1. Algorithm

1.1 Build a class named GenBase, overload the initializing function. In the function, if the inputs are valid, which means the base is integer between 2 and 16, give the value and base input to the Genbase. If the inputs are not valid, the function can raise an input error and print out which input is invalid.

1.2 Build the toBase function to convert the Genbase[‘value’](which is a decimeter number) to the base of Genbase[‘base’], and return the string. If the given value in Genbase is negative, calculate the corresponding string of its absolute value and then add the ‘-‘ at the beginning of the string.

Build a dictionary of the corresponding relationship between integer from 0 to 15(since the biggest base is 16, the biggest number on each digit is 15) and characters from ‘0’ to ‘e’. Get the modulo of value divided by the base, and find its corresponding character in the dictionary as the last character of the string. and the value become the integer divide of value by the base. Continue the process until the value is zero.

Then we can get the string based on the GenBase base from the decimal GenBase value.

1.3 Overload the add (+), subtract (-), multiply (*), integer divide (floordiv, //), and modulo (%) functions only by doing the corresponding operations to the two decimal numbers of the two GenBase objects. And return a GenBase object with the operation result as the value, together with the base of the first GenBase as the base return.

1.4 To print the GenBase out, overload the str() function by returning the proper printout format.

1.5 In the Base function, return the base of the GenBase object.

1.6 In the ChangeBase function, return a GenBase object with the new base.

2. Test Procedure

2.1 Test if the function can raise correct input error.

When the input base is too large, the code is as follows,

```
from GenBase import *
x=GenBase(19,255)
```

and it prints out “Invalid base input”;

When the input value is not integer, the code is as follows,

```
from GenBase import *
y=GenBase(8,255.1)
```

and it prints out “Invalid value input”

Thus the class can raise the proper error regarding different invalid inputs.

2.2 Test if the function can output the right answers with proper inputs. The code and the results are shown in chapter 3.2 and chapter4.

3. Code

3.1 Class Code

```
from docutils.io import InputError
class GenBase(object):
    def __init__(self, baseIn=10, valueIn=0):
        if (baseIn>=2 and baseIn <=16 and type(baseIn)==int and type(valueIn)==int):
```

```
                self.value = dict(base=baseIn, value=valueIn)
            elif(type(valueIn)!=int):
                print "Invalid value input"
                raise InputError
            else:
                print "Invalid base input"
                raise InputError
        def toBase(self): #convert a from base 10 to self.base
            x=self.value['value']
            if x<0:
                x=-x

transdict={0:'0',1:'1',2:'2',3:'3',4:'4',5:'5',6:'6',7:'7',8:'8',9:'9',10:'a',11:'b',12:'c',13:'d',14:'e',15:'f'}
            j=x % self.value['base']
            s=transdict[j]
            x=x/self.value['base']
            while x!=0:
                j=x%self.value['base']
                s=transdict[j]+s
                x=x/self.value['base']
            if self.value['value']<0:
                s='-'+s
            return s
        def __add__(self, x):
            if type(x) == GenBase:
                sum=self.value['value']+x.value['value']
                return GenBase(self.value['base'], sum)
            else:
                raise TypeError

        def __sub__(self, x):
            if type(x) == GenBase:
                s=self.value['value']-x.value['value']
                return GenBase(self.value['base'], s)
            else:
                raise TypeError

        def __mul__(self, x):
            if type(x) == GenBase:
                s=self.value['value']*x.value['value']
                return GenBase(self.value['base'], s)
            else:
                raise TypeError
        def __floordiv__(self,x):
```

```
        if type(x) == GenBase:
            s=self.value['value']//x.value['value']
            return GenBase(self.value['base'], s)
        else:
            raise TypeError
    def __mod__(self,x):
        if type(x) == GenBase:
            s=self.value['value']%x.value['value']
            return GenBase(self.value['base'], s)
        else:
            raise TypeError
    def __str__(self):
        x=self.toBase()
        return    "%(value)s (base %(base)s)" % {'value':x, 'base': self.value['base']}
    def Base(self):
        return self.value['base']
    def ChangeBase(self,x):
        return GenBase(x,self.value['value'])
```

3.2 Test Code-1

```
from GenBase import *

x=GenBase(16,255)
y=GenBase(8,255)

z=x+y
a=y+x
b=z+a

print 'type of z: %s' % type(z)
print 'type of z: %s' % type(a)
print 'type of z: %s' % type(b)
print 'base of z is %s' % z.Base()
print 'base of a is %s' % a.Base()
print 'z is %s' % z
print 'a is %s' % a
print 'b is %s' % b
```

3.3 Test Code-2

```
from GenBase import *


x=GenBase(11,22)
y=GenBase(7,36)

print x + y
```

```
print x - y
print y - x
print x * y
print x // y
print x % y
print y // x
print y % x
print 'type of x is %s' % type(x)
print 'base of x is %s' % x.Base()
print 'base of y is %s' % y.Base()
x=x.ChangeBase(8)
print 'base of x after change is %s' % x.Base()
print 'x after change is %s'% x
```

4. Test outcome

4.1 Test outcome for Test Code-1

```
type of z: <class 'GenBase.GenBase'>
type of z: <class 'GenBase.GenBase'>
type of z: <class 'GenBase.GenBase'>
base of z is 16
base of a is 8
z is 1fe (base 16)
a is 776 (base 8)
b is 3fc (base 16)
```

4.2 Test outcome for Test Code-2

```
53 (base 11)
-13 (base 11)
20 (base 7)
660 (base 11)
0 (base 11)
20 (base 11)
1 (base 7)
20 (base 7)
type of x is <class 'GenBase.GenBase'>
base of x is 11
base of y is 7
base of x after change is 8
x after change is 26 (base 8)
```