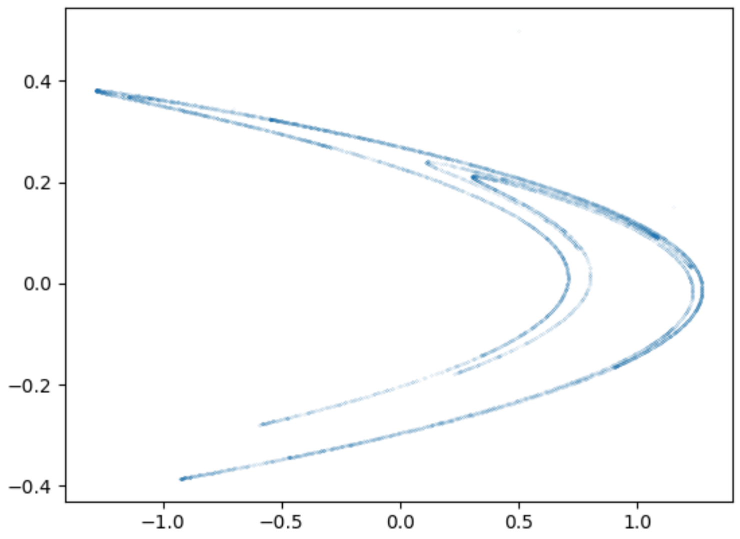

Figure 1: Strange attractor of the Hénon map

Challenge 7: write and test a function called `multiple_filter`, which takes as its arguments `ints`, assumed to be a list of ints and `n`, assumed to be an int greater than 1, and returns a list containing all the elements of `ints`, with multiples of `n` other than `n` itself removed.

```
def multiple_filter(ints, n):
    return [r for r in ints if r == n or r % n > 0]
```

Test this one yourselves!

5.6 Comments and docstrings

Here's a nice quote from a Python website:

> When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important. If you're like me, you've probably opened up old codebases and wondered to yourself, "What in the world was I thinking?" If you're having a problem reading your own code, imagine what your users or other developers are experiencing when they're trying to use or contribute to your code.

We've put this off too long! Now that our code is becoming more complex, it's going to be useful to us to add **comments** and **docstrings** to it.

5.6.1 Docstrings

A docstring is a description of what the function does. That description can be a one-liner, or something more complicated; this is a design decision. One-line docstrings begin and end with double quotes, ".

Challenge 8: write a version of henon_iterates with a one-line docstring.

```
def henon_iterates(a, b, x0, y0, n):
    "Returns x and y iterates of the Henon map."
    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)
    return (x_list, y_list)
```

The docstring is then available to users who want to find out about the function, as follows:

```
help(henon_iterates)
```

```
Help on function henon_iterates in module __main__:

henon_iterates(a, b, x0, y0, n)
    Returns x and y iterates of the Henon map.
```

Multi-line docstrings begin and end with three double quotes, """. There are "Pythonic" conventions about how a multi-line docstring should be set up, namely

```
    """
    Summary line.

    Extended description of function.

    Parameters:
    arg1 (type, if appropriate): Description of first argument
    arg2 (type, if appropriate): Description of second argument
    ...

    Returns:
    type: Description of return value

    """
```

Challenge 9: write a version of `henon_iterates` with a multi-line docstring.

```
def henon_iterates(a, b, x0, y0, n):
    """
    Returns x and y iterates of the Henon map.

    For real  parameters a and b, iterates the Henon map,
    (x, y) -> (1 - a*x**2 + y, b*x),
    n times, starting with (x, y) = (x0, y0), returning a tuple
    containing a list of x-values and a list of y-values.

    Parameters:
    a (float): A real system parameter
    b (float): A real system parameter
    x0 (float): Initial value of state variable x
    y0 (float): Initial value of state variable y
    n (int, non-negative): Number of times to iterate

    Returns:
    tuple of list of float: (x_list, y_list) where
        x_list = [x0, x1, ..., xn],
        y_list = [y0, y1, ..., yn]

    """

    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)
    return (x_list, y_list)
```

A one-line docstring is fine for simple functions, especially if it's only you who's ever going to use them. A multi-line docstring becomes appropriate if the function is more complex, especially if you're designing for clients, or end-users other than yourself. This is largely a matter of judgement.

5.6.2 Comments

Docstrings are mostly about documenting your code for users (which of course includes yourself). They also help out **developers** (which also includes yourself), but developers often need a little more. It's useful to put into your code **comments**, which are pieces of text that Python ignores but that are readable by human beings. In Python, comment lines begin with a hashtag, #.

Challenge 10: write a version of `henon_iterates` with a one-line docstring *and comments*.

```
def henon_iterates(a, b, x0, y0, n):
    """Returns x and y iterates of the Henon map."""

    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]

    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)

    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

I hope you can see how this hugely improves the readability of your code! Let's use comments, and at least single-line docstrings, from now on.

5.6.3 Commenting out

The main use of comments is in order to improve the readability of your code, both to other developers and to yourself. But there are other reasons why it might be useful to be able to make lines in a program invisible to Python without having to delete them.

For example, suppose you've written a "cosine iteration" function (yes, yet another) which delivers not a set of iterates but a final value, and another one that you think may converge more rapidly. It can be useful to insert a `print` statement into your functions, *on a temporary basis*, so that you can view the iterates and test this hunch, as in

```
def cosine_fixedpoint1(x0, tolerance):
    """Finds solution of x = cos x by iteration."""

    # import cosine function
    from math import cos

    # initial values of oldx and newx
    oldx = x0
    newx = cos(oldx)

    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate cosine, updating oldx and newx
        oldx = newx
        newx = cos(oldx)
        print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint1(1.0, 0.0005)
```

```
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

0.7389377567153446
```

As contrasted with

```
def cosine_fixedpoint2(x0, tolerance):
    """Finds solution of x = cos x by iteration."""

    # import cosine and sine functions
    from math import cos, sin

    # initial values of oldx and newx
    oldx = x0
    newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))

    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate function, updating oldx and newx
        oldx = newx
        newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))
        print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint2(1.0, 0.0005)
```

```
0.7391128909113617
0.7390851333852839

0.7390851333852839
```

So, yes, our second one is way more efficient! The thing is, now we know that, we probably don't need the `print` statements any more. But rather than deleting them (which would mean that if we did turn out to want them again, we would have to rewrite them), we could simply comment them out, as here:

```
def cosine_fixedpoint2(x0, tolerance):
    """Finds solution of x = cos x by iteration."""

    # import cosine and sine functions
    from math import cos, sin

    # initial value of oldx and newx
    oldx = x0
    newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))

    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate function, updting oldx and newx
        oldx = newx
        newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))

        # print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint2(1.0, 0.0005)
```

```
0.7390851333852839
```

This is also quite a good **debugging strategy**: if your function is going wrong and you're not sure why, consider printing the values of key variables so you can "look under the hood" (or bonnet!) and see what's really going on. Then, as you stop seeming to need those `print` statements, act cautiously by commenting them out instead of deleting them; delete them finally only when you're sure your code is working as designed.