**CEE 505**

**Jean-Luc Jackson**

**10/17/2016**

**Assignment #2**

**GeneralBase Class**

##### **Problem Statement**

1. *Expand Problem 3 from Assignment #1 to create a class GeneralBase which holds the internal variables in a dictionary as:*

*self.value = dict(base = …, value = …)*

1. *Overload math operators add, subtract, multiply, integer divide, and modulo. Assume that the two values being operated on do not have to be in the same base.*
2. *Add the following methods:*
   1. *\_\_str\_\_(self) such that:*

*>>>x = GeneralBase(16,255)*

*>>>print x*

*ff (base 16)*

* 1. *A Base() method that returns the base of an instance of the class as an integer.*
  2. *A method ChangeBase(base) that converts an instance of the class to a different base.*

1. *All functions shall accept numbers in any arbitrary base between 2 and 16 and return an object of type GeneralBase.*
2. *Provide reliable error recognition and feedback (warnings and errors).*

#### **Code Description**

##### **Implementation Details**

This code creates a *GeneralBase* class with an internal dictionary data structure containing both a value and base which are passed to the class upon instance creation (e.g. *GeneralBase(16, 255)* creates an instance with *value* = 255 and *base* = 16). This program utilizes the code written in Assignment #1, which converted values to and from binary, decimal, and hexadecimal bases. This program generalizes the previous code’s algorithms to convert between any base between 2 and 16 implemented in the new methods *FromBaseToInteger* and *FromIntegerToBase*.

The *GeneralBase* prioritizes calculations between *GeneralBase* instances over value reporting. For example, adding two *GeneralBase* instances (*a* + *b*) is prioritized over reporting the internal *base* of an instance (*print a*). This prioritization will be discussed below, but is easily summarized: an instance of *GeneralBase* stores the inputted *value* internally as a base-10 integer and only converts this *value* to the specified *base* when reporting this *value* externally, such as when the instance is called to *print*.

This code accounts for negative *values* and for a GeneralBase’s *value* being inputted in either integer form or as a string in the base matching that of the *base* input. For example, GeneralBase(16,255) can also be called as GeneralBase(16,’ff’) to produce equal instances. Error recognition includes notifying the user if a *base* larger than 16 has been inputted and if a float type *value* was used.

##### **Class Interface**

The constructor method converts the *value* passed to the class to an integer (decimal number of base 10) based on the *base* passed by the user (unless the *value* passed was already an integer). This integer *value* and *base* are saved into an internal dictionary. The dictionary saves the inputs in the order of {*base, value*} to maintain consistency with how *GeneralBase* is called. The use of the *FromBaseToInteger* function makes for low efficiency early-on with a conversion up-front, but high efficiency math operations later on. This initial function call also accounts for multiple errors, as mentioned above in Implementation Details.

Aside from the standard constructor method that creates an object of the proper class, *GeneralBase* also contains methods that return a string reporting the *value* and *base*; return a string reporting the *base*; change the *base* of the instance to a new user-specified base; and various math operations (add, subtract, multiply, integer divide, and moduo) that return a new object of class *GeneralBase* with a *value* of that calculated by the math operation and a *base* of the first value operated on. Reporting methods, such as the *\_\_str\_\_()* method, utilize the *FromIntegerToBase* function to convert the internal base-10 *value* to the specified *base* of the instance before printing said *value* as a string in the proper format. As mentioned earlier, this printing process is less efficient because the internal variable is stored in base-10 format, but allows for high efficiency math operations.

All math operations are performed in base-10 (integer values) for ease of calculation so that built-in Python functions can be utilized (+, -, \*, //, and %). Functions that return values utilize the instance’s base to convert the value to a string of the proper format (e.g. binary, octal, or hexadecimal format). For bases over 10, an internal dictionary utilizes alphabet characters from A to F for value conversions.

##### **GeneralBase Class (Python Code)**

*'''*

*Created on Oct 10, 2016*

**@author:** *jeanl*

*'''*

# import functions from Problem3 and overload necessary math operators

import Problem3 as p

from operator import \*

from operator import \_\_floordiv\_\_

class **GeneralBase**(object):

*'''*

*GeneralBase containing a base and a value, corresponding to a number in a certain base (e.g. binary, hexadecimal) with a certain value.*

*Value can be passed in either base 10 or in the base passed by the user.*

*'''*

def **\_\_init\_\_**(*self*, b = 10, v = 0):

*'''*

*Constructor*

*Value can be passed in either base 10 or in the base passed by the user.*

*'''*

*self*.value = dict(base = b, value = p.FromBaseToInteger(v,b))

def **\_\_str\_\_**(*self*):

s = *"{} (base {})"*.format(p.FromIntegerToBase(*self*.value[*'value'*], *self*.value[*'base'*]), *self*.value[*'base'*] )

return s

def **Base**(*self*):

return int(*self*.value[*'base'*])

def **ChangeBase**(*self*, NewBase ):

*self*.value.update({*"base"*:NewBase})

def **\_\_add\_\_**(*self*,y):

result = *self*.value[*'value'*] + y.value[*'value'*]

return GeneralBase(*self*.value[*'base'*], result)

def **\_\_sub\_\_**(*self*,y):

result = *self*.value[*'value'*] - y.value[*'value'*]

return GeneralBase(*self*.value[*'base'*], result)

def **\_\_mul\_\_**(*self*,y):

result = *self*.value[*'value'*] \* y.value[*'value'*]

return GeneralBase(*self*.value[*'base'*], result)

def **\_\_floordiv\_\_**(*self*,y):

result = *self*.value[*'value'*] // y.value[*'value'*]

return GeneralBase(*self*.value[*'base'*], result)

def **\_\_mod\_\_**(*self*,y):

result = *self*.value[*'value'*] - (\_\_floordiv\_\_(*self*,y).value[*'value'*] \* y.value[*'value'*])

return GeneralBase(*self*.value[*'base'*], result)

##### **FromBaseToInteger Function (Python Code)**

def **FromBaseToInteger**(value, base):

*'''*

*Used within GeneralBase to to convert incoming values of any base to integers (base 10)*

*for internal computation efficiency.*

*Currently does not handle cases where base > 16. JL20161018*

*'''*

# Stop if the base is > 16

if base > 16:

print *"Base must be within 2 and 16. Changing base from {} to 10 and proceeding with value of {}..."*.format(base,value)

base = 10

# Convert value with any base to a base10 object

# if it's already an integer, just return its originally passed value

if type(value) == int:

return value

# If user provided a float value to GeneralBase, this is the first time that we will encounter an error. Report such an error.

elif type(value) == float:

print *"FromBaseToInteger Message:"*

print *"Value cannot be float type ({:.2f}). Converting float to integer ({}) and proceeding..."*.format(value, int(value))

return int(value)

# Create dictionary to convert from any base to base10 - d must be modified for bases > 16

d = dict(a=10,b=11,c=12,d=13,e=14,f=15)

# Make sure s is string, initialize counter & collector

input = str(value)

# Initialize & Loop

i = 0

dec = 0

for n in input:

# Check if the current character is a number or letter

if d.has\_key(n):

dig = d[n] # if letter, lookup in dictionary

else:

dig = int(n) # if number, just use number

dec += dig \* pow(base, len(input) - 1 - i)

i += 1

return dec

##### **FromIntegerToBase Function (Python Code)**

def **FromIntegerToBase**(i,base):

*'''*

*Used within GeneralBase to convert internal integer "value" from dictionary*

*to necessary base value for reporting purposes.*

*This function is only called \*after\* FromBaseToInteger, so it currently does NOT handle cases where*

*the input is NOT an integer. Change as necessary. JL20161018*

*'''*

# Create DEC to HEX dictionary - d must be modified for bases > 16

d = {10:*'a'*,11:*'b'*,12:*'c'*,13:*'d'*,14:*'e'*,15:*'f'*}

# Convert input to integer value & intialize collector

v = abs(int(i))

rem = *''*

# Cover s = 0 case

if v == 0:

rem = 0

# Loop through inputted string

while v > 0:

if d.has\_key(v%base):

rem = d[v%base] + rem

else:

rem = str(v%base) + rem # add new remainder onto FRONT of string

v = v/base

# if the original input was negative, add a negative sign to the beginning of string

if int(i) < 0:

rem = *'-'* + rem

return rem

##### **Testing Procedure**

###### **Input (Python Code)**

*'''*

*Created on Oct 12, 2016*

**@author:** *jeanl*

*'''*

if \_\_name\_\_ == *'\_\_main\_\_'*:

from GeneralBase import GeneralBase

#a = GeneralBase(16,'16ced')

#b = GeneralBase(2,'11101001')

a = GeneralBase(16,*'ad4'*)

b = GeneralBase(2,*'1111011'*)

print *"Testing Format:"*

print *"(operator tested) (first value) (operator) (second value)"*

print *"(Result from GeneralBase)"*

print *"(Result from python formatting function) = python(\_)"*

print

print *"Begin Testing..."*

print

print a

print *"{:x} = python(a)"*.format(a.value[*'value'*])

print b

print *"{:b} = python(b)"*.format(b.value[*'value'*])

print

print *"Adding {} + {}"*.format(a,b)

c = a + b

print c

print *"{:x} = python(c)"*.format(c.value[*'value'*])

print

print *"Adding {} + {}"*.format(b,a)

d = b + a

print d

print *"{:b} = python(d)"*.format(d.value[*'value'*])

print

print *"Subtracting {} - {}"*.format(a,b)

e = a - b

print e

print *"{:x} = python(e)"*.format(e.value[*'value'*])

print

print *"Subtracting {} - {}"*.format(b,a)

f = b - a

print f

print *"{:b} = python(f)"*.format(f.value[*'value'*])

print

print *"Multiplying {} \* {}"*.format(a,b)

g = a\*b

print g

print *"{:x} = python(g)"*.format(g.value[*'value'*])

print

print *"Multiplying {} \* {}"*.format(b,a)

h = b\*a

print h

print *"{:b} = python(h)"*.format(h.value[*'value'*])

print

print *"Integer Dividing {} // {}"*.format(a,b)

i = a//b

print i

print *"{:x} = python(i)"*.format(i.value[*'value'*])

print

print *"Integer Dividing {} // {}"*.format(b,a)

j = b//a

print j

print *"{:b} = python(j)"*.format(j.value[*'value'*])

print

print *"Modulo {} % {}"*.format(a,b)

k = a % b

print k

print *"{:x} = python(k)"*.format(k.value[*'value'*])

print

print *"Modulo {} % {}"*.format(b,a)

l = b % a

print l

print *"{:b} = python(l)"*.format(l.value[*'value'*],)

print

print *"Code to Test per Mackenzie"*

a = GeneralBase(2,256)

b = GeneralBase(8,256)

c = GeneralBase(16,256)

d = a \* (b + c)

for i in range(2,17):

d.ChangeBase(i)

print i, d

print *"python base 2 {:b}"*.format(d.value[*'value'*])

print *"python base 8 {:o}"*.format(d.value[*'value'*])

print *"python base 16 {:x}"*.format(d.value[*'value'*])

###### **Output**

Testing Format:

(operator tested) (first value) (operator) (second value)

(Result from GeneralBase)

(Result from python formatting function) = python(\_)

Begin Testing...

ad4 (base 16)

ad4 = python(a)

1111011 (base 2)

1111011 = python(b)

Adding ad4 (base 16) + 1111011 (base 2)

b4f (base 16)

b4f = python(c)

Adding 1111011 (base 2) + ad4 (base 16)

101101001111 (base 2)

101101001111 = python(d)

Subtracting ad4 (base 16) - 1111011 (base 2)

a59 (base 16)

a59 = python(e)

Subtracting 1111011 (base 2) - ad4 (base 16)

-101001011001 (base 2)

-101001011001 = python(f)

Multiplying ad4 (base 16) \* 1111011 (base 2)

533dc (base 16)

533dc = python(g)

Multiplying 1111011 (base 2) \* ad4 (base 16)

1010011001111011100 (base 2)

1010011001111011100 = python(h)

Integer Dividing ad4 (base 16) // 1111011 (base 2)

16 (base 16)

16 = python(i)

Integer Dividing 1111011 (base 2) // ad4 (base 16)

0 (base 2)

0 = python(j)

Modulo ad4 (base 16) % 1111011 (base 2)

42 (base 16)

42 = python(k)

Modulo 1111011 (base 2) % ad4 (base 16)

1111011 (base 2)

1111011 = python(l)

Code to Test per Mackenzie

2 100000000000000000 (base 2)

3 20122210112 (base 3)

4 200000000 (base 4)

5 13143242 (base 5)

6 2450452 (base 6)

7 1054064 (base 7)

8 400000 (base 8)

9 218715 (base 9)

10 131072 (base 10)

11 8a527 (base 11)

12 63a28 (base 12)

13 47876 (base 13)

14 35aa4 (base 14)

15 28c82 (base 15)

16 20000 (base 16)

python base 2 100000000000000000

python base 8 400000

python base 16 20000