CS 61A Spring 2026

Iterators, Generators

Discussion 6: March 4, 2026

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.

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 2026.

Talk with each other about what built-in functions might be helpful, such as `map`, `filter`, `list`, `any`, `all`, etc.

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

________lambda n: n > 2026, ________
```

Q2: Repeated

Implement `repeated`, which takes in an iterator `t` and an integer `k` greater than 1. It returns the first value in `t` that appears `k` times in a row.

Important: Call `next` on `t` only the minimum number of times required. Assume that there is an element of `t` repeated at least `k` times in a row.

Hint: If you are receiving a `StopIteration` exception, your `repeated` function is calling `next` too many times.

```
def repeated(t, k):
    """Return the first value in iterator t that appears k times in a row,
    calling next on t as few times as possible.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> t = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(t, 3)
    8
    >>> u = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(u, 3)
    2
    >>> repeated(u, 3)
    5
    >>> v = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(v, 3)
    2
    """
    assert k > 1
    "*** YOUR CODE HERE ***"
```

Q3: 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 ***"
```

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.

Q4: 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).

```
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
```

Q5: 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).

```
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)
```

Q6: Church Generator

Implement `church_generator`, a generator function that takes in a function `f` as an argument. `church_generator` yields functions that apply `f` to their argument one more time than the previously generated function. (The yielded functions are known as Church numerals.)

```
def church_generator(f):
    """Takes in a function f and yields functions which apply f
    to their argument one more time than the previously generated
    function.

    >>> increment = lambda x: x + 1
    >>> church = church_generator(increment)
    >>> for _ in range(5):
    ...     fn = next(church)
    ...     print(fn(0))
    0
    1
    2
    3
    4
    """

    g = ______________________________________________
    while True:
        ______________________________________________
        ______________________________________________
```