MATH40006: An Introduction To Computation
Course notes, Section 14

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

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

14 More about objects

14.1 Attributes and properties

Here's a full listing of our `OperatorExpression` and `ExpressionTree` classes from the week before last, complete with all the methods I asked you to add in the exercises.

```
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)

    def prefixForm(self):
        "Method: returns string in the form root(left, right)"
        root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
        return root+'('+str(left)+', '+str(right)+')'

    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[0], self.contents[1], \
self.contents[2]
        return \
    '('+str(left)+') '+self.infixOperator()+' ('+str(right)+')'

    def subs(self, var, val):
        """Method: returns an OperatorExpression
        with var substituted for val"""
        root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
        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[0], self.contents[1], \
self.contents[2]
        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


class ExpressionTree(OperatorExpression):
    "A compound tree built from OperatorExpressions"

    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)

    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[0], self.contents[1], \
self.contents[2]
            return \
        root+'('+left.prefixForm()+', '+right.prefixForm()+')'

    def infixForm(self):
        """Method: returns string consisting of nested
        expressions in infix form"""
        if len(self.contents) == 1:
            return str(self.contents[0])
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return '('+left.infixForm()+') '+self.infixOperator()+' ('+\
right.infixForm()+')'
```

```
    def subs(self, var, val):
        """Method: returns an ExpressionTree
        with var substituted for val"""
        if len(self.contents) == 1:
            if self.contents[0] == var:
                return ExpressionTree(val)
            else:
                return self
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return ExpressionTree(root, left.subs(var, val), \
right.subs(var, val))

    def evaluate(self):
        "Method: returns the numerical value of an ExpressionTree"
        if len(self.contents) == 1:
            if isinstance(self.contents[0], str):
                return eval(self.contents[0])
            else:
                return self.contents[0]
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            if root == 'plus':
                return left.evaluate() + right.evaluate()
            elif root == 'subtract':
                return left.evaluate() - right.evaluate()
            elif root == 'times':
                return left.evaluate() * right.evaluate()
            elif root == 'divide':
                return left.evaluate() / right.evaluate()
            elif root == 'intdivide':
                return left.evaluate() // right.evaluate()
            elif root == 'power':
                return left.evaluate() ** right.evaluate()
            elif root == 'mod':
                return left.evaluate() % right.evaluate()
```

This is quite a nice couple of classes, but let's be a bit more ambitious. Let's set up three ExpressionTrees, corresponding to `x`, `1` and `x * 1` respectively.

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree('times', expr1, expr2)
```

If we now type

```
expr3.infixForm()
```

we get the output

```
'(x) * (1)'
```

But x times 1 is x. Wouldn't it be nice if our class performed some basic simplification in such cases?

Challenge 1: make your ExpressionTree class do some simple automatic simplification, so that $x \times 1$ automatically becomes x, $0 + a$ becomes a, etc.

14.1.1 First go: rewriting the init method

As a first stab at this, let's try rewriting the `__init__` method, so that content that can be readily simplified gets simplified at the time the class is set up. Here's a new, smarter `__init__` 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,)
        elif (
            root=='plus' and right.contents==(0,) or
            root=='subtract' and right.contents==(0,) or
            root=='times' and right.contents==(1,) or
            root=='power' and right.contents==(1,) or
            root=='intdivide' and right.contents==(1,)
        ):
            # simplify to left contents
            self.contents = left.contents
        elif (
            root=='plus' and left.contents==(0,) or
            root=='times' and left.contents==(1,)
        ):
            # simplify to right contents
            self.contents = right.contents
        elif root=='power' and left.contents==(1,):
            # simplify to 1
```

```
                self.contents = (1,)
        else:
            # parent tree, contents consist of a 3-tuple
            self.contents = (root, left, right)
```

This is fine, as far as it goes. If I now type

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree('times', expr1, expr2)
expr3.infixForm()
```

I do indeed get the output

```
'x'
```

So with this `__init__` method, our class is doing automatic simplification, where it can, at the point of **instantiation**: when a new class is set up. There's just one drawback, which is that this automatic simplification doesn't cut in when the contents of an instance are **rewritten**. If I type

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree('times', expr1, expr2)
expr4 = ExpressionTree('a')
expr5 = ExpressionTree(0)
expr3.contents = ('plus', expr4, expr5)
expr3.infixForm()
```

I might hope to get simply

```
'a'
```

but in fact I get

```
'(a) + (0)'
```

14.1.2 The general problem

The variable `contents`, which belongs to each instance of our class, is known as an **attribute**. The general problem we're trying to solve here is trying to get Python to do something clever when the value of an attribute is *set*. Rewriting the `__init__` method so that it does the clever thing feels quite natural, but actually, for the reasons we've seen, it's the wrong way to go: it works for instantiation, but not when we give an *existing* attribute a fresh value.

(If you have lots of experience with computing, but not really that much with Python, you may at this point be thinking "Ah, we need to write a setter method." Indeed we do, but the way that works in Python is highly distinctive, so stay tuned.)

The secret here is to make `contents` into a special kind of attribute called a **property**. A property consists of the following components:

- a private, “secret” attribute, that the user never accesses directly, either to set its value or to get its value;
- a **getter method**, which allows the user to get the value of the attribute, with, if necessary, some clever processing;
- a **setter method**, which allows the user to give the attribute a value, with, if necessary, some clever processing.

If the programmer wishes, she can also write a **deleter** method, but we'll skip over that.

14.1.3 Doing it right: making contents into a property

Our first move is to make the `__init__` method even simpler; absolutely everything we want to do can go into our setter method. So believe it or not, the following is fine:

```
    def __init__(self, root, left=None, right=None):
        self.contents = (root, left, right)
```

This might seem a scary thing to do, since it feels as if the `contents` atttribute will aways be a 3-tuple, and of course all our recursive methods have a base case in which it's a 1-tuple. But relax; it's fine, you'll see.

Next we have to set up our getter and setter methods. In modern versions of Python, this is done using a device called **decoration**, which uses the ‘at' character, `@`.

Here's our getter method, complete with decoration. It's also extremely simple, since we're packing all the smartness into our setter method (which we'll get to).

```
    @property
    def contents(self):
        return self.__contents
```

This belongs right at the top of our class, right below the `__init__` method.

Notice the “decoration” this lets Python know that `contents` is now a property rather than a simple attribute. Notice, too, the name `__contents`; this is the “private” attribute I was talking of, which the user never accesses directly.

Now for the business end of all this: our setter method. The trick here is to write a method that gives the private attribute `__contents` a clever value. This is really a straight copy of our smart `__init__` method from earlier, except that it's been rewritten so that it sets the value not of `contents` but of `__contents`.

```
    @contents.setter
    def contents(self, value):
        root, left, right = value
        if left==None or right==None:
            # default: childless tree, contents consist of a 1-tuple
            self.__contents = (root,)
        elif (
            root=='plus' and right.contents==(0,) or
            root=='subtract' and right.contents==(0,) or
            root=='times' and right.contents==(1,) or
            root=='power' and right.contents==(1,) or
            root=='intdivide' and right.contents==(1,)
        ):
            # simplify to left contents
            self.__contents = left.contents
        elif (
            root=='plus' and left==0 or
            root=='times' and left==1
        ):
            # simplify to right contents
            self.__contents = right.contents
        elif root=='power' and left==1:
            # simplify to 1
            self.__contents = (1,)
        else:
            # parent tree, contents consist of a 3-tuple
            self.__contents = (root, left, right)
```

Now everything works: if I type

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree('times', expr1, expr2)
expr3.infixForm()
```

I get

```
'x'
```

and if I type

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree('times', expr1, expr2)
expr4 = ExpressionTree('a')
expr5 = ExpressionTree(0)
expr3.contents = ('plus', expr4, expr5)
expr3.infixForm()
```

I now get

```
'a'
```

14.2 Instance, class and static methods

Let's leave our ExpressionTree class now, and work on something else.

Challenge 2: write a class called `Rational`, which represents a rational number as a 2-tuple of integers, `numden`. Give it a couple of methods, `num` and `den`.

Here we go:

```
class Rational(object):
    "Rational number as a 2-tuple of ints"

    def __init__(self, a, b):
        self.numden = (a, b)

    def num(self):
        return self.numden[0]

    def den(self):
        return self.numden[1]
```

This is a decent start, but it doesn't do much. To make these things behave like rational numbers, there's a list of functionality we ideally want.

- Rational numbers should always be *in their lowest terms*; we need to introduce automatic cancelling, so that `Rational(6,8)` gets stored as `(3,4)`. The right way to do this is via a setter method.
- Rational numbers should always have a *positive denominator*, so that `Rational(3,-4)` gets stored as `(-3,4)`; again, a setter method should take care of this.
- We should be able to do things with individual rational numbers, such as changing their signs, or calculating their reciprocals.
- We should be able to add, subtract, multiply and divide rational numbers.

Let's work on these in turn.

Challenge 3: convert `numden` into a property, with private attribute `__numden`, and use a setter method to implement automatic cancelling and positive denominators.

Here's the new class in its entirety.

```
class Rational(object):

    def __init__(self, a, b):
        self.numden = (a, b)

    @property
    def numden(self):
        return self.__numden

    @numden.setter
    def numden(self, value):
        from math import gcd
        a, b = value
        h = gcd(abs(a),abs(b))
        s = b//abs(b)
        self.__numden = (a*s//h,b*s//h)

    def num(self):
        return self.numden[0]

    def den(self):
        return self.numden[1]
```

Now if we test it:

```
rat1 = Rational(6,8)
print(rat1.numden)
rat2 = Rational(3,-4)
print(rat2.numden)
```

```
(3, 4)
(-3, 4)
```

14.2.1 Static methods

Challenge 4: do the same thing, but this time implement gcd yourself instead of using the gcd function from the math module.

One way to do this would be to write a free-floating *function* called gcd, but let's instead write a *method*, which will be tethered to our class, and which therefore doesn't risk clashing with existing functions called gcd from other modules (it will have a different **namespace**).

We've written methods before, but this one is a little unusual, in that it doesn't require any access to the values of the attributes of our class: our gcd method won't need to use self in any way, in other words.

Methods like that, that are bound to classes but that don't use, as their arguments, the data contained in the class, are called **static methods**. We can signal a static method using decoration. Here's our new version of the class.

```
class Rational(object):

    def __init__(self, a, b):
        self.numden = (a, b)

    @property
    def numden(self):
        return self.__numden

    @numden.setter
    def numden(self, value):
        a, b = value
        h = Rational.gcd(abs(a),abs(b))
        s = b//abs(b)
        self.__numden = (a*s//h,b*s//h)

    def num(self):
        return self.numden[0]

    def den(self):
        return self.numden[1]

    @staticmethod
    def gcd(a, b):
        while b>0:
            a, b = b, a%b
        return a
```

A few things to notice here. One is the use of the decoration `@staticmethod` to let Python know that's what it is. A second is the way the method doesn't make use of the `self` parameter–this is what makes it suitable to be a static method. And finally, note how we call it in the setter method: not simply `gcd` but `Rational.gcd`.

14.2.2 Calling the class

Challenge 5: write methods called `minus` and `reciprocal` which return, respectively, the negative of the current instance and the reciprocal of the current instance.

The obvious way to do that is to add the following code to our class:

```
    def minus(self):
        a, b = self.numden
        return Rational(-a, b)
    def reciprocal(self):
        a, b = self.numden
        return Rational(b, a)
```

And this works fine. The only slight drawback with it is that if we ever want to extend our class, say to create a class called `ComplexRational`, we might want these methods to be inherited by the new class. And if we do, we probably want the negative and the reciprocal of a `ComplexRational` rather than a `Rational`. Currently, that won't work.

To make it work, we need to make the output from `minus` and `self` not necessarily a `Rational`, but rather **whatever the current class is**. Here's how we do that:

```
    def minus(self):
        a, b = self.numden
        return self.__class__(-a, b)

    def reciprocal(self):
        a, b = self.numden
        return self.__class__(b, a)
```

Then:

```
rat1 = Rational(3,4)
print(rat1.minus().numden)
print(rat1.reciprocal().numden)
```

```
(-3, 4)
(4, 3)
```

14.2.3 Class methods

Challenge 6: write methods called `times`, `plus`, `divide` and `subtract` for `Rational` objects.

Here, we're in a slightly new situation. Our methods for `combining` `Rational` objects should, of course, produce `Rational` objects and should, like our `minus` and `reciprocal` methods, do this by referencing the current class, so that inheritance works properly (the product of two `ComplexRational` objects should generally be a `ComplexRational`, not a `Rational`).

However, unlike `minus` and `reciprocal` methods, they don't need to be an associated with a particular **instance** of a class. In that sense, they're more like the static method `gcd`.

Methods like that, which need access to the current class (like an ordinary **instance method**) but can do without access to the current instance (like a static method) are known as `class methods`. We signal them with decoration.

```
    @classmethod
    def times(cls, rat1, rat2):
        a1, b1 = rat1.numden
        a2, b2 = rat2.numden
        return cls(a1*a2,b1*b2)

    @classmethod
    def plus(cls, rat1, rat2):
        a1, b1 = rat1.numden
        a2, b2 = rat2.numden
        return cls(a1*b2+a2*b1,b1*b2)

    @classmethod
    def divide(cls, rat1, rat2):
        return cls.times(rat1, rat2.reciprocal())

    @classmethod
    def subtract(cls, rat1, rat2):
        return cls.plus(rat1, rat2.minus())
```

Notice that the class, `cls`, appears as the first argument. Like `self` for ordinary instance methods, it doesn't appear as an argument when we call the method.

Then:

```
rat1 = Rational(3,4)
rat2 = Rational(2,3)
print(Rational.times(rat1,rat2).numden)
print(Rational.plus(rat1,rat2).numden)
print(Rational.divide(rat1,rat2).numden)
print(Rational.subtract(rat1,rat2).numden)
```

```
(1, 2)
(17, 12)
(9, 8)
(1, 12)
```