More Advanced Topics

We are going to rethink the idea of fractions as rational numbers, with a new major example class Rational. First we introduce a few new features used there.

isinstance

Particularly since Python allows a variable to be associated with any type, and can change type anywhere, it is useful to check the type of an object sometimes. The Boolean function `isinstance` does this:

```
isinstance( obj, someType)
```

is true if `obj` is of type `someType` . Examples:

```
isinstance('12', int)   # False
isinstance(10+2, int)   # True
```

Also the someType can be replaced by a tuple of options. For instance Python has three numeric types, `int` , `float` , and `complex` . So

```
isinstance(val, (int, float, complex))
```

would be `True` if `val` had any of those numeric types.

Aside intended for math and engineering folks using complex numbers: An imaginary literal is an `int` or `float` literal with j or J (not i) appended. For instance `1J` is what in math is referred to as i, where i*i equals -1. A general complex literal can include a sum of a real and imaginary part, like `1 + 3.5j` .

Assert

You can spend a lot of time debugging if you made an earlier error, and it just does not made itself apparent until later. One defensive practice is to `assert` something that must be true that would later cause an error if False. For instance

```
def f(s):
    assert isinstance(s, str)
    ...
```

will cause an `AssertionError` right at the `assert` statement line if `s` is not a string (type `str`). If I were only later in the code to use a string method on s, I would get an error there, not where `'s'` was first set. It is good to see the issue immediately where `s` is set, and not have to guess how far back to look when an error is later triggered. We use assert statements liberally.

Function objects

We may have mentioned that everything in Python is an object. That includes function definitions. Earlier we said that to call a function, even with no parameters, you must include parentheses. Without parentheses the function name refers to the *function object*, which can be used as a variable and passed as a parameter. Then that parameter can be applied with the appropriate number of parameters inside of parentheses, like any function call. Consider

```
def sq(num):
   return num * num

def cube(num):
   return num ** 3

def showFuncVals(f, dataSeq):
  '''Apply function f to all items in dataSeq'''
   for x in dataSeq:
      print(f(x), end=' ')
   print()

data = [1, 3, 5]
showFuncVals(cube, data)  # pass function cube
showFuncVals(sq, data)     # pass function sq
showFuncVals(sq, range(4))     # pass function sq
```

prints

```
1 27 125
1 9 25
0 1 4 9
```

You see we pass actual parameters `sq` then `cube` *without parentheses* – function objects. They are assigned to the formal parameter `f` in `showFuncVals`, and then we can call `f` with the usual function notation, with appropriate parameter(s) in parentheses. The result depends on what function object was passed as the value for `f`.

Operator module

We use Python's standard operators all the time as special infix symbols, `2 + 3` or `5 <= 1`. Sometimes it is useful to have them as named functions, with the normal function call semantics with parenthesized parameters. The module `operator` has them all. Here is a simpler illustration, equivalent to printing the two expressions above:

```
import operator

print(operator.add(2, 3)) # prints 5
print(operator.le(5, 1)) # prints False
```

In the Rational class we will have situations that differ only in which standard binary operation to apply, and it will be convenient to pass such functions as parameters.

Underscore prefix for private

In Python, everything in an object's state is accessible, and generally changeable by other parts of running code, unlike many other programming languages. This is sometimes good, and sometimes not.

There is a naming convention for parts that you intend to be private to a particular class: Start the identifier with a single underscore.

This can be used for data or method attributes in a class. So instance attributes `_num` and `_den` would be intended to be changed directly only by code in their class's instance methods.

Python does acknowledge the privacy of a method so named in one way: There is a built-in `help` function to print out all the public doc strings for a class or module – this excludes any methods starting with a single underscore. The `help` function is mostly used as a reminder in a Python shell.

More on operators with user-defined classes

In Arithmetic Methods (ArithmeticMethods.html#arithmetic-methods) `__add__` was introduced with a simple specialized use: adding two Fractions. So if `f` and `f2` are Fractions, `f + f2` then makes sense, implicitly using `f.__add__(f2)`. Mathematically it also makes sense to add a fraction and an integer, but if `n` is an `int`, `f + n` would cause an error: `f.__add__(n)` cannot handle the parameter of `int` type. Also `n + f` would cause an error: the type of the first operand is `int`, and the `__add__` method for the built-in type `int` knows nothing about Fractions.

We can make the use of operators like `+` more flexible for user-defined types. First, with the use of `instanceof` introduced above, we can arrange to write `__add__` to handle more than one type as the parameter, including an `int`, so `f + n` would make sense. The other case, `n + f`, is trickier.

In actual fact an `__add__` method is not limited to returning a useful value or else causing an error: It can return a special object, `NotImplemented`. In that case the interpreter looks to see if the method `__radd__` for the type of the *right* operand is defined. The r suggests "right". For instance `n.__add__(f)` would return `NotImplemented`. If the Fraction class were to implement a `__radd__` method, then the interpreter would try `f.__radd__(n)`, and if that method allowed an `int` parameter, then everything would work out. We will show how this is done in the upcoming Rational class. All the other symbolic operators also can be implemented in a similar fashion. We will see many in the Rational class.

You can see that there are a number of special method names used by the system that start and end with Double UNDERscores. They are sometimes called *dunder methods* for short.

We have seen the dunder method `__str__`, which is used by the built-in function `str` to convert any object to its informal string representation. This same representation is used by `print` and in the string format method by default.

There is another variant, the dunder method `__repr__`. You might think of repr as short for represent. The convention, where possible, is for `__repr__` to return a string that could be parsed by the interpreter to represent an equal object. The global function call `repr(obj)` is implemented via `obj.__repr__()`.

A `__repr__` method for the Animal class would likely be:

```
def __repr__(self):
    return "Animal('{}')".format(self.name)
```

then we could write

```
a = Animal('Froggy')
print(repr(a)) # Animal('Froggy')
print(a)       # Animal: Froggy
```

If you use the `print` function to display a Python list object, then the elements of the list are displayed using their `repr` version. Also if you enter an expression into the Python shell to evaluate, then its `repr` version is displayed.

Static methods

Sometimes you want a function inside a class that does not needs to access instance attributes. By default any method inside a class is expected to have a first formal parameter self.

You can modify that assumption by adding the *decorator* `@staticmethod` as a line directly before the function definition.

For instance my Rational class has the `gcd` function that just is intended to take two integer parameters:

```
class Rational:

   def __init__( ... )
       ...

   @staticmethod
   def gcd(a, b):
       while b != 0:  # Euclidean algorithm
          a, b = b, a%b
       return a
```

Like any attribute of a class, `gcd` must be referenced with dot notation. For instance we could use `Rational.gcd(56, 6)`. If we had a Rational instance, like `self`, then `self.gcd(56, 6)` would be equivalent. When an instance is used before the dot for a static method, only its class is noted, not its attributes.
