CS 61A
Fall 2025

Generators

Discussion 6: October 8, 2025

If there are fewer than 3 people in your group, merge your group with another group in the room. If your group has 6 or more students, you're welcome to split into two sub-groups and then sync up at the end. If you want two separate Pensieve documents for the two sub-groups, just have one sub-group add 1000 to their group number.

Now switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

If you didn't attend for a good reason (such as being sick), fill out this form (within 2 weeks of your discussion): attendance form

Getting Started

Say your name and share a favorite place on the Berkeley campus or surrounding city that you've discovered. Try to pick a place that others might not have been yet. (But if the room you're in now is your favorite place on campus, that's ok too.)

McCone Hall has a nice view from the 5th floor balcony.

This emoticon of a guy in a cowboy hat is valid Python: `o[:-D]`

```
>>> o = [2, 0, 2, 5]
>>> [ o[:-D] for D in range(1,4) ]
[[2, 0, 2], [2, 0], [2]]
```

Generators

A *generator* is an *iterator* that is returned by calling a *generator function*, which is a function that contains `yield` statements instead of `return` statements. The ways to use an *iterator* are to call `next` on it or to use it as an iterable (for example, in a `for` statement).

Q1: Big Fib

This generator function yields all of the Fibonacci numbers.

```
def gen_fib():
    n, add = 0, 1
    while True:
        yield n
        n, add = n + add, n
```

Explain the following expression to each other so that everyone understands how it works. (It creates a list of the first 10 Fibonacci numbers.)

```
(lambda t: [next(t) for _ in range(10)])(gen_fib())
```

Then, complete the expression below by writing only names and parentheses in the blanks so that it evaluates to the smallest Fibonacci number that is larger than 2025.

Talk with each other about what built-in functions might be helpful, such as `map`, `filter`, `list`, `any`, `all`, etc. (Click on these function names to view their documentation.) Try to figure out the answer without using Python. Only run the code when your group agrees that the answer is right. This is not the time for guess-and-check.

```
def gen_fib():
    n, add = 0, 1
    while True:
        yield n
        n, add = n + add, n

____lambda n: n > 2025, ____
```

One solution has the form: `next(____(lambda n: n > 2025, ____))` where the first blank uses a built-in function to create an iterator over just large numbers and the second blank creates an iterator over all Fibonacci numbers.

Q2: Something Different

Implement `differences`, a generator function that takes `t`, a non-empty iterator over numbers. It yields the differences between each pair of adjacent values from `t`. If `t` iterates over a positive finite number of values `n`, then `differences` should yield `n-1` times.

```
def differences(t):
    """Yield the differences between adjacent values from iterator t.

    >>> list(differences(iter([5, 2, -100, 103])))
    [-3, -102, 203]
    >>> next(differences(iter([39, 100])))
    61
    """
    "*** YOUR CODE HERE ***"
```

Add to the following implementation by initializing and updating `previous_x` so that it is always bound to the value of `t` that came before `x`.

```
for x in t:
    yield x - previous_x
```

Discussion Time. Work together to explain why `differences` will always yield `n-1` times for an iterator `t` over `n` values. If you get stuck, ask a TA for help.

Intermission

We're lazy (like an iterator) and used ChatGPT to generate a generator joke…

Because it was skilled at knowing when to "return" to the recipe and when to "yield" to improvisation!

Switch roles: Whoever in your group helped type the answer to the last question should not type the answer to the next one. Instead, just ask questions and give suggestions; let other group members type the answer.

Q3: Partitions

Tree-recursive generator functions have a similar structure to regular tree-recursive functions. They are useful for iterating over all possibilities. Instead of building a list of results and returning it, just `yield` each result.

You'll need to identify a *recursive decomposition*: how to express the answer in terms of recursive calls that are simpler. Ask yourself what will be yielded by a recursive call, then how to use those results.

Definition. For positive integers `n` and `m`, a *partition* of `n` using parts up to size `m` is an addition expression of positive integers up to `m` in non-decreasing order that sums to `n`.

Implement `partition_gen`, a generator functon that takes positive `n` and `m`. It yields the partitions of `n` using parts up to size `m` as strings.

Reminder: For the `partitions` function we studied in lecture (video), the recursive decomposition was to enumerate all ways of partitioning `n` using at least one `m` and then to enumerate all ways with no `m` (only `m-1` and lower).

Hint: For the base case, yield a partition with just one element, `n`. Make sure you yield a string.

Hint: The first recursive case uses at least one `m`, and so you will need to yield a string that starts with `p` but also includes `m`. The second recursive case only uses parts up to size `m-1`. (You can implement the second case in one line using `yield from`.)

```
def partition_gen(n, m):
    """Yield the partitions of n using parts up to size m.

    >>> for partition in sorted(partition_gen(6, 4)):
    ...     print(partition)
    1 + 1 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 3
    1 + 1 + 2 + 2
    1 + 1 + 4
    1 + 2 + 3
    2 + 2 + 2
    2 + 4
    3 + 3
    """
    assert n > 0 and m > 0
    if n == m:
        yield ____
    if n - m > 0:
        "*** YOUR CODE HERE ***"



    if m > 1:
        "*** YOUR CODE HERE ***"
```

Discussion Time. Work together to explain why this implementation of `partition_gen` does not include base cases for `n < 0`, `n == 0`, or `m == 0` even though the original implementation of `partitions` from lecture (video) had all three.

Optional Question

Generator problems often appear on exams and often include recursion.

Hint: the statement `yield from t` is identical to the following:

```
for x in t:
    yield x
```

Q4: Squares

Implement the generator function `squares`, which takes **positive** integers `total` and `k`. It yields all lists of perfect squares greater or equal to `k*k` that sum to `total`. Each list is in non-increasing order (large to small).

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
def squares(total, k):
    """Yield the ways in which perfect squares greater or equal to k*k sum to total.

    >>> list(squares(10, 1))  # All lists of perfect squares that sum to 10
    [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [4, 1, 1, 1, 1, 1, 1], [4, 4, 1, 1], [9, 1]]
    >>> list(squares(20, 2))  # Only use perfect squares greater or equal to 4 (2*2).
    [[4, 4, 4, 4, 4], [16, 4]]
    """
    assert total > 0 and k > 0
    if total == k * k:
        yield ____
    elif total > k * k:
        for s in ____:
            yield ____
        yield from squares(total, k + 1)
```