# MATH40006: An Introduction To Computation
# Module notes, Section 6

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

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

## 6 More about functions

### 6.1 Lambda-expressions

You may remember that we wrote some code for iterating a general function `f`, returning a list of the iterates:

```
def nestlist(f, x0, n):
    "Returns iterates of x = f(x), x0, x1, ... xn"

    x = x0
    xlist = [x0]

    for r in range(n):
        x = f(x)
        xlist.append(x)

    return xlist
```

Now, suppose we wanted to carry out a Newton's Method iteration for $x = \cos x$, which means carrying out the iteration

$$x_{n+1} = x_n - \frac{x_n - \cos x_n}{1 + \sin x_n}.$$

One way to do that would be to write a function, and iterate that:

```
from math import cos, sin

def newtonFn(x):
    return x - (x-cos(x))/(1+sin(x))

print(nestlist(newtonFn, 1.0, 7))
```

```
[1.0, 0.7503638678402439, 0.7391128909113617,
     0.739085133385284, 0.7390851332151607,
     0.7390851332151607, 0.7390851332151607,
     0.7390851332151607]
```

But it seems a lot of trouble to go to, writing a Python function with only one line, only to use it once. There's an alternative, involving creating a function that instead of being referred to by name, is referred to via a description of what it does. Here's how that works:

```
from math import cos, sin

print(nestlist(lambda x: x - (x-cos(x))/(1+sin(x)), 1.0, 7))
```

```
[1.0, 0.7503638678402439, 0.7391128909113617,
     0.739085133385284, 0.7390851332151607,
     0.7390851332151607, 0.7390851332151607,
     0.7390851332151607]
```

The expression `lambda x: x - (x-cos(x))/(1+sin(x))` is called a **lambda-expression** (at least in Python it is; in other languages, the equivalent concept can go by the name "pure function" or "anonymous function", amongst others). It can be read as "the function that sends $x$ to $x - (x - \cos x)/(1 + \sin x)$".

Although lambda-expressions are a way of referring to functions without referring to them by name, they can actually be given names! The following also works, for example:

```
from math import cos, sin

newtonLam = lambda x: x - (x-cos(x))/(1+sin(x))

print(nestlist(newtonLam, 1.0, 7))
```

```
[1.0, 0.7503638678402439, 0.7391128909113617,
     0.739085133385284, 0.7390851332151607,
     0.7390851332151607, 0.7390851332151607,
     0.7390851332151607]
```

Ths simple idea can speed up code development a surprising amount.

## 6.2 Function arguments

### 6.2.1 Specifying arguments by keyword

Recall our `henon_iterates` function (this is the version with the 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)
```

In the past, we've called it with statements like

```
henon_iterates(1.4, 0.3, 0.5, 0.5, 5)
```

```
([0.5,
  1.15,
  -0.7014999999999995,
  0.656056850000001,
  0.18697517339530675,
  1.1478734533473132],
 [0.5,
  0.15,
  0.345,
  -0.21044999999999983,
  0.1968170550000003,
  0.05609255201859203])
```

This specifies the function's arguments **by position**. The disadvantage of doing things this way is that the user has to remember which argument goes where and stands for what. If they get mixed up and type

```
henon_iterates(0.5, 0.5, 1.4, 0.3, 5)
```

it'll all go horribly wrong.

Luckily, Python allows the user to specify arguments **by keyword** instead: the user could have typed:

```
henon_iterates(x0=0.5, y0=0.5, a=1.4, b=0.3, n=5)
```

or

```
henon_iterates(a=1.4, b=0.3, x0=0.5, y0=0.5, n=5)
```

or even something crazy like

```
henon_iterates(b=0.3, n=5, y0=0.5, a=1.4, x0=0.5)
```

and it would have worked just fine.

This kind of flexibility is kind of unusual, and is a distinctive thing that Python offers.

### 6.2.2 Default arguments

It's possible to give certain arguments **default values**, so that if the user leaves them out, Python will assume they meant those values. You've seen an example of this in the built-in **split** method, where if no substring is specified, Python assumes you mean the space character, `' '`.

If you want to equip your own function with default arguments, there's only one rule: any ordinary non-default arguments have to be listed first.

**Challenge 1**: write and test a function called `henon_iterates2` that uses by default the values $a = 1.4$, $b = 0.3$, but assigns no default values to any of the other arguments.

```
def henon_iterates2(x0, y0, n, a=1.4, b=0.3):
    """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)
```

Then

```
henon_iterates2(0.5, 0.5, 5, 1.4, 0.3)
```

and

```
henon_iterates2(0.5, 0.5, 5)
```

generate exactly the same output, but

```
henon_iterates2(0.5, 0.5, 5, 1.2, 0.1)
```

will use different values for $a$ and $b$.

Note that default arguments can be referred to by position or keyword, just like ordinary arguments: the input

```
henon_iterates2(x0=0.5, b=0.1, y0=0.5, n=5)
```

works just fine, for example (and will use the default value for $a$ but not for $b$).

### 6.2.3 Keyword-only arguments

Python always *allows* arguments to be referred to by keyword, but it's actually possible to *require* this.

**Challenge 2**: write and test a function called `henon_iterates3` that uses the keyword-only arguments $a$ and $b$, with default values $1.4$ and $0.3$ respectively.

```
def henon_iterates3(x0, y0, n, *,  a=1.4, b=0.3):
    """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)
```

Then things like

```
henon_iterates3(0.5, 0.5, 5)
```

and

```
henon_iterates3(0.5, 0.5, 5, b=0.1)
```

and

```
henon_iterates3(0.5, 0.5, 5, b=0.1, a=1.5)
```

and

```
henon_iterates3(b=0.1, a = 1.5, x0=0.5, y0=0.5, n=5)
```

all work fine, but Python won't allow a purely positional call like

```
henon_iterates3(0.5, 0.5, 5, 1.5, 0.1)
```

Our $a$ and $b$ are now strictly **keyword-only arguments**.

**Note**: In this example we've give our keyword-only arguments default values, but there's nothing to say we have to do that; this is a question of design. The following would be perfectly acceptable:

```
def henon_iterates3(x0, y0, n, *,  a, b):
    """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 user would then have to specify the values of $a$ and $b$, by name, when using the function: the followuing would work:

```
henon_iterates3(0.5, 0.5, 5, a=1.4, b=0.3)
```

but the following wouldn't:

```
henon_iterates3(0.5, 0.5, 5)
```

So when you're writing a function, you have two decisions to make concerning each of the function's arguments: do I want this argument to have a default value, and do I want it to be keyword-only? Those decisons are separate from, and independent from, each other.

That said, it's *usual* for keyword-only arguments to be given default values.

## 6.3 "Infinite" loops

When introducing `while` loops, I gave it as a rule that the condition at the top of the loop should start off **true**, and end up **false**. But in fact, this isn't an absolutely iron rule. The reason we can relax it is that every time a function returns a value it stops; that means that we don't need to be sure that our condition will become false, as long as we're sure that we'll eventually return a value.

**Challenge 1**: write and test two versions of a Hailstone sequence function, one using a terminating `while` loop and the other using an infinite `while` loop that terminates through a `return`.

First the conventional implementation:

```
def hailstone_sequence1(a):
    """Returns iterates of the Hailstone sequence."""

    alist = [a]

    # loop until value 1 is reached
    while a > 1:

        if a % 2 == 0:
            a //= 2
        else:
            a = 3*a+1

        alist.append(a)

    return alist
```

Now the wacky infinite-loop one:

```
def hailstone_sequence2(a):
    """Returns iterates of the Hailstone sequence."""

    alist = [a]

    # loop 'forever'
    while True:

        if a % 2 == 0:
            a //= 2
        else:
            a = 3*a+1

        alist.append(a)

        # if value 1 has been reached, return list of values and stop
        if a==1:
            return alist
```

Then, for example,

```
hailstone_sequence1(35)
```

and

```
hailstone_sequence2(35)
```

both return

```
[35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Note the nifty way of setting up an "infinite" loop using

```
while True:
```

Note, too, that the loop isn't really infinite; the program always terminates because all (known) Hailstone sequences hit 1 eventually, forcing the `return`. So this is only a *potentially* infinite loop, not an *actually* infinite loop. But it should go without saying that

```
while True:
```

should be handled with caution, in case your potentially infinite loop really does loop forever!

The existence of the `itertools` module even allows us to set up infinite `for` loops.

**Challenge 2**: write and test three versions of a function for calculating the **integer square root** of a non-negative integer $a$: that is, the largest integer $n$ such that $n^2 \leq a$.

First a conventional implementation with an ordinary `while` loop:

```
def int_sqrt1(a):
    """Returns integer square root of a."""

    n = 0

    # increment until n-squared > a
    while n**2 <= a:
        n+=1

    # return previous value of n
    return n-1
```

Now one using `while True`:

```
def int_sqrt2(a):
    """Returns integer square root of a."""

    n = 0

    # increment 'forever'
    while True:
        n+=1

        # if n-squared > a, return previous value of n
        if n**2 > a:
            return n-1
```

And finally a rather neat version that uses a (potentially) infinite `for` loop:

```
def int_sqrt3(a):
    """Returns integer square root of a."""

    from itertools import count

    # loop over n = 0, 1, 2, ... 'forever'
    for n in count():

        # if n-squared > a, return previous value of n
        if n**2 > a:
            return n-1
```

All three functions will behave in the same way: give them the input 10, and they'll return 3, and give them the input 20, and they'll return 4. The last one is probably my favourite.

**Note**: There's a far more efficient way of implementing the integer square root, which we may look at later in the module.

## 6.4 Recursion

So far, when we've wanted Python to do the same thing over and over again, we've written a loop, using either `while` or `for`: this is called **iteration**. There exists an entirely different approach, called **recursion**, which works by getting a function to call itself: that is, by having the function's code include the function itself.

On the face of it, this seems paradoxical; almost crazy: if a function is defined in terms of itself, is that not a circular definition? But actually, it's not really a circle; it's more like a helix! Let me show you what I mean.

**Challenge 1a**: write and test an old-skool iterative function called `square_sum1` that takes as its argument an integer `n`, and returns the sum of the squares from 0 to `n`.

```
def square_sum1(n):
    """Returns the sum of the first n squares."""
    total = 0

    for r in range(1,n+1):
        total += r**2

    return total
```

Then testing:

```
square_sum1(10)
```

```
385
```

```
square_sum1(1)
```

```
1
```

```
square_sum1(0)
```

```
0
```

**Challenge 1b**: write and test a **recursive** function called `square_sum2` that does the same thing.

```
def square_sum2(n):
    """Returns the sum of the first n squares."""

    # base case
    if n==0:
        return 0

    # recursion step
    else:
        return n**2 + square_sum2(n-1)
```

Then testing:

```
square_sum2(10)
```

```
385
```

```
square_sum2(1)
```

```
1
```

```
square_sum2(0)
```

```
0
```

So, how does this work? The answer is that when we type

`square_sum2(10)`

this becomes

`10**2 + square_sum2(9)`

and

`10**2 + 9**2 + square_sum2(8)`

and so on, until we eventually get

`10**2 + 9**2 + 8**2 + ... + 1**2 + square_sum2(0)`

and `square_sum2(0)` is defined as zero! There's a downward spiral until Python hits a value it knows.

This is the basic structure of a recursive function, then:

```
def <recursive_function>(<args>):

    <code>
    <code>
    ...

    # base case
    if <base_condition>:
        return <base_value>

    # recursion step
    else:
        <code>
        <code>
        ...
        return <recursive_function>(<changed_args>)
```

Just as with a `while` loop you have have a condition that ends up false, with a recursion you have to have a condition that ends up true: that is, you need to be sure that the base case is eventually reached.

**Challenge 2a**: write and test an iterative function called my_sqrt1 that takes as its arguments an number a, a starting value x0 and a tolerance, and returns an approximation to the square root of a, obtained by iterating

$$x_{n+1} = \frac{x_n + a}{x_n + 1}$$

until $|x^2 - a|$ is within tolerance.

```
def my_sqrt1(a, x0, tolerance):
    """Calculates an approximation to sqrt(a)."""

    x = x0

    while abs(x**2 - a) > tolerance:
        x = (x + a)/(x + 1.0)

    return x
```

Then testing:

```
my_sqrt1(5, 5, 0.000005)
```

```
2.236067059356593
```

**Challenge 2b**: write and test a **recursive** function called my_sqrt2 that does the same thing.

```
def my_sqrt2(a, x0, tolerance):
    """Calculates an approximation to sqrt(a)."""

    # base case
    if abs(x0**2 - a) <= tolerance:
        return x0

    # recursion step
    else:
        # update x
        x = (x0 + a)/(x0 + 1.0)
        # return function evaluated on new value of x
        return my_sqrt2(a, x, tolerance)
```

Then testing:

```
my_sqrt2(5, 5, 0.000005)
```

```
2.236067059356593
```

Sometimes, a recursive implementation needs two functions: an inner one, implementing the actual recursion, and an outer "wrapper". A good example is when you want to return a list; somehow, the recursion needs access to the "list so far".

**Challenge 3a**: write a recursive function called `cos_nestlist_inner` that takes as its arguments a list of iterates `xlist` and a non-negative integer `n`, and returns an iterate list for $x_{r+1} = \cos x_r$, lengthened by a further `n` iterates.

```
def cos_nestlist_inner(xlist, n):
    from math import cos

    # base case
    if n==0:
        return xlist

    # recursion step
    else:
        # last value of x
        x = xlist[-1]

        # next value of x
        x = cos(x)

        xlist.append(x)

        # return using n-1
        return cos_nestlist_inner(xlist, n-1)
```

Then testing:

```
cos_nestlist_inner([0.0, 1.0], 3)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

```
cos_nestlist_inner([0.0], 4)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

**Challenge 3b**: hence, write a function called `cos_nestlist2` that takes as its arguments an iniitial value `x0` and a non-negative integer `n`, and returns a list of the iterates

$$x_0, x_1, x_2, \ldots, x_n.$$

```
def cos_nestlist2(x0, n):
    return cos_nestlist_inner([x0], n)
```

Then testing:

```
cos_nestlist2(0.0, 4)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

Another way of dealing with the same problem, which is a little neater, uses default values.

**Challenge 3c**: write a single function called `cos_nestlist3` that takes as its arguments an iniitial value `x0`, a non-negative integer `n` and an optional keyword-only argument `list_so_far` with default value `[]`. With the default value of `list_so_far` it should return a list of the iterates

$$x_0, x_1, x_2, \ldots, x_n.$$

```
def cos_nestlist3(x0, n, *, list_so_far=[]):
    from math import cos

    # base case
    if n==0:
        return list_so_far + [x0]

    # recursion step
    else:
        return \
    cos_nestlist3(cos(x0), n-1, list_so_far=list_so_far + [x0])
```

Then testing:

```
cos_nestlist3(0.0, 4)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

(Notice how we've used the line continuation symbol, \, to deal with what would have been an overlong line of code.)

## 6.5 Output versus side-effects

As you've seen, Python functions and methods can do two things: return output, or have some kind of *side-effect*, such as changing the order of a list, or placing a graphic in the notebook. How do you make your own functions have side-effects?

There are at least two ways: have your function call side-effect functions or methods, or work with **global variables**. The former works especially well if the variable you want to change is a *list* (or any other kind of *mutable* data), and if it's one of the *arguments* of your function. The latter works for anything, mutable or immutable, and is appropriate if your variable isn't an argument, but is instead assigned a value *inside* your function.

### 6.5.1 Changing the value of a list argument

Suppose you want to write your own function for reversing the order of a list. So far, your experience would lead you to write a function that takes a list as its input and returns a list as its output:

**Challenge 1**: write a function called `my_reverse1` which takes a list as its input and returns that list, reversed, as its output.

```
def my_reverse1(lis):
    """Outputs elements of lis in reverse order"""

    new_list = []

    # iterate backwards through input list, appending to output list
    for n in range(1,len(lis)+1):
        new_list.append(lis[-n])

    return new_list
```

Now, if we type

```
my_list = [1, 2, 4, 1, 0]
my_reverse1(my_list)
```

the output is

```
[0, 1, 4, 2, 1]
```

Now, this is fine as far as it goes, but a real Python function, or method, would probably not do that; instead, it would produce no output, but would simply reorder the items in the input list.

> **Challenge 2**: write a function called `my_reverse2` which takes a list as its input and reverses that list, returning no output.

Here's a first go. Warning: this isn't going to work.

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""

    new_list = []

    # iterate backwards through input list, appending to sorted list
    for n in range(1,len(lis)+1):
        new_list.append(lis[-n])

    # set input list equal to sorted list
    lis = new_list
```

This looks good, but it fails: look.

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

```
[1, 2, 4, 1, 0]
```

Why hasn't this worked? The point of failure is the line `lis = new_list`. We want this to reset the value of the **global** variable represented by `lis`, but a line beginning "`lis =`" never does that; instead, it assigns a value to a separate **local** variable with the same name. So our solution, whatever it is, can't involve a line beginning "`lis =`".

One approach is to empty `lis` as we go, so that by the end of the loop, it's simply the list `[]`; we can then use **augmented assignment**, which does work globally in the way we want. There's a list method called pop that does exactly what we need: `lis.pop(4)` outputs element number 4 from `lis`, and removes it at the same time.

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""

    new_list = []

    N = len(lis)

    # backwards through input list, removing elements
    # and appending to sorted list
    for n in range(1,len(lis)+1):
        new_list.append(lis.pop(N-n))

    # lis is now empty; use augmented assignment to make it
    # the same as the sorted list
    lis += new_list
```

This now works:

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

```
[0, 1, 4, 2, 1]
```

We can do even better, though; how about this for a neat implementation?

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""

    N = len(lis)

    # iterate backwards through input list, removing elements
    # and appending to input list
    for n in range(1,len(lis)+1):
        lis.append(lis.pop(N-n))
```

Then

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

```
[0, 1, 4, 2, 1]
```

### 6.5.2 Global variables

By default, the variables we use within functions are **local** to those functions: they have no effect on any variables you may be using outside the function that may happen to have the same name. Nearly always, that's exactly what we want, but on occasion we may wish our function to have the **side-effect** of changing the value of a global variable. When that's the case, we need to declare it `global`.

To see how this works, compare the following two examples. Here's a function that returns a list of "Hello World!" strings, and at the same time has the utterly random effect of assigning the value 3 to the variable `x`.

```
def hw_list(n):
    x = 3
    return ["Hello World!" for r in range(n)]
```

What would you expect to see if you typed

```
x = 7
print(hw_list(5))
print(x)
```

You might think the answer would be

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
3
```

because calling the function should, as a side-effect, give `x` the value 3. But no: what you see is

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
7
```

That's because the variable called `x` inside the function isn't the same as the variable `x` outside the function: it's purely **local**, meaning that we can give it the value 3 all night; the one outside will still have the value 7.

If we want to change that, we need to explicitly declare `x` **global**:

```
def hw_list(n):
    global x
    x = 3
    return ["Hello World!" for r in range(n)]
```

Then

```
x = 7
print(hw_list(5))
print(x)
```

does indeed give us

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
3
```

But that's a slightly silly example. How about doing something useful with this stuff?

**Challenge 3**: write and test a function called `cos_and_count` which takes a list as its input a number (float or int) and returns its cosine, at the same time increasing the value of the global variable `evaluation_count` by 1.

Use your function to compare the efficiency of two implementations of a `cos_fixedpoint` function, one of which uses the condition

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

and the other of which uses

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

First the `cos_and_count` function:

```
def cos_and_count(x):
    global evaluation_count
    from math import cos
    evaluation_count += 1
    return cos(x)
```

Then

```
evaluation_count = 0
print(cos_and_count(0.5))
print(cos_and_count(0.5))
print(evaluation_count)
```

```
0.8775825618903728
0.8775825618903728
2
```

Now the tests:

```
def cos_fixedpoint_inefficient(x0, tolerance):
    x = x0
    while abs(x - cos_and_count(x)) > tolerance:
        x = cos_and_count(x)
    return x

def cos_fixedpoint_efficient(x0, tolerance):
    oldx = x0
    newx = cos_and_count(oldx)
    while abs(oldx - newx) > tolerance:
        oldx = newx
        newx = cos_and_count(oldx)
    return newx
```

Then

```
evaluation_count = 0
print(cos_fixedpoint_inefficient(1.0, 0.00001))
print(evaluation_count)
```

```
0.7390893414033927
57
```

```
evaluation_count = 0
print(cos_fixedpoint_efficient(1.0, 0.00001))
print(evaluation_count)
```

```
0.7390822985224023
29
```

The efficient version is indeed more efficient!