- week: We calculate the week for each date so the data can be grouped by week.
- price: This corresponds to the Adj Close column from the CSV; the closing price for that symbol on that date.

When we're done forming our table, we need to commit the changes to the database.

```
def create_database(db_conn):
    cur = db_conn.cursor()
    cur.execute('CREATE TABLE stocks (symbol text, year integer, month integer, day integer, week integer, price real)')
    db_conn.commit()
```

Import the Data

This function imports data for a symbol from input_file using the database connection object db_conn. This is very similar to the Structred Data exercise, except we do a SQL insert for each record. The question marks get associated with the values in the tuple (the second argument to cursor.execute after the INSERT command string).

After we've iterated over all records for the inserts, we need to commit them to the database with db_conn.commit().

```
def import_data(symbol, input_file, db_conn):
    with open(input_file, 'r') as infile:
        symbol = symbol.strip().upper()
        reader = DictReader(infile)
        cursor = db_conn.cursor()

        for record in reader:
            dt = datetime.strptime(record['Date'], '%Y-%m-%d')
            week = dt.isocalendar()[1]
            price = float(record['Adj Close'])
            cursor.execute("INSERT INTO stocks VALUES (?, ?, ?, ?, ?, ?)", (symbol, dt.year, dt.month, dt.day, week, price))

        db_conn.commit()
```

Making SQL do all the Work

In the Structured Data Exercise, the student had to manually group the data in a dictionary by week (really a (year,week) tuple). SQL can do this for us, and even calculate the average. We break down each line of the query as follows:

- SELECT: We want the year, the week, and the average for the prices for the days on that year week, so we use SELECT to pick those columns.
- FROM: We're working with the stocks table, so we say FROM stocks.

- WHERE: To put conditions on a SQL query, we use the WHERE clause. Here our only condition is that we only want data associated with a certain symbol (AAPL in this case).
- GROUP BY: We need to group the data by week, and since we don't want the same week in two different years to get grouped together, we have to group by the year and the week, hence GROUP BY year, week.
- ORDER BY: To display our data in descending order by date, we have to use ORDER BY. SQL even allows mixed ascending and descending subgroups, so we have to specify that we want both the year and the week in descending order.

We append each result from our query into an empty list, which gets returned by our function.

```
def weekly_averages(symbol, db_conn):
    cur = db_conn.cursor()
    results = []

    for result in cur.execute('''SELECT year, week, avg(price)
                                 FROM stocks
                                 WHERE symbol=?
                                 GROUP BY year, week
                                 ORDER BY year DESC, week DESC''', (symbol,)):
        results.append(result)

    return results
```

Execute!

Here we create our database in aapl.db. Note that it will raise an exception if the table has already been created in the database. If you simply comment out the create_database() call to get around this, then be careful since import_data() will insert the data again, so you'll have the double entries in your stocks table. Restart this notebook and delete aapl.db to get a truly fresh start.

```
db_file = 'aapl.db'

db_conn = sqlite3.connect(db_file)
create_database(db_conn)
import_data('AAPL', 'aapl.csv', db_conn)
```

Now that the data is in the database, we can call our weekly_averages() query function. This will just display the list of results.

```
weekly_averages('AAPL', db_conn)
```

Module: Structured Data: CSV, XML, and JSON

0

(b)(3)-P.L. 86-36

Updated over 1 year ago by [redacted] in COMP 3321

3 5 701 263

python fcs6

(U) Read, write, and manipulate CSV, XML, JSON, and Python's custom pickle and shelve formats.

Recommendations

UNCLASSIFIED

(U) Setup

(U) For this notebook, you will need the following files:

- user_file.csv
- user_file.xml.

(U) Right-click each to download and "Save As," then, from your Jupyter home, navigate to the folder containing this notebook and click the "Upload" button to upload each file from your local system.

(U) Introduction: It's Sad, But True

(U) Much of computing involves reading and writing structured data. Too much, probably. Often that data is contained in files--not even a database. We've already worked with opening, closing, reading from, and writing to text files. We've also frequently used `str` ing methods. At first, it might seem that that's all we need to work with **CSV**, **XML**, and other structured data formats.

(U) After all, what could go wrong with the following?

```
my_csv_file = open('user_file.csv', 'r')
```

```
csv_lines = my_csv_file.readlines()
```

```
comma_separated_records = [line.split(',') for line in csv_lines]
```

```
xml_formatter = """<person>
                    <name>{}</name>
                    <address>{}</address>
                    <phone>{}</phone>
                    </person>"""
```

```
xml_records = "\n".join([xml_formatter.format(*record) for record in comma_separated_records])
```

```
xml_records = "<people>" + xml_records + "</people>"
```

```
with open('file.xml', 'w') as f:
    f.write(xml_records)
```

(U) In a rapidly-developed prototype with controlled input, this may not cause a problem. Given the way the real world works, though, someday this little snippet from a one-off script will become the long-forgotten key component of a huge, enterprise-wide project. Somebody will try to feed it data in just the wrong way at a crucial moment, and it will fail catastrophically.

(U) When that happens, you'll wish you had used a fully-developed library that would have had a better chance against the malformed data. Thankfully, there are several--and they actually aren't any harder to get started with.

(U) Comma Separated Values (CSV)

(U) The most exciting things about the `csv` module are the `DictReader` and `DictWriter` classes. First, let's look at the plain vanilla options for reading and writing.

```
import csv
```

```
f = open('user_file.csv')
```

```
reader = csv.reader(f)
```

```
header = next(reader)
```

```
all_lines = [line for line in reader]
```

```
all_lines.sort()
```

```
g = open('user_file_sorted.csv', 'w')
writer = csv.writer(g)
writer.writerow(header)
writer.writerows(all_lines)
g.close()
```

(U) CSV readers and writers have other options involving *dialects* and *separators*. Note that the argument to `csv.reader` must be an open file (or file-like object), and the reading starts at the current cursor position.

(U) Accessing categorical data positionally is not ideal. That is why `csv` also provides the `DictReader` and `DictWriter` classes, which can also handle records with more or less in them than you expect. When given only a file as an argument, a `DictReader` uses the first line as the keys for the remaining lines; however, it is also possible to pass in `fieldnames` as an additional parameter.

```
f.seek(0)
d_reader = csv.DictReader(f)
records = [line for line in d_reader]
```

(U) To see the differences between reader and DictReader, look at how we might extract cities from the records in each.

```
# for the object from csv.reader
cities0 = [record[2] for record in all_lines]
# for the object from csv.DictReader
cities1 = [record['city'] for record in records]
cities0 == cities1
```

(U) In a `DictWriter`, the `fieldnames` parameter is required and headers are not written by default. If you want one, add it with the `writeheader` method. If the `fieldnames` argument does not include all the fields for every dictionary passed into the `DictWriter`, the keyword argument `extrasaction` must be specified.

```
g = open('names_only.csv', 'w')
d_writer = csv.DictWriter(g, ['name', 'primary_workstation'], extrasaction='ignore')
d_writer.writeheader()
```

```
d_writer.writerows(records)
```

```
g.close()
```

(U) Javascript Object Notation (JSON)

(U) JSON is another structured data format. In many cases it looks very similar to nested Python `dict` s and `list` s. However, there are enough notable differences from those (e.g. only single quotation marks are allowed, boolean values have a lowercase initial letter) that it's wise to use a dedicated module to parse JSON data. Still, *serializing* and *deserializing* JSON data structures is relatively painless.

(U) For this section, our example will be a list of novels:

```
import json

novel_list = []

novel_list.append({'title': 'Pride and Prejudice', 'author': 'Jane Austen'})

novel_list.append({'title': 'Crime and Punishment', 'author': 'Fyodor Dostoevsky'})

novel_list.append({'title': 'The Unconsoled', 'author': 'Kazuo Ishiguro'})

json.dumps(novel_list)  # to string

with open('novel_list.json','w') as f:
    json.dump(novel_list,f)  # to file

the_hobbit = '{"title": "The Hobbit", "author": "J.R.R. Tolkien"}'

novel_list.append(json.loads(the_hobbit))  # from string

with open('war_and_peace.json') as f:  # <-- if this file existed
    novel_list.append(json.load(f))  # from file
```

(U) By default, the `load` and `loads` methods return Unicode strings. It's possible to use the **json** module to define custom encoders and decoders, but this is not usually required.

(U) Extensible Markup Language (XML)

(U) This lesson is supposed to be simple, but XML is complicated. We'll cover only the basics of reading data from and writing data to files in a very basic XML format using the **ElementTree**API, which is just the most recent of at least three approaches to dealing with XML in the Python Standard Library. We will not discuss attributes or namespaces at all, which are very common features of XML. If you need to process lots of XML quickly, it's probably best to look outside the standard library (probably at a package called **lxml**).

(U) Although there are other ways to get started, an `ElementTree` can be created from a file by initializing with the keyword argument `file`:

```
from xml.etree import ElementTree
```

```
xml_file = open('user_file.xml')
```

```
user_tree = ElementTree.ElementTree(file=xml_file)
```

(U) To do much of anything, it's best to pull the root element out of the `ElementTree`. Elements are iterable, so they can be expanded in list comprehensions. To see what is inside an element, the **ElementTree** module provides two class functions: `dump` (which prints to screen and returns `None`) and `tostring`. Each node has a `text` property, although in our example these are all empty except for leaf nodes.

```
root_elt = user_tree.getroot()
```

```
users = [u for u in root_elt]
```

```
print(ElementTree.tostring(users[0]))
```

```
u_children = [x for x in users[0]]
```

```
u_children[2].text
```

```
u_children[2].text = 'north-x5-1234'
```

```
ElementTree.dump(users[0])
```

(U) To get nested descendant elements directly, use `findall`, which returns a list of all matches, or `find`, which returns the first matched element. Note that these are the actual elements, not copies, so changes made here are visible in the whole element tree.

```
all_usernames = root_elt.findall('user/name/username')
```

```
[n.text for n in all_usernames[:10]]
```

(U) To construct an XML document:

- make an `Element`,
- `append` other `Element`s to it (repeating as necessary),
- wrap it all up in an `ElementTree`, and
- use the `ElementTree.write` method (which takes a file *name*, not a `file` object).

```
apple = ElementTree.Element('apple')
apple.attrib['color'] = 'red'
apple.set('variety', 'honeycrisp')
apple.text = "Tasty"
ElementTree.dump(apple)
fruit_basket = ElementTree.Element('basket')
fruit_basket.append(apple)
fruit_basket.append(ElementTree.XML('<orange color="orange" variety="navel"></orange>'))
ElementTree.dump(fruit_basket)
fruit_tree = ElementTree.ElementTree(fruit_basket)
fruit_tree.write('fruit_basket.xml')
```

(U) Bonus Material: Pickles and Shelves

(U) At the expense of compatibility with other languages, Python also provides built-in serialization and data storage capabilities in the form of the **pickle** and **shelve** modules.

(U) Pickling

P.L. 86-36

```
import pickle
pickleme = {}
pickleme['Title'] = 'Python is Cool'
pickleme['PageCount'] = 543
pickleme['Author'] = '[redacted]'
```

```
with open('/tmp/pickledData.pick', 'wb') as p:
    p = pickle.dump(pickleme, p)
```

```
with open('/tmp/pickledData.pick', 'rb') as p:
    p = pickle.load(p)
```

```
print(p)
```

(U) Shelving

(U) Creating a Shelve

P.L. 86-36

```
import shelve

pickleme = {}

pickleme['Title'] = 'Python is Cool'

pickleme['PageCount'] = 543

pickleme['Author'] = '[redacted]'

db = shelve.open('/tmp/shelve.dat')

db['book1'] = pickleme

db.sync()

pickleme['Title'] = 'Python is Cool -- The Next Phase'

pickleme['PageCount'] = 123

pickleme['Author'] = '[redacted]'

db['book2'] = pickleme

db.sync()

db.close()
```

(U) Opening a Shelve

```
db = shelve.open('/tmp/shelve.dat')
z = db.keys()
a = db['book1']
b = db['book2']
print(a)
print(b)
print(z)
db.close()
```

(U) Modifying a Shelve

```
db = shelve.open('/tmp/shelve.dat')
z = db.keys()
a = db['book1']
b = db['book2']
print(a)
print(b)
print(z)
a['PageCount'] = 544
b['PageCount'] = 129
db['book1'] = a
db['book2'] = b
```

```
db.close()
```

Module: System Interaction

(b)(3)-P.L. 86-36

0

Updated over 3 years ago by [redacted] in COMP 3321

3 167 62

python fcs6

(U) Basic operating system interaction using the os, shutil, and sys modules.

Recommendations

UNCLASSIFIED

(U) Introduction

(U) Python provides several modules for interacting with your operating system and the files and directories it holds. We will talk about three: `os`, `shutil`, and `sys`.

(U) Be aware that while this notebook is unclassified, your output may not be (depending on the files you're displaying).

(U) os Module:

(U) This module helps you interact with the operating system, providing methods for almost anything you would want to do at a shell prompt. On POSIX systems, there are over 200 methods in the **os** module; we will just cover the most common ones. Be aware that the **os** module includes methods that are not cross-platform compatible; the documentation is helpfully annotated with *Availability* tags.

(U) Directory discovery and transversal is pretty basic:

```
import os
```

```
os.getcwd()
```

```
os.chdir('/tmp')  # Unix dir--choose different dir for Windows
```

```
os.listdir()
```

```
os.getcwd()
```

```
walker = os.walk(os.curdir)
```

```
type(walker)
```

```
list(walker)
```

(U) Avoid one common confusion: `os.curdir` is a module constant (`'.'` on Unix-like systems), while `os.getcwd()` is a function. Either one can be used in the method `os.walk` , which returns a generator that traverses the file system tree starting at the method's argument. Each successive value from the generator is a tuple of
`(directory, [subdirectories], [files])` .

(U) A variety of methods allow you to examine, modify, and create or remove directories and files.

```
f = open('new_temp_file.txt', 'w')
```

```
f.close()
```

```
os.stat('new_temp_file.txt')
```

```
os.mkdir('other_dir')
```

```
os.rename('new_temp_file.txt', 'other_dir/tempfile.txt')
```

(U) The **os.path** submodule provides additional functionality, including cross-platform compatible methods for constructing and deconstructing paths. Note that while it is possible to join a path completely, deconstructing a path occurs one element at a time, right to left.

```
sample_path = os.path.join('ford', 'trucks', 'f150')
```

```
sample_path
```

```
os.path.split(sample_path)
```

```
os.path.exists(sample_path)
```

(U) Information about the current environment is also available, either via specific methods or in the `os.environ` object, which functions like a dictionary of environment variables. If `os.environ` is modified, spawned subprocesses inherit the changes.

```
os.getlogin()
```

```
os.getuid()  # Unix

os.getgroups()  # Unix

os.environ

os.environ['NEW_TEMP_VAR'] = '123456'

os.uname()  # Unix
```

(U) shutil Module

(U) Living on top of the **os** module, **shutil** makes high-level operations on files and collections of files somewhat easier . In particular, functions are provided which support file copying and removal, as well as cloning permissions and other metadata.

```
import shutil

shutil.copyfile(src,dest) # overwrites dest

shutil.copymode(src,dest) # permission bits

shutil.copystat(src,dest) # permission bits and other metadata

shutil.copy(src,dest)     # works like cp if dest is a directory

shutil.copy2(src,dest)    # copy then copystat

shutil.copytree(src,dest)

shutil.rmtree(path)       # must be real directory, not a symlink

shutil.move(src,dest)     # works with directories
```

(U) sys Module

(U) The **sys** module provides access to variables and functions used or maintained by the Python interpreter; it can be thought of as a way of accessing features from the layer between the underlying system and Python. Some of its constants are interesting, but not usually useful.

```
import sys
```

```
sys.maxsize
```

```
sys.byteorder
```

```
sys.version
```

(U) Other module attributes are sometimes useful, although fiddling with them can introduce problems with compatibility. For instance, `sys.path` is a list of where Python will look for modules when `import` is called. If it is modified within a script, and then modules can be loaded from a new location, but there is no inherent guarantee that location will be present on a system other than your own! On the other hand, `sys.exit()` can be used to shut down a script, optionally returning an error message by passing a non-zero numeric argument.

Manipulating Microsoft Office Documents with win32com

(b)(3)-P.L. 86-36

0

Updated almost 3 years ago by [redacted] in COMP 3321

3 1 115 18

win32com python comp3321 pug

(U) Demonstration of using win32com to create and modify Microsoft Office documents.

Recommendations

(U) Manipulating Microsoft Office Documents with win32com

(U) Welcome To Automation with win32com!

(U) The win32com module connects Python to the Microsoft Component Object Model interface that enables inter-process communication and object creation within Microsoft Office applications.

(U) **Note: win32com only exists on Windows platforms, so this notebook will not run on LABBENCH.** In order to run this notebook, install Anaconda3 on your Windows platform and use jupyter-notebook.

(U) "Hello World" for Word

(U) We need to import the library, and open Word.

```
import win32com.client
word = win32com.client.Dispatch('Word.Application')
```

(U) `Dispatch` checks to see if Word is already open. If it is, it attaches to that instance. If you'd like to always open a new instance, use `DispatchEx`.

(U) By default, Word will start, but won't be visible. Set this to `True` if you want to see the application.

```
word.Visible = True
```

(U) Create a document and add some text, setting a font size that we like.

```
worddoc = word.Documents.Add()
worddoc.Content.Text = "Hello World"
worddoc.Content.Font.Size = 18
```

(U) Save the document and exit the application. Note that `win32com` bypasses the normal Python file object, so we need to account for the Windows directory separator.

(U//~~FOUO~~) Also, ClassifyTool may nag you for a classification. In order to prevent this, in Word, select the "ClassifyTool" tab, click on "Options", and under "When Closing Document", uncheck "Always show Classification Form", and click "Save".

```
worddoc.SaveAs('u:\\private\\jupyter\\win32com\\hello.docx')
word.Quit()
```

(U) That's it!

(U) More Elaborate Word Example

(U) There's another option for starting the application:

```
word = win32com.client.gencache.EnsureDispatch('Word.Application')
```

(U) This can take slightly longer, but enables access to win32com constants, which are required for some methods. The alternative is to look through the win32com documentation for the value of the constants you need.

(U) Let's take a look at a possible use case. Say we have reports in a particular format that we need to regularly generate. We can create a template with the sections that will be replaced. In this case, they are `ReportEvent`, `ReportTime`, and `ReportPlace`. First, download the template. Then open the template and create a dictionary with the sections and the data that will be used.

```
constants = win32com.client.constants                # save some future typing
word.Visible = True
worddoc = word.Documents.Open('u:\\private\\jupyter\\win32com\\demo_template.docx')
event_details = { "ReportEvent" : "[redacted]
                  "ReportTime"  : "[redacted]
                  "ReportPlace" : "[redacted]
                }
```

P.L. 86-36

(U) Now the magic happens. Lets iterate through the dictionary, replacing all of the sections withthe data.

```
# Execute( FindText, MatchCase, MatchWholeWord, MatchWildcards, MatchSoundsLike, MatchAllWordForms,
#          Forward, Wrap, Format, ReplaceWith, Replace)
for tag, data in event_details.items():
    _ = word.Selection.Find.Execute( tag, False, False, False, False, False,
                                     True, constants.wdFindContinue, False, data, constants.wdReplaceAll)
```

(U) We can add a couple of paragraphs of additional info, and we're done.

```
paragraph1 = worddoc.Paragraphs.Add()
paragraph1.Range.Text = 'Additional info\n'
footer = worddoc.Paragraphs.Add()
footer.Range.Text = 'Produced by me\n'
worddoc.SaveAs('u:\\private\\jupyter\\win32com\\demo_out.docx')
word.Quit()
```

(U)PowerPoint

(U) PowerPoint works very similarly. Again, download the template

```
ppt = win32com.client.Dispatch('PowerPoint.Application')
presentation = ppt.Presentations.Open('u:\\private\\jupyter\\win32com\\MyTeam_template.pptx')
```

(U) Did you notice that we didn't need to set `ppt.Visible` ? PowerPoint is always visible.

```
title = presentation.Slides(1)
```

(U) We know the first slide is the title slide, so we've set a variable to it. PowerPoint presentations are made up of slides, which in turn are collections of shapes. To modify a presentation, we need to know which shape is which. Let's take a look at title:

```
title
```

(U) Hmm. That's not very helpful. Let's see what methods we have:

```
dir(title)
```

(U) At this point you're probably realizing that COM objects don't act like normal Python objects.

```
help(title)
```

(U) So Python just takes anything you try to do with `title` and passes it on to the Windows COM library. Which means you'll need to consult Microsoft's Win32Com documentation if you have questions about something.

(U) Let's get back to working with this presentation. We still need to find out which shape is which:

```
for i, shape in enumerate(title.Shapes):
    shape.TextFrame.TextRange.Text = 'Shape #{0}'.format(i+1)
```

(U) This sets the text for each shape to its index number so we now have a number associated with each shape. You only need to do this when you're writing your script. Once you create your template, the shape numbers won't change. So the title is #1 and the subtitle #2.

(U) `Undo` a few times will remove the numbers.

(U) Let's update the title slide with today's date:

```
from datetime import date
today = date.today().strftime('%Y%m%d')
title.Shapes(2).TextFrame.TextRange.Text = today
```

(U) Now let's update the status of our two focus areas. We'll skill the step of identifying the shapes we want to modify.

```
focus1 = presentation.Slides(2)
focus1.Shapes(2).TextFrame.TextRange.Text = 'All Good, Boss'

focus2 = presentation.Slides(3)
focus2.Shapes(2).TextFrame.TextRange.Text = 'Sir, We have a problem'
```

(U) Now save the presentation with today's date, and Bob's your uncle.

```
presentation.SaveAs('u:\\private\\jupyter\\win32com\\MyTeam_{0}.pptx'.format(today))
presentation.Close()
ppt.Quit()
```

(U) Visio

P.L. 86-36

(U) Starting the application should look familiar:

```
visio = win32com.client.Dispatch("Visio.Application")
documents = visio.Documents
document = documents.Add("Basic Network Diagram.vst")    # Start with a built-in template
document.Title = "New Network Graph"                     # Add a title
pages = visio.ActiveDocument.Pages
page = pages.Item(1)
```

(U) Visio is visible by default, but can be hidden if desired.

(U) So we've created a document and grabbed the page associated with it. Visio shapes are part of stencil packages, so let's add a couple.

```
NetworkStencil = visio.Documents.AddEx("periph_m.vss", 0, 16+64, 0)
ComputerStencil = visio.Documents.AddEx("Computers and Monitors.vss", 0, 16+64, 0)
```

(U) Other stencils are:

- Network Locations: `netloc_m.vss`
- Network Symbols: `netsym_m.vss`
- Detailed Network shapes: `dtlnet_m.vss`
- Legends: `lgnd_m.vss`

(U) Other stencil names can be found on the Internet.

(U) Now we need the shape masters that we'll use.

```
pc = ComputerStencil.Masters.Item("PC")
router = NetworkStencil.Masters.Item("Router")
server = NetworkStencil.Masters.Item("Server")
connector = NetworkStencil.Masters.item("Dynamic Connector")
```

(U) The names match the names you see when you view the shapes in the stencil sidebar. Let's add a few shapes.

```
pc1 = page.Drop(pc, 2, 2)
pc1.Text = "10.1.1.1"
pc2 = page.Drop(pc, 10, 10)
pc2.Text = "10.1.1.2"
server1 = page.Drop(server, 15, 5)
server1.Text = "10.1.1.100"
router1 = page.Drop(router, 8, 8)
router1.Text = "10.1.1.250"
```

(U) Some of the shapes went off the page, so resize. You can wait until the end to do this, but it's more fun to watch the connections being drawn.

```
page.ResizeToFitContents()
page.CenterDrawing()
```

(U) Now draw the connectors.

```
arrow = page.Drop(connector, 0, 0)
arrowBegin = arrow.CellsU("BeginX").GlueTo(pc1.CellsU("PinX"))
arrowEnd = arrow.CellsU("EndX").GlueTo(router1.Cellsu("PinX"))
arrow.Text = "pc1 connection"
```

(U) We can customize a connector

```
arrow = page.Drop(connector, 0, 0)
arrowBegin = arrow.CellsU("BeginX").GlueTo(pc2.CellsU("PinX"))
arrowEnd = arrow.CellsU("EndX").GlueTo(router1.Cellsu("PinX"))
arrow.CellsU("LineColor").Formula = "=RGB(255, 153, 3)"
arrow.CellsU("EndArrow").Formula = "=5"
arrow.CellsU("EndArrowSize").Formula = "=4"
arrow.CellsU("LineWeight").FormulaU = "=5.0 pt"
arrow.Text = "pc2 connection"

arrow = page.Drop(connector, 0, 0)
arrowBegin = arrow.CellsU("BeginX").GlueTo(server1.CellsU("PinX"))
arrowEnd = arrow.CellsU("EndX").GlueTo(router1.Cellsu("PinX"))
arrow.Text = "server1 connection"
```

(U) Now resize, recenter, and save.

```
page.ResizeToFitContents()
page.CenterDrawing()

document.SaveAs('U:\\private\\jupyter\\win32com\\visio_demo.vsdx')
```

(U) Close the application.

```
visio.Quit()
```

(U) Excel

(U) win32com works with Excel too, but due to the slowness of the interface, you're probably better off using pandas.

Module: Threading and Subprocesses

0

(b)(3)-P.L. 86-36

Updated over 2 years ago by [redacted] in COMP 3321

3 3 206 34

fcs6 python

(U) Module: Threading and Subprocesses

Recommendations

(U) Module: Threading and Subprocesses

(U) Concurrence and Python's GIL - i.e. Python doesn't offer true concurrence

(U) Python's Global Interpreter Lock (GIL) means that you can really only have one true thread at one time. However, Threading in Python can be immensely helpful in speeding up processing when your script can perform subsequent steps tha do not depend on the ouptut of other steps. Basically, it gives the illusion of being able to do two (or more) things at the same time.

(U) Threading

(U) Threading allows you to spawn off "mini programs" called threads that work independently of the main program (sort of). Threading allows you to send data off to a function and let it work on getting results while you go on with your business. It

can also allow you to set up functions that will process items as you add them to a work queue. This could be especially helpful if you have parts of your program that take a long time to execute but are independent of other parts of your program. A good example is using a thread to execute a slow RESTful web service query.

(U) This adds some complexity to your life. Threads act asynchronously - meaning that you have limited control as to when they execute and finish. This can cause problems if you are depending on return values from threads in subsequent code. You have to think about if and how you need to wait on thread output which adds extra things to worry about in terms of accessing data. Python provides a thread-safe container named Queue. Queues will allow your threads access without becoming unstable, unlike other containers (such as dictionaries and lists) which may become corrupted or have unstable behavior if you access them via multiple threads.

(U) Subprocess

(U) The subprocess module is useful for spinning off programs on the local system and letting them run independently.

```
import ipydeps

modules = ['threading','queue']

for m in modules:
  installed_packages = [package.project_name for package in ipydeps._pip.get_installed_distributions()]
  if (m not in installed_packages) and (m not in ipydeps.sys.modules):
    ipydeps.pip(m)

import time
from threading import Thread, Timer, Lock
from queue import Queue
import random
```

Doc ID: 6689695

```
result_q = Queue()
work_q = Queue()
work_list = []
```

```
# The worker thread pulls an item from the queue and processes it
def worker():
    while True:
      item = work_q.get()
      do_work(item)
      work_q.task_done() #pause while until current work_q task has completed

def do_work(item):
    ## submit query process results and add result to Queue
    result_q.put( wait_random(item) )

def wait_random(t):
  time.sleep(t[1])
  print('finished task {}'.format(t[0]))

def hello():
  print("hello, world")
```

```
#Loading up our work_q and work_list with the same random ints between 1 and 10
time_total = 0
for i in range(10):
  x = random.randint(1,10)
  time_total += x
  work_q.put((i,x))
  work_list.append((i,x))
```

```
work_q.qsize()
```

```
len(work_list)
```

```
%%time
print('This should take {} seconds'.format(time_total))
for w in work_list:
  wait_random(w)
```

```
%%time
for i in range(5):
  t = Thread(target=worker)
  t.daemon = True  # thread dies when main thread exits. If we don't do this, then the threads will continue to
                   # "listen" to the work_q and take items out of the work_q and automatically process as you
                   # stick more items into the work_q
  t.start()        # you have to start a thread before it begins to execute
work_q.join()      # block until all tasks are done
```

(U) You can also use the Timer class to specify that a thread should only kick off after a set amount of time. This could be critical if you need to give some other treads a head start of for various other reasons. Remember, when we are doing threading you have to keep timing in mind!

```
%%time
# stupid little example
ti = Timer(5.0, hello)
ti.daemon = True
ti.start()  # after 5 seconds, "hello, world" will be printed
```

(U) You can mix these. The output below will most likely look like a bucket of crazy because threads execute (sort of) independently.

```
#loading up our work_q and work_list with the same random ints between 1 and 10
for i in range(10):
  x = random.randint(1,10)
  work_q.put((i,x))
```

```
%%time
for i in range(5):
  t = Thread(target=worker, )
  t.daemon = True
  t.start()

ti = Timer(5.0,hello)
ti.daemon = True
ti.start()        # ti *will probably* print 'hello, world' before all the other threads finish,
                  # or it might not it depends on the work_q contents
work_q.join()     # block until all tasks are done
```

(U) Subprocesses

(U) For most subprocess creation you will usally want to use the subprocess.run() convenience method. Please note, if you wish to access the STDOUT or STDERR output you must specify a value for the stdout and stderr arguments. Using the subprocess.PIPE constant puts the results from STDOUT and STDERR into the CompletedProcess object's attributes.

```
import subprocess
```

```
completed = subprocess.run(['ls','-l'],stdout=subprocess.PIPE,universal_newlines=True)
```

```
print("ARGS:", completed.args)
print("STDOUT:",completed.stdout)
print("STDERR: ",completed.stderr)
print("return code:",completed.returncode)
```

```
completed = subprocess.run(['ls', 'nosuchfile'],stdout=subprocess.PIPE,stderr=subprocess.PIPE,universal_newlines=True)
```

```
print("ARGS:", completed.args)
print("STDOUT:",completed.stdout)
print("STDERR: ",completed.stderr)
print("return code:",completed.returncode)
```

```
dir(completed)
```

```
type(completed)
```

(U) For finer-graned control you can use subprocess.Popen() . This allows greater flexability, and allows you to do things like kill spawned subprocesses, but be careful - you may get some unexpected behavior.

```
completed = subprocess.Popen(['ls','-l'],stdout=subprocess.PIPE,stderr=subprocess.PIPE,universal_newlines=True)
```

```
print("ARGS:", completed.args)
print("STDOUT:",completed.stdout)
print("STDERR: ",completed.stderr)
print("return code:",completed.returncode)
```

```
dir(completed)
```

```
type(completed)
```

(U) If you are just looking for a quick way to dump the output from the subprocess into a variable, you can use subprocess.check_output(). This is an older way of doing a specific type of .run() so you will see it used in Python 2. It takes many of the same parameters as .run() but has a few extra that correspond more closely to Popen()

```
completed = subprocess.check_output(['ls','-l'],universal_newlines=True)
```

```
dir(completed)
```

```
print(completed)
```

```
type(completed)
```

Distributing a Python Package at NSA

0

(b)(3)-P.L. 86-36

Updated 2 months ago by [redacted] in COMP 3321

3 9 255 30

python packages distribution git gitlab

(U//~~FOUO~~) Directions for how to make and distribute a Python package using the nsa-pip server.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U//~~FOUO~~) Module: Distributing a Python Package at NSA

(U//~~FOUO~~) At NSA, internally-developed Python packages can be installed by configuring `pip` to point to the `nsa-pip` server (`https://pip.proj.nsa.ic.gov/`). But how do you push your own package out to `nsa-pip` ? The basic steps are as follows:

1. (U) Make a python package.
2. (U) Make the package into a distribution.
3. (U) Push the project to GitLab.
4. (U//~~FOUO~~) Add a webhook for nsa-pip to the GitLab project.
5. (U//~~FOUO~~) Push the package to nsa-pip.

1. (U) Make a python package

(U) Recall from an earlier lesson that a python package is just a directory structure containing one or more modules, a special `__init__.py` file, and maybe some nested subpackages. The name of the package is just the name of the directory.

```
awesome/
|-- __init__.py
`-- awesome.py
```

(U) Example module

(U) Here are the contents of the `awesome` module (awesome.py) from before:

```
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

def cool(group):
    return "Everything is cool when you're part of {0}".format(group)

if __name__ == '__main__':
    a = Awesome("Everything")
    print(a)
```

(U) `__init__.py`

(U) Once again, the `__init__.py` file needs to be present but can be empty. It is intended to hold any code needed to initialize the package. If the package has many subpackages, it may define `__all__` to specify what gets imported when someone uses the `from <package name> import *` syntax. Other things `__init__.py` commonly includes are `import` statements for subpackages, comments/documentation, and additional code that glues together the subpackages.

(U) Since our module is small and we have nothing to initialize, we'll leave `__init__.py` empty. If we wanted, we could actually put all the awesome.py code in `__init__.py` itself, but that can be confusing for other developers. We could also define:

```
__all__ = ['Awesome', 'cool']
```

(U) But since those would get imported anyway, we don't need to do that. You only need to define `__all__` if the package is complex enough that you want to import some things and not others or ensure that subpackages get imported.

(U//~~FOUO~~) For another example, see the __init__.py file for the `parserror` module on GitLab.

2. (U) Make the package into a distribution

(U) At this point, you could tar or zip up that package, give it to someone else, and they could extract it and use it. But what directory should they put the package in? What if your package depends on other modules to work? What if you change the package and want to keep track of the version? These considerations suggest that there is some *package management* that needs to go on. Python does have the `pip` package manager to handle a lot of that. So how do we distribute the package so you can just `pip install awesome` ?

(U) First we need to add another layer of stuff around our package to help set it up for installation.

```
awesome/
|-- README.md
|-- awesome/
|   |-- __init__.py
|   `-- awesome.py
|-- setup.cfg
`-- setup.py
```

(U) You can see that our original package directory has been moved down to a subdirectory. Above it we have a new *project* directory by the same name, and alongside it there are a few more files. Let's look at each.

(U) setup.py

(U) setup.py is the most important file. It mainly contains a `setup()` function call to configure the distribution. It is also used to run the various packaging tasks via `python setup.py` . The `setup()` function comes from the `setuptools` package, which is not part of the Python standard library. You may need to `pip install setuptools` first to use it. (`setuptools` improves the legacy `distutils` package that is part of the standard library and is the officially recommended distribution tool these days.)

(U~~//FOUO~~) Our setup.py looks like this:

```
from setuptools import setup

setup(
    version='1.0.0',
    name='awesome',
    description='(U) An awesome module for awesome things.',
    long_description=open('README.md', 'r').read(),
    url='https://gitlab.coi.nsa.ic.gov/python/awesome.git',
    author='COMP3321',
    author_email='comp3321@nsa.ic.gov,
    scripts=[],
    packages=['awesome',],
    package_data={},
    install_requires=[]
)
```

(U) This just scratches the surface of the arguments you can give to `setup()`. You can find more details on the outside from the Python Packaging Authority (`https://packaging.python.org/tutorials/distributing-packages/`). For example, you can specify which versions of Python are compatible using the `classifiers` and `python_requires` options, specify dependencies with `install_requires` and other `*_requires` options, and include non-code data files.

(U//FOUO) For another example, see `parserror` 's setup.py.

(U) setup.cfg

(U) `setup.cfg` is an INI file that configures some defaults for the setup.py options. Ours is fairly simple:

```
[metadata]
description-file = README.md
```

(U) If you are using `wheel`, you may also want to set this if your code works for both Python 2 and 3:

```
[bdist_wheel]
univeral=1
```

(U) README.md

(U) README.md is just a Markdown file that gives an overview of what the package is for and describes how to install and use it. (You may also see README.rst files instead. The .rst stands for restructured text, which is a popular Python-specific way to format text.)

3. (U) Push the project to GitLab

(U//~~FOUO~~) Of course, with a project like this, you should be using version control. And since the `nsa-pip` server ties into GitLab, you'll need to use Git. Learning Git could be a course in itself, so we'll just cover the basics here.

(U) For those who are unfamiliar, Git is a distributed version control manager. It's *version control* because it allows to to track and revert changes in your text files easily in a Git *repository*. You can view the history of your changes as a tree, and even branch off from it and merge back in easily if you need to. It's *distributed* because everybody who has a copy of the git repository has a full copy of the history with all the changes. And it's a *manager* (like `pip` is a manager) because it manages all the tracking itself and gives you a bunch of commands to let you add and revert your changes.

a. (U) Install and configure Git

(U//~~FOUO~~) With any luck, your system will already have Git installed. If you're working on LABBENCH, it should already be there. If you're on another Linux system, you can `yum install git` (for CentOS/RHEL) or `apt-get install git` (for Ubuntu). If you're on Windows, you will probably want to use Git Bash or TortoiseGit or something similar. [redacted] There are additional Windows directions here.

(U//~~FOUO~~) After Git is installed, you will need to configure it and SSH to connect to GitLab. On LABBENCH, the easiest way to do that is to run this (Ruby) notebook to set up Git and SSH in jupyter-docker. It takes care of:

- making a .gitconfig,
- making an overall .gitignore, and
- making an encrypted RSA key pair based on your PKI cert for use with SSH.

(U//~~FOUO~~) If you are on another system, you will need to take care of those things yourself. For tips, see the one-time setup instructions for Git on WikiInfo. [redacted]

b. (U) Add version control to your project

(U) Now to work! Enter your local project directory (i.e. the top level where setup.py lives) and run `git init`. That will turn your project into a a Git repository and get Git ready to track your files. (Note that it doesn't actually start tracking them yet--you have to explicitly tell it what to track first.)

(U) Next, since this is a Python package, you probably want to a `.gitignore` file to your project directory containing:

```
build/
*.pyc
```

(U) That will tell Git to ignore temporary files made when you run `setuptools` commands.

(U) Now our project structure should look something like this:

```
awesome/
|-- .git/
|   `-- (...git stuff...)
|-- .gitignore
|-- README.md
|-- awesome/
|   |-- __init__.py
|   `-- awesome.py
|-- setup.cfg
`-- setup.py
```

(U) Next, you can start tracking all these files for changes by running:

```
git add *
git commit -m "Initial commit."
```

c. (U) Make a corresponding GitLab project

(U) Congratulations! Your package is now under version control. However, you have the only copy of it, so if your local copy goes away, your code is gone. To preserve it and enable others to work on it, you need to push it to GitLab.

(U//FOUO) The first step is to make a new GitLab project:

1. (U) Visit the new project page on GitLab
2. (U) Enter your package name as the Project path (e.g. `awesome`).
3. (U//~~FOUO~~) Enter the overall classification level of the code and files in your project.
4. (U) Leave the Global access level set to Reporter. That will allow others to both clone (copy) your code and file issues if there are problems with your package. See the GitLab permissions chart for a full description of the access levels.
5. (U//~~FOUO~~) Choose the Namespace. By default it will make a personal project under your name and SID. If you belong to a GitLab group (and have the right permissions), you can also add the project to that group. Consider joining the Python group and adding your project there.
6. (U) Add a short description of your package, if you want.
7. (U//~~FOUO~~) Select a visibility level. Since NSA GitLab uses PKI, there is no difference between "internal" and "public."
8. (U) Hit "Create Project."

(U) Afterwards, you can copy the URL from the main page of your new project and put it in the `url` argument to `setup()` in `setup.py`.

d. (U) Push your code out to GitLab

(U) Your new project page should have some instructions on how to push code from an existing folder or Git repository. Near the top of the page you should see a box with "SSH" highlighted to the left and a `git@gitlab.coi.nsa.ic.gov...` address in the box to the right. Copy that address. Then,

1. (U) `cd` to the top level of your project
2. (U//~~FOUO~~) `git remote add origin git@gitlab.coi.nsa.ic.gov...` (using the `.git` address you just copied)
3. (U) `git push -u origin master`

(U) Afterwards, if you visit your GitLab project page, you should see a list of your project files and the rendered contents of your README.

4. (U//~~FOUO~~) Create a tag for the package release.

(U) Back in your local repository, tag your local project with `pip-` and the release version and push that tag to GitLab. For example, if your first version is 0.1.0, run:

```
git tag -a pip-0.1.0 -m "Releasing to the pip repo"
git push origin pip-0.1.0
```

(U) Ideally, version numbers should comply with Python's PEP 440 specification. In plain English, that means they should be of the form:

```
Major.Minor.Micro
^     ^     ^
|     |     \- changes every bugfix
|     |
|     \- changes every new feature
|
\- changes every time backwards-compatibility broken
```

5. (U//~~FOUO~~) Create distributions

(U) Source distribution (sdist) and Wheel distributions are both collections of files and metadata that can be installed on a machine and then used as a Python library. Wheels are considered a "built distribution" while sdist needs one extra build step, although that is transparent to a user if you are using pip to install. For most use cases, you want to build both an sdist and bdist_wheel, upload both, and let pip work out which one to use (almost always the Wheel).

(U) An example of building and publishing the distributions is as follows:

```
pip3 install setuptools wheel twine
python3 setup.py sdist bdist_wheel
```

a. (U) Upload package to NSA-Pypi with Twine--

(U) twine is a Python library developed by the same team that maintain Pypi. It uses requests.py to make secure (https) uploads of Python packages, and offers some command-line arguments to make it easy to specify what repository, client cert, ca bundle, and username/password to use. The username and password can be blank (or anything you want) for uploading to NSA-Pypi, because NSA-Pypi will use your PKI and Casport to determine authentication/authorization.

(U) Note that the NSA-Pypi server supports XPE with Labbench, so using twine there on Labbench does not require uploading your personal cert. In non-Labbench environments, the following command should work:

```
twine upload dist/* -u '' -p '' \
                    --repository-url https://nsa-pypi.tools.nsa.ic.gov \
                    --cert /path/to/ca_bundle.pem \
                    --client-cert /path/to/your_cert.pem \
                    --verbose
```

(U) Bonus!

(U) Python Packaging Authority

(U) For years there were a variety of ways to gather and distribute packages, depending on whether you were using Python 2, Python 3, `distutils`, `setuptools`, or some obscure fork that tried to improve on one or another of them. This eventually got so messy that some developers got together and formed the Python Packaging Authority (PyPA) to establish best practices and govern submissions to PyPI. PyPA's approach is now referenced in the official Python documentation. They can be found at `http://packaging.python.org/` and have many useful tutorials and guides, including how to use `setuptools` and push a package to PyPI itself.

(U) Testing

(U) Consider using common test modules like `doctest`, `unittest`, and `nose` to add tests to your code. You can try test-driven development (TDD) or even behavior-driven development (BDD) with `behave`.

(U) Documentation

(U) Make sure your code includes docstrings where appropriate, as well as good comments and a helpful README.

(U) `virtualenv`

(U) To solve the problem of coming up with an isolated, repeatable development environment (i.e. make sure you have the dependencies you need and that they don't conflict with other Python programs), most developers use `virtualenv` and `virtualenvwrapper`.

(U) `wheel`

(U) If your package is large or needs to compile extensions, you may want to distribute a pre-built "wheel." The `wheel` module adds some functionality to `setuptools` if you `pip install wheel`. On the receiving end, when you `pip install` packages, you may occasionally see that pip is actually installing a `.whl` file -- that's a wheel. It is relatively new and is the recommended replacement for the old `.egg` format.

Module: Machine Learning Introduction

0

(b)(3)-P.L. 86-36

Created 3 months ago by [redacted] in COMP 3321

3 1 32 10

sklearn pandas numpy matplotlib comp3321 python jupyter nlp nltk pressurewave

(U//~~FOUO~~) This notebook gives COMP3321 students an introduction to the world of machine learning, by demonstrating a real-world use of a supervised classification model.

Recommendations

A Note About Machine Learning

Machine learning is a large and diverse field--this notebook is not trying to cover all of it. The point here is to give a real-world example for how machine learning can be used at NSA, just to expose students to the kinds of things that machine learning can do for them. The example below is of a supervised classification model. Supervised means that we're feeding it labeled training data. In other words, we're giving it data and telling it what it's supposed to predict. It will then use those labels to figure out what predictions to make for new, unlabeled data that we give it. It's a classification model because the labels we want it to predict are categorical--in this case, is the Jupyter notebook in question a "mission", "building block", or "course materials" notebook?

Load Required Dependencies

```
import ipydeps

ipydeps.pip(['numpy', 'pandas', 'matplotlib', 'sklearn', 'sklearn-pandas', 'nltk', 'lbpwv', 'requests-pki',
             'bs4'])

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
import requests_pki
from bs4 import BeautifulSoup
import os, re, lbpwv
import requests_pki
ses = requests_pki.Session()
from bs4 import BeautifulSoup
%matplotlib inline
```

Read in Labeled Training Data

Here we'll read in a table of what we call labeled training data. In this case, the 'label' is the 'category' field, which includes one of three string values:

- Building block
- Mission
- Course materials

The label is the target variable, what we want the machine learning model to predict. The data the model will use to make these predictions is called the features. You can see below that we have a lot of features comprising a lot of different data types--these will require some preparation before they can be used in the model.