515_02_Generators

November 14, 2024

1 Week 2: Iterators, Generators, Decorators, Caching and Memoization

1.1 Iterators

1.1.1 What is iteration?

Iteration is the process of traversing a series of objects, handling each one in turn until there are no more objects to deal with.

For example, a for loop represents a type of iteration.

In python, we often iterate through a list:

```
numbers = [1,3,5,7,9]
for number in numbers:
    print(number)
```

1.1.2 Iterable vs. iterator

- in python, an iterable object is one that implements an `__iter__` method
- in python, an iterator is an object that implements a `__next__` method
- iter() returns an iterator for an object
- next() returns the next element from an iterator (by invoking the underlying `__next__` method)
- usually the case that a class has both an `__iter__` and a `__next__` method

[1]:
```
years = [1967, 1974, 1955, 2029]
years_iter = iter(years)
print(next(years_iter))
print(next(years_iter))
print(next(years_iter))
print(next(years_iter))
next(years_iter)
```

```
1967
1974
1955
2029
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[1], line 7
      5 print(next(years_iter))
      6 print(next(years_iter))
----> 7 next(years_iter)

StopIteration:
```

[2]:
```
for year in years:
    print(year)
```

```
1967
1974
1955
2029
```

1.1.3 Python's `for` loop is really a `while` loop with an iterator!

```
for x in some_iterable:
    do_something_with(x)
```

is really

```
iterator = iter(some_iterable)
while True:
    try:
        item = next(iterator)
    except StopIteration:
        break
    yield item
```

1.1.4 Let's define a class that's iterable and that returns an iterator

(that is, one that returns an interator when `__iter__` is called)

[3]:
```
class Decades:
    def __init__(self, years):
        self.years = years
        self.index = 0
    def __iter__(self):
        return self # note that self has a __next__ method, so it's an iterator
    def __next__(self):
        if self.index >= len(self.years):
            raise StopIteration
        retval = self.years[self.index] // 10 * 10
        self.index += 1
        return retval
```

[4]:
```
decades = Decades([1971,1982,2019])
```

[5]:
```
decades_iterator = iter(decades)
```

[6]:
```
next(decades_iterator)
```

[6]: 1970

1.2 Generators

1.2.1 Generators are similar to functions, so let's start there:

Let's look at a simple function that returns a list of n elements, each of which is a string containing "Hello"

[7]:
```
def Hello(n = 0):
    ret = []
    for i in range(n):
        ret.append("Hello")
    return ret
```

[8]:
```
hellos = Hello(5)
```

[9]:
```
for hello in hellos:
    print(hello)
```

```
Hello
Hello
Hello
Hello
Hello
```

1.3 Our first generator

[10]:
```
def HelloGenerator(max = 0):
    x = 0
    while True:
        if x < max:
            yield "Hello"
            x = x+1
        else:
            break
```

[11]:
```
hellos = HelloGenerator(5)
```

[12]:
```
for hello in hellos:
    print(hello)
```

```
Hello
Hello
Hello
Hello
Hello
```

1.3.1 Generators vs. Functions

- most important difference is that generators use the `yield` statement, whereas functions use `return`
- `yield` returns a value, stops executing the code at that point and maintains state until it's called again
- when invoked, returns an object but doesn't start executing code
- implements `__iter__` and `__next__` automatically (hey, that's useful!)

1.3.2 List comprehensions vs. generator expressions

[13]:
```
# Initialize the list, in this case with a list of years
year_list = [2018, 1776, 2020, 1977, 1980, 2009, 2019]

# Find the decade corresponding to each of the years
decade_list = [x//10*10 for x in year_list]
```

[14]:
```
# same thing with a generator
decade_generator = (x//10*10 for x in year_list)
```

[15]:
```
max(decade_list)
```

[15]:
```
2020
```

[16]:
```
min(decade_generator)
```

[16]:
```
1770
```

1.4 Filtering

In the following code:

[17]:
```
decade_generator_filtered = (x//10*10 for x in year_list if x > 1900)
```

the parentheses ('()') create a generator expression

[18]:
```
max(decade_generator_filtered)
```

[18]:
```
2020
```

1.5 Memory size issues

[19]:
```
import sys
```

[20]:
```
sys.getsizeof(decade_list)
```

[20]: 120

[21]:
```
sys.getsizeof(decade_generator)
```

[21]: 112

[22]:
```
big_year_list = [x for x in range(1770,2020)]
big_decade_list = [x//10*10 for x in big_year_list]
```

[23]:
```
sys.getsizeof(big_decade_list)
```

[23]: 2200

[24]:
```
big_decade_generator = (x//10*10 for x in big_year_list)
```

[25]:
```
sys.getsizeof(big_decade_generator)
```

[25]: 112

1.6 A more complex example

[26]:
```
def lines_words_chars(text):
    yield ('lines',len(text.splitlines()))
    yield ('words',len(text.split()))
    yield ('characters',len(text))
```

[27]:
```
a = lines_words_chars("This is a text")
```

[28]:
```
next(a)
```

[28]: ('lines', 1)

[29]:
```
next(a)
```

[29]: ('words', 4)

[30]:
```
next(a)
```

[30]: ('characters', 14)

[31]:
```
next(a)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[31], line 1
----> 1 next(a)

StopIteration:
```

[32]:
```python
# skip all non-lowercased letters (including punctuation)
# append 1 if lowercase letter is "o"
# append 0 if lowercase letter is not "o"
out = []
for i in "Hello. How Are You?":
    if i.islower():
        out.append(1 if i is "o" else 0)
```

[33]:
```python
out
```

[33]:
```
[0, 0, 0, 1, 1, 0, 0, 0, 1, 0]
```

[34]:
```python
# NOTE: this is not efficient because statistics.mean() will create a list from␣
 ↪a generator
#       before proceeding with the calculation






from statistics import mean
out2 = mean(1 if char is 'o' else 0 for char in "Hello. How Are You?" if char.
 ↪islower())
out2
```

[34]:
```
0.3
```

1.7 Let's take a look at some python code:

From https://github.com/python/cpython/blob/master/Lib/statistics.py

```
# === Measures of central tendency (averages) ===
def mean(data):
    """Return the sample arithmetic mean of data.
    >>> mean([1, 2, 3, 4, 4])
    2.8
    >>> from fractions import Fraction as F
    >>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
    Fraction(13, 21)
    >>> from decimal import Decimal as D
```

```
    >>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
    Decimal('0.5625')
    If ``data`` is empty, StatisticsError will be raised.
    """
    if iter(data) is data:
        data = list(data)
    n = len(data)
    if n < 1:
        raise StatisticsError('mean requires at least one data point')
    T, total, count = _sum(data)
    assert count == n
    return _convert(total/n, T)
```

[35]:
```python
data = [1,2,3]
idata = iter(data)
```

[36]:
```python
data
```

[36]: [1, 2, 3]

[37]:
```python
idata
```

[37]: <list_iterator at 0x72f5045eee20>

[38]:
```python
def dgen():
    yield 1
    yield 2
    yield 3
```

[39]:
```python
dg = dgen()
```

[40]:
```python
dgi = iter(dg)
```

[41]:
```python
dg
```

[41]: <generator object dgen at 0x72f4fc272270>

[42]:
```python
dgi
```

[42]: <generator object dgen at 0x72f4fc272270>

[43]:
```python
if data is not idata:
    print(1)
```

1

[44]:
```python
if iter(data) is data:
    print("yes")
```

[45]:
```
data
```

[45]:
```
[1, 2, 3]
```

[46]:
```
iter(data)
```

[46]:
```
<list_iterator at 0x72f5045ee5e0>
```

1.8 Memoization

A common way to teach memoization is to use Fibonacci numbers, defined as

$ F_{0}=0,   F_{1}=1,$

and

$ F_{n}=F_{n-1}+F_{n-2},$

for $n > 1$

Thus, the first few Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Here's an implementation of the code to calculate Fibonacci numbers:

1.8.1 LEARNING CHECK POSSIBILITY: GET THEM TO WRITE A FIBONACCI FUNCTION

[47]:
```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

[48]:
```
fibonacci(35)
```

[48]:
```
9227465
```

1.8.2 Important digression: Jupyter magic commands

- sometimes you'll see a line in a Jupyter notebook that starts with a '%'
- these are "magic" commands
- we'll deal with these in more detail in a later lecture, but for now we're going to introduce %timeit
- %timeit will tell you how much time a line takes to run
- %%timeit will tell you how much time a cell takes to run

[49]:
```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
```

```
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

[50]:
```
%timeit fibonacci(32)
```

```
760 ms ± 3.83 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

1.9 Caching

- typically done with web browsers and web pages
- let's take a look at a simple caching example

1.10 Memoization: a special form of caching

- caching is a more general approach: e.g. web pages
- memozation is caching of the output of a function given a specific set of parameters

1.11 Memoization example:

[51]:
```
def memoize(func):
    cache = dict()

    def memoized_func(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    return memoized_func
```

[52]:
```
memoized_fibonacci = memoize(fibonacci)
```

[53]:
```
%timeit memoized_fibonacci(32)
```

```
The slowest run took 12.83 times longer than the fastest. This could mean that
an intermediate result is being cached.
716 ns ± 1.06 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[54]:
```
%timeit memoized_fibonacci(30)
```

```
The slowest run took 11.46 times longer than the fastest. This could mean that
an intermediate result is being cached.
673 ns ± 957 ns per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Note that the memoized version doesn't call the memoized verision when it recurses.

1.12 Important Digression: Decorators

- recall example from above where one function (memoized_fibonacci) returned another function (fibonacci)
- this is a specific form of a more general approach called decorators
- let's take a look at the simplest form of a decorator, the null decorator, which does nothing

[55]:
```python
def null_decorator(func):
    return func
```

[56]:
```python
def hello():
    return "Hello"
```

[57]:
```python
hello()
```

[57]: 'Hello'

[58]:
```python
decorated_hello = null_decorator(hello)
```

[59]:
```python
decorated_hello()
```

[59]: 'Hello'

1.12.1 Now let's look at a slightly more complicated example that takes some function (assuming it returns a string) and wraps the output in `<em>...</em>` tags

[60]:
```python
def emphasize(func):
    def wrapper():
        original_ret = func()
        modified_ret = "<em>" + original_ret + "</em>"
        return modified_ret
    return wrapper
```

[61]:
```python
emphasized_hello = emphasize(hello)
```

[62]:
```python
emphasized_hello()
```

[62]: 'Hello'

1.12.2 Using the @: wrapping functions simplified

- commonly referred to as "syntactic sugar", the @ command allows you to wrap a function with one line

[63]:
```python
def emphasize(func):
    def wrapper():
        original_ret = func()
        modified_ret = "<em>" + original_ret + "</em>"
```

```
        return modified_ret
    return wrapper
```

[64]:
```
@emphasize
def hello():
    return "Hello"
```

[65]:
```
hello()
```

[65]: 'Hello'

1.13 A slightly more complicated example: decorating functions that take parameters

- let's say we have a function that returns "Hello" in some specified language:

[66]:
```
def multilingual_hello(lang = 'en'):
    lookup = {'en':'Hello','fr':'Bonjour'}
    return lookup[lang]
```

[67]:
```
multilingual_hello('en')
```

[67]: 'Hello'

[68]:
```
multilingual_hello('fr')
```

[68]: 'Bonjour'

1.14 And now let's say we want to decorate that function with our emphasize wrapper:

[69]:
```
@emphasize
def multilingual_hello(lang = 'en'):
    lookup = {'en':'Hello','fr':'Bonjour'}
    return lookup[lang]
```

[70]:
```
multilingual_hello()
```

[70]: 'Hello'

[71]:
```
multilingual_hello('fr')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[71], line 1
----> 1 multilingual_hello('fr')
```

```
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

1.14.1 Uh oh… what just happened?

- our wrapper isn't set up to take any paramters, but our underlying function expects one (optional) one
- we can change our decorator to accommodate the optional paramter by using `*args` and `**kwargs`:

[72]:
```
def emphasize_args(func):
    def wrapper(*args,**kwargs):
        original_ret = func(*args,**kwargs)
        modified_ret = "<em>" + original_ret + "</em>"
        return modified_ret
    return wrapper
```

[73]:
```
@emphasize_args
def multilingual_hello(lang = 'en'):
    lookup = {'en':'Hello','fr':'Bonjour'}
    return lookup[lang]
```

[74]:
```
multilingual_hello('fr')
```

[74]:
```
'<em>Bonjour</em>'
```

1.15 Ok, back to memoization

- but first, another digression: functools
- functools: https://docs.python.org/3/library/functools.html
- "Higher-order functions and operations on callable objects"
- of note, `@functools.lru_cache`

[75]:
```
import functools

@functools.lru_cache(maxsize=128)
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Or equivalently:

[76]:
```
from functools import lru_cache
```

```python
@lru_cache(maxsize=128)
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

See also https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU) for LRU

[77]:
```python
%timeit fibonacci(10)
```

```
71.8 ns ± 1.21 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[78]:
```python
%timeit fibonacci(20)
```

```
71.9 ns ± 0.226 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[79]:
```python
%timeit fibonacci(30)
```

```
72.3 ns ± 1.43 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[80]:
```python
%timeit fibonacci(40)
```

```
71.4 ns ± 0.205 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[ ]: