INTRODUCTION TO PROGRAMMING
WEEK 10 - EXCEPTIONS AND GDATA

FINAL EXAM

Your final exam is **next week**! The first part of the exam is due before class next week.

To pass this class, you will need to:

- Get a grade of B or better on final exam
- Complete ALL homework assignments before 12:00AM PST on 3/29

I will be emailing everyone with the homework I am currently missing from you. Please start the take-home part of the final as early as possible as it is quite difficult. I am always available to help, feel free to send me emails, pings, or schedule time to meet up.

GETTING THE DEMO FILES

This week's class will move very fast, so I have prepared demo files for you to follow along with.

Please run the following from your `week10` directory:

```
cp -r /home/alberthwang/python/week10 demo_files
```

All the examples below reference something in the starter files. Please note that these starter files may change, but I will endeavor to keep them as up to date as possible. To refresh your version, just run:

```
rm -r demo_files
cp -r /home/alberthwang/python/week10 demo_files
```

EXCEPTIONS

While programming in python, you will often encounter 2 things that end program execution - **errors** and **exceptions**. Errors are mistakes in the code, such as forgetting to add colons to conditional statements, forgetting to indent code blocks, or misspelling the names of variables.

```
if counter == 3
  print 'this will not print!'
```

When writing more complex scripts, however, you will often find yourself encountering exceptions, which are errors detected at runtime that are not caused by an syntactical errors. This is especially true of scripts that take user input.

```
def main():
  num = raw_input('enter a number: ')
```

```
  new_num = int(num) + 4
  print 'plus 4: ' + str(new_num)
```

Run this script and when prompted, enter the string "**seven**". It will throw the following error:

```
Traceback (most recent call last):
  File "example1.py", line 9, in <module>
    main()
  File "example1.py", line 5, in main
    new_num = int(num) + 4
ValueError: invalid literal for int() with base 10: 'seven'
```

By reading the **Traceback** of the execution, we see that the exception occurred when attempting to cast the string "seven" to an integer. The exception that was raised was a ValueError type exception. **In reality, there is not a whole lot that you could have done to prevent the user from breaking the script, and it's not a pleasant user experience to have this confusing error displayed on the page. Instead, it would be much nicer to tell the user they entered an invalid integer and prompt them again.**

Exceptions can be quite disruptive, and you don't want them to end execution. What if the user had entered a list of numbers to add 4 to? It would break at the first error and not add the rest! The user would then have to re-enter all the numbers after the error!

Handling Exceptions - Try/Except

The `Try` clause tells the interpreter to attempt to execute a certain code block and raise exceptions if they occur. The `Except` clause tells the interpreter what to do if an Exception of a certain kind is raised. Alter the code in the last script to be this:

```
def main():
  num = raw_input('enter a number: ')

  try:
    new_num = int(num) + 4

  except ValueError:
    print 'oops! that wasn\'t an integer...try again.'
    num = raw_input('enter a number: ')
    new_num = int(num) + 4

  print 'plus 4: ' + str(new_num)
```

Now when you enter "seven" in the prompt, you get this:

```
enter an integer: seven
oops! that wasn't an integer...try again.
enter an integer: 7
plus 4: 11
```

Notice here that we are specifying the type of exception to be caught, `ValueError`. There

are a few common types of exceptions that you can find here:

http://docs.python.org/library/exceptions.html

While - Try/Except

One common practice with user-input based scripts is to couple `try-except` clauses with the **`while`** statement to keep prompting the user until a valid input is used. Change the main function to this:

```
def main():
  while True: # infinite loop
    try:
      num = raw_input('enter an integer: ')
      new_num = int(num) + 4
      print 'plus 4: ' + str(new_num)
      break # if valid integer, break out of while loop

    except ValueError:
      print 'oops! that wasn\'t an integer.'
```

When you run the script and enter invalid numbers:

```
enter an integer: seven
oops! that wasn't an integer.
enter an integer: 7?
oops! that wasn't an integer.
enter an integer: 7
plus 4: 11
```

Use the `try/except` clauses to to catch exceptions at run-time and handle them gracefully. When using the `except` clause, put the type of exception after "`except`" (such as `ValueError`).

You can find the exception to throw by running the script and purposefully breaking it in the way you think it will be broken.

Multiple Exception Types

When you need to catch multiple kinds of exceptions, just list `except` statements under each other, sort of like `if` and `elif` statements. Try changing the code of `main()` to this:

```
def main():
  companies = ['google', 'twitter', 'microsoft', 'intel']

  while True:
    try:
```

```
      rank = raw_input('Company Rank: ')
      print companies[rank]
      break

    except ValueError:
      print 'oops! please enter a valid int for rank'

    except IndexError:
      print 'oops! there is no company with that rank!'
```

Running the script above should give you this output:

```
enter the rank of the company you want to see: seven
oops! please enter a valid int for rank
enter the rank of the company you want to see: 7
oops! there is no company with that rank!
enter the rank of the company you want to see: 1
google
```

General Except Clause

Occasionally you might write a script where any number of kinds of exceptions might be raised, but whatever they are, you want to take the same action. For instance, if you wrote a script to process data in a CSV file and you know that the data might be malformed, you can write a general except clause that will handle all possible exceptions that are thrown in the "`Try`" code block:

```
def main():
  data = [1, 2, 'three', 0, 5, 6]
  sum = 0
  errors = []

  for datum in data:
    try:
      sum += float(10.0/datum)
    except:
      errors.append(datum)

  print 'here is a sum of the 10/values: ' + str(sum)

  print 'These values were omitted due to errors:'
  for error in errors:
    print error
```

This script will exclude 'three' and 0 from the calculations even though they raise 2 different exceptions (`ValueError`, `ZeroDivisionError`) because they are caught with a general `except` clause that handles any exception thrown in the `try` block at runtime.

While it is nice to have the general `except` clause in a script, it's often better to know the exact

nature of the exceptions being found in your script so that you can refine your script to account for them (for instance to strip punctuation or special characters). In general, it's not a good idea to have a general `except` clause unless absolutely necessary.

Terminating Execution with Exceptions

While the try-except clauses help to catch exceptions and handle them gracefully, sometimes you do actually want to exit a program if an exception is caught (or if the same exception is caught too many times). To do this, you'll want to use the `sys` module's `exit()` function which will terminate execution of the script with a message.

Take a look at this sample script that tries to take an integer input from the user 4 times.

```
import sys #1

def main():
  for try_num in range(4): #2
    try:
      num = int(raw_input('Enter an integer: '))
      break
    except ValueError:
      if try_num == 3: #3
        sys.exit('Failed to give valid integer in 4 tries.')
      else:
        print 'Invalid Integer...try again.' #4

  print 'Your number plus two - ' + str(num + 2) #5

  print 'Thanks for using plus two script!'
```

When we run the above script and purposefully put incorrect integers, the output looks like this:

```
Enter an integer: w
Invalid Integer...try again.
Enter an integer: d
Invalid Integer...try again.
Enter an integer: e
Invalid Integer...try again.
Enter an integer: p
Failed to give valid integer in 4 tries.
```

Notice how the last lines that involve adding two and printing out a "thanks!" the user aren't even executed if the user fails to enter a valid integer in 3 tries.

Comment Explanations:

1. To use `sys.exit()` you need to first import the `sys` module
2. Give the user 3 tries by asking in a for loop
3. If the user fails (except ValueError) and it's the 3rd try, then exit the script with a message
4. If it's not the 3rd try, just warn them and give them another chance

5. If the script didn't exit, then print out the number + 2 and some thanks

GDATA

GData is a Google code library for accessing google services such as spreadsheets, docs, and calendar. It's a powerful library that comes in several languages. For this class, we'll be looking at the gdata Python library for **spreadsheets**. You can use this code to read/write data to trix.

SETTING UP GDATA

From gswitch

If you are on gswitch, then you are in luck! To use gdata on gswitch, you simply need to run the following line from the command line once you begin your session:

```
export PYTHONPATH=$PYTHONPATH:/usr/local/google/third_party
```

This line tells the OS to look for Python files in `/usr/local/google/third_party` on gswitch which has the gdata library files.

Alternatively, you can also edit your `.bashrc` file which is in your home directory so that you don't have to run that line at the beginning of each session. To edit it, just use vi like so:

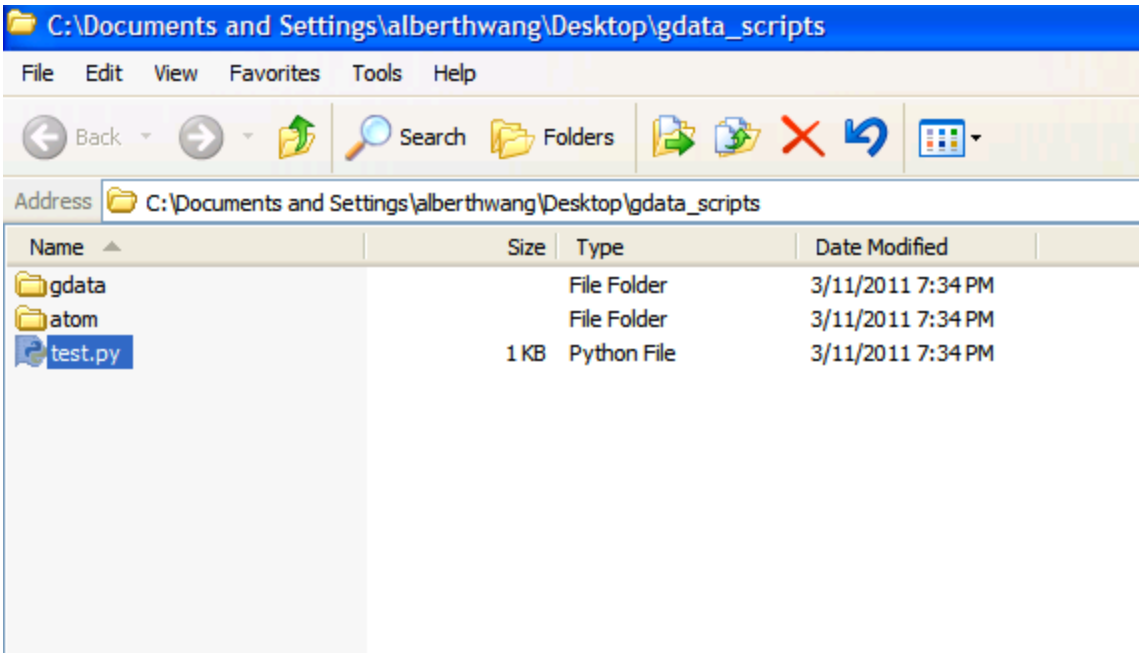
```
alberthwang@gswitch:~$ vi .bashrc
```

Once you have it opened, tack on the `export PYTHONPATH=...` line to the bottom of the file.

From your own machine

If you want to use gdata from your own machine, you have to first download the library. You can download it here.

Download the latest **zip file** and extract the files. Then, drill down into the downloaded files to get to the `src` directory (should contain 2 directories, `gdata` and `atom`). Now copy the `gdata` and `atom` directories into your working directory. All in all, your working directory should look like this:



Clients

To work with trix data, we need to first create a Client which is essentially an authenticated object that has permission to manipulate the trix page(s) in question. Create a demo file and add the following content.

```
import atom
import gdata.spreadsheet.service
import getpass

def GetTrixClient():
  gd_client = gdata.spreadsheet.service.SpreadsheetsService()#1
  gd_client.email = raw_input('ldap: ') + '@google.com' #2
  gd_client.password = getpass.getpass('password:')#3
  gd_client.source = 'i2p gdata demo'
  gd_client.ProgrammaticLogin() #4
  return gd_client
```

Comment explanations:

1. To start, we need to first create a `SpreadsheetsService` object which will be our client object that can read/write data from trix.
2. First, we prompt the user for his/her ldap
3. Next, we prompt the user for his/her account password (more explanation below)
4. Once this information has been obtained, calling the `ProgrammaticLogin` method

will attempt to authenticate the credentials for use by the `SpresheetsService` Client object.

getpass

One of the most common reasons that users need to be prompted in Python scripts is to obtain their user credentials.Of course, we could always prompt the user with:

```
raw_input('Password: ')
```

This, however, is not secure, because whatever is typed in the terminal echoes back to the user. If someone were walking behind you as you did this, your credentials would be compromised! Thus for passwords, use the python module, `getpass`.

```
# Get user password
password = getpass.getpass('Password:')
```

It works exactly like `raw_input()` except that it doesn't echo your password back. Also, to use it, you need to import the `getpass` module.

```
import getpass
```

Please use `getpass` to prompt the user for secure information and not `raw_input()` in all your scripts!!

Access Issues

The script may through an error even though you're logging in with your correct username/ password. This is because Google recently moved to a new (*sarcastic hooray*) security protocol called Application Specific Passwords (ASPs). Please us this doc to fix your access issues:

https://docs.google.com/a/google.com/document/d/1Wc_-GL__C2Dq2zwhnMnlTTEjp--hBZlMtBdKeoWW1HE/edit?hl=en#

Feeds

Data from trix pages always come in the form of feeds. A data feed is essentially a set of data that contains information about an object. Almost every object in a trix page has a feed, so we will encounter worksheet feeds, list (row) feeds, cell feeds, and more while working with trix.

Getting Worksheets Feed

When working with trix pages, we first need to identify which sheet of the trix we want to manipulate. While this would seem simple, each worksheet within a trix has its own unique ID

which needs to be extracted from a worksheets feed.

For these demonstrations i'll be using this trix.

Now add the following main() function to your script.

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient() #1

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw' #2

  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry #3

  # For each worksheet, print its title and full ID url
  for worksheet_feed in worksheet_feeds: #4
    print worksheet_feed.title.text #5
    print worksheet_feed.id.text #6
    # This is the worksheet ID we'll need
    print worksheet_feed.id.text.split('/')[-1] #7
    print ''

  # Obtain a worksheet's ID through its worksheet number
  print worksheet_feeds[0].id.text.split('/')[-1] #8
```

Comment Explanations:

1. First, we create a `SpreadsheetService` Client by calling the function that we wrote above.
2. To work with a trix, we need its key, which is a unique identifier for that trix page. A trix's key can be found in its url (in the key= parameter, up to an ampersand)
3. To fetch a feed of all the worksheet feeds that belong to that trix, we call the client's `GetWorksheetsFeed` method with the key of the trix we're working with.
4. Here we are iterating through the worksheet feeds we extracted
5. Each worksheet feed has a `title` property that holds the name of the worksheet
6. Each worksheet feed also has an `id` property that holds the worksheet's full ID url. We're actually only interested in the last part of this ID URL (after the last forward slash) which holds the actual ID of the worksheet
7. To extract the actual ID of the worksheet (last part of the ID URL) we do a split on forward slashes and get the last element of the list.
8. Thus, to get the ID of a trix worksheet, we just need to pass the numerical index of the sheet to the worksheets feed (since it's positionally indexed) and pull its ID with this formula. In this case, we are indicating that we want the ID of the first worksheet of the trix, thus the index 0.

Reading data from trix as list of dictionaries

To get the data out of trix worksheet, we need to use the Client's `GetListFeed` method which will return the trix data by row and takes 2 parameters - a trix key and a worksheet's ID. Take a look at the code below.

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient()

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw'

  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry

  # Obtain the first sheet's ID
  wks_id = worksheet_feeds[0].id.text.split('/')[-1] #1

  # Get data out of the worksheet with GetListFeed
  data_feed = trix_client.GetListFeed(trix_key, wks_id).entry #2

  # Map data in each row to a dict, and everything to list of dicts
  data = [] # to be a list of dictionaries
  for row_feed in data_feed: #3
    row_dict = {}
    for key, val in row_feed.custom.iteritems(): #4
      row_dict[key] = val.text #5
    data.append(row_dict)

  # Data in trix as a list of dicts
  print data #6
```

Comment Explanations:

1. First we get the ID of the worksheet we're interested in working with (the first worksheet, index 0).
2. Next, we get the row feeds out of that worksheet by calling the Client's `GetListFeed` method and passing the trix key and worksheet ID we're interested in pulling from
3. Iterating through each row's feed, we can extract the information from that row
4. Each row feed has a custom property that holds the row's data in a dictionary of custom objects. These custom object still need to be further decoded to get the info we want, so we have to iterate through this dictionary.
5. For each element of the row feed's custom dictionary, we want to extract the text of the custom object which holds the data we need.

6. Finally, we print all the trix data as a single list of dictionaries.

Header Normalization

One thing you might have noticed with gdata trix is that headers are always normalized (standardized) to be lowercased with only certain punctuation. If you look at the dictionaries you pulled, all the keys are in the form `headerwithnospace` from whatever form it used to be. Here are some examples of how this will happen.

Header: `Target Bonus`
Normalized: `targetbonus`

Header: `LDAP`
Normalized: `ldap`

Header: `Compa-Ratio`
Normalized: `compa-ratio`

Header: `Total_Match`
Normalized: `totalmatch`

Header: `Albert's Column`
Normalized: `albertscolumn`

What this means is that for a trix page like this:

LDAP	**Full Name**	**Target-Bonus**
satishm	Satish Musunuru	100
alberthwang	Albert Hwang	92

GDATA will extract the row data in a dictionaries like so:

```
[{'ldap': 'satishm', 'fullname': 'Satish Musunuru', 'target-bonus': '100'}
{'ldap': 'alberthwang', 'fullname': 'Albert Hwang', 'target-bonus': '92'}]
```

Then, to access the data you extracted, you need to use the normalized header.

```
print data[1]['fullname'] # 'Albert Hwang'
```

Adding a row to a trix worksheet

If you want to insert data into a trix, you need to use the client's `InsertRow` method. The code sample below demonstrates usage of this method. Change main() to this:

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient()

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw'

  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry

  # Obtain a worksheet's ID through its worksheet number
  wks_id = worksheet_feeds[0].id.text.split('/')[-1]

  # Insert a row of data into trix
  row = {'ldap': 'alonsor', #1
         'name': 'Alonso Rukibayhunga',
         'department': 'Product'}

  trix_client.InsertRow(row, trix_key, wks_id) #2

  print 'done.'
```

1. The row to be inserted must be a dictionary with the same headers as the trix worksheet itself.
2. To insert it, you just call the InsertRow method of the trix client with the data, the trix key, and the worksheet ID you want to insert into.

The data in the dictionary that you write to the trix needs to have string values, so you can't insert ints, floats, etc. without casting them with str().

Editing a Row

To edit a row in a trix page, you'll want to use the client's `UpdateRow` method which takes a row feed and the new data you want to change the row to. Change main() to this to **alter the 2nd row** of the data set:

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient()

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw'
```

```
  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry

  # Obtain a worksheet's ID through its worksheet number
  wks_id = worksheet_feeds[0].id.text.split('/')[-1]

  # Get data out of worksheet as a List Feed
  data_feed = trix_client.GetListFeed(trix_key, wks_id).entry

  # Update the 2nd row of data
  row = {'ldap': 'eimearm', #1
         'name': 'Eimear McCurry',
         'department': 'Sales'}

  trix_client.UpdateRow(data_feed[1], row) #2

  print 'done.'
```

Comment Explanations:

1. The dictionary of the row to be updated must have the same headers/keys as the original trix
2. The UpdateRow method takes two parameters, the row feed to change (in this case, the 2nd row...index 2) and the new dictionary of data to replace that row.

Deleting a row from trix

To delete a row from a trix page, you want to simply call the `DeleteRow` method of the Client. Change main() to this:

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient()

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw'

  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry

  # Obtain a worksheet's ID through its worksheet number
  wks_id = worksheet_feeds[0].id.text.split('/')[-1]

  # Get data out of worksheet with List-based feed
  data_feed = trix_client.GetListFeed(trix_key, wks_id).entry
```

```
  # Delete the 2nd row of data
  trix_client.DeleteRow(data_feed[1]) #1

  print 'done.'
```

Comment Explanations:

1. Here again, you need to just specify the row feed that you want to delete. The index, 1, here indicates that you are interested in deleting the 2nd row in the worksheet.

Querying data from a trix

Sometimes, you want to query specific rows out of a trix for manipulation, and not all the rows. I won't go into detail in this session, but you can always download my sample file from my filer. I also have the sample code for querying trix here. The following script will look for all rows who's "Department" column has the value "Sales". You can write relatively complex queries and find the data you're looking for in trix.

```
def main():
  # Create a trix client object
  trix_client = GetTrixClient()

  trix_key = 'tyzfb_k4_k_QA9R2pk8zrxw'

  # Get the worksheets feed - info for every worksheet
  worksheet_feeds = trix_client.GetWorksheetsFeed(trix_key).entry

  # Obtain a worksheet's ID through its worksheet number
  wks_id = worksheet_feeds[0].id.text.split('/')[-1]

  # Generate query
  department_query = gdata.spreadsheet.service.ListQuery()
  department_query.sq = 'department=Sales'
  feed_url = 'https://spreadsheets.google.com/feeds/list/'
  feed_url += trix_key + '/' + wks_id + '/private/full'
  department_query.feed = feed_url

  # Get Data Feed
  data_feed = trix_client.GetListFeed(trix_key, wks_id,
                                      query=department_query).entry

  # Map data in a row to a dict, and everything to list of dicts
  data = []
  for row_feed in data_feed:
    row_dict = {}
    for key, val in row_feed.custom.iteritems():
```

```
      row_dict[key] = val.text
    data.append(row_dict)

# Data in trix as a list of dicts
print data
```

The code above pulls the queried rows into a list of dictionaries much like the example above.

An Important Note on Security

Please...never, ever hard code your credentials into your scripts as tempting as that may seem. Files on your filer can obviously been seen by others so by putting your credentials into your script, you risk compromising all your data.