Lab 7

COMP9021, Session 2, 2015

1 ☞ Change-making problem: greedy solution

Write a program that prompts the user for an amount, and outputs the minimal number of banknotes needed to yield that amount, as well as the detail of how many banknotes of each type value are used. The available banknotes have a face value which is one of \$1, \$2, \$5, \$10, \$20, \$50, and \$100.

Here are examples of interactions:

```
$ python question_1.py
Input the desired amount: 10

1 banknote is needed.
The detail is:
 $10: 1
$ python question_1.py
Input the desired amount: 739

12 banknotes are needed
The detail is:
$100: 7
 $20: 1
 $10: 1
  $5: 1
  $2: 2
$ python question_1.py
Input the desired amount: 35642

359 banknotes are needed
The detail is:
$100: 356
 $20: 2
  $2: 1
```

The natural solution implements a *greedy* approach: we always look for the largest possible face value to deduct from what remains of the amount.

Suppose that the available banknotes had a face value which was one of \$1, \$20, and \$50. For an amount of \$60, the greedy algorithm would not work, as it would yield one \$50 banknote and ten \$1 banknotes, so eleven banknotes all together, whereas we only need three \$20 banknotes.

2 ☞ Change-making problem: general solution

Write a program that prompts the user for the face values of banknotes and their associated quantities as well as for an amount, and if possible, outputs the minimal number of banknotes needed to match that amount, as well as the detail of how many banknotes of each type value are used.

Note: this question is flagged but in case it is used in the exam, only the minimal number of banknotes needed to match the amount would have to be computed, not the various solutions.

Here are examples of interactions:

```
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

2 : 100
50: 100

Input the desired amount: 99

There is no solution.
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1  : 30
20 : 30
50 : 30

Input the desired amount: 60
There is a unique solution:
$20: 3
```

```
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1: 100
2: 5
3: 4
10:5
20:4
30:1

Input the desired amount: 107

There are 2 solutions:
 $1: 1
 $3: 2
$10: 1
$20: 3
$30: 1

 $2: 2
 $3: 1
$10: 1
$20: 3
$30: 1
```

The natural approach makes use of the linear programming technique exemplified in the computation of the Levenshtein distance between two words.

3 ☞ A class to work exactly with fractions

Write a program that allows one to create fractions and work precisely with them, following this kind of interaction:

```
$ python
...
>>> from question_3 import Fraction
>>> f = Fraction()
Provide exactly two arguments.
>>> f = Fraction(1, 5, 12)
Provide exactly two arguments.
>>> f = Fraction(1, 'hi')
Provide an integer and a nonzero integer as arguments.
```

```
>>> f = Fraction(3, 0)
Provide an integer and a nonzero integer as arguments.
>>> f = Fraction(0, 3)
>>> f
Fraction(numerator = 0, denominator = 1)
>>> print(f)
0 / 1
>>> f = Fraction(120, -30)
>>> f
Fraction(numerator = -4, denominator = 1)
>>> print(f)
-4 / 1
>>> f = Fraction(-310532200, -46077863832)
>>> f
Fraction(numerator = 425, denominator = 63063)
>>> print(f)
425 / 63063
>>> f1 = Fraction(123, 456)
>>> f2 = Fraction(2345, 6789)
>>> f3 = f1 - f2
>>> print(f3)
-78091 / 1031928
>>> f3 = f1 * f2
>>> print(f3)
96145 / 1031928
>>> f1 / Fraction(0, 35)
Cannot divide by 0.
>>> f1 < f2
True
>>> f1 <= f2
True
>>> f1 > f2
False
>>> f1 >= f2
False
>>> f1 == f2
False
>>> f1 != f2
True
```

To find out whether the correct number of arguments has been provided, pass an argument of the form `*args` to `__init__` (the name `args` is arbitrary; what matters is the leading star, which makes `args` a tuple consisting of all positional arguments).

The implementation makes use of the following special methods (find out about them): `__repr__`, `__str__`, `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`.