Data Abstraction

Announcements

Recursion and Strings

Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return ____0____
    elif n == 0:
        return ____1____
    else:
        return count_park(n-1) + count_park(n-1) + count_park(n-2)
```

```
count_park(3):
  %%%
  %%.
  %.%
  %..
  %<>
  ----
  .%%
  .%.
  ..%
  ...
  .<>
  ----
  <>%
  <>.
```

Spring 2023 Midterm 2 Question 5(b) [modified a lot]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **park**, which returns a list of all the ways, represented as strings, that vehicles can be parked in n adjacent parking spots for positive integer n. Spots can be empty.

```
def park(n):
    """Return the ways to park cars and motorcycles in n adjacent spots.
    >>> park(1)
    ['%', '.']
    >>> park(2)
    ['%%', '%.', '.%', '..', '<>']
    >>> len(park(4))  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return []
    elif n == 0:
        return ['']
    else:
        return ['%'+s for s in park(n-1)] + ['.'+s for s in park(n-1)] + ['<>'+s for s in park(n-2)]
```

```
park(3):
%%%
%%.
%.%
%..
%<>
---
.%%
.%.
..%
...
.<>
---
<>%
<>.
```

Manipulating Lists

The Most Important Operations on a List of Numbers

```
>>> s = [5, 7, 9, 11]  # Make a list using a list literal
>>> s[0]               # Get the first element using item selection
5
>>> s[1:]              # Get the rest using slicing
[7, 9, 11]
>>> [3] + s            # Make a longer list using addition
[3, 5, 7, 9, 11]
```

Discussion 4

Max Product

Write a function that takes in a list and returns the maximum product that can be formed using non-consecutive elements of the list. All numbers in the input list are greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be
    formed using non-consecutive elements of s.

    >>> max_product([10, 3, 1, 9, 2]) # 10 * 9
    90
    >>> max_product([5, 10, 5, 10, 5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
    if len(s) == 0:
        return 1
    elif len(s) == 1:
        return s[0]
    else:
        return max(s[0] * max_product(s[2:]), max_product(s[1:]))
```

A tip for finding a recursive process:

1. Pick an example: s = [5, 10, 5, 10, 5]
2. Write down what recursive calls will do:
   - max_product([10, 5, 10, 5]) → 10 * 10
   - max_product([5, 10, 5]) → 5 * 5
   - max_product([10, 5]) → 10
   - max_product([5]) → 5
3. Which one helps build the result?

Either include s[0] but not s[1], OR
Don't include s[0]

Choose the larger of:
multiplying s[0] by the max_product of s[2:] (skipping s[1]) OR
just the max_product of s[1:] (skipping s[0])

Sum Fun

Implement **sums(n, m)**, which takes a total **n** and maximum **m.** It returns a list of all lists:

- that sum to n,
- that contain only positive numbers up to m, and
- in which no two adjacent numbers are the same.

```
>>> sums(5, 3)
[[1, 3, 1], [2, 1, 2], [2, 3], [3, 2]]
>>> sums(5, 5)
[[1, 3, 1], [1, 4], [2, 1, 2], [2, 3], [3, 2], [4, 1], [5]]
```

[1, 3, 1] = [1] + [3, 1]
[2, 1, 2] = [2] + [1, 2]
[2, 3] = [2] + [3]
[3, 2] = [3] + [2]
~~[1, 1, 3]~~ = [1] + [1, 3]
~~[1, 2, 2]~~ = [1] + [2, 2]

```
def sums(n, m):
    if n < 0:
        return []
    if n == 0:
        sums_to_zero = []     # The only way to sum to zero using positives
        return [sums_to_zero] # Return a list of all the ways to sum to zero
    result = []
    for k in range(1, m + 1):
        result = result + [[k]+rest for rest in sums(n-k,m) if rest == [] or k != rest[0]]
    return result
```

Dictionaries

```
{'Dem': 0}
```

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

```
Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

Example: Multiples

Implement **multiples,** which takes two lists of positive numbers **s** and **factors.** It returns a dictionary in which each element of factors is a key, and the value for each key is a list of the elements of **s** that are mulitples of the key.

```
def multiples(s, factors):
    """Create a dictionary where each factor is a key and each value
    is the elements of s that are multiples of the key.

    >>> multiples([3, 4, 5, 6, 7, 8], [2, 3])
    {2: [4, 6, 8], 3: [3, 6]}
    >>> multiples([1, 2, 3, 4, 5], [2, 5, 8])
    {2: [2, 4], 5: [5], 8: []}
    """

    return {d: [x for x in ____s____ if __x % d == 0__] for d in __factors__}
```

Data Abstraction

Data Abstraction

A small set of functions enforce an abstraction barrier between ***representation*** and ***use***

- How data are represented (as some underlying list, dictionary, etc.)
- How data are manipulated (as whole values with named parts)

E.g., refer to the parts of a line (affine function) called f:

- `slope(f)` instead of `f[0]` or `f['slope']`
- `y_intercept(f)` instead of `f[1]` or `f['y_intercept']`

Why? Code becomes easier to read & revise.

(Demo)