CME 193: Introduction to Scientific Python

Winter 2017

Lecture 7: Recursion and Exceptions

Blake Jennings

`stanford.edu/~bmj/cme193`

Contents

- More pandas
- Exception handling
- Recursion

Pandas

Last time we saw the main data structures offered in the pandas package

Table: Main data structures

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure
3	Panel	General 3D labeled, also size-mutable array

Pandas - introduction

Let's switch over to another Jupyter notebook example to see some more internals of the package

Contents

- More pandas
- Exception handling
- Recursion

Wordcount

Consider a function that takes a filename, and returns the 20 most common words. (This is similar to one of the exercises you could have done.)

Suppose we have written a function `topkwords(filename, k)`

Instead of entering filename and value of k in the script, we may also want to run it from the terminal.

Parse input from command line

The `sys` module allows us to read the terminal command that started the script:

```
import sys

print sys.argv
```

`sys.argv` holds a list with command line arguments passed to a Python script.

Note that `sys.argv[0]` will be the name of the python script itself.

Parse input from command line

The `sys` module allows us to read the terminal command that started the script:

```
import sys

print sys.argv
```

`sys.argv` holds a list with command line arguments passed to a Python script.

Note that `sys.argv[0]` will be the name of the python script itself.

Back to the wordcount example

```
import sys

def topkwords(filename, k):
    # Returns k most common words in filename
    pass

if __name__ == "__main__":
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print topkwords(filename, k)
```

Issues?

Back to the wordcount example

```
import sys

def topkwords(filename, k):
    # Returns k most common words in filename
    pass

if __name__ == "__main__":
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print topkwords(filename, k)
```

Issues?

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Issues

- What if the file does not exist?
- What if the second argument is not an integer?
- What if no command line arguments are supplied?

All result in errors:

- IOError
- ValueError
- IndexError

Exception handling

What do we want to happen when these errors occur? Should the program simply crash?

No, we want it to gracefully handle these

- IOError: Tell the user the file does not exist.
- ValueError, IndexError: Tell the user what the format of the command line arguments should be.

Exception handling

What do we want to happen when these errors occur? Should the program simply crash?

No, we want it to gracefully handle these

- IOError: Tell the user the file does not exist.
- ValueError, IndexError: Tell the user what the format of the command line arguments should be.

Try ... Except

```
import sys

if __name__ == "__main__":
    try:
        filename = sys.argv[1]
        k = int(sys.argv[2])
        print topkwords(filename, k)
    except IOError:
        print "File does not exist"
    except (ValueError, IndexError):
        print "Error in command line input"
        print "Run as: python wc.py <filename> <k>"
        print "where <k> is an integer"
```

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

Try ... Except

- The try clause is executed
- If no exception occurs, the except clause is skipped
- If an exception occurs, the rest of the try clause is skipped. Then if the exception type is matched, the except clause is executed. Then the code continues after the try statement
- If an exception occurs with no match in the except clause, execution is stopped and we get the standard error

A naked except

We can have a naked except that catches any error:

```
try:
    t = 3.0 / 0.0
except:
    # handles any error
    print 'There was some error'
```

Use this with extreme caution though, as genuine bugs might be impossible to correct!

Try Except Else

Else clause after all except statements is executed after successful execution of the try block (hence, when no exception was raised)

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
# from Python docs
```

Why? Avoids catching exception that was not protected

E.g. consider `f.readlines` raising an `IOError`

Example from Python docs

Raise

We can use `Raise` to raise an exception ourselves.

```
>>> raise NameError('Oops')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Oops
```

Finally

The `finally` statement is always executed before leaving the `try` statement, whether or not an exception has occured.

```
def div(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print 'Division by zero!'
    finally:
        print "Finally clause"

print div(3,2)
print div(3,0)
```

Useful in case we have to close files, closing network connections etc.

Finally

The `finally` statement is always executed before leaving the `try` statement, whether or not an exception has occured.

```
def div(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print 'Division by zero!'
    finally:
        print "Finally clause"

print div(3,2)
print div(3,0)
```

Useful in case we have to close files, closing network connections etc.

Raising our own exceptions: Rational class

Recall the Rational class we considered a few lectures ago:

```
class Rational:
    def __init__(self, p, q=1):
        g = gcd(p, q)
        self.p = p / g
        self.q = q / g
```

What if `q = 0`?

Rational class

```
a = Rational(1,3)
b = Rational(2,3)
c = Rational(2,0)

print 'a*b = {}'.format(a*b)  # 2/9
print 'a*c = {}'.format(a*c)  # 1/0
print 'a/b = {}'.format(a/b)  # 1/2
print 'a/c = {}'.format(a/c)  # 0
print 'c/a = {}'.format(c/a)  # 1/0
```

... Not really anything!

Making the necessary change

If `q = 0`, then raise an exception!

Which one?

```
class Rational:
    def __init__(self, p, q=1):
        if q == 0:
            raise ZeroDivisionError('denominator is zero')
        g = gcd(p, q)
        self.p = p / g
        self.q = q / g
```

Making the necessary change

If `q = 0`, then raise an exception!

Which one?

```
class Rational:
    def __init__(self, p, q=1):
        if q == 0:
            raise ZeroDivisionError('denominator is zero')
        g = gcd(p, q)
        self.p = p / g
        self.q = q / g
```

Contents

- More pandas
- Exception handling
- Recursion

Back to control flow

To execute repetitive code, we have relied on `for` and `while` loops.

Furthermore, we used `if` statements to handle conditional statements.

These statements are rather straightforward and easy to understand.

Recursion

Recursive function solve problems by reducing them to smaller problems of the same form.

This allows recursive functions to call themselves...

- New paradigm
- Powerful tool
- Divide-and-conquer
- Beautiful solutions

Recursion

Recursive function solve problems by reducing them to smaller problems of the same form.

This allows recursive functions to call themselves...

- New paradigm
- Powerful tool
- Divide-and-conquer
- Beautiful solutions

First example

Let's consider a trivial problem:

Suppose we want to add two positive numbers a and b, but we can only add/subtract 1 to any number.

How would you write a function to do this without recursion? What control statement(s) would you use?

First example

Non-recursive solution:

```
def add(a, b):
    while b > 0:
        a += 1
        b -= 1
    return a
```

First example

Recursive solution:

- Simple case: b = 0, return a
- Else, we can return 1 + add(a, b-1)

```
def add(a, b):
    if b == 0:
        # base case
        return a
    # recursive step
    return add(a, b-1) + 1
```

First example

Recursive solution:

- Simple case: b = 0, return a
- Else, we can return 1 + add(a, b-1)

```
def add(a, b):
    if b == 0:
        # base case
        return a
    # recursive step
    return add(a, b-1) + 1
```

Base case and recursive steps

Recursive functions consist of two parts:

base case The base case is the trivial case that can be dealt with easily.

recursive step The recursive step brings us slightly closer (breaks the problem into smaller subproblems) to the base case and calls the function itself again.

Reversing a list

How can we recursively reverse a list ([1, 2, 3] $\rightarrow$ [3, 2, 1]).

- If list is empty or has one element, the reverse is itself
- Otherwise, reverse elements 2 to n, and append the first

```
def reverse_list(xs):
    if len(xs) <= 1:
        return xs
    else:
        return reverse_list(xs[1:]) + [xs[0]]
```

Reversing a list

How can we recursively reverse a list ([1, 2, 3] $\rightarrow$ [3, 2, 1]).

- If list is empty or has one element, the reverse is itself
- Otherwise, reverse elements 2 to n, and append the first

```
def reverse_list(xs):
    if len(xs) <= 1:
        return xs
    else:
        return reverse_list(xs[1:]) + [xs[0]]
```

Palindromes

A palindrome is a word that reads the same from both ways, such as *radar* or *level*.

Let's write a function that checks whether a given word is a palindrome.

The recursive idea

Given a word, such as *level*, we check:

- whether the first and last character are the same
- whether the string with first and last character removed are the same

Base case

What's the base case in this case?

- The empty string is a palindrome
- Any 1 letter string is a palindrome

Base case

What's the base case in this case?

- The empty string is a palindrome
- Any 1 letter string is a palindrome

Implementation

```
def is_palin(s):
    '''returns True iff s is a palindrome'''
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and is_palin(s[1:-1])
```

What is an iterative solution?

Implementation

```
def is_palin(s):
    '''returns True iff s is a palindrome'''
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and is_palin(s[1:-1])
```

What is an iterative solution?

Numerical integration

Suppose we want to numerically integrate some function f:

$$A = \int_a^b f(x)dx$$

Trapezoid rule:

$$\begin{aligned} A &= \int_a^b f(x)dx \\ &= \int_a^{r_1} f(x)dx + \int_{r_1}^{r_2} f(x)dx + \ldots + \int_{r_{n-1}}^{b} f(x)dx \\ &\approx \frac{h}{2}(f(a) + f(b)) \quad + \quad h(f(r_1) + f(r_2) + \ldots + f(r_{n-1})) \end{aligned}$$

Numerical integration

Suppose we want to numerically integrate some function f:

$$A = \int_a^b f(x)dx$$

Trapezoid rule:

$$\begin{aligned} A &= \int_a^b f(x)dx \\ &= \int_a^{r_1} f(x)dx + \int_{r_1}^{r_2} f(x)dx + \ldots + \int_{r_{n-1}}^{b} f(x)dx \\ &\approx \frac{h}{2}(f(a) + f(b)) \quad + \quad h(f(r_1) + f(r_2) + \ldots + f(r_{n-1})) \end{aligned}$$

Trapezoid rule

```
def trapezoid(f, a, b, N):
    '''integrates f over [a,b] using N steps'''
    if a > b:
        a, b = b, a
    # step size
    h = float(b-a)/N
    # running sum
    s = h/2 * (f(a) + f(b))
    for k in xrange(1, N-1):
        s += h * f(a + h*k)
    return s
```

Forget the math / code: This function approximates the area under f between a and b using N points.

Key point

- If function is flat, then we don't need many points.
- If function is very wiggly, we need a lot of points.

So:

- How many points do we need?
- What if function is flat in some areas, wiggly in others?

Adaptive integration

Idea: Adaptively space points based on local curvature of function.

Areas where function is flat: few points, areas where function is wiggly: many points.

```
def ada_int(f, a, b, tol=1.0e-6, n=5, N=10):
    area = trapezoid(f, a, b, N)
    check = trapezoid(f, a, b, n)
    if abs(area - check) > tol:
        # bad accuracy, add more points to interval
        m = (b + a) / 2.0
        area = ada_int(f, a, m) + ada_int(f, m, b)
    return area
```

Note: we do not need to use trapezoid rule.

Adaptive integration

Idea: Adaptively space points based on local curvature of function.

Areas where function is flat: few points, areas where function is wiggly: many points.

```
def ada_int(f, a, b, tol=1.0e-6, n=5, N=10):
    area = trapezoid(f, a, b, N)
    check = trapezoid(f, a, b, n)
    if abs(area - check) > tol:
        # bad accuracy, add more points to interval
        m = (b + a) / 2.0
        area = ada_int(f, a, m) + ada_int(f, m, b)
    return area
```

Note: we do not need to use trapezoid rule.

Exercise

Write a recursive function that computes a^b for given a and b, where b is an *integer*. (Do not use $**$)

Exercise

Base case: $b = 0$, $a^b = 1$.

Recursive step: (be careful) there are actually two options, one for if $b < 0$ and one for if $b > 0$.

```
def power(a,b):
  if b == 0:
    return 1
  elif b > 0:
    return a*power(a,b-1)
  else:
    return (1./a)*power(a,b+1)
```

Pitfalls

Recursion can be very powerful, but there are some pitfalls:

- Have to ensure you always reach the base case.
- Each successive call of the algorithm must be solving a simpler problem
- The number of function calls shouldn't explode. (see exercises)
- An iterative algorithm is always faster due to overhead of function calls. (However, the iterative solution might be much more complex)