MATH40006: An Introduction To Computation
Course notes, Section 12

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

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

---

12 Data, Files and a Very Brief Introduction to Objects

12.1 Sets, dictionaries and frozensets

We've already met the data structures **list**, **tuple** and **string**. Now for three we didn't get to look at back then.

A **set** is a data structure that ignores (a) order and (b) multiplicity. We create sets in Python by using curly brackets, or by wrapping the word `set` around a list or tuple or string:

```
set1 = {5, 5, 3, 1, 3, 7, 9, 1, 5, 3, 3, 7, 9, 1, 1, 7}
set2 = set([3, 5, 7, 9, 1, 3, 5, 7, 9])
set3 = set((5, 5, 5, 7, 7, 7, 7, 3, 3, 3, 3, 1, 1, 1, 1, 9, 9, 9, 9))
set4 = set('the quick brown fox jumps over the lazy dog')

print(set1)
print(set2)
print(set3)
print(set4)
```

```
{1, 3, 5, 7, 9}
{1, 3, 5, 7, 9}
{1, 3, 5, 7, 9}
{'u', 'n', 'l', 'd', 'w', 'v', 'q', 'j', 's', 'f', ' ', 'i', 'z', 'y',
'o', 'b', 'c', 'a', 'h', 't', 'k', 'e', 'r', 'x', 'p', 'g', 'm'}
```

Notice that all multiplicities have been suppressed, and the set of characters seems to be in an entirely arbitrary order. This is by design; order and multiplicity don't matter with sets. In fact, you could say that the only thing that matters about a set is whether a certain piece of data is an element of it or not; not where it appears or how often.

```
print(set1 == set2)
print(set2 == set3)
print(set3 == set1)
```

```
True
True
True
```

The set-theoretic operations of union and intersection are represented by, respectively, | and &:

```
primes = {2, 3, 5, 7, 11, 13, 17}
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
print(primes | odds)
print(primes & odds)
```

```
{1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19}
{3, 5, 7, 11, 13, 17}
```

The command `a - b` gives those elements that are in `a` but not `b`:

```
print(primes - odds)
print(odds - primes)
```

```
{2}
{1, 19, 9, 15}
```

The command `a ^ b` gives those elements that are in `a` or `b`, but not both:

```
print(primes ^ odds)
```

```
{1, 2, 9, 15, 19}
```

To check whether something is an element of a set, use `in` (this also works with lists and tuples, of course):

```
print(2 in primes)
print(2 in odds)
```

```
True
False
```

The `add` method allows you to place additional elements in a set; it's the rough equivalent of `append` for lists.

```
primes.add(23)
print(primes)
```

```
{2, 3, 5, 7, 11, 13, 17, 23}
```

The remove method deletes a specific element:

```
primes.remove(23)
print(primes)
```

```
{2, 3, 5, 7, 11, 13, 17}
```

If you try to remove an element that isn't there, an error message is thrown:

```
odds.remove(18)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-4c89fd943bbf> in <module>()
----> 1 odds.remove(18)

KeyError: 18
```

The discard method acts like remove, except that it throws no error if it fails to find the target element:

```
odds.discard(19)
print(odds)
odds.discard(18)
print(odds)
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17}
{1, 3, 5, 7, 9, 11, 13, 15, 17}
```

The add, remove and discard methods are all pure side-effects; none of them returns a value (or rather, they all return the value None). By contrast, the pop method works a bit like its counterpart for lists; it both removes and returns an element. The difference is that if the case of lists, the element returned is always the last in the list, whereas with sets, it's arbitrary and unpredictable (sometimes, this doesn't matter).

```
print(set4)
print(set4.pop())
print(set4)
```

```
{'u', 'n', 'l', 'd', 'w', 'v', 'q', 'j', 's', 'f', ' ', 'i', 'z', 'y',
'o', 'b', 'c', 'a', 'h', 't', 'k', 'e', 'r', 'x', 'p', 'g', 'm'}
u
{'n', 'l', 'd', 'w', 'v', 'q', 'j', 's', 'f', ' ', 'i', 'z', 'y', 'o',
'b', 'c', 'a', 'h', 't', 'k', 'e', 'r', 'x', 'p', 'g', 'm'}
```

The elements of a set can be iterated across:

```
for n in odds:
    print('({} + 1) / 2 is equal to {}'.format(n, (n+1)//2))
```

```
(1 + 1) / 2 is equal to 1
(3 + 1) / 2 is equal to 2
(5 + 1) / 2 is equal to 3
(7 + 1) / 2 is equal to 4
(9 + 1) / 2 is equal to 5
(11 + 1) / 2 is equal to 6
(13 + 1) / 2 is equal to 7
(15 + 1) / 2 is equal to 8
(17 + 1) / 2 is equal to 9
```

Sets, just like lists, can form the output of a comprehension:

```
new_odds = {2*n-1 for n in range(1,11)}
print(new_odds)
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

Notice that lists and sets share this property of being able to form the output of a comprehension, whereas tuples lack it.

A **dictionary** in Python is a data structure that is indexed not by a range of numbers but by a set of *keys* (which can be any kind of Python data, including strings). (Well, actually, not absolutely any kind of data, as we'll see—but many). They're a Python implementation the idea of an **associative array**.

For example:

```
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}
print(polyhedra['archimedean'])
print(polyhedra['johnson'])
```

```
13
92
```

To access the keys:

```
print(polyhedra.keys())
```

```
dict_keys(['platonic', 'archimedean', 'catalan',
'johnson (simple)', 'johnson', 'kepler-poinsot'])
```

To access the associated values:

```
print(polyhedra.values())
```

```
dict_values([5, 13, 13, 28, 92, 4])
```

To access both:

```
print(polyhedra.items())
```

```
dict_items([('platonic', 5), ('archimedean', 13), ('catalan', 13),
 ('johnson (simple)', 28), ('johnson', 92), ('kepler-poinsot', 4)])
```

The keys, values and items of a dictionary can be iterated across:

```
for poly in polyhedra.keys():
    print(f'What are the properties of {poly} polyhedra?')
```

```
What are the properties of platonic polyhedra?
What are the properties of archimedean polyhedra?
What are the properties of catalan polyhedra?
What are the properties of johnson (simple) polyhedra?
What are the properties of johnson polyhedra?
What are the properties of kepler-poinsot polyhedra?
```

```
for n in polyhedra.values():
    print(f'There are {n} of a certain kind of polyhedron.')
```

```
There are 5 of a certain kind of polyhedron.
There are 13 of a certain kind of polyhedron.
There are 13 of a certain kind of polyhedron.
There are 28 of a certain kind of polyhedron.
There are 92 of a certain kind of polyhedron.
There are 4 of a certain kind of polyhedron.
```

```
for poly, n in polyhedra.items():
    print(f'There are {n} {poly} polyhedra.')
```

```
There are 5 platonic polyhedra.
There are 13 archimedean polyhedra.
There are 13 catalan polyhedra.
There are 28 johnson (simple) polyhedra.
There are 92 johnson polyhedra.
There are 4 kepler-poinsot polyhedra.
```

12.2 More about mutability

We've met, in the course so far, two types of data structure. Lists and sets are what we call **mutable**: you can change them piecemeal. For example, lists support the operation of *appending*, and sets that of `adding`.

By contrast, tuples and strings are **immutable**. If you assign an immutable value to a variable, the only way to change it is to completely redefine it.

Individual pieces of data, such as ints, longs or floats, are also immutable in this sense (kind of by default, really, as they don't have separate components that can be changed one by one).

You may be wondering, given that sets are mutable, whether there's a set-like version of a tuple: that is, an immutable data structure that ignores multiplicity and order. There is, though it's a bit unwieldy; it's known as a **frozenset**.

```
odds = set(range(1,21,2))
primes = {2, 3, 5, 7, 11, 13, 17, 19}
frozen_odds = frozenset(odds)
frozen_primes = frozenset(primes)
print(frozen_odds)
print(frozen_odds | frozen_primes)
```

```
frozenset({1, 3, 5, 7, 9, 11, 13, 15, 17, 19})
frozenset({1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19})
```

You may remember that when we first met this distinction between mutable and immutable data, I was rather vague about why the latter was necessary. Now it can be revealed: we need immutable data because mutable data is simply too unstable to serve as dictionary keys. The keys to any dictionary don't have to be of any particular type, but they must all be immutable. (Well, it's slightly more subtle than that—but this is pretty near the truth of it.)

Mutable data (lists and sets) can't form the elements of a set or a frozenset either (so though we can have a set of frozensets, or a frozenset of frozensets, we can never have a set of sets, or a frozenset of sets). Neither can it serve as dictionary keys. Immutable data can be used in both these ways.

We've noted before an interesting effect when we change the value of mutable data. Check out the following:

```
list1 = list(range(1, 16, 2))
list2 = list1
list3 = list(range(1, 16, 2))
print(list1)
print(list2)
print(list3)
```

```
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 5, 7, 9, 11, 13, 15]
```

Then:

```
list1.append(17)
print(list1)
print(list2)
print(list3)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17]
[1, 3, 5, 7, 9, 11, 13, 15, 17]
[1, 3, 5, 7, 9, 11, 13, 15]
```

The way to explain this is that the variable names `list1` and `list2` *actually refer to the same piece of data*, whereas `list3`, which was defined separately, refers to a different piece of data that happens, at the moment, to have the same *value*. So when we change the value of `list1` using mutability, the value of `list2` also changes, but that of `list3` doesn't.

Does that mean that if we want to create a list with the same value as another list, but consisting of separate data, we have to assign the value separately like this? That could be pretty laborious. Fortunately, the answer is no. The following works just as well:

```
from copy import copy
list1 = list(range(1, 16, 2))
list2 = list1
list3 = copy(list1)
```

It's also worth noting a crucial difference between mutable and immutable data in the behaviour of *augmented assignment*. Contrast this...

```
list1 =  list(range(1, 16, 2))
list2 = list1
list1 += [17, 19, 21]
print(list1)
print(list2)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

... with this:

```
tup1 =  tuple(range(1, 16, 2))
tup2 = tup1
tup1 += (17, 19, 21)
print(tup1)
print(tup2)
```

```
(1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21)
(1, 3, 5, 7, 9, 11, 13, 15)
```

The reason augmented assignment works this way with tuples is that it has to. You can't change the value of a tuple in place, so the only thing you can do is set up a new variable with the same name. The command

```
tup1 += (17, 19, 21)
```

is exactly equivalent to

```
tup1 = tup1 + (17, 19, 21)
```

whereas the commands

```
list1 += [17, 19, 21]
```

and

```
list1 = list1 + [17, 19, 21]
```

are subtly different.

12.3 File input/output

12.3.1 Reading and writing strings

It's quite easy to write strings to, and read them from, external files. Try the following:

```
example_str = "Python and its file I/O make me feel dumbstruck etc."

fo = open("fileio_test.txt", "w")
fo.write(example_str)

fo.close()
```

(Note, `"w"` stands for "write".)

Now, locate the file `fileio_test.txt`, which should be somewhere on your computer (exactly where will depend on your Python set-up). If you read it, the string should be there.

To read the string in again, try:

```
fo = open("fileio_test.txt", "r")
new_str = fo.read()

fo.close()

print(new_str)
```

```
Python and its file I/O make me feel dumbstruck etc.
```

12.3.2 Using the eval function

OK, so that's OK for strings. But suppose we want to write to a file, and read from a file, some other kind of Python object, like a list, or a tuple, or a set, or a dictionary? One way to do that is to convert it to a string before we write it, and convert it back from a string after we're read it.

Converting a Python object into a string is often pretty easy: we can just wrap a `str()` around it. So for example:

```
# define dictionary
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}
# convert into string
polyhedra_str = str(polyhedra)

# open a new file and write to it
fo = open("fileio_test2.txt", "w")
fo.write(polyhedra_str)

# close the file
fo.close()
```

If you check the computer, this file should now have appeared, with the correct string in it.

When we read the string back in, we have to convert it back into a dictionary again. There's a function that takes any string and converts it (if it can) into a Python object, and that function is called `eval`. Here's how that works:

```
# open the file and read from it
fo = open("fileio_test2.txt", "r")
new_dict_str = fo.read()

# close file
fo.close()

# convert to dictionary
```

```
new_dict = eval(new_dict_str)

# testing:
print(new_dict['platonic'])
```

```
5
```

12.3.3 The pickle module

Sometimes using `eval` like that really is the best way to do things (an example might be when you're taking Python data from a URL). But there are several drawbacks to it. One is that the file can end up bigger than it needs to be. The amount of data in our dictionary is quite small, but the amount of memory needed to store it *as a string* might be much greater. (This doesn't matter much for our little toy case here, but if you were storing large amounts of data in a Python dictionary, it might.)

Another drawback is that some Python objects (such as functions) can't, in any case, readily be described using strings.

There's a module called `pickle` that allows us to write Python objects directly to files, and read them in again. This requires Python to convert the object to and from a readable/writeable form. If you're an old computing hand, you may have met the general idea before, and heard it called *serializing* or *marshalling*; in Python, it usually gets called **pickling**.

Here's how it works for our dictionary example above. First writing:

```
import pickle

# define dictionary
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}

# open a new file and write a pickled version of the dictionary to it
po = open("fileio_test3.pickle", "wb")
pickle.dump(polyhedra, po)

# close the file
po.close()
```

(Note that the second input we give to the `open` function is not `"w"` but `"wb"`; the 'b' stands for "bytes".)

Now reading:

```
# open the file and read from it
po = open("fileio_test3.pickle", "rb")
new_dict = pickle.load(po)
```

```
# close file
po.close()

# testing:
print(new_dict['kepler-poinsot'])
```

```
4
```

The `dump` and `load` functions in the `pickle` module are the rough `pickle` equivalents of the `write` and `read` functions from core Python. Again, notice it's not `"r"` but `"rb"`.

In the exercises, you take a deeper dive into pickling; you"ll find that just about anything Pythonic can be pickled: not just data like our polyhedra dictionary, but also functions, not to mention the thing you're about to meet, **classes** (more on them later).

12.3.4 Using the exec function

It's even possible to read, and run, Python code. For that we need a way of taking a string that represents Python code, and getting Python to run it as code. You might think that would be a job for `eval`, but in fact there are strict limits on what `eval` will do; essentially, it will *evaluate* strings representing Python *expressions*, but it won't *execute* strings representing Python *instructions*. For that we need something stronger, namely the function called `exec`.

Here's an example. Let's start by writing a code string to an external file (note the use of the triple-quote to break the long string over several lines, and the use of the special character `\n` to represent a line break *in the code*):

```
code_string = """feeling_good = input('Are you feeling good? Y/N ')\n
if feeling_good=='Y': print('Glad to hear it!')\n
else: print('Oh dear!')"""

fo = open("fileio_test4.txt", "w")
fo.write(code_string)

fo.close()
```

Now to read it and run it:

```
fo = open("fileio_test4.txt", "r")
new_code_string = fo.read()

fo.close()

exec(new_code_string)
```

This has the same effect as running the small script

```
feeling_good = input('Are you feeling good? Y/N ')
if feeling_good=='Y':
    print('Glad to hear it!')
else:
    print('Oh dear!')
```

Notice, in passing, the use of the function `input` to set up a dialog with the user.

12.4 Objects and classes: putting data and programs in the same place

This course has mostly been about the traditional form of programming, in which there is *data*, and in which there are *programs* that operate on data, and the two are entirely separate. But Python also supports what's called **object-oriented programming** (OOP), in which data, and the things that operate on data, are combined into a single entity.

If you carry on with computing next year, there'll be much, much more about OOP. But let's have a taster now.

In one way, an **object** can be thought of a bit like a dictionary, in that it consists in part of data indexed non-numerically. But an object also consists of functions that work with this data. The pieces of data are known in Python as **attributes** and the functions are called **methods**.

To create an object is a two-stage process: first one defines what's called a **class**, and then one creates an **instance** of that class.

Let's look at an example. Our aim here is to build a kind of toy SymPy: a way of representing and processing symbolic mathematical expressions. For that we'll need a new type of Python data, and some ways of doing things with them.

Challenge 1: set up a new type of Python data called `OperatorExpression`, which is capable of representing expressions like `x + 2` or `5 ** 7`.

The new type of data we'll be writing is called a **class**, and the associated representation processes are called **methods**. Here's a first go at that:

```
class OperatorExpression(object):
    """Expression of the form a <operator> b, where <operator> is
    +, -, *, /, //, ** or %"""

    def __init__(self, root, left, right):
        self.contents = (root, left, right)
```

Lots to unpack here. First, notice the syntax of the first line. The keyword `class` lets Python know to expect a class, and we've specified its name, `OperatorExpression`. (The convention with classes is to use "camel case", in which every word in the class name starts with a capital letter; we don't normally use underscores to separate words, though it's perfectly possible to do so.)

Then there's the word `object` in brackets. This lets Python know that this new class is an extension of the most general Python class, the `object`.

Next we have something that every class must have: an `__init__` method, which tells Python what to do when we **instantiate** our class; that is, create data of this new type. This particular `__init__` method shows that our class carries around one piece of data, called `contents`: a tuple containing a root, a left and a right. The root will represent our operation, whereas the left and right will represent the two terms we're operating on. so that `x + 2` could be represented by the tuple `('plus', 'x', 2)`. This piece of data attached to the class is called an **attribute**.

Let's test our class by setting up an **instance** of it:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.contents)
```

```
('plus', 'x', 2)
```

Notice that although our `__init__` method seems to have four arguments, `self`, `root`, `left` and `right`, we only include the values of `root`, `left` and `right` when we set up an instance of our class. This will be a theme: the `self` never appears as an explicit argument; instead, it signals to Python that this is a method rather than an ordinary **function**. More on that later.

Now, this is all very well, but so far not very impressive. We'd like to be able to **do** things with our new class of Python object. Let's start by representing it in a more human-readable way.

Challenge 2: write a method called `prefixForm`, in which `x + 2` is represented as the string `'plus(x, 2)'` and `5 ** 7` is represented as `'power(5, 7)'`.

We add the following to our code:

```
    def prefixForm(self):
        "Method: returns string in the form root(left, right)"
        root, left, right = self.contents
        return root+'('+str(left)+', '+str(right)+')'
```

Then the following works:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.prefixForm())
```

```
plus(x, 2)
```

The main thing to notice here is the syntax of calling our new method. As with `__init__`, the `self` is ignored; this method appears, when we call it, to have no arguments. Also, we

type not `prefixForm(expr1)` but `expr1.prefixForm()`; this, as you know, is the typical syntax for a method rather than a function.

Let's now give users an alternative, even more human-readable representation:

Challenge 3: write a method called `infixForm`, in which `x + 2` is represented as the string `'(x) + (2)'` and `5 ** 7` is represented as `'(5) ** (7)'`.

We add the following to our code:

```
    def infixOperator(self):
        """Method: returns the infix operator
        associated with the root node"""
        root = self.contents[0]
        if root == 'plus':
            return '+'
        elif root == 'subtract':
            return '-'
        elif root == 'times':
            return '*'
        elif root == 'divide':
            return '/'
        elif root == 'intdivide':
            return '//'
        elif root == 'power':
            return '**'
        elif root == 'mod':
            return '%'

    def infixForm(self):
        """Method: returns string in the form (left) op (right) where
        op is the infix operator associated with the root node"""
        root, left, right = self.contents
        return \
    '('+str(left)+') '+self.infixOperator()+' ('+str(right)+')'
```

Then the following works:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.infixForm())
```

```
(x) + (2)
```

Those parentheses are a bit annoying, but getting rid of them in a robust way is a bit of a can of worms, so let's live with them.

Let's get more ambitious.

Challenge 4: write methods called `subs`, in which you can substitute a value for a variable, and `evaluate`, which allows you to take numerical expressions like `5 ** 7` and find their values.

We add the following to our code:

```
    def subs(self, var, val):
        """Method: returns an OperatorExpression
        with var substituted for val"""
        root, left, right = self.contents
        if left == var and right == var:
            # variable present on both sides of the expression
            return OperatorExpression(root, val, val)
        elif left == var:
            # variable present on the left hand side only
            return OperatorExpression(root, val, right)
        elif right == var:
            # variable present on the right hand side only
            return OperatorExpression(root, left, val)
        else:
            # variable not present
            return self

    def evaluate(self):
        "Method: returns the numerical value of an OperatorExpression"
        root, left, right = self.contents
        if root == 'plus':
            return left + right
        elif root == 'subtract':
            return left - right
        elif root == 'times':
            return left * right
        elif root == 'divide':
            return left / right
        elif root == 'intdivide':
            return left // right
        elif root == 'power':
            return left ** right
        elif root == 'mod':
            return left % right
```

Then the following work:

```
expr1 = OperatorExpression('plus', 'x', 2)
expr2 = expr1.subs('x', 5)
print(expr2.infixForm())
print(expr2.evaluate())
```

```
(5) + (2)
7
```

Notice the way the `subs` method seems to have three arguments, `self`, `var` and `val`, but has only two, `var` and `val`, when we call it; as always, the `self` argument is never actually used, but serves only to badge this as a method rather than a function.

Let's be *really* ambitious now.

Challenge 5: write a class called `ExpressionTree` that is capable of representing not only expressions like `(x + 1)` but also more complicated expressions like `(x + 1) ** 2 - 5`

An `ExpressionTree` is, in effect, an `OperatorExpression` whose left and right branches are both `ExpressionTrees`. So its natural structure is recursive, meaning that the secret to making it work is making all our *methods* recursive.

We'll also use a trick called **inheritance**, which allows us to set up our `ExpressionTree` class as an extension to our `OperatorExpression` class. It will then inherit any methods and attributes we don't override; in particular, we won't need to rewrite the `infixOperator` method. We ensure that this happens by making sure our opening line reads like this:

```
class ExpressionTree(OperatorExpression):
```

Now for the methods. First of all we need a new `__init__` method. The idea here is that there will be two types of `ExpressionTrees`: **parent** trees, which have left and right branches with content in them, and **childless** trees (the default), which consist of a single node. Here's the method:

```
    def __init__(self, root, left=None, right=None):
        if left==None or right==None:
            # default: childless tree, contents consist of a 1-tuple
            self.contents = (root,)
        else:
            # parent tree, contents consist of a 3-tuple
            self.contents = (root, left, right)
```

Next we want a new `prefixForm` method; this one will need to work recursively. The base case is a childless tree: if you ask for the `prefixForm` of an `ExpressionTree` whose contents are something like `(1,)` or `('x',)`, you just get the string `'1'` or `'x'`. The recursion step is a parent tree: if you ask for the `prefixForm` of anything with a `root`, a `left` and a `right`, the method calls itself on `left` and `right`.

```
    def prefixForm(self):
        "Method: returns string consisting of nested expressions"
        if len(self.contents) == 1:
            # childless tree: return the root node as a string
            return str(self.contents[0])
        else:
            # parent tree: recurse down the structure
            root, left, right = self.contents
            return \
        root+'('+left.prefixForm()+', '+right.prefixForm()+')'
```

Here's how we might use it:

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree(2)
expr4 = ExpressionTree('plus', expr1, expr2)
expr5 = ExpressionTree('power', expr4, expr3)
print(expr5.prefixForm())
```

```
power(plus(x, 1), 2)
```

To complete the definition of our new `ExpressionTree` class, the methods `infixForm`, `subs` and `evaluate` will all also need to be implemented recursively. That forms part of the exercises for this week.

This has been a tiny taste of Object-Oriented Programming (OOP) in Python. Next year's module is all about that. As a preview of that module, you may be interested in the forthcoming bonus material for this module, in which we explore OOP in greater depth. This bonus material is non-examinable.