Practical sheet 1: Overview, Basics and Functions

Words below in *green italic* are important technical terms in Python programming. By the end of this session you should have some idea of what they all mean. When reviewing your understanding at the end of the sheet, ask tutors to explain any that are not clear to you.

Section 1a: What is programming?

An algorithm is a sequence of instructions or a set of rules that are followed to complete a task or calculation. For example, you probably know how to find the factorial of a positive whole number or how to find the median of a sample of data. Both of those are algorithms.

Programming is the process of taking an algorithm and encoding it so that it can be executed by a computer. Basically, we are telling the computer what to do.

In order to tell the computer what to do we have to use a special language. There are many different programming languages we might use. The language we use in this module is called Python and we use the dialect known as Python3.

BEFORE YOU START:

- Create a folder (for example `programming1` in `Documents` under `This PC`) where you will keep your work for this module. You might make a sub-folder for each week.
- Start Thonny.

Thonny is a simple IDE (integrated development environment) for Python3. It supports writing and running Python programs and also has facilities to help with debugging — finding out why your program doesn't work as you intended.

When you first open Thonny, you will see a window with two panes: one has a tab labelled `<Untitled>` and the other a tab labelled `Shell`. You can write a program in the `<Untitled>` tab and then save it to a file to keep. You can have more than one program open at a time and can create a new program at any time by selecting New from the File menu. The `Shell` is where the program actually runs and where you will see any *output* from a program.

Section 1b: A program that has already been written for you

In Ultra for this module, go to *Python files* in the *Resources* folder and scroll down to find `plot_function.py`. Right click on `plot_function.py` and save a copy to the folder where you plan to keep your programs. Now use Open from the File menu in Thonny to open this program.

For now, do not worry about trying to understand the program. Just follow the instructions.

You can run the program using Run current script from the Run menu (or just click the button that looks like a "play video" button or alternatively hit the key labelled `F5` on your keyboard). After a few seconds, a window should appear showing a (not very good) graph of $y = \sin x$ for $0 \leq x \leq 4\pi$. Close the window and look at the program. The key to what's happening here is the line

```
plot_function(my_fun, 0, 4*pi, 20)
```

Comment out this line, by putting a single # at the start of the line. Then run the program again. Python then ignores this line in the program. You'll see that no window appears. Take out the # again. Comments are used by programmers to explain their code to themselves and others and to disable code which might be wanted again later. You'll see some other lines in `plot_function.py` which are also comments and which are intended to explain a little of what is going on.

So what does this line do? It tells python to plot the *function* named `my_fun` using 20 values of x equally spaced between 0 and 4π.

And what is the *function* `my_fun`? Look in the code for

```
def my_fun(x):
    y = sin(x)
    return y
```

which defines a "Python function" called `my_fun` which calculates $\sin x$. So we are in fact plotting $y = \sin x$.

Task 1.1. The plot is not very good because there are only 20 values of x being used and the plot is drawn by "joining the dots". Try increasing the number of values of x. It should improve the plot. If you don't like the + symbols, remove the `+` from the string `'-+'` on the line which starts with `plt.plot` and see what happens. What happens if you remove the `-` instead or if you add the letter `r` after the `+`?

Task 1.2. Try also changing the range of values of x and verify that it works as would expect.

You are not expected to understand the rest of the program at this stage. By the end of the term, you should understand it fully and be able to create your own programs which do this and many more complicated tasks.

There is one further point which is worth making now: A programming task can often be made easier by relying on something which has already been done. In this example we did this in the first two lines which *import* Python functions etc which have been provided by other people. The first line imports a Python *variable* containing a good numerical approximation to π and a Python function which calculates the mathematical sine function to high accuracy. The second line imports support for creating plots.

Section 1c: Starting from basics: variables, assignment, types, operators

Task 1.3. Please type the following program into the pane labelled <Untitled> and use Save from the File menu to save the result to a file in the folder you are using for programming. You might wonder why we ask you to type rather than using "copy and paste" from the PDF in Ultra. Unfortunately, the latter often does not work very well and produces nonsense or Python that looks OK but does not work.

```
m = 3
n = 2*m
print(m, n)
m = m + 2
print(m, n)
```

Before you run the program, think about what it might be doing. The thing you need to know is that a single equal sign `=` does *assignment*: on the lhs (left hand side) is the name of a *variable* and on the rhs (right hand side) is something to be stored in the variable. A variable is basically a "box with a name" into which we can put things. So writing `x=3` in Python is much the same as writing "Let x be 3" on a piece of paper as part of a calculation.

Once you have tried to guess what the program might do, run it in the same way as you ran the first program. You should see some *output* in the `Shell` pane. Does it make sense?

- The first line stores the value 3 in the variable `m`.
- The second line first computes the *expression* `2*m` which is in fact `2*3` because `m` contains 3. In Python, `*` means multiplication and so `2*3` is 6 which is then stored in the variable `n`.
- The third line prints the values stored in the variables `m` and `n` and you should see a line in the `Shell` which reads
  `3 6`
- The fourth line might seem odd mathematically ($m = m + 2$ is false for any real number m) but in Python it makes makes perfect sense: it tells Python to take the current value of `m`, add on 2, and then store the result back in `m`.
- The fifth line does the same thing as the third line but the value of `m` has now changed to be 5 instead of 3.

Info 1.4. In Python, the arithmetic *operator*s `+`, `-` have their usual meaning, `*` means multiplication and `x**y` means x^y (you can't use `^` for powers in Python although you can in some other languages). Python respects BIDMAS/BODMAS (known formally as *operator precedence*). For example, `1+2*3` means `1+(2*3)` not `(1+2)*3`.

Task 1.5. Does `x*y**3` mean xy^3 or $(xy)^3$? Calculate both by hand using $x = 2$ and $y = 3$. Then check what Python does by writing a small program. Ask a tutor to look at what you did and discuss with them if you're not sure about it.

Task 1.6. Without typing in the following short program, decide for yourself what values will be stored in `m` and `n` at the end.

```
m = 5
n = 3*m
m = 2*m + n
n = 5*n - m
```

Now type it in a new file and save and run it. There will be no output but you can see the values of `m` and `n` in two ways:

- Type **`print(m)`** in the `Shell` and hit the `Enter` key on your keyboard. Then do the same to see the value of `n`.
- Select Variables in the View menu and a new pane labelled `Variables` will open, showing you the values of `m` and `n`.

Remark 1.7. When you Run your program, Thonny starts with a "blank slate", i.e. it forgets anything that was done in a previous program or in a previous run of the current program.

Task 1.8. Let us continue with more numerical computations. In a new file, try

```
m = 2
x = 3.14
y = m*(x + 2)
print(m, x, y)
print(type(m), type(x), type(y))
```

An important aspect of mathematics is the study of abstract objects: numbers, shapes, functions, sets, etc. It is very important to know at all times what type of object one is dealing with: integers, real, complex or more fancy numbers, vectors, matrices, etc.

The same is true in programming. To help us, in Python `type(x)` tells us what type of object a variable or *expression* `x` is. From the last line of the program, we learn that the value stored in the variable `m` is of type `int`, which is Python's internal representation of integers, while `x` and `y` both hold values of type `float`, which is Python's *approximation* of real numbers.

Task 1.9. Unlike real numbers in mathematics, `float` has limited *precision* (roughly 16 significant digits), which can lead to surprising results if one is not expecting it. Mathematically, we know that $(x + y) - y = x$ if x and y are real numbers. In Python, this may not work exactly.

In a new file, try:

```
x = 0.0002
print(x)
t = x + 1e9
print(t)
t = t - 1e9
print(t)
```

and you will see that at the end `t` does not exactly equal `x`. This is because Python is unable to store most real numbers exactly. If you don't know what `1e9` means, **ask a tutor**.

This is an important topic that will get more attention in a later practical. It is easy to compute a number using a computer. It is also easy to compute the wrong number if one is not careful, especially when using `float`s.

Task 1.10. The double equal sign `==` is a *comparison operator*. It computes the lhs ("left hand side") and the rhs, and checks if they are equal. In a new file, try:

```
m = 2
print( m==2 )
print( m==3 )
print( m*m == m+m )
print( 2*m > m+3 )
print(m)
```

and look at the output in the `Shell`. Does it make sense? Note that the spaces inside the brackets are not necessary but they may make the program easier to read. Note also that the value stored in `m` has not changed at the end of the program: the comparisons do not change the value.

Task 1.11. Go back and read Task 1.3 again. Are you clear about the difference between `=`

and ==?

Task 1.12. Now add to your file:

```
print(type( m==2 ))
print(type( m==3 ))
print(type(m*m == m+m))
```

In mathematics, $m^2 = m + m$ is a *mathematical statement* which is either true or false depending on the value of m. Math question: for what values of m is statement be true, i.e. for what values of m is m^2 equal to $2m$?

In Python, `m*m` and `m+m` are *expression*s of type **`int`** since the value in `m` is an **`int`**. Now `m==2`, `m==3` and `m*m == m+m` are also expressions, but of type **`bool`**, whose only possible values are `True` and `False`.

So: a mathematical statement becomes an *expression* in programming; it has type **`bool`** (named after George Boole) in Python.

Info 1.13. `==` is a *comparison* operator. Python's other *comparison* operators include `<`, `<=`, `>`, `>=` and `!=`, the last one meaning $\neq$. Arithmetic operators have higher precedence than comparisons, i.e. `3**2 < 2**3` means `(3**2) < (2**3)` which is `False` (check it in the `Shell`). When in doubt about precedence, use brackets. However, many unnecessary brackets can make programs harder to read.

Info 1.14. An important concept to understand is *type promotion*: if we do `+`, `-` or `*` with two **`int`**s, the result is an **`int`** but if either *operand* is a **`float`**, the result is a **`float`**. Thus `2+1` is an **`int`** but `2+1.0` is a **`float`**. Try it in the `Shell` or write a small program.

However, `/` behaves differently. The result is always a **`float`** even if the it is mathematically an integer. Try `4/2` in the `Shell`.

The result of `**` is a **`float`** unless both operands are **`int`** and the second operand is non-negative.

Exercise 1.15. With `m` and `y` as in Task 1.8, predict the types and values of the following *before* typing them in:

```
m = 3*m + 7
p = (m == 2*m + 17)
w = y - 0.28
j = m/13
u = m**2
r = m**-2
```

Now add these lines and suitable print statements to your file from Task 1.8 and run the resulting program to check your answers.

Exercise 1.16. Make a copy of `plot_function.py` with a different name. Then modify it so that it plots $y = (\sin^2 x)\cos x$. In order to do this, you need to know that we cannot square a function in python but we can square a number. In other words `sin**2(3)` will produce an error but `sin(3)**2` will first find $\sin(3)$ and will then square the results. You will also need to add something to the first line of `plot_function.py`.

On paper, find all the turning points of $y = (\sin^2 x)\cos x$ for $0 \leq x < 2\pi$. Does your

answer agree with the plot?

How many maxima are there for $0 \leq x < 2\pi$? You can read this easily from your plot.

Section 1d: Introduction to functions

A Python *function* has quite a lot in common with a mathematical function. Essentially it is a recipe for turning inputs into output. You have already used and modified a function in the first part of the practical.

Now we are going to take a closer look at what is involved in creating functions. Here's a function to compute the quadratic $q(x) = 2 + 3x + 4x^2$:

```
def q(x):
    val = 2 + 3*x + 4*x**2
    return val
```

Type it in. Note that the second and third lines of the function need to be *indented* as shown: Thonny should do this for you automatically. Also, it is good practice to add a blank line after the definition of a function.

We could then calculate $q(5)$ and show the answer by adding the line `print(q(5))` to our program. Write down on a piece of paper what you think the output should be. Then try it in Thonny. Check that the answer is correct.

Breaking the function down:

- First line does two important things:
  - it says that the name of the function is `q`
  - it says that the function has a single input which will be called `x` when doing calculations inside the function.
- Second line computes $q(x)$ and stores the result in the variable `val`
- Third line says that the output of the function is whatever is stored in the variable `val` at the moment when the **`return`** *statement* is executed.

A key additional thing to understand is that the variables `x` and `val` exist only inside the function: they are there to help define the recipe. Outside the function, the only thing that is visible is the function itself (named `q`).

Task 1.17.

Make a copy of `plotfunction.py`. Remove the lines which define `my_fun(x)` and replace them by the lines above which define `q(x)`. What do you have to change now so that the program draws the graph of $q(x)$ for x between -4 and 4?

Task 1.18.

On paper, compute the distance between the points $(1, 3)$ and $(4, -1)$ in the plane.

Type in this program

```
def dist2(x1,y1,x2,y2):
    dx = x2 - x1
    dy = y2 - y1
    return (dx**2 + dy**2)**0.5
```

and Run it. Nothing seems to happen! This is because the program defines a function but does not use it to do any calculation.

Add a blank line and then:

```
print(dist2(1, 3, 4, -1))
```

and run again. Did you get the same answer as when you did it on paper?

Add another line to the program to compute the distance between the points $(2, 1)$ and $(7, 13)$ and run the program again. Does it give the right answer?

Task 1.19. Download `perimeter4.py` from Ultra and open it. The function `perimeter4(x1,y1,x2,y2,x3,y3,x4,y4)` calculates the length of the perimeter of the quadrilateral in the plane with vertices (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4). It makes use of the function `dist2` to compute the length of each edge.

Compute by hand the length of the perimeter of the quadrilateral with vertices $(0, 0)$, $(3, 0)$, $(3, 1)$, $(0, 5)$.

What do you have to add to `perimeter4.py` to get python to compute the length of the perimeter of this quadrilateral? Hint: the answer looks like `print(perimeter4(....))` where you need figure what to write in place of the `....` part.

Exercise 1.20. Write a function `area3(x1,y1,x2,y2,x3,y3)` that returns the area of a triangle in $\mathbb{R}^2$ (the plane) whose vertices are (x_1, y_1), (x_2, y_2) and (x_3, y_3). Use "Heron's formula". If you don't know Heron's formula, look it up using your favourite search engine.

Your function should **call** `dist2` from Task 1.18 to compute the distance between two vertices. You can of course write a version of `area3` that does its own calculations of distances but what we want here is to see you **call** `dist2` three times and then do a calculation using the values returned from the 3 calls. If you're not sure what it means to **call a function**, either review the videos provided for preparation for week 1 practicals or ask to discuss with a tutor.

Choose a triangle for which you can fairly easily compute the area and write down its vertices on a piece of paper. Then calculate the area of the triangle by hand using Heron's formula. Finally, test your code by adding a line to your program to compute the area of your chosen triangle. Does your code give the right answer?

At this point, try to get the attention of one of the tutors so that you can discuss your progress with them and they can answer any questions you may have.

Section 1e: Understanding the mathematics behind the exercises

Many of the exercises this term are about mathematics. Sometimes it may be possible for you to write the program without fully understanding the maths. But you should try to understand it; that's a big part of being a mathematician.

So, if you used Heron's formula in Exercise 1.20, you should make sure to learn why it works if you don't already know. Searching the internet will lead you to several proofs.

Section 1f: Extra exercises to challenge you

If you get to the end of a practical sheet and have time to spare, you'll find more exercises on Ultra (see *Extra exercises* in the *Resources* folder). These extra questions are not a core part of the module: you don't need to do them as part of the module but they may add to what you learn, especially if you are already a confident programmer.

Section 1g: Reviewing your learning

Go through this sheet again and review these concepts: variable, type (`int`, `float`, `bool`), operators, assignment. In the next pracitical session, you'll be tested on some of the things you learned today as part of the first quiz and your tutors may ask you some questions.

Quizzes

This week sees the opening of the **first quiz** that counts towards the module mark. You should read the announcment on Ultra about the quizzes. Your tutors will tell you more at the start of your class including when you should attempt the quiz.

Practical sheet 2: Repetition, conditionals and a bit more

Section 2a: Repetition using a while loop

Task 2.1. The following program illustrates one way to repeat execution of a task. It uses a *while* loop to print out the numbers from 1 to 10 (the *indentation* of `print(i)` and `i = i+1` is important):

```
i=1
while i<=10:
    print(i)
    i = i+1
```

Try putting the program in Thonny and running it.

What happens if you change the initial value assigned to `i` or the number to which `i` is compared on the second line?

What happens if you change `i` to `j` everywhere that the variable `i` appears? What if you make the change on one line but not on others?

What happens if you move the `print(i)` below `i = i+1`?

Task 2.2. On Ultra, you will find `loop2.py`. Download it and open in Thonny. Run it. It prints some output in the `Shell`.

The indentation matters. Can you see how the indentation affects the result? Try moving `print('Finished')` to the right and see how it changes the output.

Move `print('Finished')` back to the left and also move `print(j**2)` and `j = j+1` to the left and see what that does. The program never stops. Why?

Do you understand what is happening here? If you're confident, carry on. Otherwise, ask one of the tutors to discuss it with you. If is very important to understand both what is going on in a while loop and what indentation means in Python.

Info 2.3. The indented lines

```
    print(j)
    print(j**2)
    j = j+1
```

in the original version of `loop2.py` are called the *body* of the loop and are the lines of code that are being executed repeatedly by the *while* statement. Technically they are known as a *block* in Python.

You have seen *block*s before. When defining a function using *def*, the indented lines of code are the body of the function which defines what the function actually does. Again, those lines are a *block*. The most important thing to understand is that the lines

```
while j<11:
    print(j)
    print(j**2)
    j = j+1
```

need to be considered as a single unit. The first line makes no sense on its own: it needs at least one indented line to follow, specifying the action(s) to be repeated. In the same way, the three indented lines make no sense on their own: there needs to be an unindented line before them which ends in a `:`, in this case a *while* statement.

Exercise 2.4. Write a program to print out the cubes of the integers from 10 to 20 (inclusive).

Section 2b: Updating a value during a loop

Task 2.5. Instead of printing out values, it is often more useful to do a calculation in the body. Here is an example which computes $10!$ (the factorial of 10) and then prints it:

```
val = 1
i = 2
while i<11:
    val = val*i
    i = i+1
print(val)
```

Try it. Are you sure you know how it works? Why does the first line say `val=1`? What would go wrong if it said `val=0` or `val=2`? What would go wrong if we deleted that line? If in doubt, try it and see what happens.

What happens if you add an extra (indented) line `print(val)` below `val = val*i`?

Exercise 2.6. Write a program to calculate the sum of the first 5 natural numbers: $\sum_{i=1}^{5} i$. Check that your program gives the correct answer (calculate in your head or by hand).

Section 2c: Loops inside functions

Task 2.7. We can put a loop inside a function. For example:

```
def factorial(n):
    val = 1
    i = 2
    while i<=n:
        val = val * i
        i = i+1
    return val
```

modifies the code from Task 2.5 and defines a function to compute the factorial of a natural number. Note the two levels of indentation: the block defining the function and within it the while loop block to be repeated.

Check that this works, i.e. try adding lines of the form `print(factorial(x))` for several values of x for which you know the correct answer.

Compare the code here to Task 2.5. Do you understand how the specific program for computing $10!$ has been turned into a function for computing $n!$?

Exercise 2.8. Write and test a function `sumints(n)` to compute the sum of the first n positive integers using a *while* loop. Hint: start from your answer to Exercise 2.6.

Section 2d: Two new operators

Info 2.9. The operators `//` and `%` implement division with remainder. $x\%y$ finds the remainder when x is divided by y and $x//y$ is the integer number of times y divides into x. More detail is provided later in Info 2.35.

Remark 2.10. To test simple code, we can type it in to the `Shell` directly – recall that the shell is the bottom pane in Thonny.

Task 2.11. Check that you understand what `//` and `%` do by typing numerical examples into the `Shell` in Thonny. For example, first predict what the result will be for `41%8` and `41//8` and then check by running them in the `Shell`.

Task 2.12. If x is an integer, what do we learn about x by typing $x\%2$ in Python? Hint: this divides integers into two groups which have special names in English. If you're not sure about the answer, discuss with a tutor.

Section 2e: A little more about functions

Task 2.13. On Ultra, you will find `functiondetails.py`. Download it. You will see that it defines three functions: `good`, `bad` and `bad2`. There are also four lines of code at the end which:

(i) call `good`, storing the result in `z`;
(ii) print the value stored in `z`;
(iii) calculate the square-root of `z` and store the result in `w`;
(iv) print `w`.

Before running the program, try to predict what will happen. Then run it and see if your were right. You should see that the function does what you would expect and that the result is stored in `z` for further use.

Task 2.14. Now replace `z = good(5)` by `z = bad(5)` and run it again. You will see that the square of 5 is printed in the `Shell` but that nothing is stored in `z` and so there is an error when trying to compute the value to store in `w`.

Task 2.15. Replace `z = bad(5)` by `z = bad2(5)` and run it again. This time it doesn't even print the square of 5 in the `Shell`.

Remark 2.16. What's the point? If you want to use the result of a function in subsequent computations, you must include a *return* statement to tell Python the result of the function.

Section 2f: Conditionals

Task 2.17. Very often when programming, we want to do something different depending on the value stored in a variable: this is called conditional execution. Here is a very simple example:

```
a = 2
if a%2 == 0:
    print('a is even')
else:
    print('a is odd')
```

Try it. What happens if you change the integer assigned to `a` on the first line?

Delete the last two lines and again run using several values of `a`. How did the output change?

Info 2.18. The two indented *call*s to the `print` function in Task 2.17 are both blocks. Conditionals and loops can be used in blocks of other loops and conditionals and in the body of a function.

Exercise 2.19. Write a function `isodd(n)` which returns 1 when `n` is an odd integer and returns 0 when `n` is an even integer.

Exercise 2.20. Write a function `isodd2(n)` which returns `True` when `n` is an odd integer and returns `False` when `n` is an even integer. Remember that Python has a type `bool` for which the only possible values are `True` and `False`.

Task 2.21. Test your `isodd2` function using the following code:

```
i=1
while i<101:
    if isodd2(i):
        print(i)
    i = i+1
```

which should print out the odd integers between 1 and 100.

Note that the indentation varies. Why is `i=i+1` indented less than `print(i)`?

Section 2g: Putting it all together

You may not realise it yet, but you now know enough to do many different computations in Python. The hard part is working out how to achieve what you want by combining the four key concepts: *variables*, *loops*, *conditional execution* and *functions*.

Exercise 2.22. Write a function `mymax(a,b)` which returns the maximum of the numbers `a` and `b`. Example: `mymax(5,12)` should return 12 and `mymax(2,-3)` should return 2. Your function should not make use of any other Python functions.

Exercise 2.23. Write a function `nroots(a,b,c)` that returns the number of real roots of $ax^2 + bx + c = 0$ (obviously only 0, 1 or 2 are possible). Hint: check the discriminant.

Exercise 2.24. The greatest common divisor (gcd) of two positive integers can be computed by the Euclidean algorithm. The Euclidean algorithm was discussed in detail in the "What programming is" video which can be found on Ultra in Section 1 (see also the Wikipedia

page). If you have any doubts about the Euclidean algorithm and have not watched the video, please watch it now.

A terse statement of the algorithm is: keep subtracting the smaller number from the larger number until the two numbers are equal. If this is unclear, or you are not sure about to turn this into something you can program, please review the video or the PDF of the presentation used in the video (available on Ultra).

Compute *by hand* $\gcd(456, 123)$ using this algorithm.

Write a Python function `gcd(p,q)` which returns the gcd of positive integers p and q. Test your function using `print(gcd(12,30))` and `print(gcd(456,123))`.

Section 2h: Beyond the basics

Sections 2a to 2g cover the basic knowledge required from this practical. However, if you reach here with time to spare, you should carry on to the next page to "Sheet 2: beyond the basics". The material there is core to this module. However, it's fine to defer it for now if you are not ready or feel a bit overwhelmed. We will expect you to cover it later when you are ready.

Sheet 2: Beyond the basics

Do not progress to this material until you have completed everything on the main part of the sheet (sections 2a to 2g) and resolved any areas of doubt by discussing with a tutor.

Section 2i: More exercises

Exercise 2.25. The least common multiple (lcm) of two positive integers p and q can be computed by $\text{lcm}(p,q) = pq/\gcd(p,q)$. Write a Python function `lcm(p,q)` that returns the lcm of positive integers p and q.

Exercise 2.26. If your answer to Exercise 2.25 did not use your `gcd(p,q)` function from Exercise 2.24, write a new short version of `lcm(p,q)` which does use it.

Task 2.27. [suggested by Prof. Peyerimhoff] Given an integer $n > 1$, one obtains the "hailstone sequence" starting from n as follows: replace n by $n/2$ if n is even and by $3n+1$ if n is odd, and repeat until $n = 1$. Example: from $n = 3$ we obtain $10, 5, 16, 8, 4, 2$ and 1.

We can do this by hand for small integers, but the computation soon becomes tedious. It is however quite easy using a computer. Download `collatz.py` from Ultra and check that it gives the right sequence if we start with $n = 7$. Check also the answer for $n = 3$.

Modify `collatz.py` so that, instead of printing out the sequence of numbers as it goes along, it prints out at the end how many steps it took to reach 1. For example, starting from $n = 3$ the answer is 7 steps. You will need to introduce an extra variable to count the steps.

Remark 2.28. In the definition of the "hailstone sequence", we said "until we reach 1", but do we always reach 1 for *every* starting value $n \in \mathbb{N}$? This is a famous open problem in mathematics known as the Collatz or $3n+1$ conjecture. Obviously this has been checked by computer for very many n (cf. the Wikipedia article), but this is not a proof!

Exercise 2.29. Write a function `hailstones(n)` which returns the number of steps it takes to reach 1 starting from positive integer `n`. Hint: you can start from the code in `collatz.py` and turn it into a function but you'll need to add an extra variable to count the number of steps.

Exercise 2.30. Write a loop which prints the number of steps required to reach 1 for all positive integers up to and including 1000.

Section 2j: Repetition using a for loop

Task 2.31. The following code illustrates another way to repeat execution of a task in Python. It uses a *for* loop to do the same thing as the *while* loop in Task 2.1

```
for i in range(1, 11):
    print(i)
```

The key thing to understand that the result of `range(1,11)` is effectively the following sequence of natural numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. To see that this is the case, type `list(range(1,11))` in the Shell.

The *for* loop repeatedly executes `print(i)` with `i` set in turn to be each of the numbers in

the sequence returned by `range(1,11)`.

Exercise 2.32. Redo Exercise 2.6 using a *for* loop instead of *while* loop.

Remark 2.33. The main benefit of a *for* loop is that there is slightly less for the programmer to type and it may be easier to read. However, any task that can be performed by *for* loop can also be performed by a *while* loop.

Exercise 2.34. Redo Exercises 2.8 and 2.30 using a *for* loop.

Section 2k: More about the two new operators

Info 2.35. To be precise, when `y>0` the calculations for `x//y` and `x%y` find the integer m and real r such that $x = my + r$ and $0 \leq r < y$. Then m is the result of `x//y` and r is the result of `x%y`. Both operations result in an *int* if both `x` and `y` are *int*s, otherwise the result is a *float*. If you want to know both m and r, the function `divmod(x,y)` returns the pair of values (m, r).

You may find it instructive to try what happens in simple cases when x is not an integer or when x is negative.

Section 2l: Extra exercises to challenge you

As last week, there are extra exercises on Ultra (in the "Extra exercises" section) intended to stretch those who are already confident programmers.

Please read the following before starting work on this sheet.

Please complete as far as the end of section 2g "Putting it all together" before proceeding to this sheet. You can delay the "Beyond the basics" part of sheet 2 until later if you wish.

A typographical **warning**: In many places on the practical sheets, it is important to distinguish the digit 1 ("one") from the letter l ("lowercase ell"). In the font used for Python code on these practical sheets, the digit looks like `1` and the letter looks like `l`. In particular, there are many places where `lst` is used as a variable name. This variable name begins with the letter not the digit.

Organisational **advice**: It's often OK to put the answer to more than one task/exercise in the same file. Just bear in mind the following:

- If you have an error in an earlier task, it may prevent the one you are working on from running properly.
- If you have a call to the print function in an earlier task, the output may confuse you when you are trying do to the new task.
- If you assign global variables in an earlier task, they may affect a later task.

If you get really stuck/puzzled with a particular task, it may pay to make a new clean file for it so as to avoid these issues. If you want to re-use a function from an earlier task/exercise, just copy and paste it into the new file.

Practical sheet 3: Lists and some more about functions and ranges

Section 3a: A closer look at functions

Example 3.1. So far, in examples, the *return* value from a function has nearly always been stored in a variable and then the value of that variable has been returned. However, it is perfectly OK to *return* an expression which is evaluated before returning the result. For example:

```
def double(x):
    val = 2*x
    return val
```

is the same as

```
def double(x):
    return 2*x
```

Task 3.2. Recall from lecture 2 that the *arguments* (input variables) to a function are *local* to the function, i.e. they do not exist outside the function. The same applies to any variables which are assigned in the *body* of a function (the body is the block of lines following the **`def`** line). If a variable of the same name exists outside the function, it is a completely different variable.

In the following code, determine the values of `x` and `y` at the end by hand or in your head.

Write down your answer and then type in the code, adding suitable print statements at the end, to check.

```
def fbad(x):
    x = 3*x + 1
    return x

x = 2
y = 3
y = fbad(y)
```

If your answer was wrong and you do not understand why, ask a tutor to discuss it with you.

Example 3.3. You already know that the `return` statement defines the output of a function. What you may not yet know is that the the function *exit*s when the *return* statement is executed, even if we are still in the middle of the function body. For example

```
def afun(stuff):
    return 1
    val = 2*stuff
    return val
```

is equivalent to

```
def afun(stuff):
    return 1
```

because the function exits, returning 1, before it reaches the remaining two lines of the body.

Task 3.4. As another example, here is a solution to exercise 2.22

```
def mymax(x,y):
    if (x > y):
        return x
    return y
```

Type the function into a file. You can then test it by adding print statements:

```
print(mymax(5, 12))
print(mymax(-12, 5))
print(mymax(7.5, 4))
print(mymax(1.3, 1.3))
```

If you are not sure how it works try adding `print('Hello')` before `return y` (indented by the same amount as `return y`). You should see that the program only prints `Hello` in those cases where `x` is less than or equal to `y`.

Exercise 3.5. Modify `isprime` from `primes.py` (on Ultra in "Python functions used in section 2 videos") so that it does not need the variable `prime` of type *bool*. Hint: the function can immediately return `False` if it finds a factor of `n`.

Info 3.6. If a function is missing a `return` statement, or if a `return` statement is never reached (e.g., due to a conditional), the function returns the special object `None`.

Section 3b: Lists

If you have not yet watched the "Lists" video from the section 3 content on Ultra, you may find that you need to watch it at some point while working through this section.

Info 3.7. A *list* is a finite ordered sequence of objects which can be of different types.

One uses square brackets to create a list from individually specified elements with commas between the elements. For example, we can write

```
alst = [1, 2, 0.577, False, 'Hello']
```

To create an empty list, one can write `[]` or `list()`. To access an individual element of a list, one uses square brackets, for example, `alst[0]` gives `1` and `alst[3]` is `False`.

Important: the first element of a list has index 0 not 1. Consequently, if a list has n elements, the last element of the list has index $n-1$ not n.

Task 3.8. Download load and review `lists_example.py` available from Ultra (in "Python functions used in section 3 videos"). It may help to *step through the code* one piece at at time. See instructions in the Thonny section on Ultra for how to step through a program.

3b.1 Calculations for lists of numbers

Task 3.9. A function argument can be a list. Here we want to create a function that computes the sum of a list of numbers.

1. What value should the function return for the list `[5]` (a list containing one element)? Write down your answer.
2. What value should the function return for the list `[5,8,10,12]`? Write down your answer.
3. Now think about the algorithm to use to do this. The key is to find a way to do the task sequentially, processing one element of the list at a time.
4. On Ultra, in "Python functions used in section 3 practicals", you will find `lsum.py` which defines a Python function `lsum` to compute the sum of a list of numbers. Check that the function works for the cases specified in items 1 and 2. Were the same values you wrote down returned by `lsum`?
5. Add some extra tests of your own, for example using a list of 3 numbers instead of 4.

Exercise 3.10. Now consider the problem of computing the product of the numbers in a list (product means multiplying the numbers instead of adding). Repeat steps 1 to 3 in Task 3.9 for the product. Make a copy of `lsum.py`, rename the function to `lprod` and modify it so that it returns the product of the numbers in the list. If you're not sure about this, look back at the difference between the code provided for Task 2.5 and your answer for Exercise 2.6. Finally, check that your function gives the right answers for the lists used in steps 1 and 2 and then test further using some other examples.

Exercise 3.11. Now consider finding the maximum from a given list of numbers.

Repeat steps 1 and 2 in Task 3.9 but this time finding the maximum instead of the sum.

Now proceed to step 3. Try to work out what algorithm to use. First, consider how you would find the maximum number in a long list of numbers printed on paper. Imagine that the list is

so long that you might have to take a comfort break and you might forget what was the largest number you had seen before you took a break. What would you write down on a piece of paper before taking the break? Now try to confirm your understanding by writing down an algorithm which works sequentially through the list, processing one number at a time.

Turn your algorithm into a function `lmax` that returns the maximum element of a list of numbers. Hint: this is not really hugely different from Task 3.9 and Exercise 3.10. You do have to tweak a couple of aspects but a lot of the structure can be the same.

Finally, check that your function gives the right answers for the lists used in steps 1 and 2 and then test further using some other examples.

Now write a function `lmin`, which returns the minimum element of a list of numbers.

Exercise 3.12. Write functions `lmean` and `lstdev` (each taking a single list argument) returning respectively the mean and standard deviation of the numbers in the list. If you don't remember the formula for the standard deviation, look it up on the web.

Choose a couple of lists with which to test your functions. Hint: a good choice is a list of numbers for which it is reasonably easy to compute the mean and standard deviation by hand.

3b.2 Functions that return lists

Example 3.13. So far, all our functions return a single number (or `None`). Sometimes, it is convenient to return several values:

```
def pow123(x):
    return [x,x**2,x**3]

lst = pow123(3)
print(lst)
```

The type of the value returned by `pow123` is `list`.

Exercise 3.14. Return to exercise 2.23 and write a function `roots(a,b,c)` which returns a list containing the unique real roots of the quadratic. The reason for returning a list is that the number of real roots depends on the the coefficients of the quadratic. When there are no real roots, you should return an empty list.

Info 3.15. `lst1 + lst2` does not do numerical addition. It makes a single lisst by joining two lists end to end (also called concatenation). Try it using two lists of your choice.

Info 3.16. A common *idiom* is to create an empty list and then add items to it one at at time using list concatentation. For example, `mylst = []` followed by
`mylst = mylst + [2]`, ...

Task 3.17. For example, `firstnats.py` on Ultra defines a function that returns a list of the first `n` natural numbers (starting from 1).

Test it (by adding suitable unindented print statements) for some positive values of `n`.

Look at the code. What do you think will happen if `n` is `0`? What will happen if `n` is a negative integer? Check your answers by adding more print statements.

What will happen if `n` is not an integer but is a float in between two natural integers? Again, check your answer by adding a print statement.

Exercise 3.18. Make a copy of `firstnats.py` and change the name of the function to `primes_up_to`.

Now modify it so that it returns a list of all the prime numbers less than or equal to `n`. This will be much easier if you make use of `isprime` from earlier: copy the `isprime` function into the start of this file.

If you're not sure what to do to make this function work: think about adding a suitable **`if`** statement inside the loop.

3b.3 Functions that make lists from lists

Exercise 3.19. In previous tasks and exercises, you have learned two important tricks:

- how to work through the elements of a list doing some calculation (as in `lsum`, `lmax`);
- how to make a new list starting from an empty list by adding elements one at a time (as in `firstnats`).

Challenge: combine the two tricks. Write a function `lsqr` that takes a list of numbers as its argument and returns a list, of the same length, containing the squares of the numbers.

Exercise 3.20. Use `lsqr` and `lsum` to write a shorter version of `lstdev`.

Exercise 3.21. Write a function `primes_in(lst)` which takes a list of natural numbers as its argument and which returns a new list containg only those which are prime. Again, using `isprime` from earlier will help.

If you're not clear what is being asked: `primes_in(firstnats(n))` should return the same as `primes_up_to(n)`. You can use this to test your function.

3b.4 Representing vectors using lists

Task 3.22. One convenient use of lists is to represent vectors in Python. On Ultra, you will find `dist2v.py` which re-visits task 1.18. This version of `dist2` has two arguments, one for each point. Each argument should be a list of 2 numbers, the coordinates of the points.

Test that this function gives the right answer for the points $(1, 0)$ and $(-2, 4)$.

Exercise 3.23. Write and test `distn(p1,p2)` which computes the distance between two points in $\mathbb{R}^n$ (n-dimensional space). You may assume that both arguments are length n lists. You'll need to use **`len`** to find out what n is from the arguments.

Section 3c: Beyond the basics

Sections 3.1 to 3b cover the basic knowledge required from this practical. However, if you reach here with time to spare, you should carry on to the next page to "Sheet 3: beyond the basics". The material there is core to this module. It's fine to defer it for now if you are not ready but we will expect you to cover it later when you are ready.

Sheet 3: Beyond the basics

Before carrying on with this material, make sure that:

- **you have understood everything up to the end of section 3b;**
- **you have worked through "Sheet 2: beyond the basics" from last week.**

Section 3d: More about functions

Task 3.24. Return to Task 3.2. What changes if you replace `3*x + 1` by `3*x + y`? The key to understanding this is that `y` is a *global* variable: it exists outside the function. A function can use the value of a *global* variable if there is no `local` variable with the same name, i.e. the name is not an argument of the function and is not assigned to in the function.

Task 3.25. Add the following to your code from Task 3.2 and guess the output before you run it:

```
print(type(fbad))
print(type(fbad(1.5)))
```

Confused? In mathematics, $\sin$ is a *function* — it takes in a number and spits out another number; for example $\sin(3)$ is a real number (roughly 0.141). In our example, `fbad` is a Python *function*, i.e. an object that takes in something and spits out something else . `fbad(1)` is the output of the function when the input is 1; here both input and output are *int*s.

Section 3e: More about lists

You are already familiar with the **`for`** statement which loops over each element of list.

Info 3.26. Python provides some built-in functions and operations which takes lists as arguments. You have already seen **`len`**`(lst)` which returns the length of list `lst`.

Built-in functions **`sum`**, **`max`** and **`min`** make `lsum`, `lmax` and `lmin` redundant.

`sorted``(lst)` returns a sorted *copy* of `lst`.

All of these leave the list(s) unchanged. A complete list of list functions and operations can be found in the official Python3 documentation.

Exercise 3.27. Write and test a function `lmedian` returning the median of a list of numbers. Hint: this is much easier if you first make sorted copy of the list. You may also need to refresh your knowledge of the `//` and `%` operators from sheet 2.

Info 3.28. There are also operations which *modify* lists *in place*. These use a different *syntax* in which the name of the *list* variable is followed by `.` and the name of the operation.

If `lst` is a list: `lst.append(x)` appends `x` to the end of `lst`, `lst.sort()` sorts `lst` *in place*, `lst.reverse()` reverses the elements in `lst`, `lst.pop()` returns `lst[-1]` and removes it from `lst`; for more, see the Python3 documentation.

Exercise 3.29. Return to the hailstone sequence of task 2.27 and subsequent exercises and write a function `hailstone_seq(n)` which returns a list containing the steps in the sequence starting from `n`. Example: The return from `hailstones_seq(3)` should be `[10, 5, 16, 8, 4, 2, 1]`.

Use `hailstones_seq` to make a new very short version of `hailstones(n)` from 2.29.

Example 3.30. The `for` statement can loop over arbitrary lists. For example, using the function from Exercise 3.29:

```
hs3 = hailstones_seq(3)
for x in hs3:
    print(x)
```

will print out, one at a time, the steps in the hailstone sequence starting from 3. Try it.

Exercise 3.31. Rewrite `lsum.py` to use a for loop instead of a while loop. You should be able to reduce the body of `lsum` to 4 lines if you use `for x in lst:`

Example 3.32. The following prints out the truth table (see the Analysis I module) for "(not p) and q":

```
for p in [True,False]:
    for q in [True,False]:
        print(p, q, not p and q)
```

Note that `not p and q` means `(not p) and q` because `not` has higher *precedence* than `and`. When in doubt, use brackets.

Section 3f: The `range` function

Info 3.33. The `range` function is an easy way to create a list of equally spaced integers.

Technically, the function returns something that often behaves like a *list* but is not actually a list. To turn it into a list, one needs to *cast* it. Try this in the `Shell`:

```
print(range(11))
print(list(range(11))
```

You will see that the result is the sequence of integers from 0 to 10. Note that it stops before it gets to 11.

In general, the behaviour of `range` depends on how many arguments are provided: `range(n)` is a shorthand for `range(0,n)`, and `range(m,n)` is a shorthand for `range(m,n,1)`. In its full version `m, n, step` is the sequence `m`, `m+step`, `m+2*step`, ... and the sequence stops before it gets as far as `n`.

Note that all three arguments to `range` must be integers. If we want to produce equally spaced values which are not integers, we need to do a little more work. See Exercise 3.36 below.

Task 3.34. Carefully read the information just provided and then predict the list that results from each of the following: `range(3,7)`, `range(4,1,-1)` and `range(6,0,-2)`.

Then check your answer in the `Shell`. Don't forget to add `print(list(...))`.

Task 3.35. Before executing them, write down the outputs of `list(range(4,-1))` and `list(4, 5, 2)` then try them in the `Shell`.

Exercise 3.36. Write a function `grid(a, b, npts)` which returns a list containing `npts` equally spaced numbers (of type *float*) which start at `a` and end at `b`.

Hint: each number in the list is of the form $a + i\Delta$ where the integer i runs from 0 to `npt-1` and you have to figure out what Δ is.

Example: `grid(-1,1,5)` should return `[-1.0, -0.5, 0.0, 0.5, 1.0]`

Please read the following before starting work on this sheet.

Please complete as far as the end of section 3b "Lists" before proceeding to this sheet. You can delay the "Beyond the basics" parts of sheets 2 and 3 until later if you wish.

Practical sheet 4: Practising your skills

Before starting work on this sheet, make sure that you have completed the three previous sheets (you may defer the "Beyond the basics" sections if you like). The exercises that follow are intended to consolidate the skills learned there.

For each of the following exercises, before trying to write Python code, do the following:

1. Make sure that you understand the basic mathematical idea. If you're not sure, ask a tutor to discuss it with you.
2. Once you understand the idea, check your understanding using the example(s) given in the question: you should be able to predict the answer(s).
3. Finally, before writing any code, think about the algorithm required — the computational process involved. You could try to write the algorithm down on paper.

Exercise 4.1. This appeared already on "Sheet 3: Beyond the basics" so if you did it there, you can ignore it here. **You do not need any of the "Beyond the basics" material on either sheet 2 or sheet 3 to do this exercise**.

Write and test a function `grid(a, b, npts)` that returns a list containing `npts` equally spaced numbers (of type *float*) which start at `a` and end at `b`.

Example: `grid(-1.1, 2.5, 5)` should return `[-1.1, -0.2, 0.7, 1.6, 2.5]`. Note that the approximations involved for calculations using *float*s mean that your answer may differ very slightly.

Hint: each number in the list is of the form $a + i\Delta$ where i runs from `0` to `npts-1` and you shold figure out what the formula for Δ is in terms of `a`, `b` and `npts` before writing your function.

Exercise 4.2. Write and test `divisors(n)` which computes and returns a list of all the positive divisors (also known as factors) of a positive integer n. For example `divisors(6)` should return `[1,2,3,6]`. Please test your function using more than one example.

If you're not sure how to test if one number is a divisor of another, take another look at `isprime` (used for practical sheet 3 and also in a section 2 video). If you're still not sure, discuss with a tutor. The tutors are there for that purpose.

Exercise 4.3. We call a positive integer *perfect* if it is equal to the sum of its divisors other than itself. For example, $6 = 1 + 2 + 3$ is perfect but $8 \neq 1 + 2 + 4 = 7$ and is not perfect.

Write and test a function `is_perfect(n)` which returns `True` if the positive integer n is *perfect* and `False` if it is not. For example `is_perfect(6)` should return `True` whereas `is_perfect(8)` should return `False`. If you think about it, you should be able to do some of the work by calling your `divisors` function from the previous exercise.

How many perfect numbers can you find using your function? Hint: try the infinite loop approach used in the original `sequence.py` from lecture 1.

If you find perfect numbers interesting, see the extra exercises for more about them and their relation to Mersenne primes.

Exercise 4.4.

This exercise is about computing the partial means of a list of numbers. The result is a new list which has the same number of elements as the original list. The kth partial mean is mean of the first k elements of the original list.

Expressing this using notation, let $x_1, \ldots, x_n$ be the original list. Then the partial means are $m_1, \ldots, m_n$ where $m_1 = x_1$, $m_2 = (x_1 + x_2)/2$, $m_3 = (x_1 + x_2 + x_3)/3$ and the formula for a general element is $m_k = (x_1 + x_2 + \cdots + x_k)/k$.

For example, for the list `[1,5,1.5,12.5]`, the partial means are `[1, 3, 2.5, 5]`. Before proceeding, ensure that you understand and can reproduce this calculation by hand.

Now think about how you would do this on paper if you were given a list of 100 numbers. There are different approaches, some of which are much more efficient than others.

Write and test a function `parmean(alst)` which takes as argument a list of numbers (which may be floats or integers) and which returns a new list which contains the partial means. Remember that a Python list of length `n` has indices `0`, . . . , `n-1` whereas the mathematical description above uses $1, \ldots, n$.

Check your function using the example above.

Now download `parmeantest.py` from Ultra. Put a copy of your function into the file in the place indicated and then run it.

If it takes less then a second to run, your function is very efficient and you almost certainly used `lst.append` from "Sheet 3: Beyond the basics". If it takes more than a minute, it is inefficient for a different reason and you should discuss the reason why with one of the tutors.

Exercise 4.5. The *fundamental theorem of arithmetic* states that every positive integer has a unique representation as a product of prime numbers. For example $140 = 2 \times 2 \times 5 \times 7$ and this is unique apart from the ordering of the prime factors.

Write and test a function `prime_factors(n)` which computes a returns a list containing the *unique prime factorisation* of the number n, i.e the numbers in the list should all be prime numbers and their product should equal n. The list should be in numerical order.

For example `prime_factors(140)` should return `[2,2,5,7]`.

Hint: first find and remove all factors of 2 from the original number, then all factors of 3 etc. Note that you don't need to restrict your check to prime numbers: for example, there will be no factors of 4 if you have removed all factors of 2 and no factors of 6 if you have removed all factors of 2 and 3.

Section 4a: Beyond the basics

Exercises 4.1 to 4.5 cover the basic knowledge required for now. However, if you have time, carry on to the next page to "Sheet 4: beyond the basics". The material there is core to the module. It's OK to defer it for now if you are not ready but we will expect you to cover it later.

Sheet 4: Beyond the basics

Section 4b: More about Python lists

This material is optional *for now*. You should work through *all* the material on practical sheets 1 to 3, including the "Beyond the basics" parts before working on this. Watching the section 4 video "More about lists (and assignment)" may also be a good idea.

Info 4.6. As discussed in "Sheet 3: Beyond the basics" and the section 4 video "More about lists (and assignment)", a *list* can be changed *in place* and examples were given of the `list.method(...)` syntax. Another way to change a list in place is to write something like

```
lst[2] = 'abc'
```

This is a different form of assignment: instead of assigning to variable, we are assigning to a particular location in a list, replacing whatever was there before. In this particular example, the 3rd element (remember that Python indexes from `0`) of `lst` is changed to be `'abc'`.

Task 4.7. As discussed in the section 4 videos, lists and numbers differ fundamentally. Run this:

```
s = [1,2,3]
t = s
s[1] = 6
s.append(4)
print(s, t)
s = s + [5]
print(s, t)
```

Confused? Let us go through this carefully, using the idea from the "Assignment revisited" video that assignment to a variable in Python really means *attaching a label to the object with the name of the variable on the label*.

- On the first line, the list object `[1,2,3]` is first constructed and then, in the language of the "Assignment revisted" video, the label `s` is attached to the list.
- The second line simply attaches the label `t` to the same object as `s`.
- So when the list is changed *in place* on the next two lines, the variables `s` and `t` both remain labels on that changed object.
- On the penultimate line, the rhs is first constructed from the list with label `s`, giving a new list object `[1,6,3,4,5]`. The label `s` is then removed from the original list and attached to the new list. The label `t` remains attached to the original list.
- So we end up with two lists: `[1,6,3,4]` labelled `t` and `[1,6,3,4,5]` labelled `s`.

Task 4.8. Task 4.7 begs the question of what we should do when we want create a new list which is a copy of an existing list. The standard way to copy a list is to write `t = s.copy()` or `t = s[:]`. Make the following amendment to the code for task 4.7 and see what happens:

```
s = [1,2,3]
t = s[:]
s[1] = 6
s.append(4)
print(s, t)
```

Now `t` is a copy of `s` and so does not change when `s` is changed in place.

Task 4.9. The `lst[:]` way of copying in the previous task is an example of list *slicing*. Run the following and check the value of each variable at the end:

```
s = [5,7,9,11,12,13,17]
u = s[:2]
v = s[2:5]
w = s[5:]
```

A `:` used in a list *index* indicates a *slice*. The result is a new list containing consecutive elements of the original list. If `s` is a list, then `s[a:b]` contains the elements of `s` in order, starting `s[a]` and ending with `s[b-1]`

One can omit `a` or `b` or both: `s[:b]` means `s[0:b]`; `s[a:]` means `s[a:len(s)]` and `s[:]` means `s[0:len(s)]`. Either can be negative which means that it should be subtracted from `len(s)`, i.e. `s[3:-1]` means `s[3:len(u)-1]`.

Section 4c: Mutability and copying

Remark 4.10. However, the elements of lists are themselves like variables: the elements of a list are like labels on objects (using the new metaphor for assignment introduced in the section 4 video "More about lists (and assignment)"). Consequently, when a list is copied, an extra label is tied on each object rather than copying objects. That this is true has little consequence unless an element of a list is *mutable*, for example is a list. The following code illustrates the potential issue:

```
s = [1]
t = [2,s]    # t contains a new label on [1]
u = t[:]     # copies t: copies the label.
# At this point there are three labels on [1].
# One is s and the others are elements of t and u.
s.append(3)
print(t, u) # t changed and u also changed!
```

If this behaviour is unwanted, we can avoid it by making a *deep copy*. When we deep copy a list, we also make copies of any lists inside it.

```
from copy import deepcopy
s = [1]
t = [2,s]
u = deepcopy(t)
s.append(3)
print(t, u) # t changed but u did not change
```

Task 4.11. Why all this? There are two types of objects in Python: *mutable* and *immutable*. Numbers (`int`, `float`, etc.), booleans and strings (below) are immutable, while lists are mutable. Mutable objects can change once they are created, but immutable objects cannot change: there is an important conceptual difference between

```
s = [1,2]
s.append(3)
```

and

```
s = [1,2]
s = s + [3]
```

even though they give the same result (or seem to anyway!).

The function `id()` gives the object "identity" — `x` and `y` point to the same object if and only if `id(x)==id(y)`. Now do `print(id(s))` after each line above (in both cases). You will see that `id(s)` does not change when we do `s.append(3)` which changes the existing list in place, but does change when we do `s = s + [3]` which creates a new list.

Remark 4.12. The distinction between the two cases in task 4.11 does not exist when `s` is a number and we add a number to `s`. There is no equivalent for numbers of `s.apppend`. For those of you who know about `+=`: `s += 3` does not change the number `s` in place, it is just a shorthand for `s = s + 3`.

Task 4.13. Unlike numbers, lists passed as arguments can be changed by functions:

```
t = [1,2,3]

def f(s):
    s.append(5)
    return s

print(f(t))   # f(t) is as expected
print(t)          # But t has also changed!
```

Run this and see what happens. You should observe that the list labelled `t` changed.

It is usually undesirable that a function changes its argument in this way. Modify the function so that it returns the same result but does not change its argument.

Exercise 4.14. Revisit exercise 3.19. Does your `lsqr` function modify the argument? Test this by

```
lst  = [1,2,3]
print(lst)
lst2 = lsqr(lst)
print(lst, lst2)
```

If your function does modify the argument, change your function so that it no longer modifies the argument. Test the result.

Section 4d: Extra material and exercises

The material up to this point on the sheet is all core although the "Beyond the basics" can be deferred. The material on the next page is not core and may be omitted altogether if you wish. As usual, there also extra exercises available on Ultra to keep you entertained.

Section 4e: Extras on functions*

The material in this section is not core and may be omitted.

Example 4.15. Python functions accept *default value*s for arguments:

```
def f(p,q=2):
    return p**q

print(f(3), f(3,3), f(3,4))
```

Note that all 'normal' arguments must come before any default value arguments; i.e. it is illegal to write `def f(x=0,y,z=0):`

When calling a function, it is also possible to specify the name of each argument and they can be passed in any preferred order:

```
def g(p,q,r):
    print(p, q, r)

g(1,5,7)
g(r=3, q=4, p=1)
```

Example 4.16. One can construct Python functions to take an *arbitrary* number of arguments, which appear as a tuple (an object similar to a list, but which is immutable) inside the function:

```
def ff(a,*t):
    print('ff():', a, t) # just to check
    return a*sum(t)

print(ff(5,1,2,3))
```

For more details on function arguments, see section 4.7 of the official Python3 tutorial at:
http://docs.python.org/3/tutorial/

Before you move on to the new material on this sheet, please make sure that you have succeeded with the following exercises from previous sheets: `area3`, `isodd2`, `gcd`, `lmax`, `roots`, `lsqr`, `primes_in`, `distn`, `grid`, `divisors` and `is_perfect`. These exercises develop and test key skills.

Practical sheet 5: More practice and some new skills: list comprehensions and formatting output

Section 5a: Built-in functions

Python has a number of built-in functions you have not yet seen but which can prove useful:

- `abs(x)` returns the absolute value (magnitude) of number `x`.
- `round(x, ndigits)` returns number `x` rounded to the specified number of digits after the decimal point. If `ndigits` is omitted, it rounds `x` to an integer.
- `max`, `min` and `sum` make `lmax`, `lmin` and `lsum` (from exercises on sheet 3) redundant.
- `sorted(alst)` returns a new list with the same elements as `alst` but sorted in increasing order.

Section 5b: The range function

If you have not already done so, please now work through section 3f which is part of "Sheet 3: beyond the basics". You can skip Exercise 3.36 as it was repeated as Exercise 4.1. There's no need to work through the rest of "Sheet 3: beyond the basics" yet.

Section 5c: List comprehensions

y = [f(x) for x in lst]

Before proceeding with this material, please watch the section 5 video about "List comprehensions" if you have not already done so.

Task 5.1. A *list comprehension* is a convenient shorthand for turning one list into another. Try:

```
lst1 = [0, 1, 2, 3, 4, 5, 6]
lst2 = [ i**3 for i in lst1 ]
print(lst2)
```

What's happening. The stuff between the square brackets (`[]`) on the second line defines a *list comprehension*. There are three key components:

1. `in lst1` says that a calculation is going to be done using each element of `lst1` in turn.
2. `for i` says that the variable `i` will hold each element of `lst1` in turn.
3. `i**3` says that the element of the new list corresponding to each element of the old list will obtained by calculating `i**3`

This means that we can recover the original list by

```
lst3 = [ x**(1/3) for x in lst2 ]
```

Check that this worked.

Note that we can use the **range** function in this context. So there was no need to create lst. Instead, we could just have typed:

```
lst2 = [ i**3 for i in range(7) ]
print(lst2)
```

Now try:

```
lst4 = [ lst2[j] for j in range(1, len(lst2)-1) ]
```

Do you understand this? Think first about what **range(1, len(lst2)-1)** will produce and then note that j takes each of those values and is then used to choose elements from lst2.

Modify the previous example to make a list comprehension to create a list lst5 which contains all the elements of lst except the first 3 and the last 2.

Example 5.2. Here's a more interesting example. A quadratic function $a_0 + a_1x + a_2x^2$ can be represented in Python by a list of its coefficients: a_0, a_1, a_2.

From Ultra, download quadeval.py which provides a Python function quadeval(a,x) that can be used to calculate any quadratic (coefficients in list a) at any point x.

Test the function using a quadratic of your choice and some suitable values of x. Do you understand how it works? If not, ask a tutor.

Use your grid function to make a list of 50 values of x running from -2 to 2. Then use a list comprehension and quadeval to make the corresponding list of values of y where $y = 3x^2 - 2x - 1$. How can you check if your answer is correct?

For more on the theme of polynomials in Python, see extra exercises 3.5.

Exercise 5.3. Use a list comprehension (no need to write a function) to make a list of the square-roots of the first 10 natural numbers.

Now write a function proot(n, p) which returns a list containing the p-th root (square and cube roots are $p = 2$ and $p = 3$ respectively) of each of the first n natural numbers. Use a list comprehension to do the work inside the function.

Exercise 5.4. Redo exercise 4.1 (the grid function), this time using a *list comprehension*.

Exercise 5.5. In the first practical, you imported pi and sin from the math module. In the same module is another function called factorial. Use this to write and test a function nchoosek(n,k) which computes the binomial coefficient $\binom{n}{k}$ for non-negative integers n and $k \leq n$. If you don't know what the binomial coefficient is, look it up on Wikipedia.

It can be shown mathematically that $\binom{n}{k}$ is an integer. What is the type of the value returned by your nchoosek? If it is not an integer, modify your code so that it is (look again at Info 2.9 if you are not sure how). Remember that / always produces a **float** but // does integer divison: x // y calculates the integer number of times y divides into x.

Define

$$R_n = \sum_{k=0}^{n} \binom{n}{k} \qquad \text{and} \qquad S_n = \sum_{k=0}^{n} k \binom{n}{k}$$

For example, $R_3 = \binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 1 + 3 + 3 + 1 = 8$.

Now write Python functions `R(n)` and `S(n)` using `nchoosek` and *list comprehension*s inside the functions to compute R_n and S_n for given n.

Use a further list comprehension and your function `R(n)` to make a list of R_n for $n = 1, \ldots, 10$ and then print out the list in the `Shell`. Do the same for S_n.

Mathematical aside: What do you think is the formula for R_n? Can you prove it? How about S_n? Please don't spend a lot of time on this during your practical class.

Task 5.6. A more complicated form of list comprehension includes a condition. This produces a list of the squares of those natural numbers less than 30 which are not divisble by 3. Try it in the `Shell`.

```
[ i**2 for i in range(30) if i%3 != 0 ]
```

Exercise 5.7. Use a list comprehension and your `is_perfect` function to produce a list of all the perfect numbers which are less than 10000.

Section 5d: Representing vectors by lists and matrices by lists of lists

The material in this section links to ideas you have seen in the Linear Algebra module. All of the exercises can be done without using list comprehensions but some will be easier, and the code will be neater, if list comprehensions are used.

We have already seen in exercise 3.23 that it can be convenient to represent a point in n-dimensional space, or a vector in $\mathbb{R}^n$, by a Python list containing n numbers.

Exercise 5.8. Write a Python function `zero_vector(n)` which returns the 0 vector in $\mathbb{R}^n$, i.e. it returns a list of length `n`, each element of which is 0.0. If you did not use a *list comprehension*, make a new version of your function which does use a list comprehension.

Exercise 5.9. Write `vadd(v1, v2)` to add two vectors in $\mathbb{R}^n$, `vdot(v1,v2)` to compute the dot product of vectors, and `vsmul(v, x)` to multiply a vector by a real number x. Test your functions!

Task 5.10. It can also be convenient to represent a matrix with m rows and n columns by a list of length m where each element of the list is a list of n numbers which represents a row of the matrix. For example, the 2×2 matrix $A = \begin{pmatrix} 2 & 3 \\ 5 & 1 \end{pmatrix}$ would be represented in Python by `A = [ [2, 3], [5, 1] ]`.

If I represent a matrix in this way, then row i of the matrix is `A[i-1]` and the entry in row i and column j of the matrix is `A[i-1][j-1]`. Verify that `A[1][0]` is the correct value.

Verify that `A[0][1] = 6` has the effect of changing the matrix to be $A = \begin{pmatrix} 2 & 6 \\ 5 & 1 \end{pmatrix}$.

Exercise 5.11. Write a Python function `det2(mat)` which computes the determinant of a

2×2 matrix `mat`. For example, to compute the determinant of `A` from task 5.10 one would type `det2(A)`. Verify that your function gives the correct answer.

Write and test `inv2(mat)` to compute the inverse of a 2×2 matrix `mat`. You may assume that `mat` is such that the inverse exists.

Task 5.12. From Ultra, download `zeromatrix.py` which provides a Python function which returns a $m \times n$ matrix where all entries are 0. Rewrite the function to take advantage of `zero_vector` from exercise 5.8

Exercise 5.13. Write a function `identity_matrix(n)` which returns the $n \times n$ identity matrix, i.e. the matrix where the diagonal entries are all 1.0 and all other entries are 0.0.

Hint: first create a $n \times n$ matrix of zeroes and then change the diagonal entries to be 1.0. You are not expected to use a list comprehension to do this.

5d.1 Optional exercises on vectors and matrices

Later in the term, we will look at the `numpy` module which provides more efficient and easier to use ways of representing and working with vectors and matrices in Python. The following exercises are therefore optional: do them if you have the time and motivation to do so. Otherwise, jump on to section 5e

Exercise 5.14. Write a function `diagonal_matrix(dv)` which returns the $n \times n$ matrix where the diagonal entries are given by the vector `dv` of length n and all other entries are 0.0.

Simplify your answer to exercise 5.13 by using `diagonal_matrix`.

Exercise 5.15. Write a function `transpose(A)` to compute the transpose of an $m \times n$ matrix A, i.e. the result should be the $n \times m$ matrix which has in row i and column j the number which was in row j and column i of A.

Hint: one approach is to first create a matrix where all entries are 0.0 and then change the entries using loops. Another is to use nested list comprehensions. Try both methods.

Exercise 5.16. Write functions `matsum(aa,bb)` and `matmul(aa,bb)` that implement addition and multiplication of $n \times n$ matrices. Can you adapt them for non-square matrices?

Exercise 5.17. If you are happy all this, you may see that `vadd` and `vsmul` are most of what is needed in order to implement Gauss-Jordan elimination from Ch. 3 of the Linear Algebra notes. You need also to be able to swap two rows: write `swaprows(A, r1, r2)` which returns a copy of `A` with rows `r1` and `r2` swapped. Then write a Gauss-Jordan function.

Section 5e: Formatting output

Before proceeding with this material, please watch the section 5 video about "Strings" if you have not already done so.

Task 5.18. When outputting information from a program, one may want to embed the value of a variable in some text. This can often be done by passing multiple arguments to print. Try

```
from math import factorial
print('The factorial of', 4, 'is', factorial(4))
```

An alternative is to use the **format** *method* for strings which *substitute*s values passed as arguments for occurrences of {} . For example, the previous example can be rewritten as

```
s = 'The factorial of {} is {}'
print(s.format(4, factorial(4)))
```

and you should see that the output is the same either way.

Exercise 5.19. Write a program to print out the factorials of the first 10 natural numbers in the form shown in task 5.18.

Info 5.20. The real benefit of the **format** approach is that, if we want to control the *formatting*, we can specify what we want inside the {} following a colon. For example, {:5d} says that the argument is an **int** which should become a string of 5 characters with spaces used if needed to fill it out. Similarly {:9.5f} says that the argument is a **float** to be formatted as a string with 5 digits after the decimal point and taking 9 characters in total.

Exercise 5.21. Modify your answer to exercise 5.19 so that the numbers line up nicely (right-aligned) in both columns. Hint: use the information provided in Info 5.20 (numbers will line up if the numbers in a column are all printed to be the same width).

Task 5.22. Try the following:

```
from math import pi
fstr = '{:.4g} {:.4g} {:.4g}'
print(fstr.format(pi, 2**10, 2**100))
```

Try changing the :.4g to :.8g and run again. Do the same for :.3g. What do you think this formatting using g inside the {} asks Python to do?

Section 5f: Beyond the basics

The material in sections 5a to 5e cover the basic knowledge required for now. However, if you have time, carry on to the next page to "Sheet 5: beyond the basics". The material there is core to the module. As with previous "beyond the basics" sections, it's OK to defer it for now if you are not ready but we will expect you to cover it later.

eg string:

```
from math import pi
s = 'Python thinks that pi is {}'
s = s.format(pi)
print(s)
```

Python thinks that pi is 3.141592653 5

Sheet 5: Beyond the basics

Section 5g: More extras on Lists*

Task 5.23. A further useful list operation is *slicing*. Run the following and check the value of each variable at the end:

```
t = [5, 7, 9, 11, 12, 13]
r = t[:2]
v = t[2:4]
```

A `:` used in a list *index* indicates a *slice*. The result is a new list containing consecutive elements of the original list. If `t` is a list, then `t[a:b]` is the same as

```
[ t[i] for i in range(a, b) ]
```

One can omit `a` or `b` or both: `t[:b]` means `t[0:b]`; `t[a:]` means `t[a:len(t)]` and `t[:]` means `t[0:len(t)]`. Either can be negative which means that it should be subtracted from `len(t)`, i.e. `t[3:-1]` means `t[3:len(t)-1]`.

Section 5h: Inline conditionals

Python has a neat expression syntax for doing certain kinds of conditional execution in one line: cases where the value assigned to a variable depends on a condition.

For example, instead of writing

```
if x>0:
    a = 5
else:
    a = 2*x
```

we can write on one line

```
a = 5 if x>0 else 2*x
```

One place where this can be useful is inside a list comprehension. For example, if there was no `abs` function to use and `lst` is a list of numbers, we could write

```
[ x if x>=0 else -x for x in lst ]
```

to make a list of the absolute values of the numbers in `lst`.

Section 5i: Strings

Task 5.24. On several occasions, *string*s have appeared but they have not been introduced formally. A string is like a *list* of printable characters although there are important differences: (a) to make a string we use quotation marks (`'` or `"`) rather than `[]`; (b) strings are *immutable*.

Given `ss = 'Hello world!'`, some things work like for *list*s (try these in the `Shell`):

- indexing: `ss[5]` or `ss[-1]` and slicing: `ss[3:7]`

- concatenation: `'hello'+'world'`
- functions familiar from lists: `len(ss)`, `sorted(ss)` for example.

Immutability means that one cannot modify a *string* after creation: `ss[2] = 'k'` and `ss.sort()` both give errors.

Task 5.25. From Ultra, download `ispalindrome.py`. Test it using `'Hello'`, `'kayak'` and `'tattarrattat'` (it is a word, look it up).

Example 5.26. One special use of strings in Python is at the beginning of functions to provide documentation. Download `gcd.py` from Ultra and run it. Now type `help(gcd)` in the `Shell`. You will see that it prints out a description of what the function does. This comes from the second line of `gcd.py` which defines a *docstring* for the function.

This can be helpful when trying to work out what a function does. Try this in the shell:

```
from math import acos
help(acos)
```

Info 5.27. Since strings have many applications in programming, Python provides many string *method*s to help. If `ss` is a string, `ss.capitalize()`, `ss.lower()`, `ss.split(sep)`, `s.join(lst)`; `ss.find(sub)` and `ss.replace(old, new)`, . . . , do useful things.

To find out what they do: try `help(''.lower)` for example (or just Google it).

Info 5.28. For some more about *string formatting*, see `https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3` and if that's not enough, read the Python documentation.

Exercise 5.29. Write a function `is_in(x, lst)` which returns `True` if the list contains an item which is equal to `x` and which otherwise returns `False`. If you know about Python's `x in lst` syntax, try not to use it for this exercise. Test your function using

```
t1 = is_in(1, [1,4,3])
t2 = is_in(2, [1,4,3])
t3 = is_in(1.0, [1,4,3])
print(t1, t2, t3)
```

which should print `True False True`.

Remark 5.30. The function `is_in` is not needed. We can simply write `x in lst` and the result is either `True` or `False`. In the `Shell`, try

```
t1 = 1 in [1,4,3]
t2 = 2 in [1,4,3]
t3 = 1.0 in [1,4,3]
print(t1, t2, t3)
```

Practical sheet 6: Plotting

Info 6.1. In mathematics and science, it is often useful to draw graphs and this was the first thing that we did in practical 1 using `plot_function.py`.

Open `plot_function.py` again. You will see that we used the `matplotlib` library from which we *imported* `pyplot` (a special plotting object) by typing `import matplotlib.pyplot as plt`. The `as plt` allows us to type `plt` instead of having to type `matplotlib.pyplot` or `pyplot` every time we use it.

We also *imported* a mathematical constant and a mathematical function from the `math` library.

Note: it is a good practice to put *all* your *imports* at the top of the file.

Section 6a: Plotting functions

Task 6.2. Look again at `plot_function.py` and compare it to `plot_sin.py` (on Ultra) which does the same thing without creating functions. Both are correct but are written in different styles. The first has more structure which may help understanding or modifying it. The second is more direct. Each programmer has to decide what style works for them.

Remark 6.3. `matplotlib` is not a core part of Python. It was designed to resemble a well-established software called MATLAB. As a result, its syntax and conventions are not completely consistent with standard Python. The official `pyplot` documentation at `https://matplotlib.org/api/pyplot_summary.html` can be difficult to navigate and it may be easier to use Google to find out how to do something.

Task 6.4. It is easy to control various aspects of the plot. In `plot_function.py`, the *string argument* `'-+'` to `plt.plot` makes a $+$ symbol appear at each point determined by pairs of coordinates in `xlst` and `ylst` and connects points by lines. If you replace `+` by `o`

or `.` or remove `-` or `+`, the plot will change. Try this. A single character which is the first letter of a standard colour controls the colour of points and/or lines.

An alternative way to take control is by using *named arguments* to `plt.plot`. For example, one can add `markersize=12` or `linewidth=2` or `color='grey'`. For a full list of such arguments, consult the documentation for `matplotlib.pyplot.plot` (see above).

Take a few minutes to experiment with some of these possibilities.

Task 6.5. By using `plt.plot` more than once before the `plt.show()`, we can arrange for more than one function or set of data to be plotted on the same graph.

Modify `plot_function.py` or `plot_sin.py` to show $\sin^2 x$ and $\cos x$ on the same graph using different colours.

Task 6.6. `pyplot` makes it is easy to use logarithmic axis scales. Make a new program to plot e^{-t} as a function of $t \in (0, 5)$. What happens if you insert `plt.semilogy()` before the final `plt.show()`? If you're not sure what happened, look at the `pyplot` documentation (see above)

Add labels to the axes of your plot using `plt.xlabel` and `plt.ylabel`

Plot u^{10} as a function of $u \in (0, 1)$. What happens if you insert `plt.loglog()` before the final `plt.show()`? Why?

Section 6b: Graphical root finding

Task 6.7. In mathematics, a *root* of a function f is a value of x for which $f(x) = 0$.

Suppose we want to find the solution of $e^{-x} = \frac{1}{2}$. This is equivalent to finding the root of $f(x) = e^{-x} - \frac{1}{2}$.

Produce a plot of $f(x)$ for $x \in (0, 1)$. Use `plt.axhline(0)` to add a horizontal line at $y = 0$; make sure to do this before the `plt.show()` or it won't appear.

Now you can zoom in on the plot by first clicking on the *magnifying glass* symbol and then selecting a rectangle with the mouse: click, drag and release. You can repeat the process to zoom in further and further. This should enable you to narrow down where the root is. See if you can find the root to 7 significant digits. **Take care:** the points are connected by straight lines and so, in order to achieve this accuracy, you may need to change the range over which you plot the function and/or the number of points.

Check your approximation to the root at the end by evaluating f at your chosen value.

Section 6c: Plotting a sequence

Info 6.8. When plotting a sequence of values, it is not necessary to specify the values for the x-axis: `pyplot` will automatically generate x-values as $0, 1, 2, \ldots$.

You have already seen this used in `sequenceplot.py` used in the week 1 video "Why teach you programming?" and the week 2 video "Finding primes". The line `plt.plot(primes, '+')` plotted the sequence of prime numbers in `primes` on the y-axis and the x-axis values were automatically generated.

Exercise 6.9. Make a plot of the number of divisors of natural numbers up to some suitable

value of n. Can you say anything about how this behaves as n increases? If you are interested, have a look at the Wikipedia entry for "Divisor function" and in particular at the "Growth rate" section.

If you have done "Sheet 3: Beyond the basics", you could do the same thing for lengths of hailstone sequences (see exercise 3.29 and/or exercise 2.29) and for maxima of hailstone sequences.

Section 6d: Plotting data: scatter-plots and histograms

Task 6.10. On Ultra, you will find `students.py` which contains data on an ancient sample of students. It defines a list named `students` which contains three lists: a list of heights (cm), a list of weights (kg) and a list of gender (1 or 2). Either copy and paste this into a new program or save it and then do `from students import students`.

Now you can do `plt.plot(students[0], students[1], 'o')` to produce a scatter-plot of the weights versus of the heights. This works because entries in the same position of each of the three lists in `students` correspond to a single student.

Exercise 6.11. Produce a scatter-plot of weights versus heights where the points for males use a different colour to the points for females. *Hint:* use your accumulated Python programming knowledge to split the data into two datasets and then use `plt.plot` twice, once for each dataset. From the plot, decide whether gender=1 means female or male.

Task 6.12. We can also easily produce histograms:

```
plt.hist(students[0])
plt.xlabel('Height (cm)')
plt.show()
```

Various aspects of the histogram can be controlled. For details, consult the documentation on `matplotlib.pyplot.hist` (see above).

Section 6e: A dynamical system — Fibonacci numbers

Dynamical systems are an important class of mathematical model. A simple definition is that a dynamical system models the evolution of some quantity or phenomenon over time: for more information, see `https://mathinsight.org/dynamical_system_idea`

Exercise 6.13. The Fibonacci sequence (f_n) starts with $f_0 = 1$ and $f_1 = 1$. Subsequent terms $f_2, f_3, \ldots$ are generated by a *recurrence relation* which expresses the next term as a function of previous terms: $f_{n+1} = f_n + f_{n-1}$.

Write a function `fib(n)` which calculates the first n terms in the Fibonacci sequence. For example, the output of `fib(6)` should be `[1, 1, 2, 3, 5, 8]`.

Use your function to calculate and plot the first 100 Fibonacci numbers. You may find a logarithmic axis useful.

Make a suitable scatter-plot of the points (f_n, f_{n+1}) for $n = 0, 1, \ldots 99$. *Hint*: if you have a list containing $f_0, \ldots, f_{100}$ you can use list comprehensions to produce lists of x and y values for the plot (or if you did "Sheet 5: Beyond the basics" you could use *slice indexing*).

What do you see in the plot?

Plot the sequence f_{n+1}/f_n for $n = 1, 2, \ldots$. What happens?

Section 6f: Beyond the basics

The material in sections 6a to 6e cover the basic knowledge required for now. However, if you have time, carry on to the next page to "Sheet 6: beyond the basics". The material there is core to the module. As with previous "beyond the basics" sections, it's OK to defer it for now if you are not ready but we will expect you to cover it later.

Sheet 6: Beyond the basics

Section 6g: More about dynamical systems (and fractals)

Task 6.14. Now consider another dynamical system, this time for complex numbers z. Starting with $z_0 = 0$, one iterates

$$z \mapsto z^2 + c \tag{6.1}$$

which means that $z_{n+1} = z_n^2 + c$.

What happens to z_n as n increases depends on the value of the complex number c. For some c, $|z|$ is forever bounded under the iteration (6.1). For other values of c, $|z_n|$ tends to infinity as n increases. The *Mandelbrot set* is the set of all $c \in \mathbb{C}$ such that $|z|$ is forever bounded.

Of course, neither "forever" nor "bounded" are attainable by computers. Instead, we have to use arbitrary (but hopefully sensible) finite cutoffs. On Ultra, you will find `is_mandelbrot.py` which contains a single function which determines if a point is in the Mandelbrot set. The function assumes: (a) that $|z_j| > 500$ for some j is evidence that $|z_n|$ will tend to infinity; (b) that it is sufficient to check 1000 iterations.

One problem is that this code is not particularly efficient and so we cannot process large numbers of points in this way (a later practical will show an an improved version). What we can do is pick points at random in a rectangle in the complex plane and plot the ones which are in the Mandelbrot set.

The following imports a function `random()` which generates a (pseudo-)random number in the interval $(0, 1)$ and then uses it to produce a a list of 10 random numbers in $(0, 1)$:

```
from random import random
us = [ random() for i in range(10) ]
```

Modify the example so that it produces random numbers in the interval $(-1, 1)$. *Hint:* what linear (straight line) mapping will map the endpoints of $(0, 1)$ to the endpoints of the required range of values?

Modify it again so that it numbers produced are in $(-2, 1)$.

Python knows about complex numbers and can do arithmetic with them. The key thing to know is that `1j` means $\sqrt{-1}$, what mathematicians call i. So if you want to create the complex number $4 + 3i$ in Python, you type `4 + 3*1j`. It is much more efficient to store complex numbers in python in this way, rather than using lists as in the first assessment.

Now write a program which makes lists `xlst` and `ylst` by repeating the following 10000 times:

- it generates a random value x in $(-2, 1)$;
- it generates a random value y in $(-1, 1)$;
- it uses x and y to make a complex number $c = x + iy$;
- it uses `is_mandelbrot` to determine if c is in the Mandelbrot set.
- if c is in the Mandelbrot set, it adds x to `xlst` and y to `ylst`

Now `xlst` and `ylst` are lists containing, respectively, the real and imaginary parts of complex numbers which are in the Mandelbrot set.

You can now draw an approximate version of the Mandelbrot set by producing a scatter-plot of `xlst` and `ylst`. Experiment: change the ranges of x and y and the number of points.

The Mandelbrot set, or rather its boundary, is *fractal*: with a high-quality plot (not ours), you can try zooming into various parts of the boundary, and you will find smaller copies of the bigger parts. The high-resolution colour images you find on the web take longer to compute (usually using a *compiled* language such as C). Closely related is the *Julia set*, which is harder to compute/plot. If you'd like to learn more about the Mandlebrot set, read `https://plus.maths.org/content/unveiling-mandelbrot-set`.

Practical sheet 7: Floating-Point Computations

The first part of this sheet looks at errors in floating point calculations and the unfortunate possibility that they can effectively destroy a calculation if they accumulate in the wrong way.

By contrast, the rest of the sheet looks at some useful numerical methods where the errors do not generally cause problems.

Section 7a: Errors in floating point calculations

Task 7.1. The fact that a Python *float* uses a binary representation of numbers and has limited *magnitude* and *precision* has consequences which can be surprising.

Please watch the section 7 video about "Floating point numbers" if you have not already watched it.

Review `truncation.py` on Ultra used and discussed in detail in the video. It is probably best to use the "stepping through a program" feature of Thonny.

Here is another example for you to try. Note that the first line is a short-hand which assigns `0.1` to `x`, `0.2` to `y` and `0.3` to `z`.

```
x, y, z = 0.1, 0.2, 0.3
print(x)
print(y)
print(z)
print(x + y - z)
print((x + y) - z, x + (y - z))
print((x + y) * z == x*z + y*z)
```

What do we learn from these examples?

- that *float*s only approximate real numbers.
- that the error of approximation can depend on the order in which we do calculations.
- that it is not reliable to check if two *float*s are equal.

For the last of these reasons, a *float* should *never* be compared directly against `0.0`. Instead, one chooses a small $\varepsilon > 0$ and checks if a given expression is less than ε in absolute value:

```
eps = 1e-15
closeto0 = abs(x + y - z) < eps
print(closeto0)
```

Equivalently, if `x` and `y` are *floats*, one should check if `abs(x-y) < eps` instead of checking if `x == y`. Now what ε should one take? Well, it depends on the problem! Even worse, sometimes it only becomes clear after the fact ... This does not mean that we should despair, rather that we should be alert to the possibility that errors may arise if we are not careful. Too large a value of ε may mean an unnecessarily inaccurate answer to a calculation; too small a value may lead to an infinite loop or other undesirable behaviour.

Exercise 7.2. Here's a more complex and interesting example, relating to the iteration

$$x_{n+1} = \tfrac{14}{5}\, x_n + \tfrac{3}{5}\, x_{n-1}$$

Mathematical theory: on paper, show that, if we start with $x_0 = 1$ and $x_1 = -\frac{1}{5}$ and then apply the iteration, the result is that $x_n = (-\frac{1}{5})^n$ for all natural numbers n. *Hint: use the method of induction,* i.e. verify that the values given for x_0 and x_1 agree with this general formula and then show that, if we plug in $x_{n-1} = (-\frac{1}{5})^{n-1}$ and $x_n = (-\frac{1}{5})^n$ to the iteration, we obtain $x_{n+1} = (-\frac{1}{5})^{n+1}$.

Python: Write a program (no functions needed) to:

- Use the iteration to compute a list containing values $x_0, x_1, \ldots, x_{50}$.
- Compute a second list containing the numbers you would get by directly using the theoretical solution $x_n = (-\frac{1}{5})^n$.
- Compute a third list containing the difference between the corresponding values of x_n obtained using the two methods: this is the "error" of the iteration.
- Compute a fourth list containing the relative difference between the corresponding values of x_n obtained using the two methods, i.e. the difference divided by the "true value" in the second list. This is the "relative error" of the iteration.
- Print out, one row for each value of n, the four numbers obtained for each value of n.

You should see that your first Python list follows the theoretical pattern approximately and that the error gets larger while the true answer gets smaller with severe consequences for large n; the result is that the relative error explodes.

What causes the problem here is that each iteration computes the difference between two numbers which have the same sign and are quite close to each other. This is essentially the same issue as we saw in `truncation.py`.

Although these examples show that floating point calculations can go wrong, it is often the case that they do not and the rest of this sheet is about methods that do generally work. Designing numerical calculations to avoid accumulation of errors is an important topic in its own right but we do not have time to look at it further in this module.

Section 7b: Finding roots numerically using the bisection method

Task 7.3. For a continuous function $f(x)$, a reliable way to find a root, a solution of $f(x) = 0$, is the *bisection method*.

We need to start by finding (or guessing) two numbers a and b such that $f(a)$ and $f(b)$ have opposite sign, i.e. either $f(a) > 0$ and $f(b) < 0$ or $f(a) < 0$ and $f(b) > 0$. This means that there must be some x between a and b for which $f(x) = 0$, i.e. there is a root between a and b.

Suppose for now that $f(a) > 0$ and $f(b) < 0$. You can think about how to change what follows if it's the other way round (and you will need to do so for some examples which follow).

We can now repeat the following process until a is very close to b:

- set $m = (a + b)/2$, i.e. m is half-way between a and b.
- if $f(m) > 0$, change a to be equal to m; otherwise change b to be equal to m.

Why does this work? After each iteration it is still true that $f(a) > 0$ and $f(b) \leq 0$ and so the root lies between a and b. Moreover each iteration halves the distance between a and b.

Why not repeat until $a = b$? In fact there is no guarantee that this will definitely happen.

Because *float*s have finite precision, we may end up with a and b different from each other but with no available *float* values between a and b. Then when we compute m it will either equal a or equal b. This may lead to an infinite loop. To avoid this problem, we stop when $|a-b|$ is less than some specified threshold which may need to depend on the context.

For example, suppose that we want to find the solution of $\mathrm{e}^{-x} = \frac{1}{2}$. Letting $f(x) = \mathrm{e}^{-x} - \frac{1}{2}$, we note that $f(0) > 0$ and $f(1) < 0$, so we take these as our starting points: $a = 0$ and $b = 1$. Write a Python program to find a root of $f(x)$ using the method described above and which stops when $|a-b| < 10^{-15}$. Remember that the way to write 10^{-15} in Python is `1e-15`.

In your program, print out a, b, m and $f(m)$. To print them nicely, you might want to look back at the later parts of section 5e

Exercise 7.4. Pretend that Python cannot compute arbitrary powers of numbers but can only do the basic arithmetic operations: `+`, `-`, `*` and `/`.

Compute $\sqrt{2}$ to 15 significant figures by solving $x^2 = 2$ using the bisection method and note the number of iterations needed (starting with $a = 1$ and $b = 2$).

Apply the same approach to compute $2^{1/3}$ to 15 significant figures. You need to choose a function $f(x)$ which can be calculated without using `**` and which has $2^{1/3}$ as a root.

Exercise 7.5. Write a function `bisect(f,a,b,eps)` taking as arguments a function whose root is sought, two suitable starting points, and an ε. You may assume that the starting points are OK, i.e. that $f(a)f(b) < 0$. A good test for your function would be

```
from math import cos, pi
x = bisect(cos, 1, 2, 1e-15)
print(x-pi/2)
```

Exercise 7.6. Our examples so far all have solutions which are in fact easily computable directly using standard Python operators and functions: $2^{1/2}$, $2^{1/3}$ and $\log 2$. Now $f(x) = \mathrm{e}^{-x} - x = 0$ has no solution in closed form, yet it is easy to find it using bisection. Do so, and again note the number of iterations needed.

Section 7c: Finding roots numerically using the Newton-Raphson method

Exercise 7.7. The Babylonians computed $\sqrt{a}$ as follows. Starting with some $x_0 > 0$, let

$$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n} \tag{7.1}$$

and stop when $|x_{n+1} - x_n|$ is small enough.

Using this method, the Babylonians found that $\sqrt{2} \simeq 30547/21600$, which is correct to six decimal places.

Code this in Python (using *float*s not fractions) to approximate $\sqrt{2}$. You can start with $x_0 = 1$ or $x_0 = 2$ and stop when $|x_{n+1} - x_n| < 10^{-15}$.

How many iterations does your code need? Compare this to the number of iterations needed in exercise 7.4. You should find that the bisection method needed many more iterations.

Task 7.8. The iteration in the previous exercise is a particular form of the *Newton–Raphson method*: to find a solution of differentiable $f(x) = 0$, we start at some suitable x_0 and iterate

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.2}$$

until a desired accuracy is reached. For this to work, one must have $f'(x) \neq 0$ in some interval near the solution.

Verify that, for function $f(x) = x^2 - 2$, equation (7.2) gives equation (7.1).

Now apply the Newton-Raphson method to the function you used with the bisection method to approximate $2^{1/3}$:

- obtain $f'(x)$ and deduce the iteration implied by equation (7.2)
- implement the iteration in Python starting from $x = 1$.

Compare the number of iterations required[1] to the number required using the bisection method in exercise 7.4.

Remark 7.9. Newton–Raphson can be *very* efficient. However, unlike bisection, it can be temperamental: sometimes it can "jump away" even when starting quite close to the desired solution, e.g., try to find the smallest positive solution of $\cos x - 0.99 = 0$ starting with $x_0 = 0.001$. *Always* do a sanity check when using Newton–Raphson.

Exercise 7.10. Revisit exercise 7.6 and find the solution of $\mathrm{e}^{-x} = x$ using Newton-Raphson. Again compare the number of iterations needed for the bisection and Newton-Raphson methods.

Section 7d: Series expansions

Task 7.11. Recall that the *Taylor series* for $\sin x$ is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{7.3}$$

which converges absolutely (cf. later in Analysis) for all x. The formula for the coefficient of x^{2i-1} is $(-1)^{i-1}/(2i-1)!$ for positive integer i.

Write a function `sintn(x,n)` which calculates the sum of the first n terms in the expansion for specified x.

Plot `sintn(x,n)` for $x \in (-\pi, \pi)$ and several values of n (on the same plot with different colour lines). You should be able to see the convergence of the Taylor series to $\sin x$ as n increases.

[1] Impressed? Dr Wirosoetisno says that with the `clnum` library, it takes 7.5s to compute $2^{1/3}$ to a million digits on his desktop. A billion digits take a couple of days.

Section 7e: Numerical approximation of integrals

Task 7.12. Please watch the section 7 video about "The trapezoidal rule" if you have not already watched it.

In the video, the "trapezoidal rule" for approximating an integral was presented and the Python code used is presented in `trapezoidal.py` on Ultra.

Another method for approximating an integral is the "mid-point rule".

If you know what the mid-point and trapezoidal rules are, then skip to the next paragraph. Otherwise, Google "midpoint rule" and read the page on www.dummies.com. Then Google "trapezoidal rule" and read the Wikipedia page.

Now review `trapezoidal.py`.

Modify a copy of `trapezoidal.py` so that it implements the mid-point rule instead of the trapezoidal rule.

Exercise 7.13. Compare the performance of the mid-point and trapezoidal rules for approximating $\int_{-1}^{1} e^{2x}\,dx$. In other words, how does the accuracy depend on the number of "slices" for both rules. Which is better?

Section 7e: Numerical approximation of integrals

Task 7.12. Please watch the section 7 video about "The trapezoidal rule" if you have not already watched it.

In the video, the "trapezoidal rule" for approximating an integral was presented and the Python code used is presented in `trapezoidal.py` on Ultra.

Another method for approximating an integral is the "mid-point rule".

If you know what the mid-point and trapezoidal rules are, then skip to the next paragraph. Otherwise, Google "midpoint rule" and read the page on www.dummies.com. Then Google "trapezoidal rule" and read the Wikipedia page.

Now review `trapezoidal.py`.

Modify a copy of `trapezoidal.py` so that it implements the mid-point rule instead of the trapezoidal rule.

Exercise 7.13. Compare the performance of the mid-point and trapezoidal rules for approximating $\int_{-1}^{1} e^{2x}\,dx$. In other words, how does the accuracy depend on the number of "slices" for both rules. Which is better?

Practical sheet 8: Numerical computations with numpy

[Contributions from Professors Dorey, Smith and Yeates]

Please watch the short section 8 video about "Numerical python with numpy" if you have not already watched it.

A commonly used Python library is numpy (short for "numerical python"), which provides objects called "arrays". These are similar to the now familiar Python lists, but have many useful features for numerical computation. In all the following examples and exercises it is assumed that you have first imported numpy using the standard abbreviation np by

```
import numpy as np
```

and many also assume `import matplotlib.pyplot as plt`.

Task 8.1. For a terse start with numpy, see `some_numpy.py` on Ultra used in the "Numerical python with numpy" video.

Task 8.2. One way to create a numpy array is from a standard list:

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0])
print(x, type(x), x[1], x[2:4])
```

However, there are a number of other functions in the numpy library for creating arrays. For example, we can create arrays of zeros or ones with the eponymous functions:

```
print(np.zeros(3))
print(np.ones(5))
```

A particularly useful function is linspace, which creates an array of equally spaced values in some interval:

```
print(np.linspace(0.0, 2.0*np.pi, 10))
```

Note that the start and end points are both included in the result. This is simpler than the methods seen previously for creating a list of points for plotting which used the **range** function and a loop or list comprehension or your own grid function from exercises 3.36 and 4.1

Section 8a: Operations on numpy arrays

Task 8.3. A major advantage of numpy arrays over standard lists is that we can do mathematical operations (addition, multiplication etc.) on the whole array, rather than working on one element at a time. This often removes the need for loops or list comprehensions.

Arithmetic operations (+, −, *, /, **) work on two arrays of the same size. Try:

```
x = np.array([1.0, 2.0, 3.0, 4.0])
y = np.array([0.0, 1.0, 0.0, 1.0])
print(x + y)
print(x * y)
print(x**y-x/y)
```

Notice how the operations are just applied separately to the corresponding elements to make the new array. As a result, the arrays involved must all be the same size.

We can also very simply perform the same operation on every element of an array, for example multiplication by a constant, addition of a constant, etc. Try:

```
print(2*x, x+2, 2**x, x**2)
```

Section 8b: Functions that work for `numpy` arrays

Task 8.4. In addition, `numpy` provides its own versions of mathematical functions like $\sin$, $\exp$, etc. that work on arrays. This makes plotting of functions much easier than before. Try

```
x = np.linspace(0.0, 8*np.pi, 100)
plt.plot(x, np.sin(x))
plt.show()
```

Add `plt.gca().set_xlim(0,8*np.pi)` before `plt.show()`. What changed?

Section 8c: Applications of `numpy` arrays

Not all numerical calculation tasks are easier/faster using `numpy` arrays. For example, sequential calculations such as root-finding using bisection and Newton-Raphson methods don't really benefit much if at all. The big benefit is for tasks where you want to do the same thing with every element of an array and where the calculation for a single element does not depend on the outcomes for other elements.

Exercise 8.5. Plot $\sin^2 x$ and $\cos x$ on the same plot for $x \in [-2\pi, 2\pi]$ using `numpy` arrays rather than lists. How would you make the y axis cover the range from -3 to 2?

Task 8.6. It is also easy to plot parametrised curves of the form $\boldsymbol{x}(t) = (x(t), y(t))$. For example, what will the following produce? **Think first before running the code.**

```
t = np.linspace(0.0, 2*np.pi, 100)
plt.gca().set_aspect('equal')
plt.plot(np.cos(t), np.sin(t))
plt.show()
```

There is another new `matplotlib` command here:
`plt.gca().set_aspect('equal')` forces the x and y axes to use the same number of pixels per unit.

Exercise 8.7. Adapt the last piece of code to plot $\boldsymbol{x}(t) = (\cos(t) + \cos(5t), \sin(t) + \sin(5t))$. Try to predict what the curve will look like *before* running your program!

Task 8.8. Revisit tasks 7.11 and 7.12 and use `numpy` arrays. You may find that you can simplify a little.

Task 8.9. In this task we will compute and plot an approximation to the derivative of a function $f(x)$. Given a numpy function `f` (which calculates using arrays), a numpy array `x` and a `float h`, this function returns an array containing an approximation to its derivative at the points in `x`:

```
def deriv(f, x, h):
    return ( f(x + h) - f(x) ) / h
```

Can you see how this estimates the derivative? We could now use this to approximate the derivative of $\sin x$ on $[0, \pi]$ by

```
deriv(np.sin, np.linspace(0, np.pi, 100), 0.01)
```

Now plot the function $g(x) = \sin(x)$ and its approximate derivative. What should the derivative $g'(x)$ be? Did `deriv` work? Can you add a plot of the correct derivative to compare with? What happens if `h` is too large? Or too small?

Exercise 8.10. Now do what you just did for the function $h(x) = e^{(\sin x)/(x^3+1)}$. First, find $h'(x)$ using your knowledge of calculus (hint: first find the derivative of $(\sin x)/(x^3 + 1)$ and then use the chain rule). Use `deriv` and `matplotlib` to check if you got the answer right.

Section 8d: Two-dimensional `numpy` arrays ("matrices")

Task 8.11. So far we have seen only 1-D arrays. But `numpy` arrays can have more dimensions. We will consider 2-D arrays, which have much in common with matrices.

One way to create a 2-D array is to convert a list of lists, e.g.

```
a = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0]])
```

Now `print(a)` and the check the number of dimensions and the shape of an array by `print(a.ndim)` and `print(a.shape)`.

For 2-D arrays the `shape` describes the number of rows and columns of the array. We can also use `zeros` or `ones` to create an array with any shape we want:

```
b = np.ones((2,3))
```

and as for 1-D arrays we can easily perform arithmetic operations element by element:

```
print(a+b)
```

Exercise 8.12. Predict the value of each of the following expressions and then check using Python: `3*b`, `a**2`, `a**b`.

Task 8.13. We can refer to individual elements of a 2-D array or extract subarrays using expressions similar to those for lists:

```
print(a[0, 1])
```

Try printing each of the following: `a[1, 1:]`, `a[0, :]`, `a[:, 0]`. Make sure you understand what each of these expressions means, and remember that as for lists the counting starts with element 0. Note that for each dimension choosing an element removes the dimension while a "slice" (even of size 1) using the `:` notation keeps the dimension.

Exercise 8.14. Predict and then print the value, number of dimensions and shape of each of `a[0, 1]`, `a[0:1, 1]`, `a[0, 1:2]` and `a[0:1, 1:2]`.

Exercise 8.15. Create a 1-D array which is the sum of the columns of `a`. Create another

which is the sum of the rows. Hint: look at `some_numpy.py`.

Exercise 8.16. Create the following matrix M and vector $\boldsymbol{b}$ as `numpy` arrays.

$$M = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 1.0 & 4.0 & 5.0 \\ 1.0 & 6.0 & 8.0 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 1.0 \\ 4.0 \\ 9.0 \end{pmatrix}.$$

We can now do matrix multiplication very easily: `m @ b`.

What do `b @ b` and `m @ m` give?

Task 8.17. What do you think the functions `np.linalg.inv` and `np.linalg.det` will do to M? Try both. Warning: these are numerical approximations and can fail.

Exercise 8.18. Solve the following system of linear equations using Python. Hint: having done exercise 8.16 and the task that follows it, it should now take just one short line of Python code.

$$\begin{aligned} x + 2y + 3z &= 1 \\ x + 4y + 5z &= 4 \\ x + 6y + 8z &= 9 \end{aligned}$$

Section 8e: Broadcasting in `numpy`

This section is definitely "beyond the basics" for `numpy`. But it shows some of the power that `numpy` brings if one can learn how to use it well.

Task 8.19. One handy way to make a 2d array is by *broadcasting* from 1d arrays. Earlier, we said that arrays had to have the same shape to combine them. That is in fact not true: a dimension with length 1, i.e. where there is only a single row or a single column, is treated specially.

First find out what `b[:,None]` and `b[None,:]` are in the `Shell`. This use of `None` to create an additional dimension in an array is specific to `numpy`. We have seen `None` occasionally before as what gets returned from a function if we don't include a return statement or if there is a return statement but it doesn't get executed. This is a completely different use of `None`.

Now try `b[:,None] + b[None,:]` in the `Shell` as well as `m + b[:,None]` and `m + b[None,:]`. The way in which a two-dimensional array with either a single row or single column or both gets treated as though there more rows or columns is known as *broadcasting* (Note: broadcasting is actually more general than this—see the numpy manual for more details.).

Task 8.20. Download `contour.py` from Ultra which shows one way to plot a function $f(x, y)$ of two variables: we first create two 1-D arrays `x` and `y` which each contain 50 values in the interval $[-4, 4]$. Then we create a 2-D array `aa` which contains the corresponding values of $f(x, y) = x^2 + y^2$. Finally, we draw a contour plot of $f(x, y)$.

What is a "contour" of $f(x, y)$? Why do you see circles in the plot?

This example uses *broadcasting*. What is the shape of `aa`? Why does `aa` end up being two-dimensional when `x` and `y` are both one-dimensional?

Exercise 8.21. Use broadcasting to construct a 2-D array in Python which corresponds to the 4×4 matrix $x_{ij} = (i - j)^{i+j}$ for $i, j \in \{0, 1, 2, 3\}$.

Practical sheet 9: Introduction to Classes

Please watch the section 9 video about "Classes" if you have not already watched it.

In all of the following it will be important to test everything that you do. So as you add methods to your classes, also add tests and print statements at the bottom.

Section 9a: The Dummy class example

Task 9.1. Download `dummy-example.py` from Ultra (Python files for the section 9 video) and review it.

Exercise 9.2. Add another instance to check your understanding. Add a new method to the class to print out the `y` attribute of an instance and test the method using the instances already created.

Section 9b: The Complex class example

Task 9.3. Download `complex.py` from Ultra and review it.

Exercise 9.4. Add a method `argument` to the `Complex` class to compute and return the "argument" of the complex number. If you're not sure what the argument of a complex number is, see the Newcastle University Maths department's web page about the modulus and argument of a complex number.

You may also find it helpful to read the documentation for the `atan2` function in the `math` module. See docs.python.org/3/library/math.html

Exercise 9.5. Add a new method `conjugate` to the `Complex` class that does the same thing as the `conjugated` method except that it changes the existing instance "in place" instead of creating a new instance.

Exercise 9.6. Add a new method `multiply` to the `Complex` class that multiplies the instance by another `Complex` number. Hint: start from a copy of the `add` method.

Exercise 9.7. Add a new method `__mul__` to the `Complex` class so that `*` works for complex numbers.

Task 9.8. Review the Python documentation about special methods for classes that emulate numbers and consider which of the methods listed you would want to implement for the `Complex` and what would be involved. There's no need to spend a lot of time on this, just to think a bit about the process.

Section 9c: Making a class to represent sets

Exercise 9.9. Python already has a **`set`** class but you might like to try to implement your own class, called (say) `MySet`. The idea is that the class should have an attribute which is a list containing the elements of the set. In what follows, that attribute is named `contents`.

- Start by making a class `MySet` which has initialiser `__init__(self, alst)`. The initialiser should make a new list in which each element in the original list appears only once and save the new list as the attribute `self.contents`.

- Define the `__str__(self)` method to return something sensible based on the `self.contents` attribute. This will help you to test what later methods are doing.
- Define a `len(self)` method to return the number of elements in the `MySet`.
- Define a `insert(self, element)` method to add an element to the set. It should check if `element` is already in the set (in the list stored as `self.contents`.). If it's already there, the method does nothing. Otherwise, it should add it to `self.contents`.
- Define a `intersection(self, set2)` method which should return a new `MySet` containing the elements of `self` that are also in `set2`. You should assume that `set2` is also an instance of `MySet`.
- Define a `union(self, set2)` method which should return a new `MySet` containing all the elements of `self` and `set2`. Again, Pyset2 is an instance of `MySet`.
- Define a `setdiff(self, set2)` method which should return a new `MySet` containing all the elements of `self` that are not in `set2`. Again, Pyset2 is an instance of `MySet`.

Exercise 9.10. You can now make the set of all natural numbers (including 0) less than 50 that are multiples of 5 by

```
mult5 = MySet(range(0,50,5))
```

In the same way, make the set of all natural numbers less than 50 that are multiples of 3.

Now use your `MySet` class methods to make and print the the following subsets of the natural numbers less than 50:

- those that are multiples of 5 but not of 3;
- those that are multiples of either 5 or 3 or of both 5 and 3;
- those that are multiples of both 5 and 3;
- those that are multiples of either 5 or 3 but are not multiples of 15.

Exercise 9.11. The Python built-in `set` class implements `+` for union of set, `*` for intersection and `-` for set difference. Do the same for your `set` class and test that everything works.

Practical sheet 10: Games and Complexity

Section 10a: Bisection

Task 10.1. [Prof. Peyerimhoff]

Download `ngame.py` from Ultra and run it (perhaps few times).

You should be able to beat the computer if you are smart about how to search for the number. Hint: think about how the bisection method works for finding a root in section 7b.

Now look at the code to see how it works.

Exercise 10.2. Write a function `binseek(x,t)` that, given a *sorted* list `t` of numbers and a number `x`, returns the index (position) of (the first occurence of) `x` in `t` or `None` if `x` is not in `t`. For example if `tst=[1.0, 3.2, 5.0, 7.1, 7.1, 10.0]`, then `binseek(3.2, tst)` should return `1`, `binseek(7.1, tst)` should return `3` and `binseek(6.0, tst)` should return `None`.

If you write this correctly, it should be able to run much faster than if the list was not (known to be) sorted. If this is not apparent to you, discuss with one of the tutors or a classmate who has worked it out.

Remark 10.3. It should be clear from these simple examples that, given a sorted list of length N, it should not take much more than $\log_2 N$ tries to arrive at the answer.

Mathematicians say that searching a sorted list takes $O(\log N)$ amount of time. This is an example of "big O notation" which is commonly used to describe the limiting behaviour of sequences and functions. You can think of $O(x)$ as meaning "of the order of" although there is of course a more formal definition (see `https://en.wikipedia.org/wiki/Big_O_notation`).

Section 10b: Sorting

Task 10.4. Recall Python's very useful `lst.sort()` and `sorted(lst)` functions. Supposing these weren't available, how would we sort lists?

First review `naivesort.py` on Ultra. The algorithm was discussed in the section 10 presentation on Ultra. You should be able to understand how it works. If necessary, add some print statements and run some test examples.

Now download `bsort.py` from Ultra. Try this out with some (small) lists and understand how it works. As usual, some `print` diagnostics may help.

If you print out the list at each step, you will see why it is called *bubble sort*.

Now try both `naivesort` and `bsort` on a list of 10^3 random elements which you can create using `random.random`. How long does each take? Make sure that you use the same list to test both functions. To time the execution, do something like

```
import time
t0 = time.time()
newlst = bsort(lst)
print(time.time() - t0)
```

This assumes that the only source of delay is the execution of the code, which is not always the case with any computer, but especially for CIS classroom computers. Using your own laptop may give a better idea of the actual execution time.

How long does each take to sort a list of 10^4 random elements?

How long would it take with 10^6 elements?

By looking at the code (and some thought), one can see that, like `naivesort`, `bsort` should take roughly cN^2 time units to run for a list of size N when N is large, for some constant c (the two functions will have different values of c.) Mathematicians say that the execution time of `bsort` is $\mathsf{O}(N^2)$.

Exercise 10.5. Write a function `mergelists(t1,t2)` that, given two *sorted* lists `t1` and `t2` of numbers, returns a sorted list of all the elements of `t1` and `t2`. The point is to do this *without* using either Python's built-in sorting functionality or either of the sorting functions above. You should exploit the fact that `t1` and `t2` are both already sorted in order. Complete this before moving to the next exercise.

Task 10.6. Returning to sorting lists, now if we split the list into two halves and sort each half separately using `bsort`, this would take $2 \cdot cN^2/4 = cN^2/2$ time units. We must then merge the two (sorted) lists, but this can be done in kN time units, for a total of $cN^2/2 + kN \sim N^2/2$ time units for N large. But then of course it should not take $cN^2/4$ time units to sort each half, since applying the same logic, we could do it in $c \sim N^2/8$ time units, giving $c \sim N^2/4$ as the total execution time.

How far can we go? Well, obviously we can stop when the list either is empty or has one element, by which point there is no need for `bsort` at all! A little thought shows that there are $\lceil \log_2 N \rceil$ levels of subdivisions, and since merging two lists of length $\tilde{N}$ each is $\mathsf{O}(\tilde{N})$, the entire sort is $\mathsf{O}(N \log_2 N)$.

This is the principle behind *merge sort*. Download `msort.py` from Ultra and replace the comment in the code by a single line which does the task of merging `p1` and `p2` to make `ps`.

Now test `msort` in the same way as you did `naivesort` and `bsort`. When you are sure that it works, carry out some timing comparisons between the three algorithms

***Remark 10.7.**

The advantage of merge sort is that it is *always* $\mathsf{O}(N \log N)$, while its disadvantage is that it needs $2N$ storage units (actually, $3N$, the way we did it above). Variants exist that are more space-efficient. Python's `sort()` is (usually) implemented using *timsort*, which is (heuristically) faster for real-life data (which are often not completely random).