Input, Output, and Options

Jamie Saxon

Introduction to Programming for Public Policy

October 10, 2016

Introduction

- Our python scripts have been ‘self-contained.’
- It is time to open them to the world:
  - **Input/Output**: Play with data!
  - **Options**: Modify them on the fly!

'Canonical' Reads

Great news: you already know how!

```
#!/usr/bin/env python

for line in open("salaries.csv", "r"):
  print(line.strip())
```

- The syntax is a `for` loop – nothin' to it!
- Just `open()` the file with a '`r`' flag, for 'read.'
- You get one line at a time, and can do whatever you want with them.
- Use `strip()` to remove any additional whitespace.

'Canonical' Writes

Good news: fairly similar...

```
#!/usr/bin/env python

yum = ["pineapple", "watermelon", "blueberry",
       "apricot", "chirimoya", "grapefruit",]

output = open("output.txt", "w")
for y in yum: output.write(y + "\n")
output.close()
```

- The difference is that we're iterating over something else.
- The output file is just an object, that we write to.
- There is also `a`, for 'append' (write at end of file).

Using and Filtering Inputs

- Let's start by reproducing our 'high salaries grep' from day 1.

```
#!/usr/bin/env python

for line in open("salaries.csv", "r"):

  if "$" not in line: continue
  line = line.replace("$", "").strip()

  # split the line into a list
  spline = line.split(",")

  # pull off the salary, as a float
  if float(spline[-1]) > 200000: print(line)
```

Python: Beyond Single Lines

- Using bash, we were limited in our 'global' view.
- Though we could `sort`, we mainly looked at one line at a time.

- Python lets us store variables and manipulate the entire dataset.*
  - In future weeks we'll learn more and more tools for doing this.

*Truth be told, bash allows this too; it's just less fun.

Aggregating at a Basic Level

- What was the expenditure on salaries in the fire department?

```python
total = 0
for l in open("salaries.csv"):

  if "FIRE" not in l: continue

  sl = l.strip().split(",")
  total += float(sl[4][1:])

print("Total fire salaries:")
print("  ${:.2f}".format(total))
```

- Open `ex/a/dept_salaries.py`, and modify it to print the total, average, and max salaries, and the number of employees.
  - Use `len()`, `sum()`, and `max()`.

‘Canonical’ Writes: `with`

ex/a/canonical_write_with.py

- I sometimes use ‘`with`’ to specify a block in which to write.
- The file ‘snaps shut’ at the end of the block.

```
#!/usr/bin/env python

yum = ["pineapple", "watermelon", "blueberry",
       "apricot", "chirimoya", "grapefruit"]

with open("output.txt", "w") as output:
  for y in yum: output.write(y + "\n")
```

Formats

CSV: Comma Separated Values

- Common, simple, flat, but non-standardized format.
    - Text in columns separated by a delimiter (‘escape’ by quotes).
    - Can be read directly by e.g., Excel.

```
Name,Position Title,Department,Employee Annual Salary
"AARON,  ELVIA J",WATER RATE TAKER,WATER MGMNT,$90744.00
"AARON,  JEFFERY M",POLICE OFFICER,POLICE,$84450.00
"AARON,  KARINA",POLICE OFFICER,POLICE,$84450.00
"AARON,  KIMBERLEI R",CHIEF CONTRACT EXPEDITER,GENERAL SERVICES,$89880.00
"ABAD JR,  VICENTE M",CIVIL ENGINEER IV,WATER MGMNT,$106836.00
"ABARCA,  ANABEL",ASST TO THE ALDERMAN,CITY COUNCIL,$70764.00
"ABARCA,  EMMANUEL",GENERAL LABORER - DSS,STREETS & SAN,$41849.60
"ABASCAL,  REECE E",TRAFFIC CONTROL AIDE-HOURLY,OEMC,$20051.20
"ABBASI,  CHRISTOPHER",STAFF ASST TO THE ALDERMAN,CITY COUNCIL,$49452.00
"ABBATACOLA,  ROBERT J",ELECTRICAL MECHANIC,AVIATION,$93600.00
"ABBATEMARCO,  JAMES J",FIRE ENGINEER-EMT,FIRE,$100320.00
```

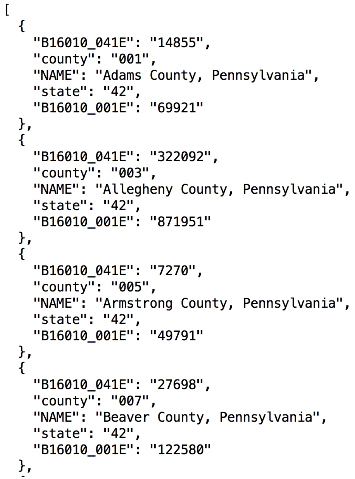
CSV: The Module

- I tend to just use a for loop, but there is a csv module.

```
import csv
with open('salaries.csv') as f:
  reader = csv.reader(f)
  next(reader) # skip the header
  for row in reader:
    print(float(row[3][1:])) # salaries
```

JSON, or, dictionaries and lists revisited.

- Officially stands for JavaScript Object Notation, but now used in many languages.
- Common format for transmitting formatted data on the internet.
- Readily manipulable in Python: just dictionaries and lists.
  - Can be ‘nested’ dictionaries – much like classes.
  - Often, data is packaged with metadata, and you have to ‘navigate down’ to a list of <u>actually</u> useful data.



```
[
  {
    "B16010_041E": "14855",
    "county": "001",
    "NAME": "Adams County, Pennsylvania",
    "state": "42",
    "B16010_001E": "69921"
  },
  {
    "B16010_041E": "322092",
    "county": "003",
    "NAME": "Allegheny County, Pennsylvania",
    "state": "42",
    "B16010_001E": "871951"
  },
  {
    "B16010_041E": "7270",
    "county": "005",
    "NAME": "Armstrong County, Pennsylvania",
    "state": "42",
    "B16010_001E": "49791"
  },
  {
    "B16010_041E": "27698",
    "county": "007",
    "NAME": "Beaver County, Pennsylvania",
    "state": "42",
    "B16010_001E": "122580"
  },
```

Sample JSON Objects: Dictionaries in Lists

JSON: Exploring and Accessing Data

- Let's explore some JSON data. Please open a python prompt in ex/a/, and type this:

```
import json

with open("narcotics.json") as data:
  narcotics = json.load(data)
```

- Use `narcotics.keys()` to find the data (a list).
- What are the most common drug offenses?
  - Use a set comprehension (curly branches) to get the types.
  - Use a for loop to ask how many of each type there are.

JSON: Loading and Processing

```
import requests, json

# we'll cover this in a few weeks.
j = requests.get("...").json

# writing to a file
with open("narcotics.json", "w") as out:
  out.write(json.dumps(j, indent=2))

# reading a file
with open("narcotics.json") as data:
  narcotics = json.load(data)
```

Pickle: Storing Arbitrary Objects

- If you have a time consuming step in your code, you can ‘pickle’ a python object and later pick up where you left off.
- Where is JSON human-readable and portable, pickle is not – it’s a python-specific, binary file.
- But pickle files are typically less than half of a JSON object, and can be read much faster (in its most-recent release).

```python
import pickle

with open('data.pickle', 'wb') as f:
  pickle.dump(narcotics, f) # writing

with open('data.pickle', 'rb') as f:
  data = pickle.load(f) # reading
```

- Many of the operations we've done, could be done in Excel.
- But in the next few weeks, we'll see a lot that can't...
- And a lot (most?) data comes in a format that is not immediately usable, and needs to be tinkered with (munged).

Arguments

What Are Arguments

We have already used arguments to:

- perform functions on different inputs.
- modify the behavior of programs (e.g., grep -i or sort -r).

Now let's modify the behavior of our scripts!

The idea is that we want the same code to be able to do many things, without rewriting it.

The argparse module

Unapproachable **documentation** but a good **tutorial**.

- Try running the simplest example, with '-h':

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("number", help="A number!")
args = parser.parse_args()

print(args)
```

The argparse module

Unapproachable **documentation** but a good **tutorial**.

- Try running the simplest example, with '-h':

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("number", help="A number!")
args = parser.parse_args()

print(args)
```

Add a requred argument:

- `parser.add_argument("number", help="A number!")`

The argparse module

Unapproachable **documentation** but a good **tutorial**.

- Try running the simplest example, with '-h':

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("number", help="A number!")
args = parser.parse_args()

print(args)
```

Add a requred argument:

- `parser.add_argument("number", help="A number!")`

Add an optional argument (dashes, or 'required'):

- `parser.add_argument("--extras", default = "yay!")`

Controling the Types

Please follow along with these.

Controling the Types

Please follow along with these.

By default, all arguments are strings; ‘cast’ them on the fly with ‘type.’

- `parser.add_argument("number", type = float, help="A number!")`

`type` takes a function; lambda functions, str.lower, etc. work too.

Controling the Types

Please follow along with these.

By default, all arguments are strings; ‘cast’ them on the fly with ‘type.’

- `parser.add_argument("number", type = float, help="A number!")`

`type` takes a function; lambda functions, str.lower, etc. work too.

Specifying default values is easy:

- `parser.add_argument("number", type = float, default = 3)`

Controling the Types

Please follow along with these.

By default, all arguments are strings; ‘cast’ them on the fly with ‘type.’

- `parser.add_argument("number", type = float, help="A number!")`

`type` takes a function; lambda functions, str.lower, etc. work too.

Specifying default values is easy:

- `parser.add_argument("number", type = float, default = 3)`

Or an `action="store_true"` (default is the opposite):

- `parser.add_argument("--store", action="store_true")`

Using the Arguments

- After running `args = parser.parse_args()`, the arguments become accessible as variables, via the long-form versions of the argument names, `args.var_name`.
- We can then pass these variables into a function or class, to run our script with varying behavior.

Let's experiment with `options_example.py`.