CS 61A Summer 2025 — Mutability, Iterators, Generators

Discussion 5: July 10, 2025

Mutability

There are many built-in methods that we can use to mutate lists. Here are some of the most useful ones:

- `append(el)`: Appends `el` to the end of the list and returns `None`.
- `extend(lst)`: Extends the list by concatenating it with `lst` and returns `None`.
- `remove(el)`: Removes the first occurence of `el` from the list and returns `None`.
- `insert(i, el)`: Inserts `el` at index `i` and returns `None`.
- `pop(i)`: Removes and returns the element at index `i`. If no index is given, removes and returns the last element in the list.

Let's see these methods in action:

```
>>> l = [3, 5, 6]
>>> l.append(10)    # Append 10 to the end of the list
>>> l
[3, 5, 6, 10]
>>> l.extend([30, 40])
>>> l
[3, 5, 6, 10, 30, 40]
>>> l.remove(5)     # Remove the first occurrence of 5
>>> l
[3, 6, 10, 30, 40]
>>> l.insert(2, -2) # Insert -2 at index 2
>>> l
[3, 6, -2, 10, 30, 40]
>>> l.pop()         # Remove and return the last element
40
>>> l
[3, 6, -2, 10, 30]
>>> l.pop(2)        # Remove and return the element at index 2
-2
>>> l
[3, 6, 10, 30]
```

Take note of two things:

- The name `l` refers to the same list object during this entire session; it is never reassigned. The reason the output looks different each time we call a method is because the list that `l` evaluates to is being *mutated*.
- The only method here that has a return value is `pop`! All of the other methods return `None`.

Q1: Copying Copies

Draw the environment diagram on paper or a tablet (without having the computer draw it for you)! Then, check your work by stepping through the diagram with PythonTutor

```
def chain(g):
    g(True, g)

def add_copy(p, then):
    copy = result
    if p:
        copy.append(1)
        result.append(list(copy))
        return then(not p, add_copy)
    else:
        copy.append(2)

result = [5]
chain(add_copy)
print(result)
```

See the web version of this resource for the environment diagram.

Iterators

An **iterable** is any value that can be iterated through, or gone through one element at a time. One construct that we've used to iterate through an iterable is a for statement:

```
for elem in iterable:
    # do something
```

In general, an **iterable** is an object on which calling the built-in `iter` function returns an *iterator*. An **iterator** is an object on which calling the built-in `next` function returns the next value.

For example, a list is an iterable value.

```
>>> s = [1, 2, 3, 4]
>>> next(s)       # s is iterable, but not an iterator
TypeError: 'list' object is not an iterator
>>> t = iter(s)   # Creates an iterator
>>> t
<list_iterator object ...>
>>> next(t)       # Calling next on an iterator
1
>>> next(t)       # Calling next on the same iterator
2
>>> next(iter(t)) # Calling iter on an iterator returns itself
3
>>> t2 = iter(s)
>>> next(t2)      # Second iterator starts at the beginning of s
1
>>> next(t)       # First iterator is unaffected by second iterator
4
>>> next(t)       # No elements left!
StopIteration
>>> s             # Original iterable is unaffected
[1, 2, 3, 4]
```

You can also use an iterator in a `for` statement because all iterators are iterable. But note that since iterators keep their state, they're only good to iterate through an iterable once:

```
>>> t = iter([4, 3, 2, 1])
>>> for e in t:
...     print(e)
4
3
2
1
>>> for e in t:
...     print(e)
```

There are built-in functions that return iterators. These built-in Python sequence operations are said to compute results lazily.

```
>>> m = map(lambda x: x * x, [3, 4, 5])
>>> next(m)
9
>>> next(m)
16
>>> f = filter(lambda x: x > 3, [3, 4, 5])
>>> next(f)
4
>>> next(f)
5
>>> z = zip([30, 40, 50], [3, 4, 5])
>>> next(z)
(30, 3)
>>> next(z)
(40, 4)
```

Q2: WWPD: Iterators

What would Python display?

```
>>> s = "cs61a"
>>> s_iter = iter(s)
>>> next(s_iter)
```

'c'

```
>>> next(s_iter)
```

's'

```
>>> list(s_iter)
```

['6', '1', 'a']

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

1

```
>>> s.append(5)
>>> next(i)
```

5

```
>>> next(j)
```

2

```
>>> list(j)
```

[3, 4]

```
>>> next(i)
```

StopIteration

Q3: 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.

```python
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
    count = 0
    last_item = None
    while True:
        item = next(t)
        if item == last_item:
            count += 1
        else:
            last_item = item
            count = 1
        if count == k:
            return item
```

Generators

We can create our own custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**. Generator functions have `yield` statements within the body of the function instead of `return` statements. Calling a generator function will return a generator object and will *not* execute the body of the function.

For example, let's consider the following generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

Calling `countdown(k)` will return a generator object that counts down from `k` to 0. Since generators are iterators, we can call `iter` on the resulting object, which will simply return the same object. Note that the body is not executed at this point; nothing is printed and no numbers are outputted.

```
>>> c = countdown(5)
>>> c
<generator object countdown ...>
>>> c is iter(c)
True
```

So how is the counting done? Again, since generators are iterators, we call `next` on them to get the next element! The first time `next` is called, execution begins at the first line of the function body and continues until the `yield` statement is reached. The result of evaluating the expression in the `yield` statement is returned. The following interactive session continues from the one above.

```
>>> next(c)
Beginning countdown!
5
```

Unlike functions we've seen before in this course, generator functions can remember their state. On any consecutive calls to `next`, execution picks up from the line after the `yield` statement that was previously executed. Like the first call to `next`, execution will continue until the next `yield` statement is reached. Note that because of this, `Beginning countdown!` doesn't get printed again.

```
>>> next(c)
4
>>> next(c)
3
```

The next 3 calls to `next` will continue to yield consecutive descending integers until 0. On the following call, a `StopIteration` error will be raised because there are no more values to yield (i.e. the end of the function body was reached before hitting a `yield` statement).

```
>>> next(c)
2
>>> next(c)
1
>>> next(c)
0
>>> next(c)
Blastoff!
StopIteration
```

Separate calls to `countdown` will create distinct generator objects with their own state. Usually, generators shouldn't restart. If you'd like to reset the sequence, create another generator object by calling the generator function again.

```
>>> c1, c2 = countdown(5), countdown(5)
>>> c1 is c2
False
>>> next(c1)
5
>>> next(c2)
5
```

Here is a summary of the above:

- A *generator function* has a `yield` statement and returns a *generator object*.
- Calling the `iter` function on a generator object returns the same object without modifying its current state.
- The body of a generator function is not evaluated until `next` is called on a resulting generator object. Calling the `next` function on a generator object computes and returns the next object in its sequence. If the sequence is exhausted, `StopIteration` is raised.
- A generator "remembers" its state for the next `next` call. Therefore,
  - the first `next` call works like this:
    1. Enter the function and run until the line with `yield`.
    2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
  - And subsequent `next` calls work like this:
    1. Re-enter the function, start at **the line after the `yield` statement that was previously executed**, and run until the next `yield` statement.
    2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
- Calling a generator function returns a brand new generator object (like calling `iter` on an iterable object).
- A generator should not restart unless it's defined that way. To start over from the first element in a generator, just call the generator function again.

Another useful tool for generators is the `yield from` statement. `yield from` will yield all values from an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2, 3, 4])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
StopIteration
```

Q4: 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
```

Try to understand the following expression. (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 2024.

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

next(filter(lambda n: n > 2024, gen_fib()))
```

Q5: 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
    """
    last_x = next(t)
    for x in t:
        yield x - last_x
        last_x = x
```

Q6: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in Discussion 1. (The recursive implementation of `is_prime` is shown here.)

First approach this problem using a `for` loop and using `yield`.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    for num in range(n, 1, -1):
        if is_prime(num):
            yield num
```

Now that you've done it using a for loop and `yield`, try using `yield from`! Hint: For your base case, remember that 1 isn't a prime.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if n == 1:
        return
    if is_prime(n):
        yield n
    yield from primes_gen(n-1)
```

Q7: 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.

```
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 str(n)
    if n - m > 0:
        for p in partition_gen(n - m, m):
            yield p + ' + ' + str(m)
    if m > 1:
        yield from partition_gen(n, m-1)
```