Operator Overloading

Consistency

When writing new classes you need to also think of consistency

A new class should be consistent with the rules of the language.

It should respond to standard messages, it should behave properly with typical functions (assuming the type allows that kind of call).

Example: Fraction class

Consider writing a class to represent a **Fraction (rational number)**

It should respond to:

- construction
- printing
- arithmetic ops (+, -, *, /)
- comparison ops (<, >, <=, >=)

Example: Fraction class

```
f1 = Fraction(1,2)          # create the fraction 1/2
f2 = Fraction(3,2)          # create the fraction 3/2
f3 = Fraction(3)            # default denominator is 1, so
                            # really creating 3/1
```

```
f_sum = f1 + f2             # use "+" in a familiar way
print(f_sum)                # use " print " in a familiar way
4/2
```

Example: Fraction class

```
if f1 == f2:                    # use equality check "==" in a
                                # familiar way
    print('equal')
else:
    print('not equal')



print(f3 - f2)                  # combine arithmetic and
                                # printing in a familiar way
3/2
```

Introspection

Python does not have a type associated with any variable, since each variable is allowed to reference any object

- Unlike C and Java, for example, (int i)

However, we can query any variable as to what type it presently references

This is called introspection, that is, while the program is running we can determine the type a variable references

Python introspection ops

`type(variable)`

returns its type as an object

`isinstance(variable,type)`

returns a boolean indicating if the variable is of that type

Example

```
1  def special_sum(a,b):
2      ''' sum two ints or convert params to ints
3      and add. return 0 if conversion fails '''
4      if type(a)==int and type(b)==int:
5          result = a + b
6      else:
7          try:
8              result = int(a) + int(b)
9          except ValueError:
10             result = 0
11     return result
```

So what does var1+var2 mean?

So what does `var1+var2` mean?

So what does var1+var2 mean?

So what does `var1+var2` mean?

Answer: it depends

What it depends on is the type:

The + operation has two operands. What are their types?

Python uses introspection to find the type and then select the correct operator

We've seen this before

What does var1+var2 do?

- with two **strings**, we get concatenation
- with two **integers**, we get addition
- with an **integer** and a **string** we get:

```
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
 1+'a'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Operator overloading

The plus operator + is overloaded

That means the operator can do/mean different things (have multiple/overloaded meanings) depending on the types involved

If python does not recognize the operation and that combination of types, you get an error

Python overload ops

Python provides a set of operators that can be overloaded. You can't overload all the operators, but you can overload many

Like all the special class operations, they use the two underlines before and after

They come in three general classes:

- numeric type operations (+,-,<,>,print etc.)
- container operations ([ ], iterate, len, etc.)
- general operations (printing, construction)

Python special method names

Math-like Operators		
Expression	**Method name**	**Description**
x + y	`__add__()`	Addition
x − y	`__sub__()`	Subtraction
x * y	`__mul__()`	Multiplication
x / y	`__div__()`	Division
x == y	`__eq__()`	Equality
x > y	`__gt__()`	Greater than
x >= y	`__ge__()`	Greater than or equal
x < y	`__lt__()`	Less than
x <= y	`__le__()`	Less than or equal
x != y	`__ne__()`	Not equal
Sequence Operators		
len(x)	`__len__()`	Length of the sequence
x in y	`__contains__()`	Does the sequence y contain x?
x[key]	`__getitem__()`	Access element *key* of sequence x
x[key]=y	`__setitem__()`	Set element *key* of sequence x to value y
General Class Operations		
x=myClass()	`__init__()`	Constructor
print (x), str(x)	`__str__()`	Convert to a readable string
	`__repr__()`	Print a Representation of x
	`__del__()`	Finalizer, called when x is garbage collected

TABLE 12.1 Python Special Method Names

How does v1+v2 maps to _ _add_ _

```
c1 + c2
```

is converted by Python into

```
c1._ _add_ _(c2)
```

These are exactly equivalent expressions. It means that the first variable calls the _ _add_ _ method with the second variable passed as an argument

c1 is bound to `self`, c2 is bound to `param2`

Class Circle

Each circle has a radius

Adding two circles will return a circle with radius that is the sum of both radii

```
class Circle():

    def __init__(self, radius):
        self._rad = radius

    def __str__(self):
        return 'Circle with radius ' + str(self._rad)

    def __add__(self, c):
        return Circle(self._rad + c._rad)

c1 = Circle(1)
print(c1)

c2 = Circle(4)
c3 = c1 + c2
print(c3)
```

Exercise

Try out the example of a Circle

Add a method that will allow to multiply two circles – the result is a circle with radius that is the multiplication of the two radii

Class Fraction

```
class Fraction():

    def __init__(self, num, denom):
        self._num = num
        self._denom = denom

    def __str__(self):
        return str(self._num) + '/' + str(self._denom)
```

Rational number: Addition

Remember how we add fractions:

- if the denominator is the same, add the numerators
- if not, find a new common denominator that each denominator divides without remainder.
- modify the numerators and add

The lcm and gcd

The least common multiple (lcm)

- finds the smallest number that each denominator divides without remainder
- For example, lcm(3,4)=12, lcm(4,6)=12

The greatest common divisor (gcd)

- finds the largest number two numbers can divide into without remainder
- For example, gcd(3,6)=3, gcd(8,12)=4

LCM in terms of GCD

$$LCM(a,b) = \frac{a * b}{GCD(a,b)}$$

OK, how to find the gcd?

GCD: The Algorithm

One of the earliest algorithms recorded was the GCD by Euclid in his book Elements around 300 B.C.

GCD(a,b) algorithm

1. If one of the numbers is 0, return the other and halt
2. Otherwise, find the integer remainder of the larger number divided by the smaller number
3. Reapply GCD(a,b) with
   a ← smaller and
   b← the remainder from step 2

GCD: Walk-through example

Using the values from the previous example, let's apply Euclid's algorithm to find **GCD(8,20).**

Step 1 Neither number is 0; proceed.
Step 2 The integer remainder of 20/8 is 4 (i.e., 20%8 = 4).
Step 3 Reapply the algorithm using the previous smaller number (8) and the remainder (4)
to find the **GCD(8,4).**

Step 4 Neither number is 0; proceed.
Step 5 Remainder of 8/4 is 0 (i.e., 8%4 = 0).
Step 6 Reapply the algorithm to find the **GCD(4,0)**.

Step 7 One number is 0; return the other (**4**) and halt.

gcd( ) and lcm( ) functions

```
def _gcd(self, bigger, smaller):
    """Calculate the greatest common divisor of two positive integers . """
    if not bigger > smaller:
        # swap If necessary so bigger > smaller
        bigger, smaller = smaller, bigger

    while smaller != 0:    # 1. if smaller == 0, halt
        # 2. find remainder
        remainder = bigger % smaller
        bigger, smaller = smaller, remainder

    return bigger

def _lcm(self, a, b):
    """Calculate the lowest common  multiple of two positive integers . """

    # From the previous equation,
    # // ensures an int is returned
    return (a*b) // self._gcd(a,b)
```

__add__ method

```
    def __add__(self, fraction):

        # find a common denominator (lcm)
        the_lcm = self._lcm(self._denom, fraction._denom)

        # multiply each by the lcm, then add
        numerator_sum = (the_lcm * self._num / self._denom) + (the_lcm * fraction._num / fraction._denom)

        return Fraction(int(numerator_sum), the_lcm)
```

Equality

The equality method is `_ _eq_ _`
(no spaces between the underscores!)

It is invoked with the == operator

- ½ == ½ is equivalent to ½._ _eq_ _(½)

It should be able to deal with non-reduced fractions:

- ½ == ½ is True
- 2/4 == 3/6 is True

Fraction(1,2) + 1

If you try

```
f1 = Fraction (1,2)
print(f1+1)
```

you'll get an error along the lines of

```
AttributeError: 'int' object has no attribute '_denom'
```

Example: Problem!

We said the add we defined would work for two fractions, but what about

- f1 + 1 # Fraction plus an integer
- 1 + f1 # commutativity

Neither works right now. How to fix it?

First let's look at `f1 + 1`

Introspection in __add__

The **add** operator is going to have to check the types of the parameter and then decide what should be done

- If the type is an **integer**, convert it to fraction.
- If it is a **Fraction**, do what we did before.
- **Anything else** that is to be allowed needs to be checked

Modified __add__

```
def __add__(self, fract):

    if type(fract) == int:
        fract = Fraction(fract)

    if type(fract) == Fraction:
        # find a common denominator (lcm)
        the_lcm = self._lcm(self._denom, fract._denom)

        # multiply each by the lcm, then add
        numerator_sum = (the_lcm * self._num / self._denom) + (the_lcm * fract._num / fract._denom)

        return Fraction(int(numerator_sum), the_lcm)
    else:
        #for any other type the + is not defined
        print("Unknown type of parameter for +")
        raise TypeError
```

Questions?