Lecture 7: Assorted Topics

Chirag Sakhuja

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

`import` keyword

- Similar to `#include` in C/C++
- Local Python files can be `import`ed
  - As long as they have a `.py` extension
- Python also has many built-in libraries, some of which we've seen
- Package managers, such as `pip` and `conda`, allow you to `import` more libraries that don't come installed by default

Report-style notebook

- Similar interface to Mathematica and Matlab
- Breaks down program into "cells" for organizational purposes
  - A cell's execution is persistent throughout the notebook
  - Cells do not necessarily execute in order, even though that is most often the logical way of using a notebook
- Consist of a mix of Markdown and Python
  - Markdown is a simple typesetting language
- Integrate well with plotting libraries and other graphics
- Useful for sharing results

Demo

See 'numpy (HTML)' on Canvas

Demo

See 'matplotlib (HTML)' on Canvas

Demo

See 'pandas (HTML)' on Canvas

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

Running shell commands in Python

```
import subprocess

command = 'ls -l'
res = subprocess.run(command.split(' '),
                     stdout=subprocess.PIPE)
print(res.stdout)
```

Not going to go into much detail since there are a lot of complicated details…

Building a batch rename utility

- The purpose is to take a directory of files and do a simple string replacement on all of their names
- We'll start off simple and then build something more robust
- The slides are structured based on how my thought process would go while writing the script

Assume we `import`

```
import os
import sys
```

Walking a directory structure recursively

```
for root, dirs, files in os.walk('.'):
    print(files)
    # => <all files, recursively>
```

Specify directory

`os.walk` iterates through all directories and subdirectories (recursively, depth-first) and for each directory it's in it will return a 3-tuple

Walking a directory structure recursively

```
for root, dirs, files in os.walk('.'):
    for file in files:
        print(os.path.join(root, file))
        # => <all files, recursively, relative path>
```

`os.path.join` creates OS-agnostic paths
Windows likes \, *nix systems like /

Building a batch rename utility

```
for root, dirs, files in os.walk('.'):
    for file in files:
        file_new = rename(file)
        file_rel = os.path.join(root, file)
        file_new_rel = os.path.join(root, file_new)
        os.rename(file_rel, file_new_rel)
```

We still need to write this

Building a batch rename utility

```
def rename(file):
    return file.replace('.c', '.cpp')
```

Building a batch rename utility

```
def rename(file, old, new):
    return file.replace(old, new)
```

Building a batch rename utility

```
# ...
file_new = rename(file, ???, ???)
# ...
```

Building a batch rename utility

```
# ...
file_new = rename(file, sys.argv[2], sys.argv[3])
# ...
```

Starting at index 2 because we also want to specify path as an argument eventually

Building a batch rename utility

```
for root, dirs, files in os.walk(sys.argv[1]):
    # ...
```

Building a batch rename utility

```
def rename_batch(path, old, new):
    for root, dirs, files in os.walk(path):
        for file in files:
            file_new = rename(file, old, new)
            file_rel = os.path.join(root, file)
            file_new_rel = os.path.join(root, file_new)
            os.rename(file_rel, file_new_rel)
```

Building a batch rename utility

```
def main():
    path = sys.argv[1]
    old = sys.argv[2]
    new = sys.argv[3]
    rename_batch(path, old, new)

if __name__ == '__main__':
    main()
```

Building a batch rename utility

```
if len(sys.argv) != 4:
    sys.exit(1)

path = sys.argv[1]
old = sys.argv[2]
new = sys.argv[3]
rename_batch(path, old, new)
```

Building a batch rename utility

```
>> python3 rename.py . '.c' '.cpp'
```

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

Assume we `import`

```
from optparse import OptionParser
```

Adding a verbose flag

```
parser = OptionParser()
parser.add_option('-v', '--verbose', dest='verbose',
                  action='store_true',
                  default=False,
                  help='print out filename details')
options, args = parser.parse_args()
if len(args) != 3:
    sys.exit(1)
rename_batch(args[0], args[1], args[2], options)
```

Adding a verbose flag

```
def rename_batch(path, old, new, knobs):
    for root, dirs, files in os.walk(path):
        for file in files:
            # ...
            if knobs.verbose:
                print(file_new_rel)
```

Adding a verbose flag

```
>> python3 rename.py . '.c' '.cpp'
>> python3 -v rename.py . '.c' '.cpp'
# => <list of new file names>
>> python3 rename.py --help
Usage: rename.py [options]

Options:
-h, --help      show this help message and exit
-v, --verbose   print out filename details
```

`optparse` also adds help message

Fixing the usage string in the help message

```
usage = '%prog [options] path old-name new-name'
parser = OptionParser(usage=usage)
# ...
```

Fixing the usage string in the help message

```
>> python3 rename.py --help
Usage: rename.py [options] path old-name new-name

Options:
-h, --help      show this help message and exit
-v, --verbose   print out filename details
```

Other common flags

```
parser.add_option('-n', '--dry-run', dest='dry_run',
                  action='store_true',
                  default=False,
                  help='do not actually rename')
parser.add_option('-d', '--depth', dest='depth',
                  action='store', type='int',
                  default=0,
                  help='subdir recursion depth')
```

Other common flags

```
>> python3 rename.py –-help
Usage: rename.py [options] path old-name new-name

Options:
-h, --help              show this help message and exit
-v, --verbose           print out filename details
-n, --dry-run           do not actually rename
-d DEPTH, --depth=DEPTH
                        subdir recursion depth
```

LBE: Building a batch rename utility

I've attached the full file on Canvas as `rename.py`

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

Unit testing

- The concept of treating each individual unit (often a function) of your code and writing tests based solely on its input and output
- Encourages writing code in a modular fashion without global state or side effects
- Very important for interpreted languages because even little typos won't be exposed until the code is actually run
  - What if there are multiple code paths in a function, and there's a typo in a less frequently used one?
  - Write better tests!

Test driven development

- A programming methodology in which you write a comprehensive suite of tests to describe the expected behavior of your code units
- Write the actual code after thinking of the tests
- May not be the best approach
  - What happens when your code passes all the tests, but you forgot some critical tests?
  - Encourages a "debug into existence" mentality
- It does end up being fast, as the required types of inputs can be specified in unit tests and then the minimal code to make the test case pass is considered acceptable

Continuous integration (regression testing)

- Automatically run unit tests on some trigger event
- Usually used in combination with some version control system
  - Every commit triggers a regression test
- Popular continuous integration frameworks include Jenkins and TravisCI (free for public Github repositories)

The goal: this fancy badge on Github

pytest

- pytest is a testing framework for Python
- Assertion based testing
  - Set up a test case and then assert a condition that should be true
- Automatically picks up unit tests by looking for functions that begin with `test_` in files with either a `test_` prefix or a `_test` suffix
  - For example it will pick up a function called `test_withdraw` in a file called `test_atm.py` as a unit test

Installs a command line tool and a Python library

```
conda install pytest
```

Assume we `import`

```
import pytest
```

Simple unit tests

```python
def increment(x):
    return x + 1

def test_increment_positive():
    assert increment(1) == 2
def test_increment_negative():
    assert increment(-1) == 0
```

Running pytest

Running in the same directory as the file from the previous slide

```
>> pytest -q
.......                                  [100%]
2 passed in 0.02 seconds
```

A failing test

```
def increment(x):
    return x + 1

def test_increment_positive():
    assert increment(1) == 2
def test_increment_negative():
    assert increment(-1) == 1
```

A failing test

```
......F                                                  [100%]
====================== FAILURES ======================
_____________ test_increment_negative _____________
def test_increment_negative():
>       assert increment(-1) == 1
E       assert 0 == 1
E        +  where 0 = increment(-1)
test_sample.py:7: AssertionError
1 failed, 1 passed in 0.05 seconds
```

LBE: ATM unit tests

```
class ATM:
    def __init__(self, init_value=0):
        self.value = init_value
    def deposit(self, amount):
        self.value += amount
    def withdraw(self, amount):
        if amount > self.value:
            raise NegativeBalanceException()
        self.value -= amount
```

LBE: ATM unit tests

```
class NegativeBalanceException(Exception):
    def __str__(self):
        return 'Attempting to withdraw more cash \
                than present'
```

LBE: ATM unit tests

```python
def test_empty_construct():
    obj = ATM()
    assert obj.value == 0

def test_init_construct():
    obj = ATM(100)
    assert obj.value == 100
```

LBE: ATM unit tests

```
def test_deposit():
    obj = ATM()
    obj.deposit(100) assert obj.value == 100
```

LBE: ATM unit tests

```python
def test_withdraw():
    obj = ATM(100)
    obj.withdraw(20)
    assert obj.value == 80

def test_withdraw_too_much():
    obj = ATM()
    with pytest.raises(NegativeBalanceException):
        obj.withdraw(20)
```

LBE: ATM unit tests

```
>> pytest -q
.......                                                  [100%]
5 passed in 0.01 seconds
```

Test fixtures to reduce code duplication

```
@pytest.fixture
def default_atm():
    return ATM()

@pytest.fixture
def init_atm():
    return ATM(100)
```

Test fixtures to reduce code duplication

```
def test_empty_construct(default_atm):
    assert default_atm.value == 0


def test_withdraw(init_atm):
    init_atm.withdraw(20)
    assert init_atm.value == 80
```

Include the test fixture

No need to create ATM

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

Python bindings into C/C++

- Python can call into C/C++ libraries
  - Build your C++ project with Boost.Python
  - Use a tool, such as SWIG, to create an interface that Python can use
  - Write hybrid code and use Cython
- Lets you write C++ where you need speed and Python where you need convenience
- Good way of prototyping in Python and then slowly integrating production code (assuming product is in C++)
- Python can bind into other languages, though C++ is most common

Python GUIs

- Easy to make graphical applications with Python
- Common libraries include:
  - PyQt (built on top of the popular Qt toolkit)
  - PyGTK (build on top of the popular GTK library)
  - PyGUI

Make standalone Python applications

- PyInstaller lets you create an installer that works on all three major platforms and doesn't require the user to have Python or any Python packages installed
  - Most useful on Windows, where Python is not installed by default or easy to use
- py2exe is another alternative that "compiles" Python code into a native executable for Windows

Python for web development

- Flask and Django are used to build websites/webapps
  - Django includes entire system, such as built-in admin panels, etc.
  - Django is easier when building common sites such as blogs
  - Both come with built-in web servers, although they pale in comparison to something like nginx
- `requests` is the standard Python library to interact with RESTful APIs (i.e. pretty much all APIs hosted on servers)

Python for mobile app development

- Not very common, but it is possible
- Kivy helps make a cross-platform GUI, but doesn't look native
- BeeWare is also cross-platform and looks native, but is less mature

Overview

Recap

Automation/scripting

optparse

pytest

Python everywhere

Conclusion

Final announcements

- All assignments have been posted
  - Due by March 18 @ 11:59 PM for no penalty; by May 4 @ 11:59 PM for 15 point penalty
- Please fill out eCIS near the end of the semester!
- I'll still read Piazza and respond to emails at chirag.sakhuja@utexas.edu

Thank you, and I hope you enjoyed the class!