Lecture 12: Iterators and Generators

July 11th, 2022

Cooper Bedin

Announcements

- Attendance is temporarily unpublished from howamidoing while we fix a bug that was preventing some folks from seeing their scores
  - This is just a display bug we have your attendance recorded I promise
  - If you have any concerns you should check in with your TA directly atm
- CATS checkpoint due tomorrow
- Lab 4 due tomorrow, sections happening today
- HW 3 due Wednesday
- Midterm Thursday!
  - If you require exam alterations you should email us at cs61a@berkeley.edu *immediately*
  - We'll be having review time in discussion, lab, and lecture the next couple days, but we also strongly encourage you to practice yourself by doing past exams
  - Today and tomorrow's lectures are not in scope for the midterm, but they are in scope for the class so you do have to learn them eventually

Iterators

What are some iterable objects we've already seen?

Lists:	`[1, 2, 3]`
Tuples:	`(1, 2, 3)`
Dictionaries:	`{1:'a', 2:'b', 3:'c'}`
Ranges:	`range(1, 4)`
Strings:	`'123'`

What can we do with iterables?

One of the most common places we see iterables is in for-loops. The below would execute with exactly the same output for all five examples we saw on the previous slide*

```
for i in <iterable>:
    print(i)
```

*we'll talk about dictionaries in a bit

We also have seen a couple of functions, like `max` and `min`, that can take in an iterable as an argument:

```
max(<iterable>)
```

The `iter` function

`iter` is a function in Python that can take in any iterable object as an argument, and will return an object called an **iterator** over that object

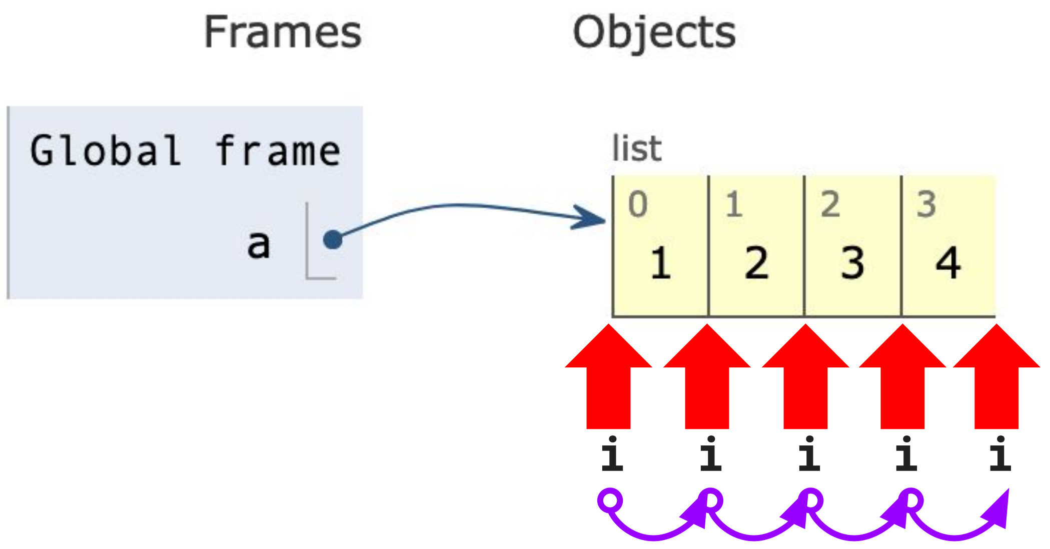
```
>>> a = [1, 2, 3, 4]
>>> i = iter(a)
>>> i
<list_iterator object at 0x7f9623429fa0>
```

We can get objects out of our iterator by calling the function `next`, and passing in the iterator as an argument

```
>>> next(i)
1
>>> b = next(i)
>>> b
2
```

Visualizing `iter`

```
a = [1, 2, 3, 4]
i = iter(a)
next(i)
next(i)
next(i)
next(i)
```



What happens when we hit the end?

```
>>> def test_stop_iteration():
...    i = iter([1, 2, 3, 4])
...    while True:
...            print(next(i))
...    print("Done!")
...
>>> test_stop_iteration()
1
2
3
4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in test_stop_iteration
StopIteration
```

Other types of iterators

`map` takes in a one-argument mapping function and an iterable object, and returns an iterator that gives back values from the iterable with the mapping function applied

```
>>> a = [1, 2, 3, 4]
>>> m = map(lambda x: x * x, a)
>>> next(m)
1
>>> next(m)
4
```

`filter` takes in a predicate function and an iterable object, and returns an iterator that gives back values from the iterable for which calling the predicate function returns `True`

```
>>> f = filter(lambda x: x % 2 == 0, a)
>>> next(f)
2
>>> next(f)
4
```

Other types of iterators (cont'd)

`zip` takes in *multiple* iterable objects, and returns an iterator that yields element-wise tuples from those iterables

```
>>> a = [1, 2, 3, 4]
>>> b = range(5, 10, 2) # 5, 7, 9
>>> z = zip(a, b)
>>> next(z)
(1, 5)
>>> next(z)
(2, 7)
>>> for i, j in zip(a, b):
...     print(i, j, i + j)
...
1 5 6
2 7 9
3 9 12
```

There's also `reversed`, which returns an iterator that visits items in reverse order

Fun tip: when you call iter on an iterator, it just returns the iterator

```
>>> a = [1, 2, 3]
>>> i = iter(a)
>>> i
<list_iterator object at 0x7fbb13b38520>
>>> j = iter(i)
>>> j
<list_iterator object at 0x7fbb13b38520>
>>> b = [4, 5, 6]
>>> k = zip(a, b)
>>> k
<zip object at 0x7fbb13b3c640>
>>> l = iter(k)
>>> l
<zip object at 0x7fbb13b3c640>
```

Exercises!

Termified

```
def termified(n, term):
    """Returns every the result of calling TERM on each
    element in the range from 0 to N (inclusive).

    Implement WITHOUT using list comprehensions

    >>> termified(5, lambda x: 2 ** x)
    [1, 2, 4, 8, 16, 32]
    """
```

Termified (solution)

```
def termified(n, term):
    """Returns every the result of calling TERM on each
    element in the range from 0 to N (inclusive).

    Implement WITHOUT using list comprehensions

    >>> termified(5, lambda x: 2 ** x)
    [1, 2, 4, 8, 16, 32]
    """
    return list(map(term, range(n + 1)))
```

Divisors

```
def divisors(n):
    """Returns all the divisors of N.

    Implement WITHOUT using list comprehensions

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
```

Divisors (solution)

```
def divisors(n):
    """Returns all the divisors of N.

    Implement WITHOUT using list comprehensions

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
    return list(filter(lambda x: n % x == 0, range(1, n)))
```

Under the hood

For loops

Now that we know what iterators are, we can finally understand how for-loops *actually* work

```
for <name> in <expression>:
    <suite>
```

For-loop execution follows these steps:

1. Creates an iterator over `<expression>`, which should be an iterable object
2. On each iteration of the for-loop, call `next` on that iterator and binds the result to `<name>`, then execute `<suite>`
3. Repeat until you encounter a `StopIteration`

WWPD?

```
>>> lst = [1, 2, 3, 4, 5]
>>> count = 0
>>> i = iter(lst)
>>> next(i)
1
>>> next(i)
2
>>> for j in i:
...     count += j
...
>>> count
12
```

Functions that take in iterables as arguments

`sum` returns the result of combining all elements in an iterable with Python's addition (+) operator

`max` returns the *largest* element in an iterable, based on Python's greater-than (>) operator

`min` returns the *smallest* element in an iterable, based on Python's less-than (<) operator

`list` returns a list containing all the elements in an iterable

`sorted` returns a list containing all of the elements in an iterable in ascending order, based on Python's comparison operators

any returns `True` if *any* elements in an iterable are truthy, and `False` otherwise

`all` returns `True` if *all* of the elements in an iterable are truthy, and `False` otherwise

WWPD? (again)

All of these functions work by constructing an iterator over their argument, and calling next on that iterator until they've answered their problem

```
>>> lst = [1, 2, 3, 4, 5]
>>> i = iter(lst)
>>> next(i)
1
>>> next(i)
2
>>> sum(i)
12
>>> next(i)
StopIteration
```

A fun one

```
>>> lst = [0, 0, 1, 2]
>>> i = iter(lst)
>>> any(i)
True
>>> next(i)
2
```

any short-circuits! Python is weird

Iterating over dictionaries

```
>>> d = {'positions':1004331994, '7 rings':1757050774}
>>> next(iter(d.keys()))
'positions'
>>> next(iter(d.values()))
1004331994
>>> next(iter(d.items()))
('positions', 1004331994)
>>> next(iter(d))
'positions'
```

As of Python 3.6, dictionaries in Python are ordered by the order key/value pairs were added in—however it's usually fine to treat them as just unordered

Another exercise!

Palindrome

```
def palindrome(s):
    """Return whether s is the same sequence backward and forward.

    Do so without constructing a list- no list comprehensions or
    calls to the list function

    >>> palindrome([3, 1, 4, 1, 5])
    False
    >>> palindrome([3, 1, 4, 1, 3])
    True
    >>> palindrome('seveneves')
    True
    >>> palindrome('seven eves')
    False
    """
```

Palindrome (solution)

```python
def palindrome(s):
    """Return whether s is the same sequence backward and forward.

    Do so without constructing a list- no list comprehensions or
    calls to the list function

    >>> palindrome([3, 1, 4, 1, 5])
    False
    >>> palindrome([3, 1, 4, 1, 3])
    True
    >>> palindrome('seveneves')
    True
    >>> palindrome('seven eves')
    False
    """
    return all(map(lambda x: x[0] == x[1], zip(s, reversed(s))))
```

don't write code that looks like this pls

Generators

Custom iterators

Just like with objects, Python has a way to make custom iterators that give back whatever we want to define—these are called **generators**

The syntax looks like this:

```
def thirtyfour_thirtyfive():
    yield 34
    yield 35
```

That probably looks weird—let's look at how to use it!

Using generators

```
>>> def thirtyfour_thirtyfive():
...         yield 34
...         yield 35
...
>>> i = thirtyfour_thirtyfive()
>>> next(i)
34
>>> next(i)
35
>>> next(i)
StopIteration
>>> for j in thirtyfour_thirtyfive(): # generators are iterable :0
...         print(j)
...
34
35
```

Generator evaluation

Whenever Python executes a function that has `yield` statements in its body, it will return a generator object, rather than a piece of data like we're used to (generally we won't see `yield` and `return` statements in the same function)

Whenever we call `next` on a generator, Python will evaluate the body of the generator function up until the next `yield` statement, give back whatever it is told to `yield`, and then pause evaluation until the next time `next` is called

If `next` is called on a generator function after the entire body of code is executed, a `StopIteration` is thrown

This is called **lazy evaluation**—it's neat because Python will only do as much work as it needs to get to the next item in generation! This allows us to do cool things like having infinite generators

Infinite generators :O

```
def side_to_side():
    while True:
        yield 1
        yield 0
        yield -1
        yield 0

def naturals():
    i = 0
    while True:
        yield i
        i += 1
```

```
def vir_fib_gen():
    prev, curr = 0, 1
    while True:
        yield prev
        prev, curr = curr, \
            prev + curr
```

Scary Generator Stuff

`yield from`

yield from is a special syntax that takes in an iterable object and repeatedly yields values from it until a `StopIteration` is encountered

```
def thirtyfour_thirtyfive():
        yield from [34, 35]

def side_to_side():
    while True:
        yield from [1, 0, -1, 0]
```

Recursive generators

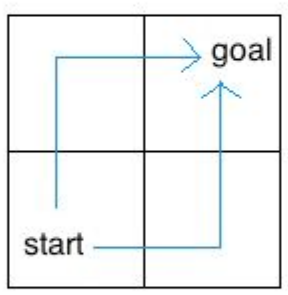
Like other functions, generators might have a self-referential definition, and lucky for us we can also define generators recursively:

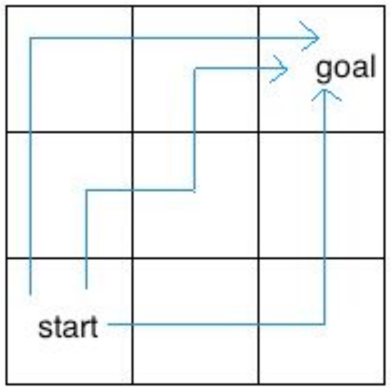
```
def naturals(i=0):
    yield i
    yield from naturals(i + 1)

def naturals():
    yield 0
    for i in naturals():
        yield i + 1
```

One more problem!

Define a function `positions` which takes in two parameters `x` and y, and returns a generator function which yields all the different paths one could take between the position `(x, y)` on a grid to `(0, 0)`, with the constraint that at each step you can only take one step down or one step to the left at each position.





Doctests

```
>>> for p in sorted(positions(1, 1)):
...     print(p)
...
[(1, 1), (0, 1), (0, 0)]
[(1, 1), (1, 0), (0, 0)]
>>> for p in sorted(positions(2, 1)):
...     print(p)
...
[(2, 1), (1, 1), (0, 1), (0, 0)]
[(2, 1), (1, 1), (1, 0), (0, 0)]
[(2, 1), (2, 0), (1, 0), (0, 0)]
```

Skeleton

```
def positions(x, y):
    if ___________ and ___________:
        yield ___________
    elif ___________ or ___________:
        return
    else:
        for path in ___________:
            yield ___________
        for path in ___________:
            yield ___________
```

Solution

```python
def positions(x, y):
    if x == 0 and y == 0:
        yield [(0, 0)]
    elif x < 0 or y < 0:
        return
    else:
        for path in positions(x - 1, y):
            yield [(x, y)] + path
        for path in positions(x, y - 1):
            yield [(x, y)] + path
```

thank u, next()