Lecture 12: Iterators and Generators

CS5001 / CS5003:
Intensive Foundations
of Computer Science

```
>>> for c in "python":
...     print(c)
...
p
y
t
h
o
n
```

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

PDF of this presentation

Lecture 12: Iterators and Generators

Today's topics:

1. Iterators
2. Generators
3. Lambda functions
4. The `set` class

Examples for today's lecture borrowed from:
https://anandology.com/python-practice-book/iterators.html

PDF of this presentation

Lecture 12: Iterators

We have seen many types of iteration in this class so far:

Iterate over a list:

```
>>> for s in ["These", "are", "some", "words"]:
...     print(s)
...
These
are
some
words
```

Iterate over a string:

```
>>> for c in "python":
...     print(c)
...
p
y
t
h
o
n
```

Iterate over a dict (keys only):

```
>>> for k in {"x": 1, "y": 2}:
...     print(k)
...
y
x
```

Iterate over a file:

```
>>> with open("a.txt") as f:
...     for line in f.readlines():
...         print(line[:-1])
...
first line
second line
```

These are all called *iterable objects*.

Lecture 12: Iterators

We can create an *iterator* from an iterable object with the built-in function, **`iter`**. Then, we can use the **`next`** function to get the values, one at a time:

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> next(x)
1
>>> next(x)
2
>>> next(x)
3
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Lecture 12: Iterators

We can create our own iterator using a class. The following iterator behaves like the **`range`** function:

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

The **`__iter__`** method is what makes an object iterable. Behind the scenes, the iter function calls **`__iter__`** method on the given object.

The return value of **`__iter__`** is an iterator. It should have a **`__next__`** method and raise **`StopIteration`** when there are no more elements.

By the way: **`raise`** means to call an exception, which can be caught in a **`try/except`** block.

Lecture 12: Iterators

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

Let's try it out:

```
>>> my_r = myrange(3)
>>> next(my_r)
0
>>> next(my_r)
1
>>> next(my_r)
2
>>> next(my_r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __next__
StopIteration
```

Lecture 12: Iterators

Example: write an iterator class called **`reverse_iter`**, that takes a list and iterates it from the reverse direction. It should behave like this:

```
>>> it = reverse_iter([1, 2, 3, 4])
>>> next(it)
4
>>> next(it)
3
>>> next(it)
2
>>> next(it)
1
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Lecture 12: Iterators

Example: write an iterator class called **`reverse_iter`**, that takes a list and iterates it from the reverse direction. It should behave like this:

```
class reverse_iter:
    def __init__(self, lst):
        self.lst = lst

    def __iter__(self):
        return self

    def __next__(self):
        if len(self.lst) > 0:
            return self.lst.pop()
        else:
            raise StopIteration()
```

Lecture 12: Generators

If we want to create an iterator, we can do it the way that we have just seen, or we can use a simpler method, called a *generator*. A generator simplifies the creation of iterators. It is a function that produces a sequence of results instead of a single one:

```
def myrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

This is a *generator function* that produces a *generator*.

Every time the yield statement is executed, the function generates a new value:

```
>>> my_r = myrange(3)
>>> my_r
<generator object yrange at 0x401f30>
>>> next(my_r)
0
>>> next(my_r)
1
>>> next(my_r)
2
>>> next(my_r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Lecture 12: Generators

Let's see how this is working internally, with a modified generator function that has some extra print statements in it:

```
>>> def foo():
...     print("begin")
...     for i in range(3):
...         print("before yield", i)
...         yield i
...         print("after yield", i)
...     print("end")
...
```

```
>>> f = foo()
>>> next(f)
begin
before yield 0
0
>>> next(f)
after yield 0
before yield 1
1
>>> next(f)
after yield 1
before yield 2
2
>>> next(f)
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

When a generator function is called, it returns a generator object without even beginning execution of the function. When the **`next`** method is called for the first time, the function starts executing until it reaches a **`yield`** statement. The yielded value is returned by the **`next`** call.

Lecture 12: Generators

Let's see how this is working internally, with a modified generator function that has some extra print statements in it:

```
>>> def foo():
...     print("begin")
...     for i in range(3):
...         print("before yield", i)
...         yield i
...         print("after yield", i)
...     print("end")
...
```

```
>>> f = foo()
>>> next(f)
begin
before yield 0
0
>>> next(f)
after yield 0
before yield 1
1
>>> next(f)
after yield 1
before yield 2
2
>>> next(f)
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

When a generator function is called, it returns a generator object without even beginning execution of the function. When the **`next`** method is called for the first time, the function starts executing until it reaches a **`yield`** statement. The yielded value is returned by the **`next`** call.

Lecture 12: Generators

Let's see some more generator examples:

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1
```

```
    ints = integers()
    for i in range(5):
        print(next(ints))
```

```
1
2
3
4
5
```

```
def squares():
    for i in integers():
        yield i * i
```

```
    sqs = squares()
    for i in range(5):
        print(next(sqs))
```

```
1
4
9
16
25
```

```
def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(next(seq))
    except StopIteration:
        pass
    return result
```

```
take(5, squares())
```

```
1
4
9
16
25
```

Lecture 12: Generators

We can create a generator using a *generator expression*, which is much like the list expressions we are used to.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

Notice that we've replaced the list comprehension brackets with parentheses -- this creates the generator.

Remember: a generator does not produce all of its values at once, and this can lead to better performance.

Lecture 12: Generators

Here is an interesting example. Let's generate the first *n* pythagorean triplets. A *pythagorean triplet* meets the requirement that for a triplet

$$(x, y, z)$$

$$x^2 + y^2 = z^2$$

Or, in Python, **`x*x + y*y = z*z`**

Here is a generator for pythagorean triplets (assuming we have the **`integers`** generator previously described):

```
pyt = ((x, y, z) for z in integers()
                 for y in range(1, z)
                 for x in range(1, y) if x*x + y*y == z*z)
```

```
take(5, pyt)
[(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15, 17)]
```

Lecture 12: Lambda Functions

A *lambda function* is a type of function that does not have a name associated with it. Lambda functions are also known as *anonymous functions* for this reason.'

In Python, lambda functions are created using the **`lambda`** keyword, and they are limited to expressions (not statements). Here is an example:

```
>>> square = lambda x: x * x
>>> square(5)
25
>>> square(8)
64
>>>
```

Lambda functions do not have an explicit return statement, and they simply perform the expression on the parameters and return the value. The above example creates a **`square`** function made from a lambda function that takes **`x`** as a parameter, and returns **`x * x`**.

Lecture 12: Lambda Functions

- You might be asking yourself, *why would we use lambda functions if it is easy enough to create a regular function?* This is a good question! But, sometimes it is useful to be able to craft a tiny function to take in certain parameters and call another function (for example) with other parameters.
- Here is an example from the Snake assignment:

```
self.bind("<Up>", lambda e: self.keypress(e, "Up"))
self.bind("<Down>", lambda e: self.keypress(e, "Down"))
self.bind("<Left>", lambda e: self.keypress(e, "Left"))
self.bind("<Right>", lambda e: self.keypress(e, "Right"))
```

The **`bind`** function is part of the Tkinter graphics library, and it requires a function that takes a single argument. However, I wanted to use a function called **`keypress`** that takes *two* arguments, so I could pass in the direction to the function. I could have created four separate functions that each took a single argument, and then passed them on to the keypress function with an extra argument, but it was much cleaner to use a lambda function for the purpose.

Lecture 12: The set class

- One data structure that we haven't yet covered is the *set*. A set is a collection that only allows unique elements. Here is an example of how it works:

```
>>> text = """I DO NOT LIKE THEM IN A HOUSE
... I DO NOT LIKE THEM WITH A MOUSE
... I DO NOT LIKE THEM HERE OR THERE
... I DO NOT LIKE THEM ANYWHERE
... I DO NOT LIKE GREEN EGGS AND HAM
... I DO NOT LIKE THEM, SAM-I-AM""".replace('\n',' ')
>>> text
'I DO NOT LIKE THEM IN A HOUSE I DO NOT LIKE THEM WITH A MOUSE I DO NOT LIKE THEM HERE OR
THERE I DO NOT LIKE THEM ANYWHERE I DO NOT LIKE GREEN EGGS AND HAM I DO NOT LIKE THEM,
SAM-I-AM'
>>> s = set()
>>> for word in text.split(' '):
...   s.add(word)
...
>>> print(s)
{'NOT', 'OR', 'I', 'THEM,', 'ANYWHERE', 'HAM', 'DO', 'A', 'THERE', 'WITH', 'LIKE',
 'SAM-I-AM', 'HERE', 'THEM', 'MOUSE', 'GREEN', 'AND', 'HOUSE', 'IN', 'EGGS'}
>>> len(text.split(' '))
44
>>> len(s)
20
```

Only the unique words are stored in the set.

Lecture 12: The set class

- The set has many useful functions. Here is part of the help file for set:

```
|  add(...)
|      Add an element to a set.
|
|      This has no effect if the element is already present.
|
|  clear(...)
|      Remove all elements from this set.
|
|  difference(...)
|      Return the difference of two or more sets as a new set.
|
|      (i.e. all elements that are in this set but not the others.)
|
|  intersection(...)
|      Return the intersection of two sets as a new set.
|
|      (i.e. all elements that are in both sets.)
|
|  isdisjoint(...)
|      Return True if two sets have a null intersection.
|
|  issubset(...)
|      Report whether another set contains this set.
|
|  issuperset(...)
|      Report whether this set contains another set.
|
|  pop(...)
|      Remove and return an arbitrary set element.
|      Raises KeyError if the set is empty.
|
|  union(...)
|      Return the union of sets as a new set.
|
|      (i.e. all elements that are in either set.)
|
```

Lecture 12: The set class

- Here are some examples using set functions:

```
>>> print(s)
{'MOUSE', 'SAM-I-AM', 'HERE', 'A', 'DO', 'NOT', 'HAM', 'WITH', 'THEM', 'IN', 'OR', 'THERE', 'AND', 'HOUSE',
 'GREEN', 'I', 'THEM,', 'LIKE', 'ANYWHERE', 'EGGS'}
>>> print(s2)
{'FOX', 'AND', 'BOX', 'GREEN', 'HAM', 'EGGS'}
>>> s.difference(s2)
{'THEM,', 'LIKE', 'MOUSE', 'THEM', 'IN', 'SAM-I-AM', 'HERE', 'HOUSE', 'OR', 'A', 'ANYWHERE', 'DO', 'NOT',
 'I', 'THERE', 'WITH'}
>>> s2.difference(s)
{'FOX', 'BOX'}
>>> s.intersection(s2)
{'AND', 'HAM', 'GREEN', 'EGGS'}
>>> s.union(s2)
{'MOUSE', 'FOX', 'AND', 'SAM-I-AM', 'HERE', 'HOUSE', 'A', 'DO', 'NOT', 'GREEN', 'HAM', 'I', 'WITH', 'THEM,',
 'LIKE', 'THEM', 'IN', 'OR', 'BOX', 'ANYWHERE', 'THERE', 'EGGS'}
>>> s - s2
{'THEM,', 'LIKE', 'MOUSE', 'THEM', 'IN', 'SAM-I-AM', 'HERE', 'HOUSE', 'OR', 'A', 'ANYWHERE', 'DO', 'NOT',
 'I', 'THERE', 'WITH'}
>>> s2 - s
{'FOX', 'BOX'}
>>>
```

Lecture 12: The set class

- Here are some more examples using set functions:

```
>>> print(s)
{'MOUSE', 'SAM-I-AM', 'HERE', 'A', 'DO', 'NOT', 'HAM', 'WITH', 'THEM', 'IN', 'OR', 'THERE', 'AND', 'HOUSE', 'GREEN
>>> print(s3)
{'TREE', 'CAR'}
>>> print(s4)
{'SAM-I-AM', 'NOT'}
>>> s.isdisjoint(s3)
True
>>> s.isdisjoint(s4)
False
>>> s.issuperset(s3)
False
>>> s.issuperset(s4)
True
>>> s3.issubset(s)
False
>>> s4.issubset(s)
True
>>>
```