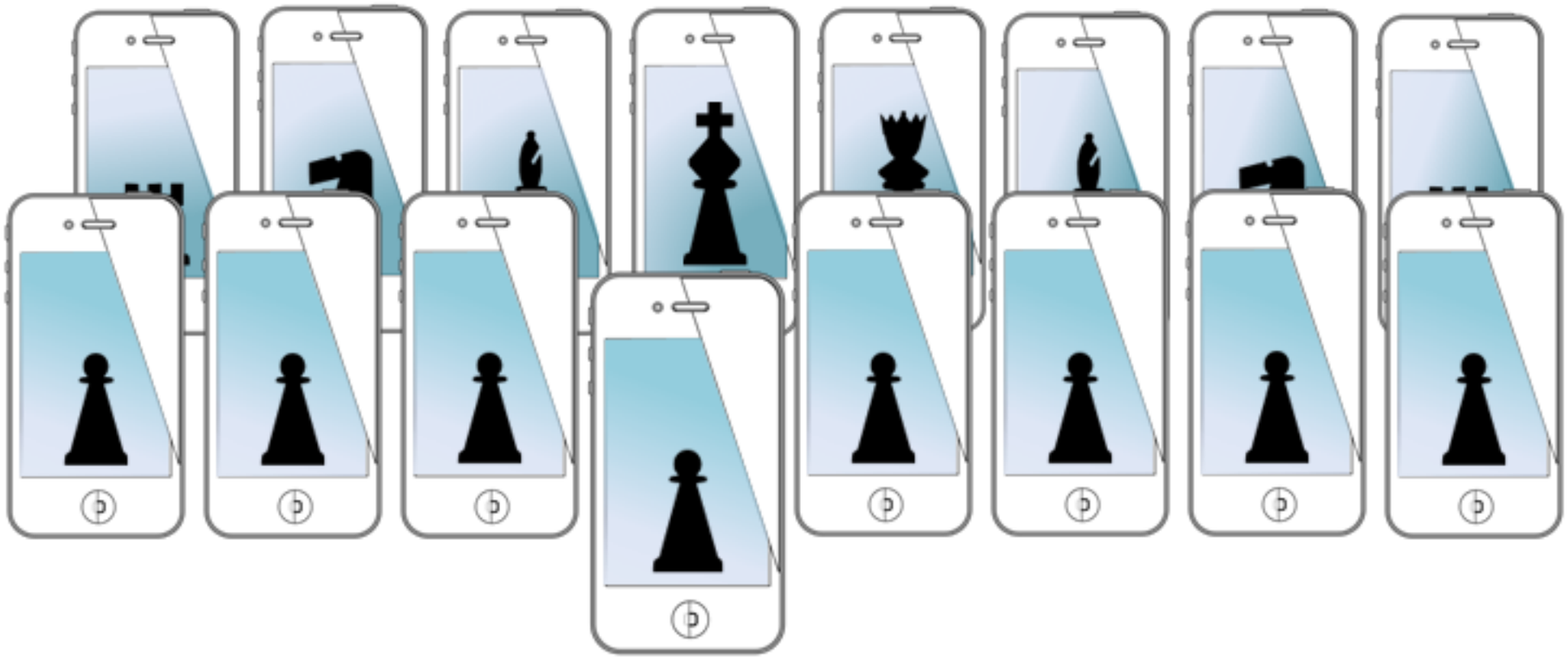
MOBILE SENSING LEARNING



CS5323 & 7323

Mobile Sensing and Learning

python crash-course, fastapi

Eric C. Larson, Lyle School of Engineering,
Computer Science, Southern Methodist University

course logistics

- **lab four due**: images
- following that
  - **lab five due**: machine learning as a service
  - **final project proposal due**

agenda

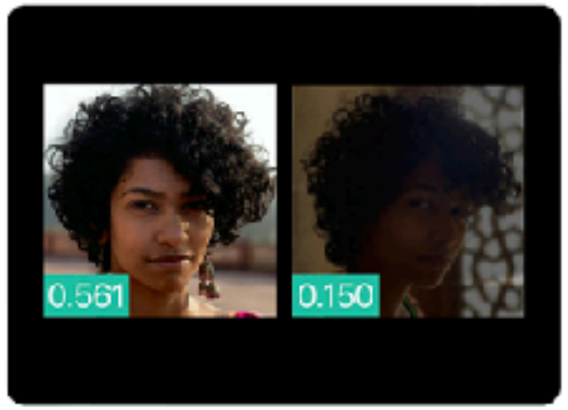
- last time: **AppleVision**
- image lab explanation
- history of python
- syntax
  - pythonic conventions
- document databases
- building our own REST API

Lab Four Explanation

Face Landmarks

Find facial features in images by detecting landmarks on faces.

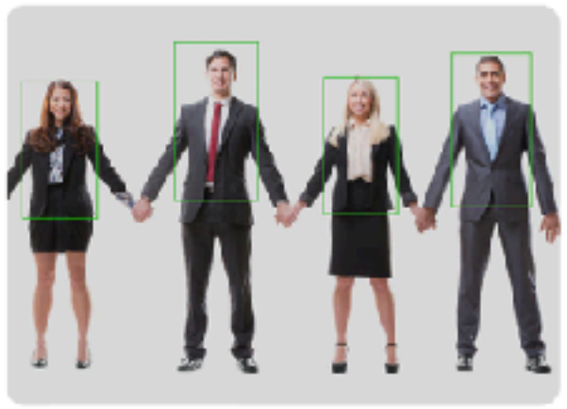
View API ›



Face Capture Quality

Compare face capture quality in a set of images.

View API ›



Human Body Detection

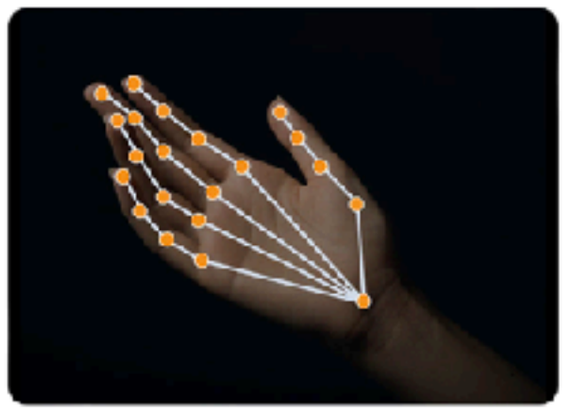
Find regions that contain human bodies in images.

View API ›

Body Pose

Detect landmarks on people in images and video.

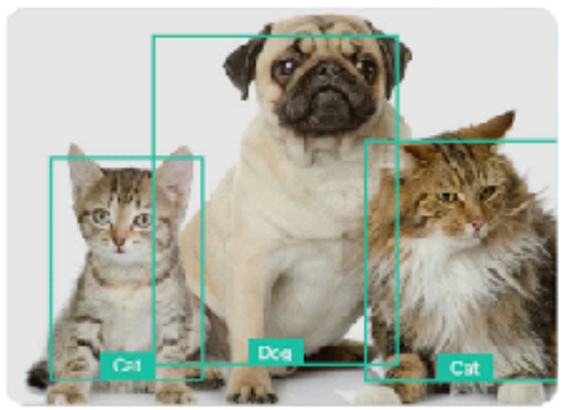
View API ›



Hand Pose

Detect landmarks on human hands in images and video.

View API ›



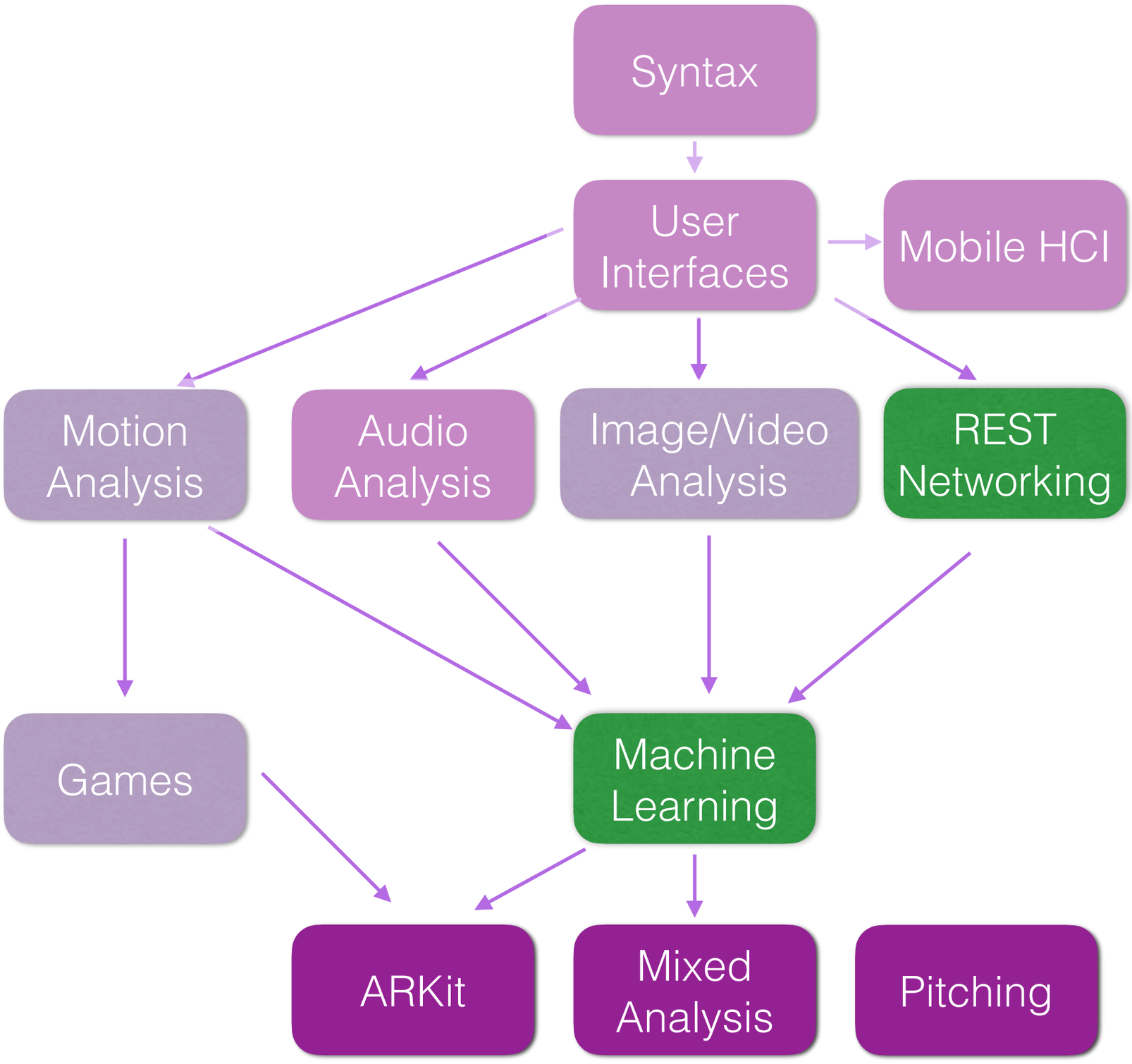
Animal Recognition

Find cats and dogs in images.

View API ›

https://developer.apple.com/machine-learning/api/

class overview



why are we learning python?

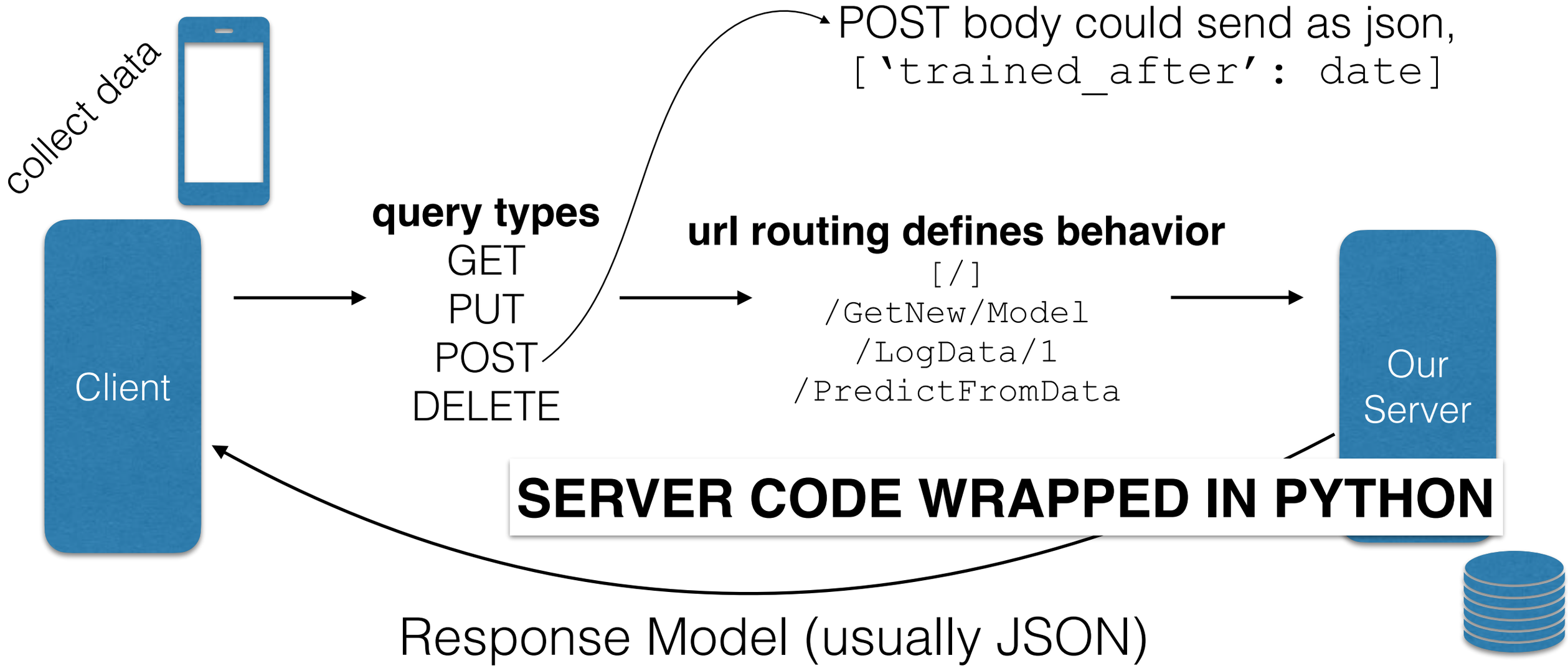
- its the glue for:
  - mongodb, fastapi for managing http requests
  - machine learning (blackbox approaches)
  - saving data for use in CreateML (via swift)

what are we doing?

- **preparing for lab 5**, need HTTP server that can:
  - accept (any) data
  - save it into a database
  - learn a (ML) model from stored data
  - mediate queries and predictions of the model
- **fastapi** is the event-driven architecture for interpreting the commands, routing the data, etc.
- our focus is building a deployment server, not an advanced ML algorithm (take DM or ML courses for that)

GET: Retrieves data from the server. No other effect.
PUT: Replaces target resource with the request payload. Update or create a new resource.
POST: Performs *resource-specific processing* on the payload. Can be used for different actions including creating a new resource, uploading a file, or training an ML model.
DELETE: Removes data from the server.

•Specifies a design for how we can interact remotely with a server to post and query data. REST does one thing at a time… We will abuse this a little in our implementation.



Response Model (usually JSON)

```
['model' : 'user_model', 'trained': date]
```

python

- Guido van Rossum

From wikipedia:

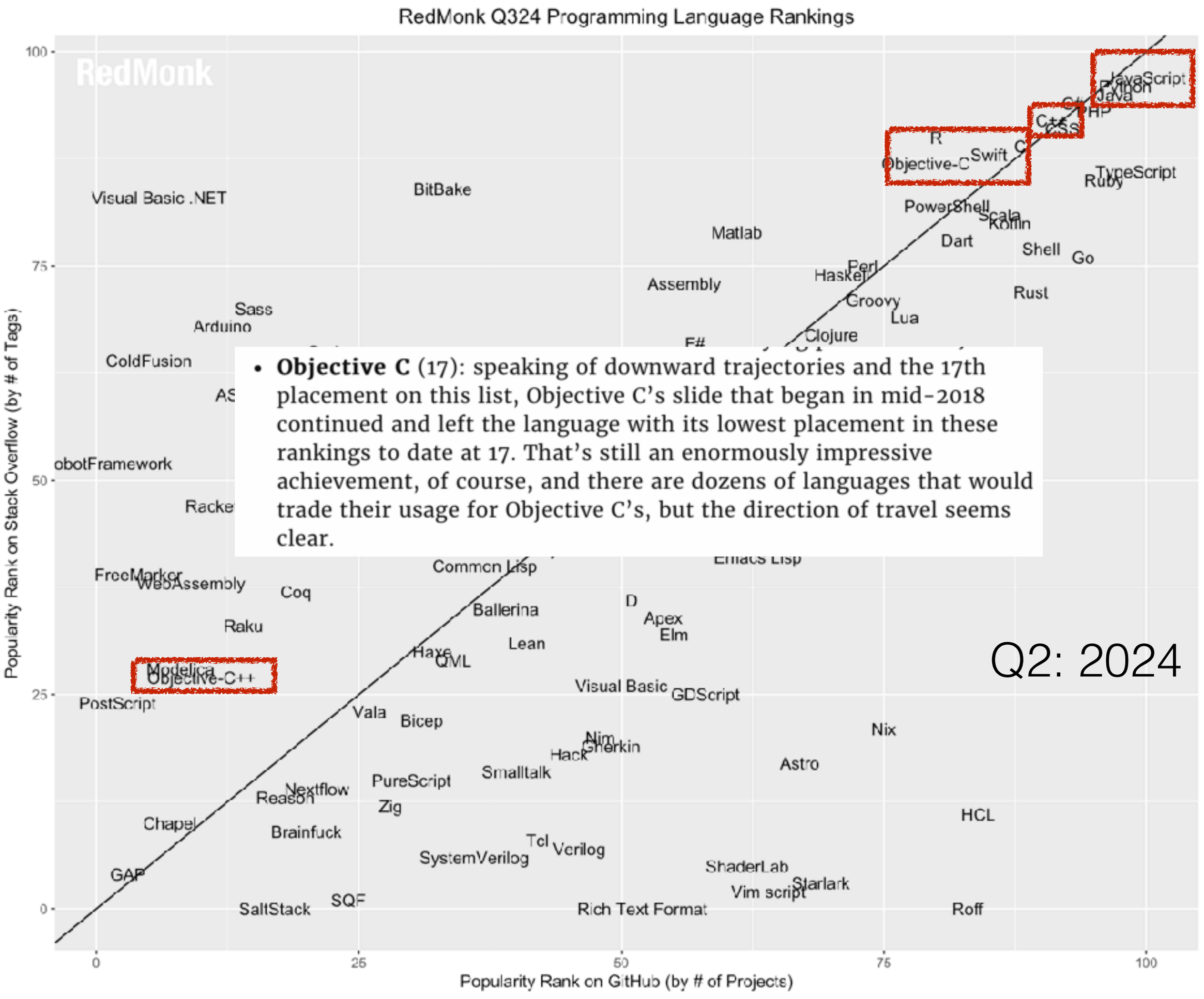
Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

-Guido van Rossum in 1996

python adoption

- appears in every programming top three list
- 2019: Tops every list, beating out Java and Javascript
  - IEEE Spectrum, ACM, Others
- 2022: Python and Javascript swap intermittently

Top Programming Languages 2022 ›
Python's still No. 1, but employers love to see SQL skills

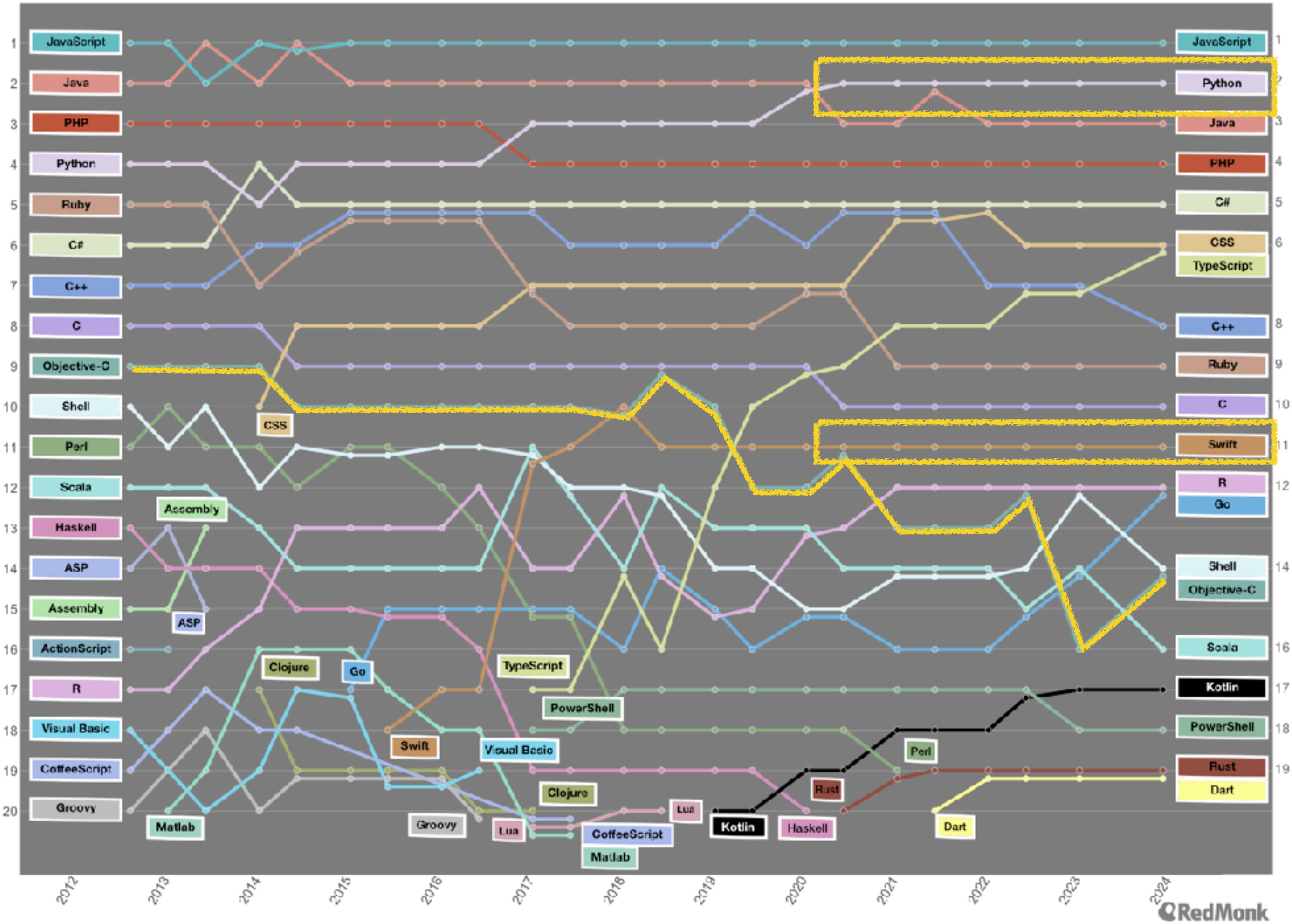


- **Objective C** (17): speaking of downward trajectories and the 17th placement on this list, Objective C's slide that began in mid-2018 continued and left the language with its lowest placement in these rankings to date at 17. That's still an enormously impressive achievement, of course, and there are dozens of languages that would trade their usage for Objective C's, but the direction of travel seems clear.

Q2: 2024

RedMonk Language Rankings

September 2012 - January 2024



installation

- install anaconda and use pip3
  - **use python 3.8** (for compatibility with later packages)
  - use **conda environments** or virtual environment
  - see the `InstallPythonEnvironment.txt` on the repository
    - there are some packages we will use that **require older versions** to work with CoreML like sklearn, Turi
- pick the IDE you want
  - Jupyter or sublime (not an IDE, but good for editing)
  - PyCharm, very good, supports breakpoints and watch
  - XCode can also be used, but is more limited

what to install for this class?

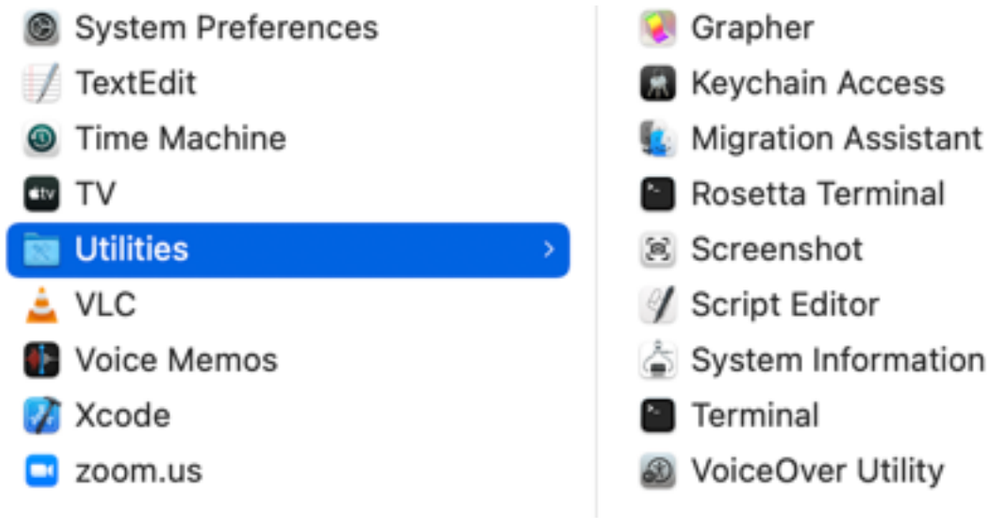
- look at installation packages list in tornado branch

From the Rosetta terminals:
CONDA_SUBDIR=osx-64 conda create "python38env" python=3.8

conda create -n "python38env" python=3.8

Instructions for all Macs:
Note that numpy must be an older version to be compatible with Turi ...

conda activate python38env
python3 -m pip install --upgrade pip
pip3 install numpy==1.23.1
pip3 install pandas
pip3 install matplotlib
pip3 install scikit-learn
pip3 install seaborn
pip3 install jupyter
pip3 install coremltools
pip3 install turicreate
pip3 install pymongo



going through syntax…

```
if reference_slide:
    print(“make me skip that slide”)
```

debugging

- the python debugger
  - http://docs.python.org/2/library/pdb.html
  - or use break points in pycharm
- `import pdb, pdb.set_trace()`
- command line arguments
  - `s`(tep), `c`(ontinue), `n`(ext), `w`(here), `l`(ist), `r`(eturn), `j`(ump)
  - and much more… like `print, p, pp`
  - can set numbered break points by running from python window
    - `python -m pdb your_function.py`

syntax, python 3

reference slide

- numbers
  - int or float
- complex numbers

```
>>> 7*5
35
>>> 5/7
0.7142857142
>>> 7/5
1.4
>>> 7.0/5
1.4
```

```
>>> tmpVar = 4
>>> print (tmpVar)
4
>>> tmpVar/8
0.5
>>> tmpVar/8.0
0.5
```

```
>>> 1+1j
(1+1j)
>>> (1+1j)*5
(5+5j)
>>> 1+1j + 4
(5+1j)
```

syntax

reference slide

- strings
  - immutable

```
>>> 'single quotes'
'single quotes'
>>> "double quotes"
'double quotes'
>>> 'here is "double quotes"'
'here is "double quotes"'
>>> 'here is \'single quotes\''
"here is 'single quotes'"
>>> "here are also \"double quotes\""
'here are also "double quotes"'
```

```
>>> someString = 'MobileSensingAndLearning'
>>> someString[:5]
'Mobil'
>>> someString[5:]
'eSensingAndLearning'
>>> someString+'AndControl'
'MobileSensingAndLearningAndControl'
>>> someString*3
'MobileSensingAndLearningMobileSensingAndLearningMobileSensingAndLearning'
>>> someString[-5:]
'rning'
>>> someString[:-5]
'MobileSensingAndLea'
>>> someString[5]
'e'
>>> someString[-1]
'g'
>>> someString[-2]
'n'
```

```
>>> someString[5] = 'r'

Traceback (most recent call last):
    File "<pyshell#32>", line 1, in <module>
        someString[5] = 'r'
TypeError: 'str' object does not support item assignment
```

syntax

reference slide

- tuples
- lists
  - highly versatile and mutable
  - containers for more abstract data and objects

```
>>> aTuple = 45, 67, "not a number"
>>> aTuple
(45, 67, 'not a number')
```

immutable

```
>>> aList = ["a string",5.0,6,[4,3,2]]
>>> print(aList)
['a string', 5.0, 6, [4, 3, 2]]
>>> aList[0]
'a string'
>>> aList[2]
6
>>> aList[-1]
[4, 3, 2]
```

```
>>> anotherList = []
>>> i=0
>>> i+=1
>>> i
1
>>> while i<1000:
   anotherList.append(i)
   i+=i

>>> print anotherList
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```
>>> len(aList)
4
>>> len(aList[-1])
3
>>> aList[0:1]=[]
>>> print(aList)
[5.0, 6, [4, 3, 2]]
>>> aList[0:2]=[]
>>> print(aList)
[[4, 3, 2]]
```

syntax loops

- for, while
  - indentation ~~matters~~ *is the only thing* that **matters**

```
i=0
while i<10:
    print (str(i) + ' is less than 10')
    i+=1
else:
    print (str(i) + ' is not less than 10')
```

```
0 is less than 10
1 is less than 10
2 is less than 10
3 is less than 10
4 is less than 10
5 is less than 10
6 is less than 10
7 is less than 10
8 is less than 10
9 is less than 10
10 is not less than 10
```

```
classTeams = ['Team','Monkey','CHC',
              'ThatGuyInTheBack',42]

for team in classTeams:
    print (team * 4)
else:
    print ('ended for loop without break')
```

```
TeamTeamTeamTeam
MonkeyMonkeyMonkeyMonkey
CHCCHCCHCCHC
ThatGuyInTheBackThatGuyInTheBackThatGuyInTheBackThatGu
168
ended for loop without break
```

Aug 28, 2018 — Even Python's retired creator **Guido van Rossum** has stated that he would not include **loop-else** in Python if he had to do it over.

syntax loops

reference slide

- for, while
  - indentation ~~matters~~ *is the only thing* that **matters**

```
for i in range(10):
    print (i)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
for j in range(2,10,2):
   print (j)
```

```
2
4
6
8
```

data structures

- **lists** can be used as a stack

```
>>> classTeams = ['Team','Monkey','CHC','ThatGuyInTheBack',42]
>>> classTeams.pop()
42
>>> classTeams.pop()
'ThatGuyInTheBack'
>>> classTeams.sort()
>>> classTeams
['CHC', 'Monkey', 'Team']
```

- or import **queues**

```
>>> from queue import Queue
>>> q = Queue(maxsize=20)
>>> q.put('Team'), q.put('Eric'), q.put('Rocks')
>>> q.get()
'Team'
```

- **dictionaries**

```
>>> myDictionary = {"teamA":45,"teamB":77}
>>> myDictionary
{'teamA': 45, 'teamB': 77}
>>> myDictionary["teamA"]
45
```

lists and loops

- comprehensions

```
>>> timesFour = [x*x*x*x for x in range(10)]
>>> timesFour
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]
```

```
from random import randint
grades = ['A','B','C','D','F']
teamgrades = [grades[randint(0,4)] for t in range(8)]
print (teamgrades)
```

```
['C', 'A', 'B', 'F', 'A', 'C', 'A', 'D']
```

can be nested as much as you like!

only **pythonic** if it makes the code **more readable**

```
>>> timesFour = {x:x*x*x*x for x in range(10)}
>>> timesFour
{0: 0, 1: 1, 2: 16, 3: 81, 4: 256, 5: 625, 6: 1296, 7: 2401, 8: 4096, 9: 6561}
```

can use comprehensions with dictionaries too!

lists and loops

reference slide

```
>>> timesFour = {x:x*x*x*x for x in range(10)}
>>> timesFour
{0: 0, 1: 1, 2: 16, 3: 81, 4: 256, 5: 625, 6: 1296, 7: 2401, 8: 4096, 9: 6561}
```

can use comprehensions with dictionaries too!

```
from random import randint

teams = ['CHC','Team','DoerrKing','MCVW','etc.']
grades = ['A','B','C','D','F']
teamgrades = {team:grades[randint(0,4)] for team in teams}
teamgrades

{'etc.': 'F', 'CHC': 'A', 'DoerrKing': 'B', 'MCVW': 'B', 'Team': 'A'}
```

pop quiz!

- add the numbers from 0 to 100, not including 100

```
sumValue = 0
for i in range(100):
    sumValue += i

print (sumValue)
print (sum(range(100)))
print (100*(100-1)/2)
```

more pythonic?

or use real math

now, print the **index** and **value** of elements in a list

```
list = [1,2,4,7,1,5,6,8]

for i in range(len(list)):
    print(str(list[i]) + " is at index " + str(i))

for i,element in enumerate(list):
    print(f"{element} is at index {i}")
```

```
1 is at index 0
2 is at index 1
4 is at index 2
7 is at index 3
1 is at index 4
5 is at index 5
6 is at index 6
8 is at index 7
```

more pythonic

conditionals

self test slide

- if, elif, else, None, is, or, and, not, ==

```
a=5
b=5

if a==b:
    print (“Everybody is a five!”)
else:
    print (“Wish we had fives…”)


a=327676
b=a

if a is b:
    print (“These are the same object!”)
else:
    print (“Wish we had the same objects…")

a=327676
b=327675+1

if a is b:
    print (“These are the same object!”)
else:
    print (“Wish we had the same objects…")

a=5
b=4+1

if a is b:
    print (“Everybody is a five!”)
else:
    print (“Wish we had fives…")
```

```
Everybody is a five!
```

```
These are the same object!
```

```
Wish we had the same objects
```

small integers are cached
strings behave the same

```
Everybody is a five!
```

conditionals

reference slide

```
teacher = "eric"

if teacher is not "Eric":
    print (“Go get the prof for this class!”)
else:
    print (“Welcome, Professor!”)
```

```
Go get the prof …
```

```
teachers = ["Eric","Paul","Ringo","John"]

if "Eric" not in teachers:
    print (“Go get the prof for this class!”)
else:
    print (“Welcome, Professor!”)
```

```
Welcome!
```

```
teachers = ["Eric","Paul","Ringo","John"]
shouldCheckForTeacher = True

if "Eric" not in teachers and shouldCheckForTeacher:
    print (“Go get the prof for this class!”)
elif shouldCheckForTeacher:
    print (“Welcome, Professor!”)
else:
    print (“Not checking”)
```

```
Welcome!
```

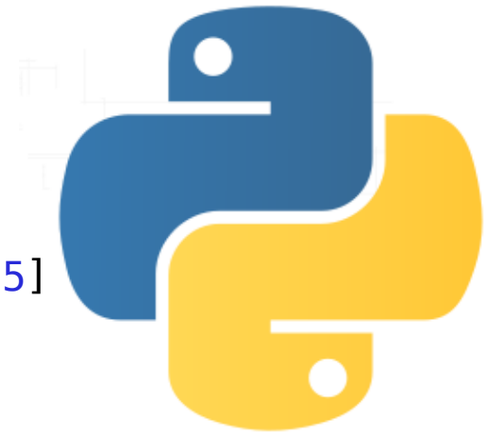
functions

reference slide

- def keyword
  - like c, must be defined before use

```
def show_data(data):
    # print the data
    print (data)

some_data = [1,2,3,4,5]
show_data(some_data);
```

```
[1, 2, 3, 4, 5]
```



```
def show_data(data,x=None,y=None):
    # print the data
    print data
    if x is not None:
        print (x)
    if y is not None:
        print (y)

some_data = [1,2,3,4,5]
show_data(some_data);
show_data(some_data,x='a cool X value')
show_data(some_data,y='a cool Y value',x='a cool X value')

def get_square_and_tenth_power(x):
    return x**2,x**10

print (get_square_and_tenth_power(2))
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
a cool X value
[1, 2, 3, 4, 5]
a cool X value
a cool Y value

(4, 1024)
```

python types, inputs, optionals

```python
from typing import Optional

def greet(name: Optional[str] = None) -> str:
    if name:
        return f"Hello, {name}!"
    else:
        return "Hello, Mobile Sensing!"

print(greet("Alice"))
print(greet())
```

```
Hello, Alice!
Hello, Mobile Sensing!
```

named, typed input arguments

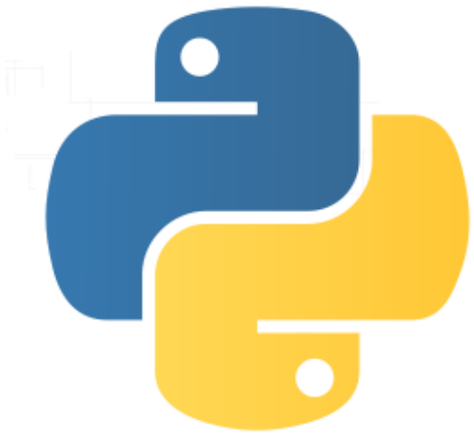
specified return type

f-strings

Over time, python and swift syntax has become more and more similar

python decorators

- wrap your method inside another method
- the wrapper changes some functionality



```
from time import sleep

def sleep_decorator(function):
    def wrapper(*args, **kwargs):
        sleep(2)
        return function(*args, **kwargs)
    return wrapper

@sleep_decorator
def print_number(num):
    return num

print(print_number(222))

for num in range(1, 6):
    print(print_number(num))
```

Can also have arguments and decorate from instance

```
app = SomeSpecialClass()

@app.get("/change_behavior/")
def func_to_wrap():
    …
```

python syntax “with”

reference slide

- the “with” statement
- defines an “enter” and an “exit” protocol
- used commonly for opening files, where “open” adopts the “with” protocol

```
file = open("/some_file.txt")
try:
    data = file.read()
finally:
    file.close()



with open("/some_file.txt") as file:
    data = file.read()
```

classes

reference slide

- multiple inheritance
- “self” is always passed as first argument

```
class BodyPart(object):
    def __init__(self,name):
        self.name = name;

class Heart(BodyPart):
    def __init__(self,rate=60,units="minute"):
        self.rate = rate
        self.units= units
        super().__init__("Heart")

    def __str__(self):
        print (“name:" + str(self.name) + " has " + str(self.rate) + " beats per " + self.units)

    @static
    def utility(x):
        return x**2

myHeart = Heart(1,"second")
print(myHeart)
```

utility decorator

pydantic data stores

- framework for working with data, with types

```
from pydantic import ConfigDict, BaseModel, Field
from typing import Optional, List



class LabeledDataPoint(BaseModel):
    # Container for a single labeled data point.

    feature: List[float] = Field(...) # feature data as array
    label: str = Field(...) # label for this data
    dsid: int = Field(..., le=50) # dataset id, for tracking


class LabeledDataPointCollection(BaseModel):
    # A container holding a list of instances.
    datapoints: List[LabeledDataPoint]
```

`Field` is generic metadata for a data entry, can also set default values and aliases

python generators

- kinda like static variables
- used to create iterables
- lots more that you can do, like send in values

```
def get_primes(number):
    while True:
        if is_prime(number):
            yield number
        number += 1

total = 2
for next_prime in get_primes(3):
    if next_prime < 2000000:
        total += next_prime
    else:
        break
```

yield allows iteration

https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/

python async/await

- introduced in python 3.5: awaitable objects

co-routine:
awaitable methods

```python
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happ                    ".
    # A coroutine object is created but not awaited,
    # so it *won't
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

nothing happens

await output

tasks:
awaitable objects

```python
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now
    # can simply be aw                    te:
    await task

asyncio.run(main())
```

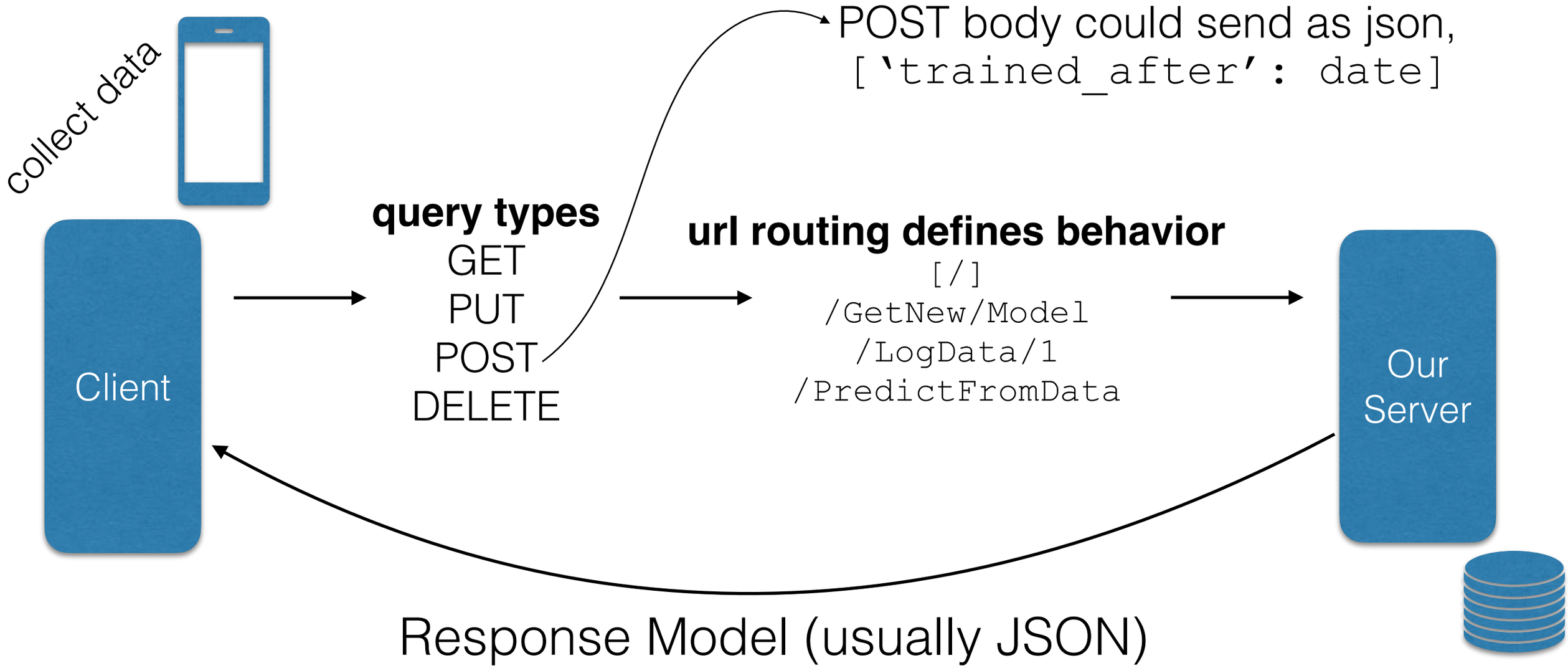
await tasks

How we will use it

```python
async def show_max_dsid():
    """
    Get the maximum dsid currently used
    """

    if (
        datapoint := await app.collection.find_one(sort=[("dsid", -1)])
    ) is not None:
        return {"dsid":datapoint["dsid"]}

    raise HTTPException(status_code=404, detail=f"No datasets currently created.")
```

GET: Retrieves data from the server. No other effect.
PUT: Replaces target resource with the request payload. Update or create a new resource.
POST: Performs *resource-specific processing* on the payload. Can be used for different actions including creating a new resource, uploading a file, or training an ML model.
DELETE: Removes data from the server.

•Specifies a design for how we can interact remotely with a server to post and query data. REST does one thing at a time… We will abuse this a little in our implementation.



Response Model (usually JSON)

```
['model' : 'user_model', 'trained': date]
```

fastAPI

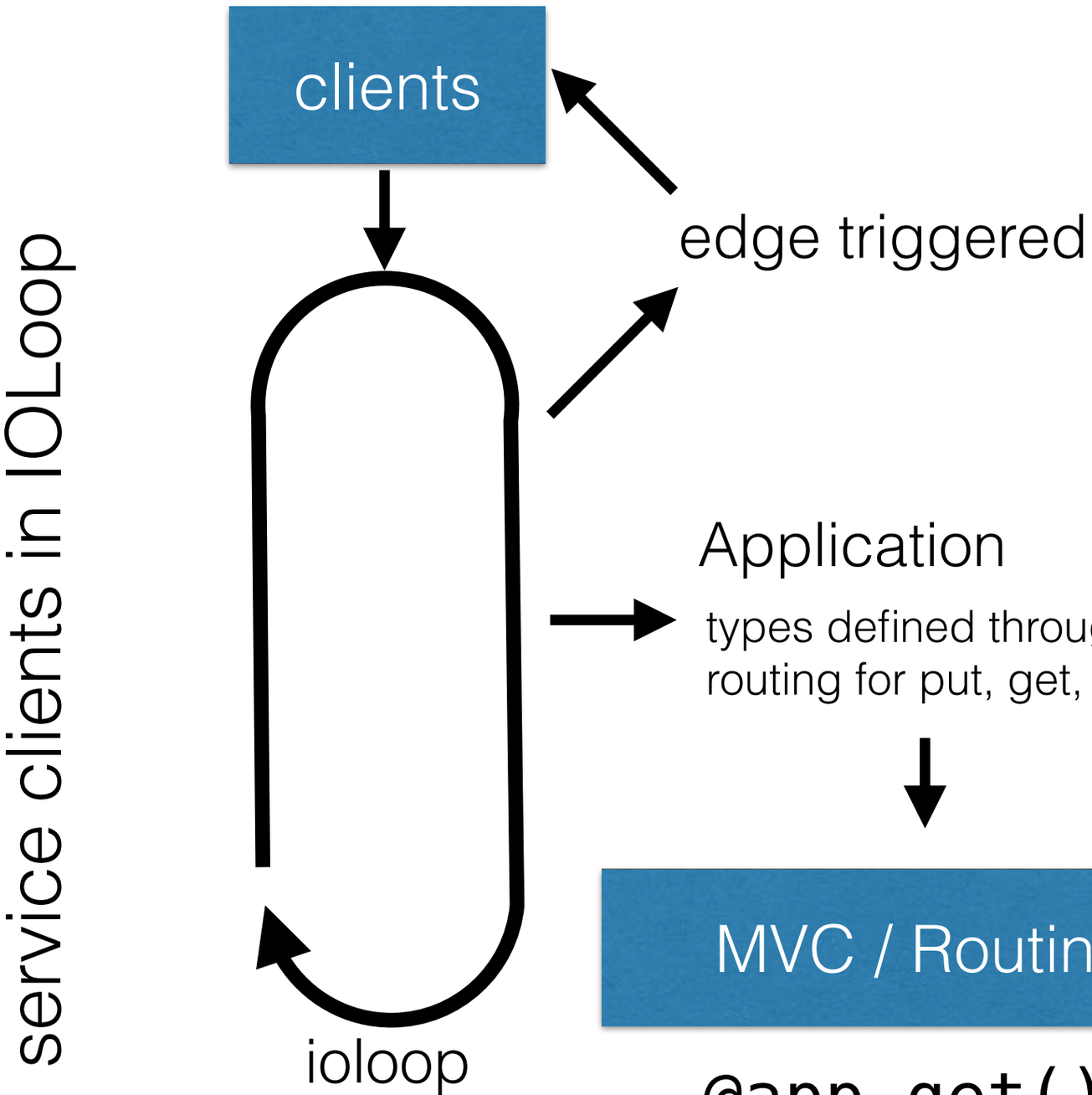
- previous classes used “tornado” but this has now been surpassed in speed and usability by fastAPI
- first released in 2018.
- uses Pydantic and type hints to validate, serialize and deserialize data.
- automatically generates OpenAPI documentation

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return "Hello World!"
```

generators control route and function

https://en.wikipedia.org/wiki/FastAPI

fastAPI



all custom actions should be asynchronous (for performance)

mongodb

- hu**mongo**us data
- NoSQL database (vs relational database)
  - its a document database
- everything stored as a document
  - more or less json
  - key: value/array
- schema is dynamic
  - the key advantage of NoSQL

mongodb install

Instructions also in Repository
`InstallMongoDB.txt`

- install it
  - `brew tap mongodb/brew`
  - `brew update`
  - `brew install mongodb-community@6.0`
- you can also **run as a service** (`./mongo`)
  - `brew services start mongodb-community@6.0`
  - its running! localhost
  - `brew services stop mongodb-community@6.0`

Mongo Clients

Interface to outside world
listen on large port number

Organizational Structure in MongoDB

mongodb

- a document, as stated by mongodb

Document Database

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

```
{
   name:  "sue",                          ← field: value
   age: 26,                               ← field: value
   status: "A",                           ← field: value
   groups: [ "news", "sports" ]           ← field: value
}
```

A MongoDB document.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

docs and collections

Database: MSLC_creations

default limit on size of each document: ***16MB***

apps_collection

```
{
  app: "mongoApp",
  users: 100005,
}

{
  app: “StepCount",
  users: 45,
  rating: 2.6,
}
....

{
  app: “Trench",
  users: 4050000,
  rating: 5,
}
```

teams_collection

```
{
  team: "mongo",
  members: [ "Eric", “Ringo”, “Paul” ],
  numApps: 21,
  website: “teammongo.org",
}
{
  team: “ran off",
  members: [ "John", “Yoko” ],
  website: “flewthecoop.org",
}
....
{
  team: “21 Pilots",
  members: [ "Tyler", “Nick” ],
  numApps: 4,
  website: “RollingStone.com”,
}
```

Mongo Client

Database

Document Collection

pymongo

- python wrapper for using mongo db

```
client  = MongoClient() # localhost, default port
db = client.some_database # access database
```

create this database, if it does not exist

```
collect = client.some_database.some_collection # access a collection
```

Mongo Client

Database

Document Collection

nothing is **created** until the first insert!!!

```
db.collection_names()
[u'system.indexes', u'some_collection']
```

get collections

pymongo (add data)

- insertion

```
dbid = db.some_collect.insert_one(
    {“key1”:values,"key2":more_values,
      “coolkey”:with_cool_values}
)
```

unique key, _id

doc to insert

- update

```
db.some_collect.update(  {“thiskey":keyValue},
    {  "$set": {"keyToSet":valueToSet}  },
    upsert=True)
```

where ever this key is…

equal to this

set

this key to this value

insert if it does not exist (put/post)

pymongo (get data)

- find one datum in database

```
a = db.some_collect.find_one(sort=[("sortOnThisKey", -1)])
newData = float( a['sortOnThisKey'] );
```

could be list of keys!

access the result

sort with this key

return last element

- iterate through many results

```
f=[];
for doc in db.some_collect.find({"keyIWant":valueOfKeyIWant}):
    doc['key1'] # entire document, is available
    f.append( str(doc['keyToGrabDataWith']) )
```

return iterator to loop over

each iteration gives one document

- lots of advanced queries are possible

https://api.mongodb.org/python/current/

teams example

```
>>> from pymongo import MongoClient
>>> client = MongoClient()

>>> db = client.some_database
>>> collect1 = db.some_collection
>>> collect1.insert_one({“team”:"TeamFit","members":["Matt","Mark","Rita","Gavin"]})
ObjectId('53396a80291ebb9a796a8af1')

>>> db.collection_names()
[u'system.indexes', u'some_collection']

>>> db.some_collection.find_one()
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'],
u'team': u'TeamFit'}

>>> collect1.insert_one({“team":"Underscore","members":["Carly","Lauryn","Cameron"]})
ObjectId('53396c80291ebb9a796a8af2')

>>> db.some_collection.find_one()
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'],
u'team': u'TeamFit'}

>>> db.some_collection.find_one({"team":"Underscore"})
{u'_id': ObjectId('53396c80291ebb9a796a8af2'), u'members': [u'Carly', u'Lauryn', u'Cameron'],
u'team': u'Underscore'}
```

bulk operations

```
from pymongo import MongoClient

client = MongoClient()
db=client.some_database
collect1 = db.some_collection

insert_list = [{"team":"MCVW","members":["Matt","Rowdy","Jason"]},
               {“team":"CHC", "members":["Hunter","Chelsea","Conner"]}]

obj_ids=collect1.insert_many(insert_list)
```

anything iterable

```
for document in collect1.find({"members":"Matt"}):
  print(document)
```

```
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'], u'team': u'TeamFit'}
{u'_id': ObjectId('53397331291ebb9afdd3cd2f'), u'members': [u'Matt', u'Rowdy', u'Jason'], u'team': u'MCVW'}
```

```
document = collect1.find_one({"members":"Matt","team":"MCVW"})
print (document)
```

```
{u'_id': ObjectId('53397331291ebb9afdd3cd2f'), u'members': [u'Matt', u'Rowdy', u'Jason'], u'team': u'MCVW'}
```

async mongodb (+ tornado)

- we will use pymongo and fastapi
  - mongodb runs localhost, fastapi mediates access
  - and asynchronous calls (decorators or async/await)

```
# Motor imports
from bson import ObjectId
import motor.motor_asyncio

app.mongo_client = motor.motor_asyncio.AsyncIOMotorClient()

# remember the db collection, named “database” and “mycollection"
app.collection = app.mongo_client.mydatabase.get_collection(“mycollection”)
```

```
new_label = await app.collection.insert_one(
        datapoint
    )
```

https://motor.readthedocs.io/en/stable/tutorial-tornado.html

what we have

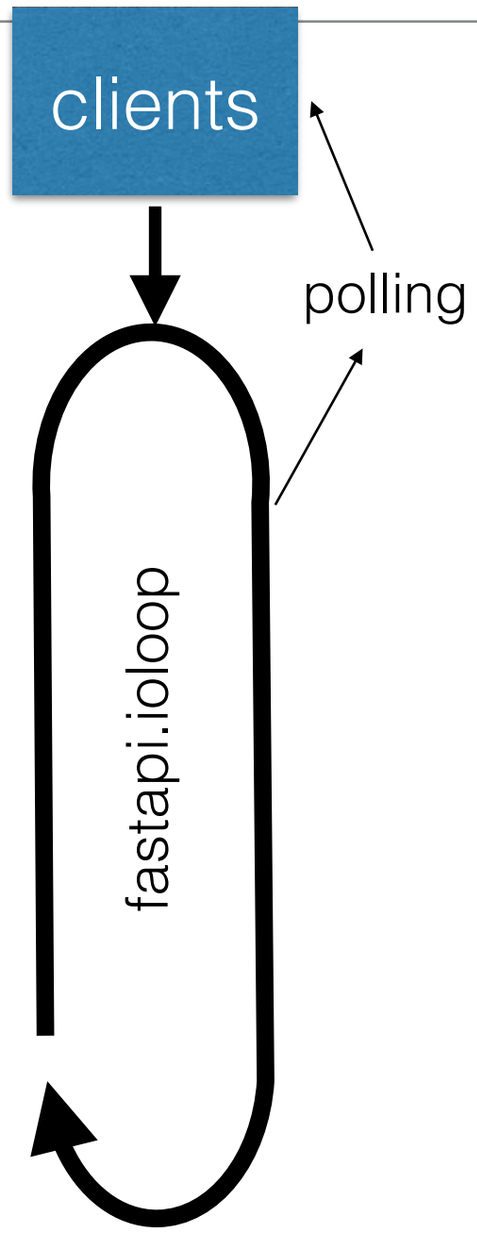
- good working knowledge of python syntax
- an asynchronous database for storing info

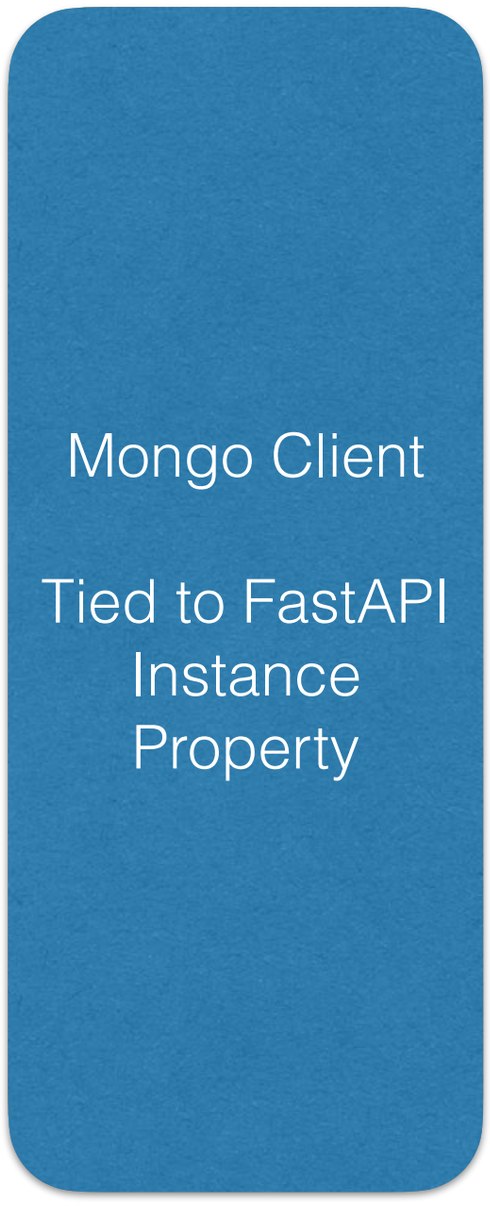
what we need

- we just need a server interface!

```
fastapi run fastapi_example.py
```

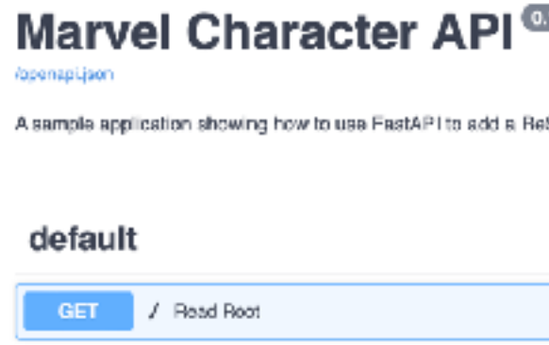
1. `brew services start mongodb-community@6.0`





2. run
fastapi.app()

`fastapi run fastapi_example.py`



3. test local queries
in a browser with docs
(if you want)

ifconfig | grep "inet "

4. make queries
from external sources
via external facing IP

fastapi workflow

```
# Motor API allows us to directly interact with a
# In this example, we assume that there is a singl
# First let's get access to the Mongo client that
client = motor.motor_asyncio.AsyncIOMotorClient()

# new we need to create a database and a collectio
# collection if they haven't been created yet. The
db = client.mcu
character_collection = db.get_collection("characte
```

- setup “app”
- setup database client and store in app (or global…)
- setup pydantic data stores (and sync id with database client)
- setup get, post, put, delete requests
  - url for particular request
  - what response model to use (from pydantic)
  - status code to send
  - input format to function

```
class CharacterModel(BaseModel):
    """
    Container for a single marvel ch
    """

    # The primary key for the Charac
    # This will be aliased to `_id`
    # but provided as `id` in the AP
    id: Optional[PyObjectId] = Field
    name: str = Field(...) # our cha
    power: str = Field(...) # a supe
    level: int = Field(..., le=5) #
    kind: str = Field(...) # Enum fo
```

```
# Create the FastAPI app
app = FastAPI(
    title="Marvel Characte
    summary="A sample app
)
```

```
@app.post(
    "/characters/",
    response_description="Add new character",
    response_model=CharacterModel,
    status_code=status.HTTP_201_CREATED,
    response_model_by_alias=False,
)
async def create_character(character: CharacterModel = Body(...)):
```

fastapi workflow

```python
# Create the FastAPI app
app = FastAPI(
    title="Marvel Character API",
    summary="A sample application showing
)
```

fastapi obiect

```python
class CharacterModel(BaseModel):
    """
    Container for a single marvel character reco
    """

    # The primary key for the CharacterModel, st
    # This will be aliased to `_id` when sent to
    # but provided as `id` in the API requests a
    id: Optional[PyObjectId] = Field(alias="_id"
    name: str = Field(...) # our character's nam
    power: str = Field(...) # a super power, if
    level: int = Field(..., le=5) # class of cha
    kind: str = Field(...) # Enum for good, bad,
```

data format

```python
# Motor API allows us to directly interact with a host
# In this example, we assume that there is a single cl
# First let's get access to the Mongo client that allo
client = motor.motor_asyncio.AsyncIOMotorClient()

# new we need to create a database and a collection. T
# collection if they haven't been created yet. They ar
db = client.mcu
character_collection = db.get_collection("character")
```

async MongoDB

document collection

```python
@app.post(
    "/characters/",
    response_description="Add new character",
    response_model=CharacterModel,
    status_code=status.HTTP_201_CREATED,
    response_model_by_alias=False,
)
async def create_character(character: CharacterModel = Body(...)
    """
    Insert a new character record.
    er by that name already exi
    r to the connected client

    A uniqu  `id` will be created and provided in the
    """

    new_character = await character_collection.find_one_and_update(
        {"name":character.name},
        {"$set":character.model_dump(by_alias=True, exclude=["id"])},
        upsert=True, # insert if nothing found.
        return_document=ReturnDocument.AFTER)

    return new_character
```

post routing from “/“

expected output format

expected input format

async find and update
fastapi poll while mongo is running

upsert = True

serialize pydantic as dictionary

return response as json, using pydantic model

motor + fastapi

ifconfig | grep "inet "

- demo:
  - store data inside mongodb for different characters