CS 61A
Summer 2024

Iterators, Generators, Efficiency

Discussion 5: July 9, 2024

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.

```
>>> 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)
```

Q1: 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

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
    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

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).

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 to create a new generator.

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
```

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 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()))
```

One solution has the form: `next(____(lambda n: n > 2024, ____))` 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.

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

Presentation Time. Explain why `differences` will always yield `n-1` times for an iterator `t` over `n` values.

Q4: 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 3.

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`!

```
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)
```

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 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 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)
```

Yield a partition with just one element, `n`. Make sure you yield a string.

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`.)

Presentation Time. 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.

Efficiency

Recall that the order of growth of a function expresses how long it takes for the function to run, and is defined in terms of the function's input sizes.

For example, let's say that we have the function `get_x` which is defined as follows:

```
def get_x(x):
    return x
```

`get_x` has one expression in it. That one expression takes the same amount of time to run, no matter what x is, or more importantly, how large x gets. This is called constant time.

The main two ways that a function in your program will get a running time different than just constant time is through either iteration or recursion. Let's start with some iteration examples!

The (simple) way you figure out the running time of a particular while loop is to simply count the cost of each operation in the body of the while loop, and then multiply that cost by the number of times that the loop runs. For example, look at the following method with a loop in it:

```
def foo(n):
    i, sum = 1, 0
    while i <= n:
        sum,i = sum + i, i + 1
    return sum
```

This loop has one statement in it `sum, i = sum + i, i + 1`. This statement is considered to run in constant time, as none of its operations rely on the size of the input. Individually, `sum = sum + 1` and `i = i + 1` are both constant time operations. However, when we're looking at order of growth, we take the maximum of those 2 values and use that as the running time. In 61A, we are not concerned with how long primitive functions, such as addition, multiplication, and variable assignment, take in order to run - we are mainly concerned with *how many more times a loop is executed* or *how many more recursive calls* occur as the input increases. In this example, we execute the loop n times, and for each iteration, we only execute constant time operations, so we get an order of growth of linear.

Here are a couple of basic functions, along with their running times. Try to understand why they have the given running time.

1. Constant

```
def bar(n):
    i = 0
    while i < 10:
        n = n * 2
        i += 1
    return n
```

2. Logarithmic

```
def bar(n):
    i = 1
    while n:
        i = i * 3
        n = n // 2
    return i
```

3. Linear

```
def bar(n):
    i, a, b = 1, 1, 0
    while i <= n:
        a, b, i = a + b, a, i + 1
     return a
```

4. Quadratic

```
def bar(n):
    sum = 0
    a, b = 0, 0
    while a < n:
        while b < n:
            sum += (a*b)
            b += 1
        b = 0
        a += 1
    return sum
```

5. Exponential

```
def bar(n):
    if n == 0:
        return 1
    else:
        return bar(n - 1) + bar(n - 1)
```

Q6: Bonk

Describe the order of growth of the function below.

```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

Logarithmic.

Explanation: As we increase the value of `n`, the amount of time needed to evaluate a call to `bonk` scales logarithmically. Let's use the number of iterations of our while loop to illustrate an example. When `n = 1`, our loop iterates 0 times. When `n = 2`, our loop iterates 1 time. When `n = 4`, we have 2 iterations. And when `n = 8`, a call to `bonk(8)` results in 3 iterations of this while loop. As the value of the input scales by a factor of 2, the number of iterations increases by 1. This indicates that this function runtime has a logarithmic order of growth.

Q7: Boom

What is the order of growth in time for the following function boom? Use big- notation.

```
def boom(n):
    sum = 0
    a, b = 1, 1
    while a <= n*n:
        while b <= n*n:
            sum += (a*b)
            b += 1
        b = 0
        a += 1
    return sum
```

$(n4)$

We can come to this answer by noticing that either `b = 0` or `b = 1` when we start the `while b <= n * n` loop, and we increase `b` by 1 every time, so this loop takes about `n * n` time. Thus, the body of the `while a <= n * n` loop takes `n * n` time, and therefore the whole function takes `n * n * n * n` time, or $(n4)$

Optional (Bonus Efficiency Problem)

This next one is significantly harder (a little beyond what you might expect to see on exams) but try it out for a fun challenge! ### Q8

This question is very challenging. This is much beyond what we expect you to know for the exam. This is here merely to challenge you.

```
def boink(n):
    if n == 1:
        return 1
    sum = 0
    i = 1
    while i < n:
        sum += boink(i)
        i += 1
    return sum
```

$O(2n)$

Submit Attendance

You're done! Excellent work this week. Please be sure to ask your section TA for the attendance form link and fill it out for credit. (one submission per person per section).