DATA C88C
Spring 2026

Iterators, Generators
Discussion 10: April 13, 2026

Iterators

In Python, **iterables** are formally implemented as objects that can be passed into the built-in `iter` function to produce an *iterator*. An **iterator** is another type of object that can produce elements one at a time with the `next` function.

- `iter(iterable)`: Returns an iterator over the elements of the given iterable.
- `next(iterator)`: Returns the next element in an iterator, or raises a `StopIteration` exception if there are no elements left.

For example, a list of numbers is an iterable, since `iter` gives us an iterator over the given sequence, which we can navigate using the `next` function:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
>>> next(lst_iter)
1
>>> next(lst_iter)
2
>>> next(lst_iter)
3
>>> next(lst_iter)
StopIteration
```

Iterators are very simple. There is only a mechanism to get the next element in the iterator: `next`. There is no way to index into an iterator and there is no way to go backward. Once an iterator has produced an element, there is no way for us to get that element again unless we store it.

Note that iterators themselves are iterables: calling `iter` on an iterator simply returns the same iterator object.

Q1: 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 ***"
```

Generators

We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**.

A generator function looks like a normal Python function, except that it has at least one `yield` statement. When we call a generator function, a ***generator** object* is returned without evaluating the body of the generator function itself. Note that this is different from ordinary Python functions. While generator functions and normal functions look the same, their evaluation rules are very different!

When we first call `next` on the returned generator, we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point the next time we call `next`.

As with other iterators, if there are no more elements to generate, then calling `next` on the generator raises a `StopIteration` exception.

Q2: Filter-Iter

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of `iterable` for which `f` returns `True`. **Remember, `iterable` could be infinite!**

```
def filter_iter(iterable, f):
    """
    Generates elements of iterable for which f returns True.
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call
     to filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    "*** YOUR CODE HERE ***"
```