Python

Ray's Rules

- Rules picked up throughout time in industry
- In small components
- Functions are as small as possible
- 1) Build small
  - Test often
- Never assume something is happening - print every stage
- 2) Build it, link test it
  - Test it in context
- Once you've built something small, you need to link it to something
- Invoke your function from the expected context - Ray refers to this as linking it
- 3) Print everything
- 4) Write down your errors
- 5) Know your definitions
  - Be able to explain them deeply
- Many of your errors are common errors.
- If you note the errors, and how you solved them, you'll be ok.
- There are keywords and definitions that are awesome in python
- Invalid syntax
- Multi dict error
- 6) Practice algorithms **one extra hour a day**
  - There are 4 key functions you should be able to perform on any data structure.
  - Stack
  - Dict(!)
  - FIFO
  - Priority Queue
  - You can split these into getters and setters, and you should know how to do each.
  - It will make all other algorithms easier
  - If you forget the built in functions, you can always write your own.
  - Also allows you to build your own data structures
- Test whether you can explain something to someone.
- 7) If you can teach it, you can do it
  - 'Teach Ray'
  - It reveals very easily the places when you don't know something.

The instructors

- James is the other python God
- Ray
  - Bio
    - Loves socker
    - Volleyball league
    - Big on sports
    - Tuesday and thursday he leaves early
    - Masters in business administration and computer science
    - Was also a Guitar Teacher
    - Ray was a recording ba, music theory ba, and marketing.
  - Times
    - Arrives most days at 7:15ish
    - Leaves most days at 7:38PM

Methods and Functions

Strings

- string.count(substring)
  - Checks the count of the substring in the string
  - Returns integer
- string.isalnum()
  - Checks if string only contains alphanumeric characters
  - Also checks if length is greater than 0
  - Returns bool
  - There are similar methods such as .isalpha or .isdigit, .islower, or .isupper
- .join(list)
- .split(char)
  - Will split string into a list based on char val
  - Default is to use space " "
- .find(sub)
  - Returns integer index of first occurance
- .endswith(substring)
  - returns bool
  - Whether string end with sub.
- .replace(new, old)
- .isalpha()
  - checks if a string is only alphanumeric

Lists

- Splice using list[1:5]
  - Will return a new list between the indexes mentioned
  - new_list = [1,2,3,4,5,6]
  - new_list[1:] will return [2,3,4,5,6]
  - new_list[:3] will return [1,2,3] - note that index 3 in the 4th position, so picks out 4, and the method returns everything BEFORE not including that value.
- Functions (good for any sequence, such as tuples or strings)
  - Useful in a 'for loop' context to return a list of tuples for each item in the list you pass in.
  - enumerate()
  - The tuple is two items long, the index followed by the value at that index.
  - map(sbFunction, sequence)
  - Returns lowest value in list
  - min(sequence)
  - Produces a symbol delineated string from the sequence
  - 'symbol'.join(sequence)
  - Returns a sorted sequence
  - list.sort() will on the other hand modify the list itself.
  - sorted(sequence)
  - max(sequence)
  - len(sequence)
  - sum(sequence)
- Both operands known as iterables
- Methods
  - You can also concatenate two lists using + or +=
  - .extend(list2)
  - Whatever object is passed is appended as a single item at the end of the list
  - .append(object)
  - Extend instead concatenates the lists
  - .insert(index, value)
  - Removes the value at that index, defaults to last value
  - .pop(index)
  - .index(value)
  - Reverses a list IN PLACE
  - .reverse
- .join(separator, values.list)
  - Create a list of tuples from lists of strings

Dict or object

- functions
  - cmp (dict1, dict2)
    - If the two are equal, return 0
    - dict1 is larger than dict2, get 1, else -1
    - This is based first on a comparison of keys
    - Then on a comparison of values
  - len(dict)
  - str(dict)
    - Returns string in form of 'Key' : Value}
  - type(dict)
    - Returns 'dict'.
  - key in dict
    - Returns bool based on whether the key is int he dict
  - del dict[key]
    - Deletes the key from the dictionary
- methods
  - Can use them via dict.method(dictName) or dictName.method()
  - .clear() removes all elements
  - .copy() returns shallow copy
  - .fromkeys(keys, values)
    - Will create new keys based on a sequence of keys
    - If provided, will also pair them with a default value
  - .get(key, default)
    - Will return the value of a key for given key
    - If key doesn't exist, will return default
  - .setdefault(key, default)
    - Will return the value for key
    - If no key there, will create that key and fill it will the default.
  - .has_key(key)
  - .update(dict2)
    - Adds dict2's key values to an existing dict
  - Converting to string
    - .items()
      - Returns list of key/value tuple pairs
    - .keys()
      - Returns a list of keys
    - .values()
      - Returns list of dict values
    - print data.items()
      - #[('house', 'Haus'), ('red', 'rot'), ('cat', 'Katze')]
    - print data.keys()
      - #['house', 'red', 'cat']
    - print data.values()
      - #['Haus', 'rot', 'Katze'].

Random Atoi

- Import the random module
- Integer within range
  - randint(a, b)
  - random.choice(list)
  - random.shuffle(x[, random])
- For sequences
  - Random.sample(sequence, how_many)
  - Will return a list of int length.
  - Removes items from the sequence with our replacement
- Core randomness
  - random.random()
  - Floating point from 0-1
  - Way to do population sampling.

Bermuda Time-angle

- Time conversion **https://docs.python.org/3.3/library/time.html?highlight=time.strftime#time.strftime**
- Convert strings back into datetimes
  - Use time.strptime(string, format)
  - The format you provide should match the format of the string used.
- Simple guide of printing formatted time stamps
  - Import the time module
  - Specifically as#time and gmtime
  - Generate a time stamp with time.gmtime()
  - Format the display of the timestamp with time.strftime
  - time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime(ts))
  - time.strftime(FORMAT, TIME)

Python OOP

Intro

- Ray's Lecture
- Objects are collections of key value pairs
- Object literal would be whole object written out- so curly bracket notation
- Literal
- Non literal
- If you want to look up a value, you have to use square brackets
- Creating classes
- Classes are things for creating things
- You need to call Classes if you actually want things
- Need to always be passing self
- we use them because looking up is super fast
- We can keep lots of elements under one name
- Obj."prop"=val, val."method"= func ( )
- Obj = {}
- Obj.prop = val
- obj['method'] = function()

Inheritance

- class ChildClass(Parent):
- To inherit another call, replace the object parameter with your desired parent
- Implicit inheritance will pull across the parent's methods and attributes
- If you want to run the parent's method, you can use the following syntax
  - **super(ChildClassName, self).parent_method()**
- Accessing the parent with Super
  - superChild(Name, self)...init...()
  - None- this is called from within a ChildName method
- Within the child class, we can access the parent's init function
- We can then overwrite any of the consequences of super init which we don't want.

Basics

- class ClassName(object):
- def __init__(self, parameters):
- Create JP instance like class attributes
- Self = parameter which can be assigned when creating instance. Refers to the Class instance
- If you change the parent object's attributes, all the instances of that object will also change
- def method(self, params):
- All methods will be passed a reference to self as the first argument
- Implicit passage of self.
- init
- def __init__(self, params) is the constructor method in python
- You can use it to setup the class instance with case-specific attributes and methods
- You can chain methods like (classInstance.login().show().logout())
- This only works if you use return self at the end of your methods
- class User(object):
- def login(self):
- // your code goes here...
- return self
- Attributes are characteristics shared by all instances of a class
- Attributes
- Methods are actions which a class instance object can perform. They must be called ON an object
- Created from the appropriate class.
- Ref values using dot notation
- self.property_name = var

Modules & Packages

- Modules
- You can import by listing a file name, and it will execute the file.
- All the functions/classes/variables within it are then available
- We can see which modules were implemented by a given module with dir()
- Diagnosing modules
- help(nonsenseModule) provides even more detailed info
- Packages
- A package is a collection of modules in directories, or from the programs point of view, a namespace containing packages and modules
- The directories give the package hierarchy
- Essentially directories
- If we have this file, even empty, python will allow you to import modules appropriately
- All packages must have __init__.py in their root
- **ie, import my_mod.test_mod** or **from my_mod import test_mod**
- This assumes test_mod.py is a file adjacent to __init__.py in a folder called my_mod adjacent to the caller's file (ie)
- def ... = "test_mod";
- This file can also filter which packages export as an API
- The above would limit the modules in the 'import' all command to just be test_mod

Namespaces

- name
- Gives us the current namespace.
- If the sile is self executed, it's "__main__"
- If any other file, "filename"
- Can use conditionals to make sure certain code only runs when in a certain namespace
- if __name__ = "__main__":

Regex

- Island of Regex
- Essentials
- Matches for things that start with a
  - ^a
  - Apple
  - matches
  - apple
  - asasas
  - failures
  - Tobby
  - bob
  - .a$
- Can use it for single char too z[3] would match any word with exactly 2 z's in a row
- .{3,} three and up
- .{,3} three and down
- .{3} three and only
- Checks that the preview condition has matched at least 'x' times
- .{3,5}
- You can specify a range of times with {3,5} (between 3 and 5 repetitions) or {,8} (up to 8 repetitions)
- .+a
- .*a
- 10-9977
- myBigSocks
- matches
- wwwww
- 4
- myBooks
- failure
- 9 9 9
- Wordlettes
- Matches anything in range, should be characters not strings
- [0-5]
- 0
- 45
- matches
- 27bn33a
- NanOtaNite
- 14
- 1sock
- failures
- NotMuch
- [A-C] [a-c]
- Acid
- Rich
- 2Fit
- Zeal
- Mat
- Ore
- anA
- matches
- failures
- Matches for things ending in b
- Bob
- cob
- matches
- slob
- Boo
- Bobby
- failures
- blobbbbb
- \b
- Will match any digit followed by ' mow'
- 'he 3 mow'
- matches
- Ooh 9 mow bon
- mow
- I love mow
- fails
- I love 3mow
- Non-digit: '\D'
- Alphanumeric '\w'
- Non-alphanumeric '\W'
- Simular
- Whitespace '\s'
- Non-whitespace '\S'
- Select the boundary between word and non-word chars
- Most systems will allow you to reference your previously captured groups
- '\0 will usually match the whole original input
- 'n then matches the nth group
- Bonus Round
- '\0 \1 \2'
- Creating regex objects
- Using the re library, you can create an object from a regex selector
- NAME = re.compile('EXPRESSION')
- NAME.match(STRING)
- The object has a .match(..) method which can be used to look for any occurrences it's pointed to.
- re.compile('^\w (A-Za-z0-9 - _]+@[a-zA-Z0-9 -]+\.[a-zA-Z]+$')
- Capturing things ( )
- Talk it out
- Everything is in brackets, so everything will be return
- + means that any number of chars of any kind will match at the start
- Second set of brackets indicate a sub-group, it will be returned in addition to the whole group its a part of.
- \0-9] is looking for any digit.
- {4} looks for 4 repetitions
- So, this will match anything that lends in 4 numbers
- It will return two strings, the whole match (including the 4 nums), and just those 4 numbers
- \d{4}
- Or |
- expression|express will match expression or express
- Brackets contain matching rules.
- Any characters matched by them will be returned
- NOT and OR
- Brackets and hat are a NOT
- [^n-q] will match any char except letters n to q
- Whitespace \w
- The whitespace selector picks out space , tabs \t, line breaks \n and carriage return \r
- '\n\t\r\n\n\n\t\t\t\t' will match for 'anything-else' except 'something-else'
- Optional?
- Whatever is preceded by a ? is optional
- What the hell is Regex?
- Regular expressions are a way of excepting/matching code
- Useful for validation, for example checking if there's an @ symbol followed by a .
- Regular expressions are used to check if a set of conditions apply to a particular sequence of characters.
- If your query were a plain string, it will search for ANY match of it, no matter if the match is in quotes or nested in brackets
- Knowing regex is a matter of knowing the syntax for building your selectors.
- Klean stir and Keen plus
- '.' will match any character, to match against a period, use '\.'
- anyRegexThing' will match for any number of repetitions of that regex thing
- anyRegex+ will match for AT LEAST ONE of that thing
- "\d+" would match strings with at least one digit

Flask

Basic Flask App

- Get flask
- You need a python virtual environment
- Activate with **$ source flaskEnv/bin/activate**
- File Structure to create for basic app
- app.py
- templates/
- index.html
- static/
- style.css
- script.js
- You need to import **'from flask import Flask'**
- Additional extras
- render_template
- For calling templates
- redirect
- For calling a new route
- request
- For getting form data.
- session
- flash
- You need to create an instance of a flask app with **app = Flask(__name__)**
- We can then add routes with **@app.route('/')**
- Routes
- Intro
- A route is akin to a variable, different routes will trigger different behaviours from the server
- Also known as a handler for particular HTTP requests
- Routes have two parts – the HTTP method, and the URL
- Routes default to get
- Advanced routing
- You can store additional queries added to your route to change how your route processes the request
- E.g. /users is the route, but /users/NAME could adapt per NAME
- Can write function to accept parameter with name of var_name
- @app.route('/route_name/<var_name>')
- def route_function(var_name):
- The return value is what gets printed to the client's screen
- And finally you tell the app to run with **app.run(debug=True)**

Forms

- In your template, the html action attribute specifies the route the form should be sent to
- Everything **within a form tag** is submitted via a post route.
- Can access the data within the post route for that form
- **request.form** is a dict, with the form input names as the keys
- request.form['name_of_input']
- Hidden fields
- Allow us to differentiate forms sent to the same route
- Create a input with a type of hidden, and whatever value you set it, while invisible to the user, can be used in the backend
- <input type="hidden" name="form_type" value="register" />
- Example
- if request.form['form_type'] == 'register':
- // do registration process
- elif request.form['form_type'] == 'login':
- // do login process
- **return redirect('/different_route')**
- **return render_template('name.html')** - no adding a slash

Sessions

- What is a session in Django?
- Getting started
- You can't use sessions until you've migrated your apps.
- Call manage.py from project root.
- **$ python manage.py makemigrations**
- **$ python manage.py migrate**
- Now restart your server
- Dictionary object, so can use dict notation or brackets to set and retrieved named values.
- Stored under the request object, just like form data stored under POST
- **request.session['name'] = request.POST['name']**
- **request.session['counter'] = 100**
- In Django, sessions are always saved in the database.
- You should know that these already run your dict methods.
- **del request.session['foo']**
- Delete a session key.
- Note that it will throw an error if the key doesn't exist
- **'key' in request.session**
- Check for a key
- Returns bool (True or False)
- Handy session methods/functions
- Within your templates, you can access session data just like any variable **{{request.session.name}}**
- The session object is only updated if Django detects that it has changed.
- Appending to lists
- Modifying a sub-list from a multi-dimensional list
- In various circumstances, that check will fail.
- Force python to update the session with **request.session.modified = True**
- In settings.py, you can change the default behaviour and have it save every time.
- Sometimes you modify a session key, but this change isn't saved properly
- **SESSION_SAVE_EVERY_REQUEST = TRUE**

Forms

- Forms need a CSRF token
- Data is accessible under request.GET or request.POST
- Rendering data is very similar to in flask/jinja
- Designed to only allow templates to fill in information- very limited logic abilities.
- **https://docs.djangoproject.com/en/3.1/ref/templates/language/**
- Can be setup to use Jinja2
- Uses Django templating language or DJ
- {% csrf_token %}
- Inhibits clients sending forms not generated by the app. An AJAX form spammer would not be able to do so.
- **request.method** shows which request GET or POST was used.
- variableName = **request.POST**['formInputName']
- Similar to Jinja

Rendering templates

- Passing data to a template from the view is easy using a third parameter
- Render takes 3 parameters, a http request object, a template, and a additional arguments
- Note that the key values in context are bound to the template directly, not via context
- {{ username }}
- {% csrf_token %}

Routing / urls.py

- urls.py is a routing file
- How to do this
- Make sure to import the include module into your project level urls.py
- When you pass the url on for further routing, use (dot notation)
- 'urlpaged.expression, **included('apps.NAME.urls')**
- In your urls.py expression, the first parameter a should specify which urls we're listening for.
- When you pass through a route, the part of the url that gets matched is chopped off the url
- Passing on to another route
- Choosing which route for what requests
- We use regex, so wrap your regex selectors in r'
- url('**REGEX**', whatToCallIt)
- 'r/some/the' passed through the home route, will pass ON a url of 'some'
- Not currently sure how this works early on it routing, but **when we're calling a view's function,** we can state it matches as named variables.
- Name the parameter with r'?P<name>.*/'
- 'P' for naming Parameters, use <> round the name
- These are then passed onto the view as named parameters
- How to save and return matches
- You can save and later use the values that you're chopping off your url.
- In app urls.py - url r'(?P<name>.*)/',
- views.viewFunc),
- In views.py - def viewFunc(request, name):

URLs, Views and Templates

- The views
- Archetype views file is a series of functions.
- Assuming the views file is in the same directory as your urls.py (it should be if you're using the standard structure)
- Import the views file to the urls
- **from . import views**
- To call a view from your urls.py
- Use not notation to pick out the function you want
- Make a url call to the function within views.
- Wrap the whole query in quotes
- **url(r'^$', 'views.give_time')**
- **Store your data in an object known as context**
- **return render**(request, path/template.html, **context**)
- So to call context.keyname in the template, you just use keyname
- Passing data to a template from the view is easy using a third parameter
- Render takes 3 parameters, a http request object, a template, and a additional arguments
- Note that the key values in context are bound to the template directly, not via context
- Rendering templates

Django

Ray's lecture

- Static Files
- Setup
- In templates
- **{% load static %}**
- **href="{% static 'ourApp/main.css' %}"**
- NOTE that when you deploy, you should change the method used to serve your static files'
- Django finds which files to use based on a STATICFILES_FINDERS variable
- You can specify its methods in settings
- (
- 'django.contrib.staticfiles.finders.FileSystemFinder',
- 'django.contrib.staticfiles.finders.AppDirectoriesFinder'.
- First line checks the static directories defined in STATICFILES_DIRS
- The second line tells the project to check each app's static directory.
- More info on how static files are found and grouped by Django https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-STATICFILES_FINDERS
- Project settings.py
- To define extra directories for Django to import static files from, define a list in settings.py
- **STATICFILES_DIRS = [os.path.join(BASE_DIR, "static"), '/var/www/static/',**
- **]**
- The directory is not relative to the project, so you need to do shit like this path.join
- Need to run this block before we request static files.
- Shared files
- Make sure you set the directory name for your static content - **STATIC_URL = '/static/'**
- Make sure order installed apps you have **django.contrib.staticfiles** should always be there.
- Command
- {% static 'ourApp/css/main.css' %}
- Open a code block using Jinja, call the static operator, and specify the path within static you want.
- You could access another app's files if you wanted, by looking in their directory
- If you don't use app specific directories, you'll have conflict issues with your files.
- Remember, all apps' static files, like templates, are combined into one directory.
- Squares are folders, dashed lines are files we manually create.
- Note that this does not list ALL the files, just the ones relevant to a basic example.
- File Structure
- project_name
- manage.py
- app_name
- apps.py
- models.py
- tests.py
- views.py
- admin.py
- __init__.py
- migrations/
- __init__.py
- project_name
- settings.py
- urls.py
- wsgi.py
- templates/
- index.html
- static/
- urls.py
- Core Picture
- Object sent as packets through internet tubes (no cats)
- Project_name
- HTTP Request
- Request Flow
- Project specific settings, like which apps are activated
- First tier routing, points to urls file for specific apps
- Second tier routing, imports views and calls a view
- Core role is running functions and returning templates
- apps/app_name/views.py
- apps/app_name/urls.py
- project_name/urls.py
- project_name/settings.py
- manage.py
- index.html
- templates/app_name
- view.like
- Model
- templates/app_2
- other bits
- apps/app_2/views.py
- apps/app_2/urls.py
- When a template is rendered, it is passed through manage.py and sent back to client.
- It handles your apps requests and points them to the right views file.
- There is always a project level routing file in the project_name directory.
- We setup per-app urls.py files too, since then we can break up our routing specific to each app.
- urls.py is a routing file.
- Routing / urls.py
- Passing on to another route
- Choosing which route for what requests
- Make sure to import the include module into your urls.py
- How to do this
- Setup home_app urls by copying the project urls.py file into home_app/urls.py
- Do a test with your test def
- Make route mid
- Import the views file into the app's urls file
- **from . import views**
- Change the test route to call views.test
- View file needs to import **render** from django.shortcuts and have the test function
- If the test works, move the test function from urls.py to your view
- Render your template
- Within your app's view.py, change the test function to **return render(request, 'home_app/index.html')**
- Note that all the templates from all your apps are combined, and we can pick out our app's templates only because we added the home_app subdirectory to templates.
- Download test app
- https://github.com/thisjohn/review-coding-dojo/blob/master/Python/Django/ray_app.zip
- Names slightly different, but works based in python/django virtual env.
- Build by following Ray's exact instructions
- Shut your folders as your go.
- Build it, link it, test it
- Do not randomly flail for the cause of the error, use logical steps
- Test from base up
- project urls.py
- apps urls
- view.py
- template file
- CRUD is the basic model our web apps are using
- Create
- Read
- Update
- Destroy
- All your routes will largely call under one of these slightly expanded operations
- Temporary session dependent stuff doesn't fit in will like login
- show_all
- show_one
- edit
- update
- delete
- post
- create

Building your app

- Create Django instance
- **$ django-admin startproject django_instance**
- **$ cd django_instance**
- Create app
- Creating an apps folder is equivalent to creating an Class object in Django
- Since it's a class it needs an init function
- Once you have that, you can tell Django to create apps inside it and Django will know what to do.
- **$ pwd -> django_instance**
- **$ mkdir apps && cd apps**
- **$ touch __init__**
- **$ django-admin startapp first_app**
- Supplement app structure
- **$ cd first_app**
- **$ touch urls.py**
- Will do app specific routing when called by the project urls.py file.
- In theory, project urls could route to the app templates
- **$ mkdir templates && cd templates**
- **$ mkdir first_app && cd first_app**
- Need to have our templates in a app names sub directory
- This is because Django will combine all the template directories across apps
- **$ touch index.html**
- Configure and Start Django Instance
- Open settings.py
- Under installed apps, add **'apps.first_app',**
- Run the server
- **$ pytuhg manage.py runserver**
- Setup a route to app
- Navigate to the 'project root' directory
- Navigate to the 'project root/project name'
- **$ open urls.py**
- You have a url patterns list
- They're regex conditionly - if the condition match, do a thing
- If you try to access a route that hasn't been setup, you'll get a list of the routes tried, and an error that no route matched.
- Test you can print through your route
- Add url patterns of attr="$", test.
- r = regex expression
- ^ = for any start.
- $ = end of string.
- Nothing between the two, so looking for routes of nothing.
- Add a a def test(request): function which just prints a string
- When you try the route, you should get an error in your browser AND your print statement in your console.
- While in your project's urls.py, add **include** import from django.conf.urls
- Add route under urlpatterns pointing to apps.home_app.urls
- url(r'expression', include('apps.NAME.urls')),
- Can get rid of **admin route**, and the **admin import**, ADD the **include import**
- Make route mid
- Download test app
- Build it, link it, test it
- Build your folders as your go.
- Test from base up
- Setup a route to app
- Create app
- Create Django instance
- Supplement app structure
- Configure and Start Django Instance
- Building your app
- CRUD is the basic model our web apps are using
- Create
- Read
- Update
- Destroy
- Command
- Project settings.py
- Setup
- In templates
- Shared files
- Static Files
- Getting started
- Sessions
- URLs, Views and Templates
- Routing
- Forms
- Modules & Packages
- Namespaces
- Inheritance
- Basics
- Intro
- Python OOP
- Strings
- Lists
- Dict or object
- Methods and Functions
- Flask
- Basic Flask App
- Forms
- Sessions
- Ray's Lecture
- Django

Islands of Regex

- Essentials
- Matches for things that start with a
  - ^a
  - Apple
  - matches
  - apple
  - asasas
  - failures
  - Tobby
  - bob
  - .a$
- Can use it for single char too z[3] would match any word with exactly 2 z's in a row
- .{3,} three and up
- .{,3} three and down
- .{3} three and only
- Checks that the preview condition has matched at least 'x' times
- .{3,5}
- You can specify a range of times with {3,5} (between 3 and 5 repetitions) or {,8} (up to 8 repetitions)
- .+a
- .*a
- 10-9977
- myBigSocks
- matches
- wwwww
- 4
- myBooks
- failure
- 9 9 9
- Wordlettes
- Matches anything in range, should be characters not strings
- [0-5]
- 0
- 45
- matches
- 27bn33a
- NanOtaNite
- 14
- 1sock
- failures
- NotMuch
- [A-C] [a-c]
- Acid
- Rich
- 2Fit
- Zeal
- Mat
- Ore
- anA
- matches
- failures
- Matches for things ending in b
- Bob
- cob
- matches
- slob
- Boo
- Bobby
- failures
- blobbbbb
- \b
- Will match any digit followed by ' mow'
- 'he 3 mow'
- matches
- Ooh 9 mow bon
- mow
- I love mow
- fails
- I love 3mow
- Non-digit: '\D'
- Alphanumeric '\w'
- Non-alphanumeric '\W'
- Simular
- Whitespace '\s'
- Non-whitespace '\S'
- Select the boundary between word and non-word chars
- Most systems will allow you to reference your previously captured groups
- '\0 will usually match the whole original input
- 'n then matches the nth group
- Bonus Round
- '\0 \1 \2'
- Creating regex objects
- Using the re library, you can create an object from a regex selector
- NAME = re.compile('EXPRESSION')
- NAME.match(STRING)
- The object has a .match(..) method which can be used to look for any occurrences it's pointed to.
- re.compile('^\w (A-Za-z0-9 - _]+@[a-zA-Z0-9 -]+\.[a-zA-Z]+$')
- Capturing things ( )
- Talk it out
- Everything is in brackets, so everything will be return
- + means that any number of chars of any kind will match at the start
- Second set of brackets indicate a sub-group, it will be returned in addition to the whole group its a part of.
- \0-9] is looking for any digit.
- {4} looks for 4 repetitions
- So, this will match anything that lends in 4 numbers
- It will return two strings, the whole match (including the 4 nums), and just those 4 numbers
- \d{4}
- Or |
- expression|express will match expression or express
- Brackets contain matching rules.
- Any characters matched by them will be returned
- NOT and OR
- Brackets and hat are a NOT
- [^n-q] will match any char except letters n to q
- Whitespace \w
- The whitespace selector picks out space , tabs \t, line breaks \n and carriage return \r
- '\n\t\r\n\n\n\t\t\t\t' will match for 'anything-else' except 'something-else'
- Optional?
- Whatever is preceded by a ? is optional
- What the hell is Regex?
- Regular expressions are a way of excepting/matching code
- Useful for validation, for example checking if there's an @ symbol followed by a .
- Regular expressions are used to check if a set of conditions apply to a particular sequence of characters.
- Knowing regex is a matter of knowing the syntax for building your selectors.
- Klean stir and Keen plus
- '.' will match any character, to match against a period, use '\.'
- anyRegexThing' will match for any number of repetitions of that regex thing
- anyRegex+ will match for AT LEAST ONE of that thing
- "\d+" would match strings with at least one digit

The Django ORM Canal

ORM Commands from the shell

- Start Django shell with **$ python manage.py shell**
- Load the relevant tables (app specific) with **$ from apps.NAME.models import ***
- **https://opensourcehacker.com/2014/06/13/turbocharge-your-python-prompt-and-django-shell-with-ipython-notebook/**
- Shell Access
- Better shell
- All commands are written as Table.objects.command(...)
- **Create** records
- Immediately **create** a record
- Parameters of **fieldName = "value"**
- $ Var = TableName(**"value", field2="val2"**)
- Store the new record under a variable and save/**create** it when ready
- $ Var.name ="new Value"
- **$ Var.save()**
- -Only adds to DB when you save
- Basic **Retrieval**
- Returns all table elements in an order
- You can reverse the order by using "-fieldName"
- Returns multiple records
- You can use dunders to build this out
- For opposite, use **.exclude(...)**
- Returns a single record
- **.get("SQL query")**
- Can optionally provide values, returns it similar to the get method
- Will return a list of dicts, the field names as key, and field as value
- values_list() will return a list of tuples, with the id in position 0
- Can provide extra argument of flat=True to just get a list of the id's
- Set a **relationship**
- Based on the naming conventions laid out by Django
- The field name the foreign table's name. Pluralisation depends on type of relationship
- The value it needs to be passed is the ID of within the foreign table of the record
- **.createThisWhatever = Table.objects.get(id=NUM)**
- Pro-tip, look up the related record and store it in a variable first.
- **Retrieval** from related tables
- $ firstPost = Post.objects.first()
- First you have to select a record
- Then you select the foreign key field, which will do a join to that record's model
- $ theComments = var.comments
- Can chain this all together, to say, list all the comments for the first post.
- $ Post.objects.first().comments.all()
- **Updating/Delete**
- Basically a method call on the record you want to bail with.
- $ Var = Table.objects.first()
- First call and store the record
- $ Var.name = "New Name"
- Then change a value
- $ Var.save()
- Then save/delete
- $ Var.delete()

How to build the database

- In app_name/models.py
- Tables be created with classes
- class TableName(**models.Model**):
- colName1 = **models.SomeDatatype(**params**)**
- colName2 = **models.ForeignKey(**params**)**
- created_at = **models.DateTimeField**(auto_now=True)
- Column types are created as methods of the models object. E.g. models.TextField()
- **CharField(max_length=)**
- max_length = 255.
- **TextField()**
- Unlike CharField has no set length so can take massive amounts of text.
- **IntegerField()**
- **BooleanField()**
- **DateTimeField(...)**
- auto_now=True will update to now whenever the object is modified
- auto_now_add=True will set the time to now value (and only when) the object is created
- To add relationships
- Add a col with the other table's name.
- The way you name it is specific to the kind of relationship
- If am connecting blogs and comments, then since each comment can only go to a single blog, the name you should be "blog" singular
- If I am linking the admins list to my blogs, then since each blog can have many admins, the name would be "admins" plural.
- Column types for relationships
- models.**ForeignKey(...)** for simple many to one relationships
- Takes the name of the foreign table, and the name by which the current table should be referred.
- blog = models.ForeignKey(Blog, related_name = "comments")
- models.ManyToManyField(...) for many to many relationships.
- Takes same params of ForeignKeyField, spelled...
- models.OneToOneField
- Once database is created, need to run migrations
- A major roll of migrations in general
- They're stored in a migrations STRUCTURE back in the directory
- This is a major reason to use a framework, since they support migrations
- makemigrations
- When you run makemigrations it will show you a log of the changes you'll make
- Will also create a file with instructions to be run.
- **NEVER DELETE**
- Run this any time you change models.py
- 'Migrate' will run the file.

Bits and bobs

- Need to get a sqlite viewer/browser to deal with output of queries
- Django hides the face that each app is had its own table prefixes
- If you need to interact with the DB outside of Django, remember this.
- Set .__repr__ methods for your models so that when you print it, it spews an informative one/line.
- Or, when dealing with a set of records from the DB, you can print each one with a single field selected.

Persistent data

Intro

- ORM = Object Relational Manager
- Makes dealing with database (DB) much easier for simple joins etc.
- Represents data in an object oriented way for the language in question (Python here)
- SQL requires query strings and table data.
- Very powerful
- To use within Django/python directly means concatenating very long strings each time.
- ORM's give data as objects with well defined methods and properties.
- Means you can interact with database as if it were a python object
- In Django, the model is controlled from the project_root//apps/app_name/**models.py** file.
- Django uses a SQLite database by default
- Very fast, since the DB is stored as a local system file.
- Not as much functionality as full MySQL
- Great when in development phase, maybe not when deploying
- **With an ORM it doesn't matter what DB language is actually used- the ORM, and hence our way of interacting with the DB, will remain the same.**

Sessions

- Start Session as an object which is hashed and stored as a cookie by Flask
- It can be used in the template files also with Jinja
- Flash data is stored across one redirect only
- Flash storage stores messages intended for display on the next rendered page (for example, a validation error).
- flash("string") will add a message into the flash storage
- {% with messages = get_flashed_messages() %}
- {% if messages %}
- {% for message in messages %}
- {{message}}
- {% endfor %}
- {% endif %}
- {% endwith %}

Apps

- Following your route, you write your functions **def function():**
- Flask uses Jinja2
- Template files are what is parsed and returned to use user.
- Within a template, Jinja can interpret some of pythons logic, but not all.
- Can also interpret variables
- Limit your code logic in template your logic should not be done if the view!
- For another, if makes your logic harder to see and keep track of.
- More info at **http://jinja.pocoo.org/docs/dev/templates/**
- Can pass named variables to the templates from the view.
- Double curly brackets for output {{ param_name }}
- How to use python code inside templates
- Views / Templates
- Run silent operations using {% %}
- {% for x in range(0,5) %}
- For loop
- {% x %}
- {% endfor %}
- {% if condition %}
- If statement
- {% endif %}
- {{ variable to be printed if condition met }}
- **{% url 'for'static'./filename=./style.css' %}**
- url_for filename argument is relative to app root
- In your HTML templates you use special variable print notation with a special url_for
- Usually sub-divide static file into .img / .js / .css
- Static files
- You need to setup a folder to be used for static files, like css and js

Route methods

- Get Method
- Also in search forms, when the search engine will send an HTTP GET request for a set of results.
- Not secure, since all the arguments are visible in the URL query
- Request object contains all the data in the html message body.
- Post Method
- CRS
- **@app.route('/process', methods=['POST'])**
- You need to dir/data.that a route can accept post methods.
- Secure
- Post content is never cashed, saved in history, or bookmarkable, unlimited in size.
- Suitable for form data or any sensitive data
- Always redirect rather than returning a template
- Otherwise every time they refresh the page, they re-submit the post request
- Once you've handled a post request, you should really send the client on somewhere.