Functions

Until now we've been putting our code all in one place. This is fine for small toy programs but if we want to write something bigger it becomes incredibly difficult to follow. This is where the idea of functions comes in - they're basically mini programs that you can run or *call* in your *main* one that provide us with various advantages. In order to understand them we must first see how functions work in practice. Here's a few examples:

```
def myNameRet():
    return "My name is Anton"
print(myNameRet())     # Will print "My name is Anton"

def myNameNoRet():
    print("My name is Anton")
myNameNoRet()       # Will also print "My name is Anton"

def square(x):
    return x * x

print(square(5))    # Will print "25"
print(square(10))   # Will print "100"

def rectArea(width, height):
    return width * height
print(rectArea(3, 4))   # Will print 12
```

1. We *define* a function with `def`.
2. We follow that with the function *name*.
3. Next we have a pair of parentheses in which we list the *arguments* to the function - these are equivalent to the *x* and *y* you might be familiar with from functions in Maths. We can have as many arguments as we want and inside the function they're treated as variables with some value already stored in them.
4. After the parentheses we have a colon signifying that what follows is an indented block, similar to what we have for loops.
5. That block is where we write our "mini-program", also known as the *body* of our function. Inside the body we may have the `return` keyword - this signifies the end of a function and whatever is right of `return` will be the *return statement* of the function - think of it as the thing that replaces the function at the place where it was called. A function that doesn't end up calling `return` automatically returns `None` which is a special value with type `NoneType`.

Note: You could have multiple return statements in a function - consider using an `if`/`else` to return different values depending on some condition. You should try to keep the type of all return statements in a function the same because otherwise you might get type errors. Consider the example:

```
def getShapeArea(shape_name, x, y):
    if shape_name == "rectangle":
```

```
        return x * y
    elif shape_name == "triangle":
        return x * y / 2

if getShapeArea("rectangle", 5, 10) > getShapeArea("tringle", 5, 10): # typo intentional
    print("The rectangle has a larger area")
```

Here a typo in the name of the shape can result in `getShapeArea` not returning anything which has type `NoneType` and you can't compare `NoneType` with `int`, causing an error.

Back to the usefulness of functions - they let us do 4 useful things:

- We can split our code into multiple chunks that are easier to reason about. Otherwise big programs would become one big chunk of code that would be incredibly difficult to understand.
- A useful property of functions is that they can also call functions. They can even call themselves. This is known as recursion and, sadly, it is outside of the scope of this course but is so useful that some other programming languages have completely removed loops in favour of recursion.
- On a similar note, when getting errors Python gives us a very convenient *call stack* - basically a list of all the functions that were called but haven't ended when we get the error. This makes tracking down bugs far easier.
- We can reuse functions as many times as we want, reducing code repetition

A good rule of thumb is to keep functions below 200 lines of code. This includes the *main* function (i.e. the non-indented code). Anything larger should be split up into multiple functions.

Exercises

For today's session we have a large number of different exercises that you can choose from. They are roughly ordered by difficulty. If you find an exercise easy, consider skipping a few. It is somewhat difficult to finish most of them by the end of the session so we'll begin the next session with some exercise time as well after the recap. We would also encourage doing some of the exercises at home. In that case you could check the `solutions.py` file and the recordings from a previous run of the course here.

Each exercise has some tests associated with it so you can test your solution. To run the tests, download the associated file, open it in IDLE, and edit the function at the top. An example of how to do exercise 1 will be shown during the lecture.

Exercise 1: Summing a list

Write a function `sum` that takes a list of numbers and returns the sum of all the numbers in the list.

```
def sum(xs):
    # return the sum of all the elements in the list xs
```

Download Test Cases

Exercise 2: Product of a list

Write a function `product` that takes a list of numbers and returns the product of all the numbers in the list.

```
def product(xs)
    # return the product of all the elements in the list xs
```

Download Test Cases

Exercise (not part of the tests): what should the product of the empty list be?

Exercise: can you optimise this if you find a zero in the list?

Exercise 3: Mean of a list

```
def mean(xs):
    # return the mean of all the elements in the list xs
```

Download Test Cases

Exercise 4: Flattening lists

Lists can contain lists, so take a list `[ [ a, b, c, ... ], [ d, e, f, ... ], ... ]` and flatten it to `[ a, b, c, ... , d, e, f, ... ]`.

```
def flatten(xs):
    # return a flatted copy of xs
```

Hint: The code for this is *almost* exactly the same as one of the earlier exercises.

Hint:* The empty list is `[]`

Download Test Cases

Exercise 5: Pairwise sum

Given two lists `xs` and `ys`, return a new list `zs` where each element is the sum of the corresponding elements in `xs` and `ys`, i.e. `sum2([1, 2, 3], [10, 11, 12]) == [11, 13, 15]`. You may assume that the lists are the same length.

```
def sum2(xs, ys):
    # return a list pairwise summing the elements
```

Download Test Cases

Exercise 6: Pairwise sum

This is exactly the same as the previous exercise, only you may not assume that the lists are of the same length. The list returned should be the length of the longer list, as if the remainder of the shorter list were zeroes. For example, `sum2([1, 2], [1, 2, 3, 4]) == [2, 4, 3, 4]`.

```
def sum2(xs, ys):
    # pairwise sum assuming that xs and ys aren't the same length
```

Download Test Cases

Exercise 7: Divisibility

Given a number `n`, compute a list of all the integers `x` divisible by 3 or 5 that are `0 <= x < n`. For example, `three_or_five(10) == [0, 3, 5, 6, 9]`.

```
def threes_or_fives(n):
    # return a list of all the integers greater than or equal to zero that
are
    # divisible by three or five and less than n
    # You might want to look up the modulus (%) operator, which computes
the
    # remainder of division
```

Exercise: See if you can combine this function with your solution to exercise 1 for solving the first Project Euler problem.

Download Test Cases

Exercise 8: Filtering a list

Given a list of numbers, return a copy of the list that only includes the even numbers.

```
def filtered_even(xs):
    # Return a copy of the list that only includes even numbers
```

Download Test Cases

Exercise 9: Filtering a list of strings

Given a list of strings, return a copy of the list that only includes the strings that start with an lowercase letter.

```
def filtered_text(xs):
    # Return a copy of xs that only contains the strings that start with a
    # lowercase letter
```

Hint: you can treat strings like lists, so to get the first character of a string you can do `s[0]` where `s` is a string variable.

Hint: you can compare characters to other characters, e.g. `"a" <= "b"` is `True` but `"a" >= "z"` is `False`; you need to include the quote marks around the character still.

Download Test Cases

Exercise 10: Rotating a list

Given a list `xs` and an integer `n`, produce a list where each element is rotated around by `n`, i.e. `rot([1, 2, 3], 1) == [2, 3, 1]` and `rot([1, 2, 3, 4, 5], 2) == [3, 4, 5, 1, 2]`.

```
def rot(xs, n):
    # return a list of xs rotated by n
```

Download Test Cases

Sorting lists

For the second set of the exercises you'll need to be able to sort a list into ascending order. To do this in Python you need to use the `sorted` function:

```
beatles = [ "John", "Paul", "George", "Ringo" ]
beatles2 = sorted(beatles) # == [ "George", "John", "Paul", "Ringo" ]

nums = [13, 56, 26, 2, 12, 12, 2, 4]
nums2 = sorted(nums) # == [2, 2, 4, 12, 12, 13, 26, 56]
```

Exercise 11: Removing duplicates

Given a list `xs`, return a new sorted list with all the duplicate elements removed.

```
def uniques(xs):
    # return a list of all the unique elements
```

Download Test Cases

Exercise 12: Binary search

Note: This exercise is harder than the others.

This exercise requires you to adapt the binary search algorithm seen last week. Given a sorted list of strings, find the least index of an element with that value. One way we could do this is:

```
def search(xs, x):
    for i in range(0, len(xs)):
        if xs[i] == x:
            return i
    return None
```

However, this isn't particularly efficient. The binary search reduces the number of comparisons we have to do from `len(xs)` to `log2(len(xs))` where `log2` is the base 2 logarithm.

```
def bin_search(xs, x):
    # return the least index of an element equal to x in the sorted list
xs
```

Note: don't worry about what you return if the item isn't in the list. Often when implementing binary search it is useful to return the index that an element *would* be at, *were* it in the list (i.e. so it could be inserted whilst keeping the list in sorted order).

Download Test Cases

Exercise 13: Merging lists

Given two sorted lists, return a sorted list containing the contents of the two lists merged in order, i.e. `merge([0, 1, 1, 2, 3, 5, 8], [1, 2, 3, 4, 5, 6]) == [0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 8]`. You should not assume that the lists are the same length.

```
def merge(xs, ys):
    # return a sorted list with the merged contents of the sorted lists xs
and ys
```

Download Test Cases

Exercise 14: Merge sort

Merge sort is a common sorting algorithm, and you can implement it using the merge function from the previous exercise. The result of this function should be the same as the result of the `sorted` function:

```
def merge_sort(xs):
    # A sorted copy of xs
```

Download Test Cases

Hint: In Python, if you want to split an array in two you can do:

```
first_half = xs[:(len(xs) // 2)]
second_half = xs[(len(xs) // 2):]
```

Mathematical exercises

The remainder of the exercises are intended for maths students or those that have studied maths at some point (A-Level Core Maths should be enough).

Exercise 15: Representing polynomials

For the next few exercises we will represent polynomials with a list, where each element of the list represents the corresponding coefficient of `x`. For example, the list `[1, 2, 3, 4]` represents the polynomial `1x^0 + 2x^2 + 3x^3

- 4x^4 = 1 + 2x^2 + 3x^3 + 4x^4`.

In this exercise you should just write a function that takes a polynomial represented in this way and outputs the equivalent equation, as above.

Can you make this nicer by not printing a term if the coefficient is zero?

Note: This exercise doesn't come with a test case; the intention is that you write some code that you can use later for debugging. If you get stuck however, the solution is here.

Exercise 16: Adding polynomials

Given two lists that represent a polynomial, return a new list that represents the polynomial that is the sum of those polynomial. For example, `poly_sum([1, 2, 3], [4, 5, 6]) = [5, 7, 9]` because `(1 + 2x + 3x^2) + (4 + 5x + 6x^2) = 5 + 7x + 9x^2`. Be careful to consider the result of adding two polynomials of different degrees.

```
def poly_sum(xs, ys):
    # return the list representing the sum of the polynomials represented by the
    # lists xs and ys
```

Download Test Cases

Hint: This is *exactly* the same as one of the earlier exercises.

Exercise 17: Multiplying polynomials

Given two lists that represent a polynomial, return a new list representing the product of those two polynomials. For example, `(1 + x)(2 + x) = 2 + 3x + x^2`, so we want `poly_prod([1, 1], [2, 1]) == [2, 3, 1]`.

```python
def poly_prod(xs, ys):
    # return the list representing the product of the polynomials
represented by
    # the lists xs and ys
```

Download Test Cases

Hint: you can create a list of `0`s of length `n` in Python with `[0] * n`.