generators, context managers, decorators

krasch

November 8th, 2016;

```
# generators
```

let's read a HUGE file line by line and do something with each line

```
f = open('data.csv')
data = f.readlines()
f.close()

for line in data:
    print(line.upper())
```

let's read a HUGE file line by line and do something with each line

```
f = open('data.csv')
data = f.readlines()
f.close()

for line in data:
    print(line.upper())
```

no good because we read the whole file into memory

let's read a HUGE file line by line and do something with each line, second try

```
f = open('data.csv')
for line in f.readlines():
    print(line.upper())
f.close()
```

let's read a HUGE file line by line and do something with each line, second try

```
f = open('data.csv')
for line in f.readlines():
    print(line.upper())
f.close()
```

no good, because readlines reads the whole file into a list (i.e. into memory)

let's read a HUGE file line by line and do something with each line, third try

```
f = open('data.csv')
for line in f:
    print(line.upper())
f.close()
```

let's read a HUGE file line by line and do something with each line, third try

```
f = open('data.csv')
for line in f:
    print(line.upper())
f.close()
```

now we are good, because we always only have one line in memory (because f provides a generator)

generators perform **lazy** evaluation

- only retrieve the data / compute what is needed right now

generators produce **iterators**

- you can use them in a *for* or *while* loop just like a list

function to produce squares - list version

```
def squares_list(n):
    result = []
    for i in range(n):
        result.append(i*i)
    print("Finished producing squares")
    return result
```

function to produce squares - list version

```
def squares_list(n):
    result = []
    for i in range(n):
        result.append(i*i)
    print("Finished producing squares")
    return result
```

using function to print squares of numbers from 0 to 4

```
for square in squares_list(5):
    print(square)

# Finished producing squares
# 0
# 1
# 4
# 9
# 16
```

function to produce squares - generator version

```
def squares_gen(n):
    for i in range(n):
        yield i*i
    print("Finished producing squares")
```

function to produce squares - generator version

```
def squares_gen(n):
    for i in range(n):
        yield i*i
    print("Finished producing squares")
```

using function to print squares of numbers from 0 to 4

```
for square in squares_gen(5):
    print(square)
# 0
# 1
# 4
# 9
# 16
# Finished producing squares
```

your task - what happens if you do?

```
print(squares_gen(5))

print(list(squares_gen(5)))
```

your task - what happens if you do?

```
print(squares_gen(5))

# <generator object squares at 0x7f7ccb14e4c0>


print(list(squares_gen(5)))

# Finished producing squares
# [0, 1, 4, 9, 16]
```

your task: write your own zip function (list version and generator version)

```
# take two lists with same number of elements
list1 = [0, 1, 2, 3, 4]
list2 = ['a', 'b', 'c', 'd', 'e']
for pair in myzip(list1, list2):
    print(pair)

# expected output
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
```

list version

```
def myzip_list(list1, list2):
    results = []
    for i in range(len(list1)):
        results.append((list1[i], list2[i]))
    return results
```

generator version

```
def myzip_gen(list1, list2):
    for i in range(len(list1)):
        yield list1[i], list2[i]
```

your task generator in generator in generator

write three generators:

1. myrange(n) counts from zero to n
2. triple(numbers) multiplies every number in the *numbers* iterator with 3
3. even(numbers) keeps only those numbers in the *numbers* iterator that can be divided by 2 (`if number % 2 == 0`)

and stack them like this

```
for i in even(triple(myrange(10))):
    print(i)
```

reason about the memory usage of stacking the generators like this

one possible implementation for generator in generator in generator

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

def triple(numbers):
    for i in numbers:
        yield i*3

def even(numbers):
    for i in numbers:
        if i % 2 == 0:
            yield i
```

the infinite generator

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

how do we get out of there?

the infinite generator

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

how do we get out of there?

```
for counter in count_endlessly():
    if counter > 100:
        break
```

generator slicing

```
squares = squares_gen(100000)

# this will give us an error:
# TypeError: 'generator' object is not subscriptable
print(squares[0:10])

# this works but we have to put all squares into a list
# which is what we wanted to avoid in the first place
print(list(squares)[0:10])

# need to use islice
from itertools import islice
slice = islice(squares, 5)

# slice is again a generator, hooray lazyness!
print(list(slice))
```

generator length

```
# this will give an error
# TypeError: object of type 'generator' has no len()
print(len(squares_gen(10000)))

# this works but again we have to make a list
print(len(list(squares_gen(10000))))

# there is no way to get the length
# without consuming the whole generator
```

unexpected differences between python2 and python3

python2

```
print(range(10))
# produces a list: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

python3

```
print(range(10))
# produces a generator: range(0, 10)
```

since generators add minimal memory overhead (lazy evaluation!), I like to use them to structure my code

```
f = open('data.csv')
for line in f:
    # remove comments
    if line.startswith('#'):
        continue
    print(line.upper())
f.close()
```

same functionality using a remove_comments generator

```
def remove_comments(lines):
    for line in lines:
        if not line.startswith('#'):
            yield line

f = open('data.csv')
for line in remove_comments(f):
    print(line.upper())
f.close()
```

generators in python

- enable lazy evaluation
- are a good tool when dealing with large files, large database results, endless data streams, etc
- can be used for structuring code into functions without introducing memory overhead

```
# context managers
```

let's write some pairs to a file

```
# lists are not same length so myzip will fail!
list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b']

f = open('pairs.txt', 'w')

# uuuhoooh when getting third pair we get an:
# IndexError: list index out of range
for pair in myzip_gen(list1, list2):
    f.write(str(pair))

# will never reach f.close() -> never close the file handler
# might lose what we have written to the file so far
# plus operating system might be sad
f.close()
```

have to wrap pair generation into a try/finally

```
list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b']

f = open('pairs.txt', 'w')
try:
    for pair in myzip_gen(list1, list2):
        f.write(str(pair))
finally:
    # we will get here in success and in error case!
    f.close()
```

let a context manager do the try/finally/close for you!

```
list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b']

with open('pairs.txt', 'w') as f:
    for pair in myzip_gen(list1, list2):
        f.write(str(pair))
```

```
from contextlib import contextmanager

@contextmanager
def open_for_writing(filename):
    fh = open(filename, 'w')
    try:
        yield fh
    finally:
        fh.close()

list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b']
with open_for_writing('pairs.txt') as f:
    for pair in myzip_gen(list1, list2):
        f.write(str(pair))
```

```
from contextlib import contextmanager

@contextmanager
def open_for_writing(filename):
    fh = open(filename, 'w')
    try:
        yield fh
    except:
        print("Some error occured")
        raise
    finally:
        fh.close()

list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b']
with open_for_writing('pairs.txt') as f:
    for pair in myzip_gen(list1, list2):
        f.write(str(pair))
```

your task: database connection manager

0. take a look at fakedb.py
1. in the code below, which errors can happen and where?
2. rewrite code below to use built-in context manager 'open' for handling meetups.txt file
3. write your own context manager called 'connect' for handling the database connection
4. rewrite code below to use your context manager from (3)

```
conn = FakeDatabase("meetups.db")
try:
    f = open('meetups.txt', 'w')
    try:
        for row in conn.query("SELECT * FROM meetups"):
            f.write(row)
    finally:
        f.close()
finally:
    conn.close()
```

your task database connection manager

2. rewrite code to use built-in context manager ‘open’ for handling meetups.txt file

```
conn = FakeDatabase("meetups.db")
try:
    with open('meetups.txt', 'w') as f:
        for row in conn.query("SELECT * FROM meetups"):
            f.write(row)
finally:
    conn.close()
```

your task database connection manager

3. write your own context manager called ‘connect’ for handling the database connection

```
from contextlib import contextmanager

@contextmanager
def connect(db_name):
    db_conn = FakeDatabase("meetups.db")
    try:
        yield db_conn
    finally:
        db_conn.close()
```

your task database connection manager

4. rewrite code to use your context manager from (3)

```
with connect('meetups.db') as conn:
    with open('meetups.txt', 'w') as f:
        for row in conn.query("SELECT * FROM meetups"):
            f.write(row)
```

your task database connection manager

4. rewrite code to use your context manager from (3)

```
with connect('meetups.db') as conn:
    with open('meetups.txt', 'w') as f:
        for row in conn.query("SELECT * FROM meetups"):
            f.write(row)
```

```
with connect('meetups.db') as c, open('meets.txt', 'w') as f:
    for row in c.query("SELECT * FROM meetups"):
        f.write(row)
```

built-in context managers: **closing**

calls the `close` method of whatever you give it

```
from contextlib import closing
with closing(FakeDatabase("meetups.db")) as conn:
    for row in conn.query("SELECT * FROM meetups"):
        print(row)


@contextmanager
def closing(thing_to_close):
    try:
        yield thing_to_close
    finally:
        thing_to_close.close()
```

built-in context managers: **tempfiles**

makes a temporary file and removes it when is not needed anymore

```
from tempfile import TemporaryFile

with TemporaryFile(mode="w") as f:
    f.write('writing some stuff into my tempfile')
```

context managers

- ensures that resources are properly managed (in particular that stuff is closed even if errors happen)
- help you separate resource handling from your actual logic

“‘ {.python} # decorators

timing how long execution of a function took

```
from datetime import datetime
from time import sleep

def my_slow_function():
    sleep(10)

start = datetime.now()
my_slow_function()
spent = datetime.now() - start
print("Execution took {}".format(spent))
```

re-usable timer

```
from datetime import datetime
from time import sleep

def timed(function_to_time):
    start = datetime.now()
    result = function_to_time()      # actual execution
    spent = datetime.now() - start
    print("Execution took {}".format(spent))
    return result

def my_slow_function():
    sleep(10)

timed(my_slow_function)
```

re-usable timer as a decorator

```
from datetime import datetime
from time import sleep

def timed(function_to_time):
    def execute_and_log_time():
        start = datetime.now()
        result = function_to_time()    # actual execution
        spent = datetime.now() - start
        print("Execution took {}".format(spent))
        return result
    return execute_and_log_time

@timed
def my_slow_function():
    sleep(10)

my_slow_function()
```

decorator order

```
@logging("out.log")
@timed
def my_slow_function():
    sleep(10)

my_slow_function()
# timed is executed first
# (because it wraps my_slow_function)
# logging is executed second
# (because it wraps the wrapped my_slow_function)
```

some useful decorators

```
# give warning to user that function will be removed soon
@deprecated

# you can put this on any class method to mark it static
@staticmethod

# pip install profilehooks
# profiling where in the function you spend your time
@profile

# pip install nose-parameterized
# runs same unit test with different parameters
@parameterized(["DE", "UK", "SP"])
def test_load_csv(country):
    data = load_csv(country)
    assert_data_correct(data)
```

subjective: use decorators lightly

- do not use them to change functionality majorly
- better as markers e.g. @deprecated
- avoid using more than one because order is somewhat counter-intuitive

the end

thank you