You, Me and Python Env

Getting a handle on the advanced features of Python

Announcements

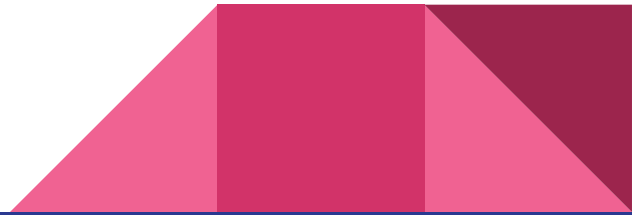
- Projects require meeting with Us, the teaching staff
  - I (Arav) will meet with those who want to do the Website Project and NewLangWhoDis
  - Sowgandhi will meet with those who want to do the Data Science Project
  - Can arrange meetings with me (Arav) out of our OH times
    - Need at least a few days notice

Overview

- Python Virtual Environments
- Advanced Python Features/Tips
  - List Comprehensions
  - Tuple Unpacking
  - Dictionaries, of other kinds

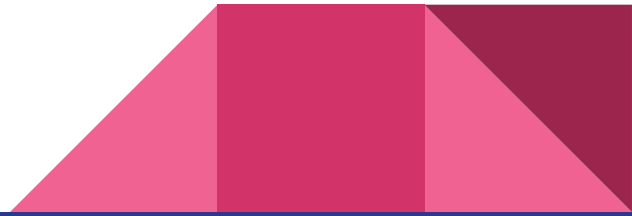
Python Virtual Environments

- Using Python as your default language can be difficult
- Installing outside packages can require certain versions of other packages
- Getting two python installations to use the same software can be hard!
- Enter - Python venv



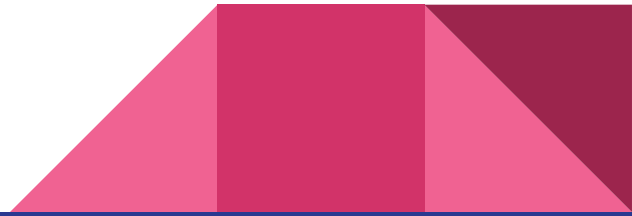
Python venv

- Python’s native tool to install virtual environments
- Allows you to have a “separate” Python installation
  - Each has different packages
  - Each is separate from each other
  - Allows for easy sharing through “requirements.txt” documents



Creating your own virtual environment

- `python -m venv env`
  - Creates a virtual environment without any packages at `./env`
  - Contains a copy of the Python interpreter and standard library
  - Contains no external packages, like `matplotlib` and `pandas`
- `source env/bin/activate`
  - Activates the current Python venv
  - Switches the `python` command to the Python interpreter in the virtual environment
- `deactivate`
  - Deactivates the current Python venv
  - Switches it back to the system installation

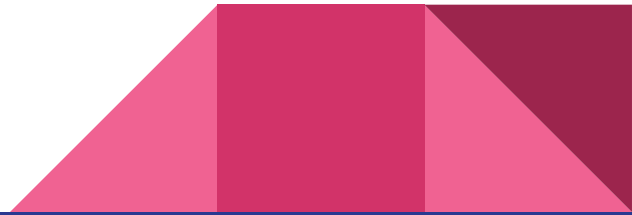


Common Pitfalls with Virtual Environments

- Need to upgrade pip, setuptools, and wheel
  - `pip install --upgrade pip setuptools wheel`
- Don’t commit `./env` to your Git repository
  - Commits the Python interpreter and associated binaries to the repo
  - Binaries not universal to all OS
- Using Anaconda’s python version to create a virtual environment
  - Sets the `PYTHONPATH` environment variable
  - Messes with standard installation of virtual environments
  - Instead, use `/usr/local/bin/python3 -m venv env`

Sharing Python Virtual Environments

- Instead of sharing the `/env` folder, share `requirements.txt`
  - List of all of our Python packages we installed in our local environment
  - Much, much, much smaller than the full `/env` folder
  - Generate it via `pip freeze > requirements.txt` with your current environment activated
- Installing a python virtual environment using `requirements.txt`
  - `pip install -r requirements.txt` from within a fresh virtual environment



Pythonic Code - List Comprehensions

Python and “Pythonic” Code

- Believe it or not, but Python has a lot to make code easier to write
- Full of features that aren’t taught in most classes, but are useful
  - List comprehensions, but in more depth
  - Tuple Unpacking
  - Advanced Dictionaries
  - Decorators
  - Like the last Python lecture - this is only a taste of what’s available
  - (Refer to either the Python docs or “Fluent Python” for more!)

List Comprehensions

- Being lazy is great!
- Common to loop through a list to change data points, one by one
- List Comprehensions allow us to do this, lazily!
- General Syntax
  - [<operations on item> for item in <list-like object>]
  - Use only if you intend on generating a List
    - Use map if you only care about side-effects, like printing to stdout
    - Use filter if you want to remove values based on some condition

List Comprehensions - Example 1

```
# Let's say we want to get the squares of the first 10 numbers into a list
# We can do it using the following for loop:
input_list = list(range(10)) # Generate a list from 0 to 9
output_list = []
for i in input_list:
    output_list.append(i**2)

# Instead, we can do the following:
output_list = [i**2 for i in list(range(10))]
```

List Comprehensions - Example 2

```
# Let's say we want to generate a list of 5 lists, each of length 6, with values -1
# We can do it using the following nested for loop:
output_list = []
for i in range(5):
    inner_list = []
    for j in range(6):
        inner_list.append(-1)
    output_list.append(inner_list)

# Instead, we can do the following:
output_list = [[-1 for j in range(6)] for i in range(5)]
```

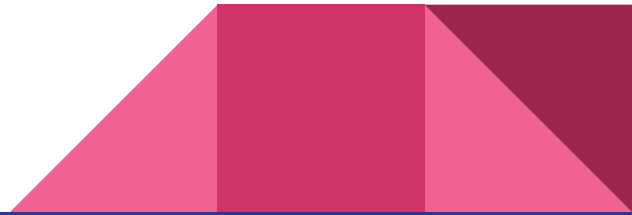
List Comprehensions - Example 3

```
# We can even use if statements in list comprehensions!
# If we want to get only the squares of the odd numbers
# from 1 to 100, we can do it using the following nested for loop:
output_list = []
for i in range(100):
    if i%2 == 1:
        output_list.append(i**2)
# Instead, we can do the following:
output_list = [i**2 for i in range(100) if i%2 == 1]
```

Pythonic Code - Tuple Unpacking

Tuple Unpacking

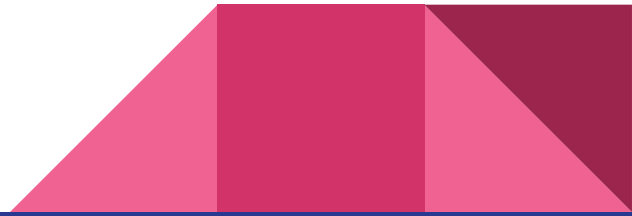
- More of a Python hack, but leads to way cleaner code
- Recall that tuples are a “data structure of ordered fields”
  - a_tuple = (1,2,3)
  - Seemed relatively useless
    - Can’t edit them
    - Can’t extend them
  - While they have their normal uses, seemed relatively esoteric
- However, the truth is a LOT of pythonic code relies on tuples!



Tuple Unpacking

Tuple Unpacking

- Python generates tuples whenever it sees something like the following:
  - a,b = b,a
    - Unique construct to Python.
    - Python's does a swap of the two variables
  - How?
    - Python evaluates the RHS first
      - Creates a tuple (b,a)
    - Python then assigns each value in the tuple to the LHS
      - a gets assigned the past value of b
      - b gets assigned the past value of a



Tuple Unpacking

- Not unique to swaps, either!
  - Does anyone got any ideas what this does?
    - `a, b = 1,1`
    - `a, b = b, a+b`
    - `a = 1,1,2,3,5,8,.....`
- Leads to code that’s easier to read and work on
  - Less temp variables
  - Less noise on what’s happening in between

Tuple Unpacking

- Even useful for your functions!
- Let’s say you want to take any number of arguments

```
def allTheArgs(*argv):
    for arg in argv:
        print("I have you now, ", arg)


allTheArgs("Peter Pan", "Wendy", "Tinker Bell")
```

- Here, * is a term known as the “splat” operator

The “Splat” Operator

- Term coined from Ruby
- Serves to “unpack argument lists and tuples”
- Example usage:

```
a, *b, c = (1,2,3,4)
# a = 1
# b = (2,3)
# c = 4
```

Pythonic Code - Advanced Dictionaries

Advanced Dictionaries

- Deep inside the standard library, there is a package called collections
  - Contains tools that some might seem … useful
  - Tend to make certain tasks easier
    - Counting objects
    - Assigning values to objects
    - Maintaining queues with keys
  - Also lead to faster interview code, if you need to write code for a screener!

Advanced Dictionaries - Counter

- Do you want to simply count the number of objects that you process?
- Introducing collections.Counter
  - Create via c = Counter(iterable)
    - Iterable is any python type you can loop over
      - Lists
      - Dicts
      - Etc.
    - C[key] returns the number of occurrences of key in iterable
  - Can do much more than simply count!

Advanced Dictionaries - Counter

- c.elements()
  - Returns the elements of iterable, returned in the order first encountered
  - Repeats as many times as value occurs
- c.most_common([n])
  - Returns a list of the n most common elements in iterable
- c.update(next_iterable)
  - Processes elements in next_iterable one by one
  - Adds them to the counter
- You can even add and subtract Counters!

Advanced Dictionaries - DefaultDict

- Do you want your dictionary to have default values for missing entries?
- Enter: collections.defaultdict
  - Acts like any normal dictionary in all but missing data cases
  - Creates an entry for the missing key, and assigns it a value you decide
- Example:

```
d = defaultdict(lambda: -1)
# d["a"] -> -1
```

Advanced Dictionaries - ?????

- The rest of collections is cool, but more application specific
  - RTM!
- See something you want in a Python dict, but it's not there?
  - Create it!
  - Python dicts are just fancy Python classes that implement the following methods
    - Look into "dunder methods", for more information!
    - E.g. __setitem__ , __getitem__
    - If it walks like a dict, talks like a dict, then by golly it's a dict!

Pythonic Code - Decorators

Decorators

- Functions in Python are first-class objects
  - You can assign them directly to variables
    - There ARE NOT pointers, but literally functions assigned to functions!
  - You can pass them into objects
    - Again, not as pointers, but as literal functions
  - You can even return them from other functions!
    - Seems dumb, but it’s really, really important!

Decorator

- In a lot of high-level Python code, you'll see something like:

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

Decorators

- Here, make_pretty is a decorator
  - It takes in a function, and returns another function
  - In this case, when you run ordinary, you're going to get a little more than it!
  - Common in web development via Flask, and some data science
- These allow for meta-programming, and open up new design spaces
  - Won't go into here for time
  - Look in "Fluent Python" for much, much, more!

Decorators

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner


@smart_divide
def divide(a, b):
    print(a/b)
```

That’s All!
Good Luck on Finals!