Introduction to Programming Homework 3 Solutions

Note : There are (many) diffenrent ways to answer many of these questions. I have placed the solutions in-line with the questions, however, they should be contaiend in their own module files. I also provided alternate solutions in Lecture 3.

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.

In [13]:

```python
def individual_strategy(boxes, num_tries, num_to_find) :
    """ Implements the cycle strategy for a player to
    look for their number (num_to_find) in a list (boxes_list) with
    a limited number of tries (num_tries). The function
    returns True if the number is found, False otherwise.
    The algorithm treats boxes[i] as a permtuation and
    looks through the cycle containing i = person. """
    count = 0
    idx = num_to_find
    while count < num_tries :
        curr = boxes[idx]
        if curr == num_to_find :
            return True
        idx = curr
        count += 1
    return False
```

- **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**.

In [14]:

```python
import random
def play_game(num_ppl, num_tries, strategy, num_runs = 1) :
    """ A tool to simulate many random rounds of the box game.
    Parameters
    ----------
    num_ppl : int, required
        Number of contestants.
    num_tries : int, required
        Number of tries each contestant has to find their box.
    strategy : function(boxes, num_tries, person), required
        A strategy function to be used for each contestant.
    num_runs : int, optional
        Number of times to play the game. Defaults to 1.

    Returns
    ----------
        rate of victory with given strategy function
    """
    boxes = list(range(num_ppl))
    ppl = list(range(num_ppl))
    wins = 0
    games_played = 0
    while games_played < num_runs :
        # Shuffle the boxes once per game
        random.shuffle(boxes)
        result = True
        for person in ppl :
            result = strategy(boxes, num_tries, person)
            if result is False : # we have lost !!!
                break # stops the inner for loop
        # remember that int(True) = 1,
        # and int(False) = 0
        wins += result
        games_played += 1
    # we have played all the games
    return wins/num_runs
```

- **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!

In [15]:

```python
play_game(10, 5, individual_strategy, 10000) > 0.3
```

Out[15]:

```
True
```

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 [16]:

```
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`.

In [17]:

```
def float_repr(x) :
    """ Given a float x, returns a pair of ints (c,q)
    such that x = c*2**q exactly.  """
    # if a function returns a tuple, you can read off
    # the multiple inputs using this syntax
    m, n = x.as_integer_ratio()
    # since n is a power of 2
    q_minus =  bad_log_base_2(n)
    if q_minus < 0 :
        raise RuntimeError("Unexpected denominator in "
                           "float.as_integer_ratio().")
    return (m,-q_minus)
```

- **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`.

In [18]:

```
def float_repr_54(x) :
    """ Given a float x, returns a pair of ints (c,q)
    such that x = c*2**q exactly and c has 54 or less
    bits of percision, including sign.  """
    c, q = float_repr(x)
    c_bits = bits_needed(c)
    while c_bits > 54 :
        if c % 2 != 0 :
            raise RuntimeError("Numerator not a multiple of 2.")
        c //= 2 # Use integer division!
        q += 1
        c_bits -= 1
    return (c,q)
```

Remark

You can see that python uses 54 bits from the following test.

In [6]:

```
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")
    ```

In [19]:

```
def valid_disjoint_cycle_rep(cycles) :
    """ Returns True if the input is a tuple of tuples representing
    a disjoint cycle decomposition of a permutation. """
    if type(cycles) is not tuple or len(cycles) == 0 : return False
    # we pull all the values out of cycles and then see if
    # they have no repeats and are in the right range
    values = []
    for c in cycles :
        if type(c) is not tuple : return False
        for x in c :
            if type(x) is not int : return False
            values.append(x)
    values = sorted(values)
    expected_range = range(max(values))
    for i in range(len(values)-1):
        if values[i] not in expected_range or \
          values[i] == values[ i + 1 ] :
           return False
    return True
```

```
    return True

def max_or_minusone(t) :
    """ Tries to returns max(t) ,
    or -1 if max(t) fails. """
    # if you want, you could also just check len(t) > 0
    try :
        return max(t)
    except :
        return -1

def min_perm_size(cycles) :
    """ Returns the minimum permutation size given a tuple
    of tuples representing a disjoint cycle decomposition.
    Raises a SyntaxError on bad input. """
    if not valid_disjoint_cycle_rep(cycles) :
        raise SyntaxError("Bad disjoint cycle representation")
    else :
        return max(map(max_or_minusone, cycles)) + 1

# A more efficient version of valid_disjoint_cycle_rep using sets
def valid_disjoint_cycle_rep_v2(cycles) :
    """ Returns True if the input is a tuple of tuples representing
    a disjoint cycle decomposition of a permutation. """
    if type(cycles) is not tuple or len(cycles) == 0 : return False
    # keep track of seen integers
    seen = set()
    for c in cycles :
        if type(c) is not tuple : return False
        for i in c :
            if type(i) is not int or i < 0 :
                return False
            elif i in seen :
                return False
            else :
                seen.add(i)
    return True
```

- **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")`.

In [11]:

```
def func_from_disjoint_cycle_rep(cycles) :
    """ Returns the function representation of a permutation
    given a tuple of tuples representing a disjoint cycle
    decomposition. Raises a SyntaxError on bad input. """
    # this will validate input, so I don't have to here
    perm_size = min_perm_size(cycles)
    # start with idenity and modify
```

```
    # start with identity and modify
    f = list(range(perm_size))

    for cycle in cycles :
        cycle_len = len(cycle)
        for i in range(cycle_len) :
            f[cycle[i]] = cycle[ (i+1) % cycle_len ]
    return tuple(f)

def cycle_rep_from_func(f) :
    """ Returns the disjoint cycle representation of a permutation
    given a tuple as a function reprepresentation.
    Raises a SyntaxError on bad input. """
    if not is_perm(f) :
        raise SyntaxError("Bad function representation")
    # we use seen to keep track of the values visited
    cycle_rep = []
    n = len(f)
    seen = [0]*n
    while 0 in seen :
        # find the first non-visited int
        c_start = seen.index(0)
        cycle = [c_start]
        seen[c_start] = 1
        c_next = f[c_start]
        while c_next != c_start :
            cycle.append(c_next)
            seen[c_next] = 1
            c_next = f[c_next]
        if len(cycle) > 1 :
            cycle_rep.append(tuple(cycle))
    if len(cycle_rep) > 0 :
        return tuple(cycle_rep)
    else :
        return ((),)

# Again, sets make things a little shorter,
# cleaner and easier to read
def cycle_rep_from_func_v2(f) :
    """ Returns the disjoint cycle representation of a permutation
    given a tuple as a function reprepresentation.
    Raises a SyntaxError on bad input. """
    if not is_perm(f) :
        raise SyntaxError("Bad function representation")
    # we use unseen to keep track of the values not visited
    cycle_rep = []
    n = len(f)
    unseen = set(range(n))
    while len(unseen) > 0 :
        c_start = unseen.pop()
        cycle = [c_start]
        c_next = f[c_start]
        while c_next != c_start :
            cycle.append(c_next)
            unseen.discard(c_next)
            c_next = f[c_next]
        if len(cycle) > 1 :
            cycle_rep.append(tuple(cycle))
    if len(cycle_rep) > 0 :
```

```
        return tuple(cycle_rep)
    else :
        return ((),)
```

- **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]).

In [12]:

```
def get_first(x) :
    """ Returns the fist element of x. Assumes x is not empty. """
    return x[0]

def get_second(x) :
    """ Returns the second element of x. Assumes x as at
        least 2 elements. """
    return x[1]

def rot_to_min(x) :
    """ Given a list or tuple x, returns its cyclical rotation
        starting with the minumum value.
        Return type is the same as x. """
    # enumerate(x) returns an iterable of paits (idx, val)
    min_idx, x_min = min(enumerate(x), key = get_second)
    # return the rotated version
    return x[min_idx:] + x[:min_idx]

def canonical_disjoint_rep(cycles) :
    """ Returns the canonical cycle decomposition given a tuple
        of tuples representing a disjoint cycle decomposition. All
        empty tuples and singletons are removed. Raises a
        SyntaxError on bad input. """
    if not valid_disjoint_cycle_rep(cycles) :
        raise SyntaxError("Bad disjoint cycle representation")
    # the identity is already canonical
    if cycles == ((),) : return cycles
    # delete any empty or singleton tuples in p
    clean_cycles = [ c for c in cycles if len(c) > 1 ]
    # now we can cyclically rotate each tuple in p
    inner_sorted = map(rot_to_min, clean_cycles)
    # sorting the lists in the inner_sorted iterator
    outer_sorted = sorted(inner_sorted, key = get_first)
    return tuple(outer_sorted)
```

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.

In [10]:

```
def primes_less_than(n) :
    """ Given an integer n, returns the
    list of primes less than n. """
    if type(n) is not int or n < 3 : return []
    # We use a sieve algorithm to
    # elimiane all multiples of
    # numbers in increasing order
    primes = [2]
    sieve = list(range(3,n,2)) # all odds less than n
    # We will remove all elemnts of the form
    # prime*(prime + 2k) wheren k = 0,1,...
    while len(sieve) > 0 :
        prime = sieve.pop(0)
        primes.append(prime)
        for x in range(prime**2, n, 2 * prime) :
            if x in sieve :
                sieve.remove(x)
    return primes


# Here is a more efficient way using sets and some
# clever(er) math

def int_sqrt(n) :
    """ If n > 0, returns the larges x with x**2 <= n. """
    assert n > 0
    # smallest integer less than n
    if type(n) is not int :
        m = int(n//1)
    else :
        m = n
    # We use Newton's method for the function
    # f(x) = x^2 - n. We start with x_0 = int(n)+1, and apply
    # x_{k+1} = x_k - f(x_k)/f'(x_k) = (x_k + n/x_k)/2
    prev, curr = 0, m
    while True:
        prev, curr = curr, (curr + m // curr) // 2
        # Notice that (curr + n//curr) // 2 is at most
        # 1 less than (curr + n/curr)/2.
        # Thus, by convexity, the first time we are
        # f(curr) is negative, we have out answer
        if curr**2 <= m :
            return curr

def primes_less_than_v2(n) :
    """ Given an integer n, returns the list of primes less than n. """
    if type(n) is not int or n < 3 : return []
    # We start wil all odd numbers less than n
    candidates = set(range(3,n,2))
```

```
    candidates = set(range(3,n,2))
    # For odd number x = 3, 5, ..., int_sqrt(n), we will
    # remove x^2, x(x+2), x(x+4),... < n from candidates.
    # Notice that we are removing only the odd ones.
    # Why this works : assume y = a*b with 1 < a <= b
    # and y < n, then a < int_sqrt(n), so y will be eliminated.
    for i in range(3, int_sqrt(n) + 1, 2) :
        candidates.difference_update(range(i**2, n, 2*i))
    candidates.add(2)
    return sorted(candidates)
```