Introduction to Programming Homework 3

Due Wednesday Oct 12 by 14h00

You will turn in your homework via e-mail (andrew.yarmola@uni.lu). For this homework, you will work in a text editor of your choosing. See instructions from the previous homework on how to write modules. **All of your functions in this homework MUST have docstrings.**

Exercise 1 (Box game)

Create a module `box_game.py`. At the beginning of the file, `import random`. Consider the following game from Lecture 3.

A game show has a team of `n` contestants, which are numbered `0,...,n-1`.

In the game room, there are `n` boxes, also numbered `0,..., n-1`. Each box contains one of the numbers `0,..., n-1` and no two boxes contain the same number. The game show host makes the following bet with the contestants

- the team pays €100 to play
- the numbers inside the boxes are randomly shuffled **once** for the entire game.
- one by one, contestants enter the room and are given `num_tries` chances to find their number (i.e. they get to open any `num_tries` boxes they choose, one at a time).
- after a contestant has either found their number or opened `num_tries` boxes, the room is reset just as it was before the contestant went inside
- if all the contestants find their number, the game show will award the team €3000, however, if any contestant has failed to find their number, the game show keeps all the money

Players can decide on a strategy ahead of time, but cannot communicate once the game begins.

- **a.** Write a `individual_strategy` function that has three inputs :
  - the list of boxes for a given round
  - the number of boxes allowed to open (i.e. `num_tries`)
  - the contestant's number to look for

  Your function should return `True` if the number is found, and `False` otherwise.

- **b.** Write a function called `play_game` to play the game multiple times and return the **victory rate** (wins/rounds) with a given strategy. The input of `play_game` should be :
  - the number of contestants
  - the number of boxes allowed to open (i.e. `num_tries`)
  - a strategy function which will be used for every contestant
  - the number of rounds the game should be played, which should **default to 1**

    In between each round, be sure to **randomly shuffle the box contents**.

- **c.** Improve your `individual_strategy` so that `box_game.play_game(10, 5, box_game.individual_strategy, 10000) > 0.3` will (usually) be true. Here the number of contestants is 10, number of tries is 5, and the number of rounds is 10000. Don't hesitate to email me if you need a hint!

Exercise 2 (Base 2 continued)

Start with your module `base_2.py` from Homework 2. You are welcome to make updates to your code from the Homework 2 Solutions. You should use the functions you wrote in Homework 2 in your answers below.

Notice that a `float` has an instance method called `.as_integer_ratio()`. Given a float `x`, the return value of `x.as_integer_ratio()` is a `tuple` of the form `(m,n)` such that `n` is power of `2` and `x == m/n` **as floating point numbers**. That is, in the computer representation, $x = m/n$ mathematically. For example

In [5]:

```
x = 1.42
as_ratio = x.as_integer_ratio()
print(as_ratio)
print(x == as_ratio[0]/as_ratio[1])
```

```
(799388933858263, 562949953421312)
True
```

- **a.** Write a function called `float_repr` which takes a floating point number `x` and returns a tuple `(c,q)` such that $x = c \cdot 2^q$ mathematically. For example `base_2.float_repr(0.1)` can return `(3602879701896397,-55)`. Note, depending on the float, there might be several valid choices for `c` and `q`.

- **b.** Write a function called `float_repr_54` which takes a floating point number `x` and returns a tuple `(c,q)` such that $x = c \cdot 2^q$ mathematically **and `base_2.bits_needed(c)` is 54 or less**. For example, `base_2.float_repr_54(123192210012943262.)` can return the tuple `(7699513125808954, 4)`.
    - Hint : python internally uses at most 54 bits to store `c`, so you should always be able to recover this from `m` and `n`.

Remark

You can see that python uses 53+1 bits from the following test.

In [1]:

```
under = float(2**53-1)
over = float(2**53+1)
print("{0:d} is clearly the ratio of {1}".format(2**53-1,
                                              under.as_integer_ratio()))
print("{0:d} is clearly NOT the ratio of {1}".format(2**53+1,
                                              over.as_integer_ratio()))
```

```
9007199254740991 is clearly the ratio of (9007199254740991, 1)
9007199254740993 is clearly NOT the ratio of (9007199254740992, 1)
```

In particular, the closest floating point to the integer `2**53+1` is the floating point number `2**53`.

Exercise 3 (Permutations continued)

Start with your module `perms.py` from Homework 2. You are welcome to make updates to your code from the Homework 2 Solutions.

In this exercise, you will write code to convert between permutations in **function representation** (using tuples as in the previous homework) and permutations in **cycle representation** (see [https://en.wikipedia.org/wiki/Permutation#Cycle_notation]). Our permutations will again permute the set {0,...,n-1}

To model the cycle representation of a permutation, we will use a **tuple of tuples**. For example, consider the permutation σ = (0 3)(1 2 4) in **mathematical cycle representation**. This corresponds to the map f_σ : {0,...,4} → {0,...,4} with

$$f_\sigma(0) = 3,\ f_\sigma(1) = 2,\ f_\sigma(2) = 4,\ f_\sigma(3) = 0,\ f_\sigma(4) = 1.$$

So in **function representation**, we write σ as the tuple `(3,2,4,0,1)`. For the **cycle representation**, we will use `((0,3),(1,2,4))`. As you can see, this is a tuple of tuples in python.

For cycle representations with just **one** cycle, we write `((a_1, a_2,...,a_k),)` (note the ,). The identity permutation is therefore just `((),)`.

- **a.** Write the following functions :
  - `valid_disjoint_cycle_rep` which takes a tuple of tuples (of integers) and returns `True` if the supplied input represents a mathematical **disjoint** cycle representation of some permutation of {0,1,...}. Returns `False` otherwise.
  - `min_perm_size` which takes a tuple of tuples (of integers) and returns the minimal permutation size (i.e. `n`) if the input is a valid disjoint cycle representation. If the input is **not** valid, **instead of a `return` statement**, use

    ```
    raise SyntaxError("Bad disjoint cycle representation")
    ```

- **b.** Write the following functions :
  - `cycle_rep_from_func` which takes a function representation and returns the disjoint cycle representation the corresponding permutation. If the input is not in function representation, `raise SyntaxError("Bad function representation")`
  - `func_from_disjoint_cycle_rep` which takes a disjoint cycle representation and returns the function representation of the corresponding permutation. If the input is not in disjoint cycle representation, `raise SyntaxError("Bad disjoint cycle representation")`.

- **c.** Disjoint cycle representations have a **canonical** form where each cycle is sorted from least to greatest and the cycles themselves are sorted by their smallest elements. Write a function `canonical_disjoint_rep` which takes a tuple of tuples and returns the **canonical** cycle representation if the input is a valid disjoint cycle representation. On bad input, `raise SyntaxError("Bad disjoint cycle representation")`
  - Hint : learn about the optional `key` argument to the function `sorted` (see [https://docs.python.org/3/howto/sorting.html#sortinghowto]).

Exercise 4 (Primes)

Write a module called `primes.py` with the function `primes_less_than` which takes a positive integer `n` and returns the list of primes less than `n`. If `n < 2`, returns the empty list.