Computational Structures in Data Science

Iterators and Generators

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC SA

Announcements

- Ants out today.
  - *Please start early!*
  - Read before programming.

- (random) Cool YouTube Video
  - https://www.youtube.com/watch?v=nmgFG7PUHfo
  - Signal Processing / History / Algorithmic Complexity

Today & Next Lecture

- Wrap up trees!
- Sequences vs Iterables
- Using iterators without generating all the data
- Magic methods
  - __next__
  - __iter__
- Generator concept
  - Generating an iterator from iteration with yield
- Iterators – the iter protocol
- __getitem__ protocol
- Is an object iterable?
- Lazy evaluation with iterators

Computational Structures in Data Science

Trees:
Practice With Recursion:
`traverse_recursive`

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC

Computational Structures in Data Science

Trees:
Counting Each Node

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC SA

How do we count nodes?

- The "root" or top of the tree is one node.
  - (We assume we can't have a tree of 0 nodes!)
- For each subtree we... Count the nodes!
  - Doesn't this sound like recursion?
- Hard Part: How do we group the results of recursion?
- Remember our recursive algorithm:
  - Base case
  - Recursive Case:
    - Divide
    - Invoke
    - Combine

```
def count_nodes(t):
    """The number of leaves in tree.

    >>> count_nodes(fib_tree(5))
    8
    """
    if t.is_leaf():
        return 1
    else:
        return 1 + sum(map(count_nodes,
t.branches))
```

Computational Structures in Data Science

Trees:
Practice With Recursion:
`print_tree`

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC ©

Computational Structures in Data Science

Trees:
Advanced Topics: Searching
Optional!

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC SA

Computational Structures in Data Science

Iterators & Genators

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Why?

- Iterators and similar patterns exist in many languages
  - We'll see more examples when we work with SQL
- Often times, with large data we can't compute a result immediately.
  - What if we have infinite data?
- A template for iteration makes solving (some) problems *easy.*

 © CC BY-NC-SA

Review: Why Object-Oriented Design?

- Approach creation of a class as a design problem
  - Meaningful behavior => methods [& attributes]
  - ADT methodology
  - What's private and hidden? vs What's public?
- Design for composition:
  - Use consistent patterns to solve problems more easily.

- Anticipate exceptional cases and unforeseen problems
  - try … catch
  - raise / assert

Review: What is a **sequence**? [Docs]

- Sequence is an "ordered set"
  - list
  - tuples
  - ranges
  - strings
- Some common operations:
  - Slicing syntax: `data[1:3]`
  - Membership: `'cs88' in courses`
  - Concatenation: `breakfast_foods + lunch_foods + dinner_foods`
  - Count Items: `'cs88'.count('8')`

Iterable - an object you can iterate over

- **iterable**: An object capable of yielding its members one at a time.
- **iterator**: An object representing a stream of data.
- We have worked with many iterables as sequences
  - i.e. We haven't yet cared about the more generic forms.

Functions that return iterables

`map, filter, zip`

- These objects are **not** sequences.
- They are *iterables*. A "stream" of data we can iterate over.
- Why?
  - Can't directly slice into them.
  - Don't know their length
- If we want to see all the elements at once, we need to explicitly collect them, by using list() or tuple()

Using an iterator

```
data = map(lambda x: x*x, range(5))
# Iterate with for loops
for num in data:
    print(num)

data = map(lambda x: x*x, range(5))
next(data) # returns 0
next(data) # returns 1 …
next(data) # eventually raises StopIteration error
```

How do `for`, `list`, `tuple` Work?

- Python's built in tools *use* the iterator pattern to work!
- for internally calls next() repeatedly
- list() internally calls repeatedly
- They handle the stop condition, adding to a list, etc.

 © CC BY-NC-SA

Demo

Computational Structures in Data Science

Generator Expressions

Berkeley
UNIVERSITY OF CALIFORNIA

cc

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Generator Expressions

- We've used them as list comprehensions
- **Generator Expressions return iterators**
  - access items by calling next()
- An expression which computes its values on demand
- Can be used in place of many sequences, like in for loops, map, etc.

```
>>> nums = (x * x for x in range(20))
>>> next(nums)
0
>>> next(nums)
1
```

 © CC BY-NC-SA

Generator Expressions and Generators

- Calling list() works, but it builds the result in one go.
  - This loses the benefits when we have large data!
- Generator Expressions are a short-hand to make iterators
- Generators allow us to successively ***generate*** (get it?) the next result!

Computational Structures in Data Science

Generator Functions

Berkeley
UNIVERSITY OF CALIFORNIA

cc

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Terminology [Docs]

generator

A function which returns a *generator iterator*. It looks like a normal function except that it contains yield expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function.

generator iterator

An object created by a generator function.

Generators: turning iteration into an iterable

- Generator functions use the **`yield`** keyword
- Generator functions have no return statement, but they don't return None
  - They *implicitly* return a generator object
- Generator objects are *just* iterators

```
def squares(n):
    for i in range(n):
        yield (i*i)
```

Spongebob Case

```
def spongebob_case(text):
    caps = True
    for letter in text:
        if caps:
            yield letter.upper()
        else:
            yield letter.lower()
        caps = not caps
```

- Generate one letter at a time.
- Explore how caps changes with each iteration.

 © CC BY-NC-SA

Nest iteration

```
def all_pairs(x):
    for item1 in x:
        for item2 in x:
            yield(item1, item2)
```

Order of Execution

- Our generator function executes until we hit `yield`
- Once we hit `yield`, execution is *paused*
- Explore this with print statements

 © CC BY-NC-SA

Computational Structures in Data Science

Iterators

Berkeley
UNIVERSITY OF CALIFORNIA

CC BY NC SA

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

What's an Iterator? [Docs]

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead.

iterable

An object capable of returning its members one at a time. Examples of include all sequence types and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements sequence semantics.

Next element in generator iterable

- Iterables work because they implement some "magic methods" on them. We saw magic methods when we learned about classes,
  - e.g., __init__, __repr__ and __str__.
- The first one we see for iterables is `__next__`

- `iter()` – transforms a sequence into an iterator
  - Usually this is not necessary, but can be useful.

Iterators: The `iter` protocol [Docs]

- In order to be iterable, a class must implement the iter protocol
- The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:
  - __iter__: Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
    - This method returns an iterator object (which can be self)
  - __next__ : Return the next item from the container. If there are no further items, raise the StopIteration exception.

The Iter Protocol In Practice

- Classes get to define how they are iterated over by defining these methods
  - containers (objects like lists, tuples, etc) typically define a Container class and a separate ContainterIterator class.
- Lists, Ranges, etc are *not* directly iterators
  - We cannot call next() on them.
  - However, they implement an `__iter__` method, and `list_iterator`, `range_iterator` class, etc.

Iterables

Demo

Computational Structures in Data Science

Building a Range Iterator

Berkeley
UNIVERSITY OF CALIFORNIA

cc

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Making a Range Iterator

- What does a range need?
  - Start value
  - Stop
  - (We'll ignore step sizes)
- keep track of the current value
- An __iter__ method
- A __next__ method

Example

```
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:
            current = self.i
            self.i += 1
            return current
        else:
            raise StopIteration()
```

 © CC BY-NC-SA

Computational Structures in Data Science

The GetItem Protocol

Berkeley
UNIVERSITY OF CALIFORNIA

cc

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Get Item protocol

- Another way an object can behave like a sequence is indexing: Using square brackets "[ ]" to access specific items in an object.
- Defined by special method: `__getitem__(self, i)`
  - Method returns the item at a given index

```
class myrange2:
    def __init__(self, n):
        self.n = n

    def __getitem__(self, i):
        if i >= 0 and i < self.n:
            return i
        else:
            raise IndexError

    def __len__(self):
        return self.n
```

Computational Structures in Data Science

Iterators and Generators Review

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Terms and Tools

- **Iterators:** Objects which we can use in a for loop
  - Anything that can be looped over!
  - Sometimes they're lazy, sometimes not!
- **Generators:** A shorthand way to make an iterator that uses yield
  - a function that uses `yield` is a *generator function*
  - a generator function returns a *generator object*
  - Generators do **not** use return
- **Sequences:** A particular type of iterable
  - They know they're length, support slicing
  - Are *not* lazy

Computational Structures in Data Science

Type Checking

Berkeley
UNIVERSITY OF CALIFORNIA

cc

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Determining if an object is iterable

- from collections.abc import Iterable
- isinstance([1,2,3], Iterable)

- This is more general than checking for any list of particular type, e.g., list, tuple, string…