CS 61A
Summer 2026

Mutability, Iterators, Generators

Discussion 5: July 9, 2026

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

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

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

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

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

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

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

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

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

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).

Q3: 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, ________
```

Q4: 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 ***"
```

Q5: 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 function 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 ***"
```

Q6: 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)
```

Q7: 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:
        ____________________________________________
        ____________________________________________
```