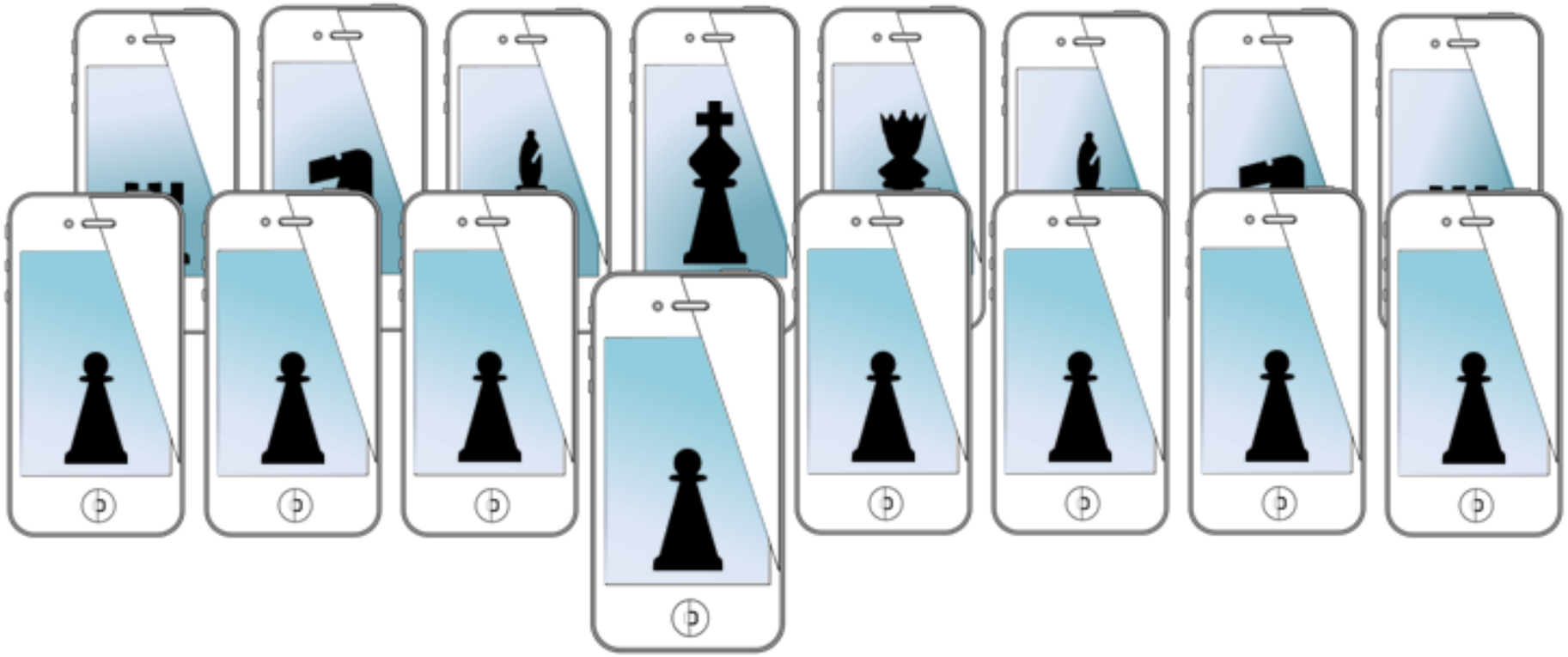
MOBILE SENSING LEARNING



CS5323 & 7323

Mobile Sensing and Learning

python crash-course, tornado

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

course logistics

- lab three was due this week
  - motion and game
- end of next week
  - lab four due: images
- two weeks following that
  - lab five due: machine learning as a service
  - final project proposal due

agenda

- last time: OpenCV
- history of python
- syntax
  - pythonic conventions
  - simple examples
- web handling with tornado
- document databases

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

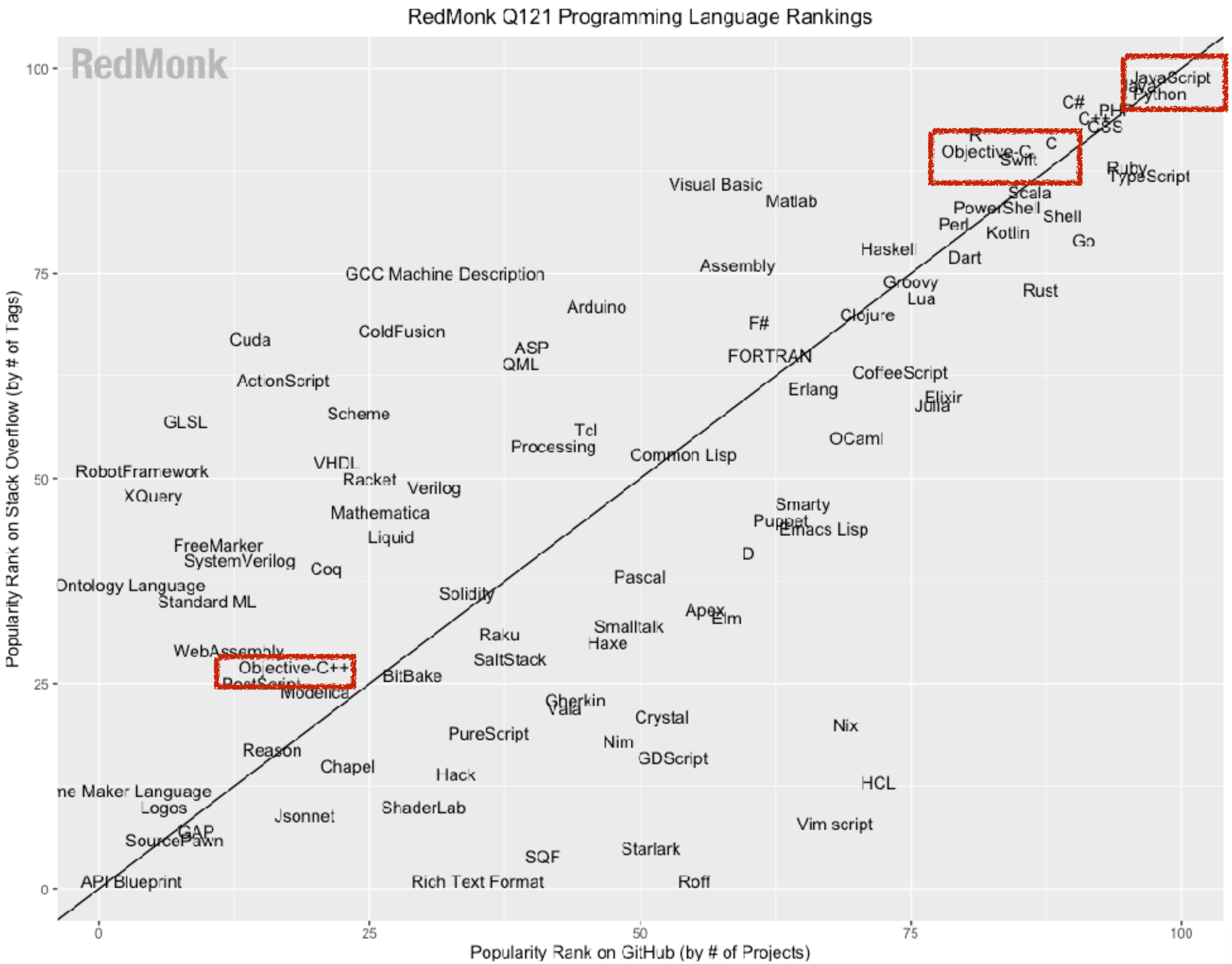
python adoption

- appears in every programming top ten list
- 2019: Tops every list, beating out Java and Javascript
  - IEEE Spectrum
  - ACM
  - Others

It's Python's world; we all just live in it.

Top programming languages in 2019: Python sweeps the board

September 9, 2019 Sarah Schlothauer


RedMonk Q121 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
100
JavaScript
Python
Java
C#
PHP
C++
CSS
R
Objective-C
C
Swift
Ruby
TypeScript
Visual Basic
Matlab
Scala
PowerShell
Shell
Perl
Kotlin
Go
Haskell
Dart
Assembly
GCC Machine Description
Groovy
Lua
Rust
Arduino
Clojure
F#
Cuda
ColdFusion
ASP
QML
FORTRAN
CoffeeScript
ActionScript
Erlang
Elixir
Julia
Scheme
GLSL
Tcl
Processing
OCaml
Common Lisp
VHDL
RobotFramework
Racket
Verilog
XQuery
Smarty
Mathematica
Puppet
Emacs Lisp
Liquid
FreeMarker
D
SystemVerilog
Coq
Pascal
Ontology Language
Standard ML
Solidity
Apex
Elm
Smalltalk
Raku
Haxe
WebAssembly
SaltStack
Objective-C++
BitBake
PostScript
Modelica
Gherkin
Vala
Crystal
Nix
PureScript
Nim
Reason
GDScript
Chapel
Hack
HCL
ne Maker Language
Logos
ShaderLab
Jsonnet
Vim script
GAP
SourcePawn
Starlark
SQF
API Blueprint
Rich Text Format
Roff

python disclaimers

- weakly typed variables (dynamic)
- its an interpreter (kinda)
  - loops are slow
  - until they are not (compile it)
- can't use parallel instructions natively
- many syntax similarities to swift
- can be the glue for your different codebases

installation

- install anaconda
- use latest python 3
- use conda environments

- pick the IDE you want
- Jupyter (not an IDE, but good for editing)
- PyCharm, very good, supports breakpoints and watch
- XCode can also be used, but is more limited than pycharm

python

- many different coding “styles”
- “best” styles get the distinction of “pythonic”
  - ill formed definition
  - changes as the language matures
- pythonic code is:
  - simple and readable
  - uses dynamic typing when possible
- …or to quote Tim Peters…

python zen

type this

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality be
Errors should never pass
Unless explicitly silenc
In the face of ambiguity ss.
There should be one-- ar s way to do it.
Although that way may no ou're Dutch.
Now is better than never.
Although never is often better tha right now.
If the implementation is hard to e plain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

get this

python is quirky
but, don't assume that means
it is not a **serious** tool

syntax, python 3

- 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

- 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

- tuples
- lists
  - highly versatile and mutable
  - containers for anything

```
>>> 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
```

syntax loops

- 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 can import queues
  - append(value)
  - pop_left(), dequeue first element
- **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 (str(element) + " is at index " + str(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

reference 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…")
```

```
Everybody is a five!
```

```
a=327676
b=a

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

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

```
a=327676
b=327675+1

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

```
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…")
```

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

- 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')
```

```
[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
```

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

print (get_square_and_tenth_power(2))
```

```
(4, 1024)
```

debugging

- the python debugger
  - http://docs.python.org/2/library/pdb.html
  - if you have not used it, I just changed your life
- `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`

python demos

- more demos:

http://sandbox.mc.edu/~bennet/python/code/index.html?utm_source=twitterfeed&utm_medium=twitter

classes

- 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)


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

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 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()
```

python decorators

reference slide

- 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))
```

used a **bunch** in web applications before python 3.5

python async/await

- new in python 3.5: awaitable objects

```python
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

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

asyncio.run(main())
```

co-routine:
awaitable methods

```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 be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete
    await task

asyncio.run(main())
```

tasks:
awaitable objects

```python
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

futures
gathering awaitable routines

why are we learning python?

- its the glue for:
  - tornado
  - mongodb
  - http requests in iOS

what are we doing?

- **preparing for A5**, need HTTP server that can:
  - accept (any) data
  - save it into a database
  - learn a (ML) model from that database
  - mediate queries and training of the model
- tornado 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)

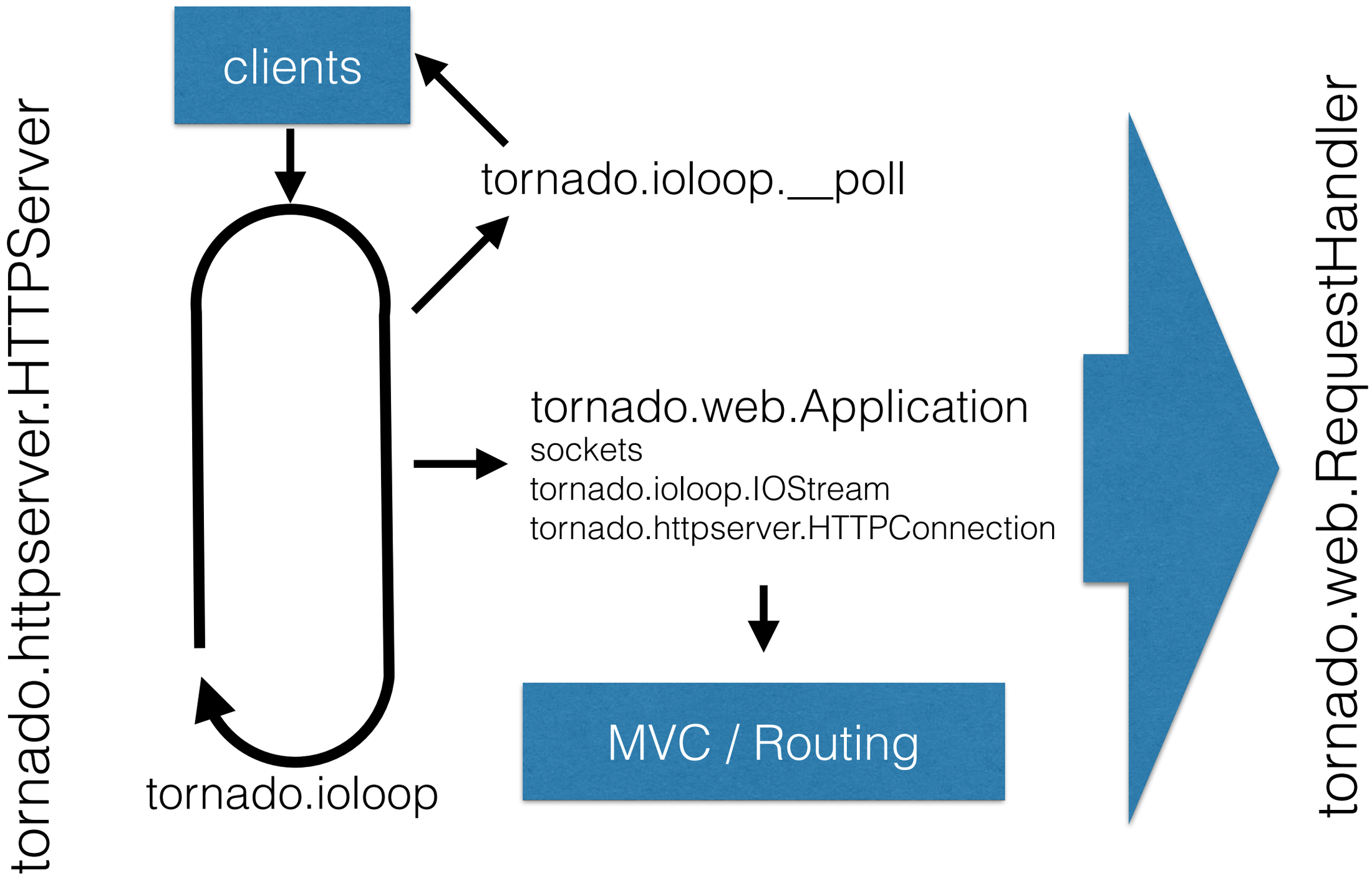
tornado web

- non-blocking web server
  - built for short-lived requests (pipelined)
  - and long lived connections
- built to scale
  - an attempt to solve the 10k concurrent problem
- has a python implementation
  - open sourced by Facebook after acquiring friendfeed.com
  - originally developed by the developers of gmail and google maps (the original releases)
- uses IOLoop and callback model

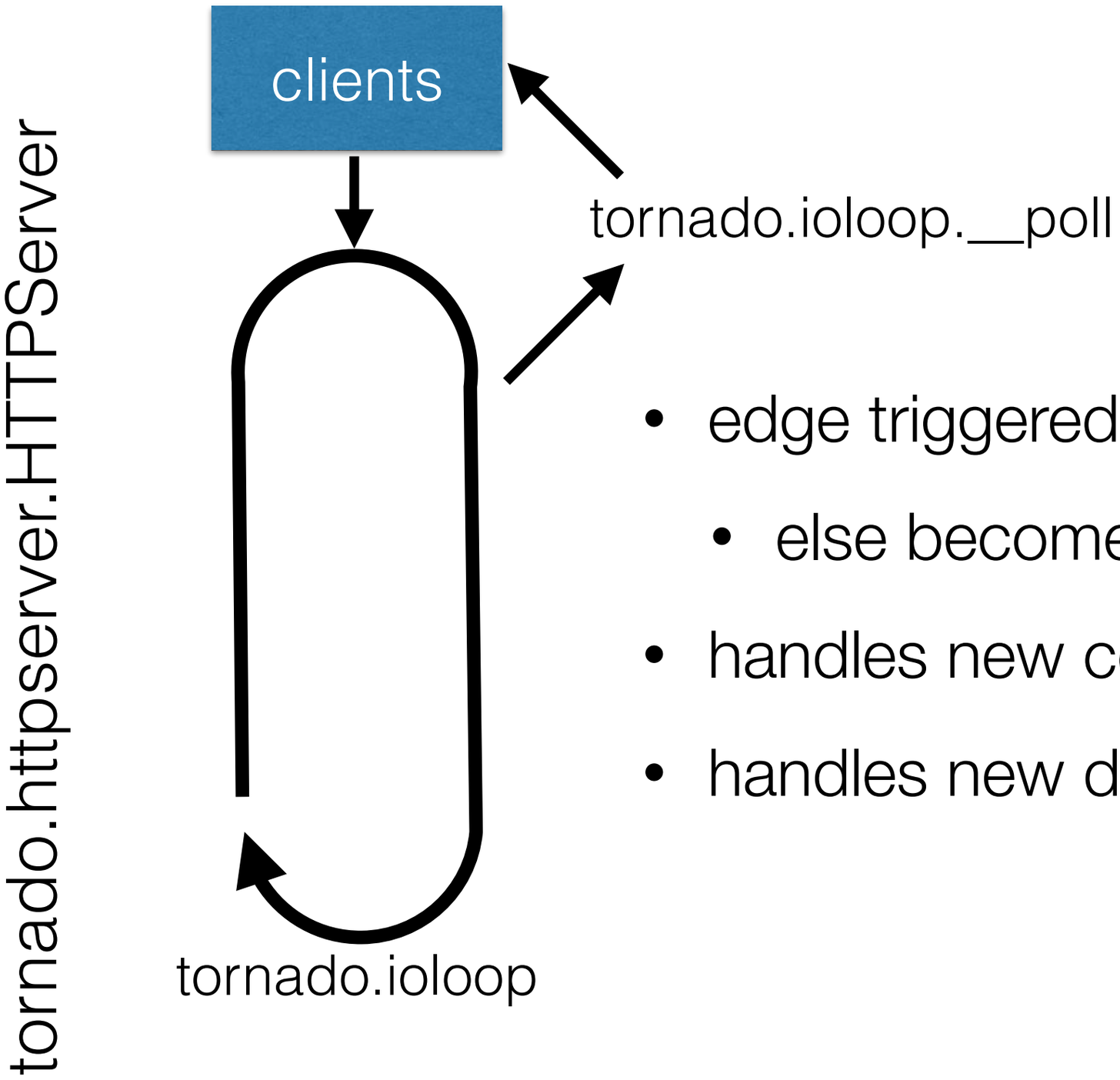
install tornado

- anaconda
  - conda install tornado
- pip
  - pip install tornado

tornado


tornado.httpserver.HTTPServer
clients
tornado.ioloop.__poll
tornado.ioloop
tornado.web.Application
sockets
tornado.ioloop.IOStream
tornado.httpserver.HTTPConnection
MVC / Routing
tornado.web.RequestHandler

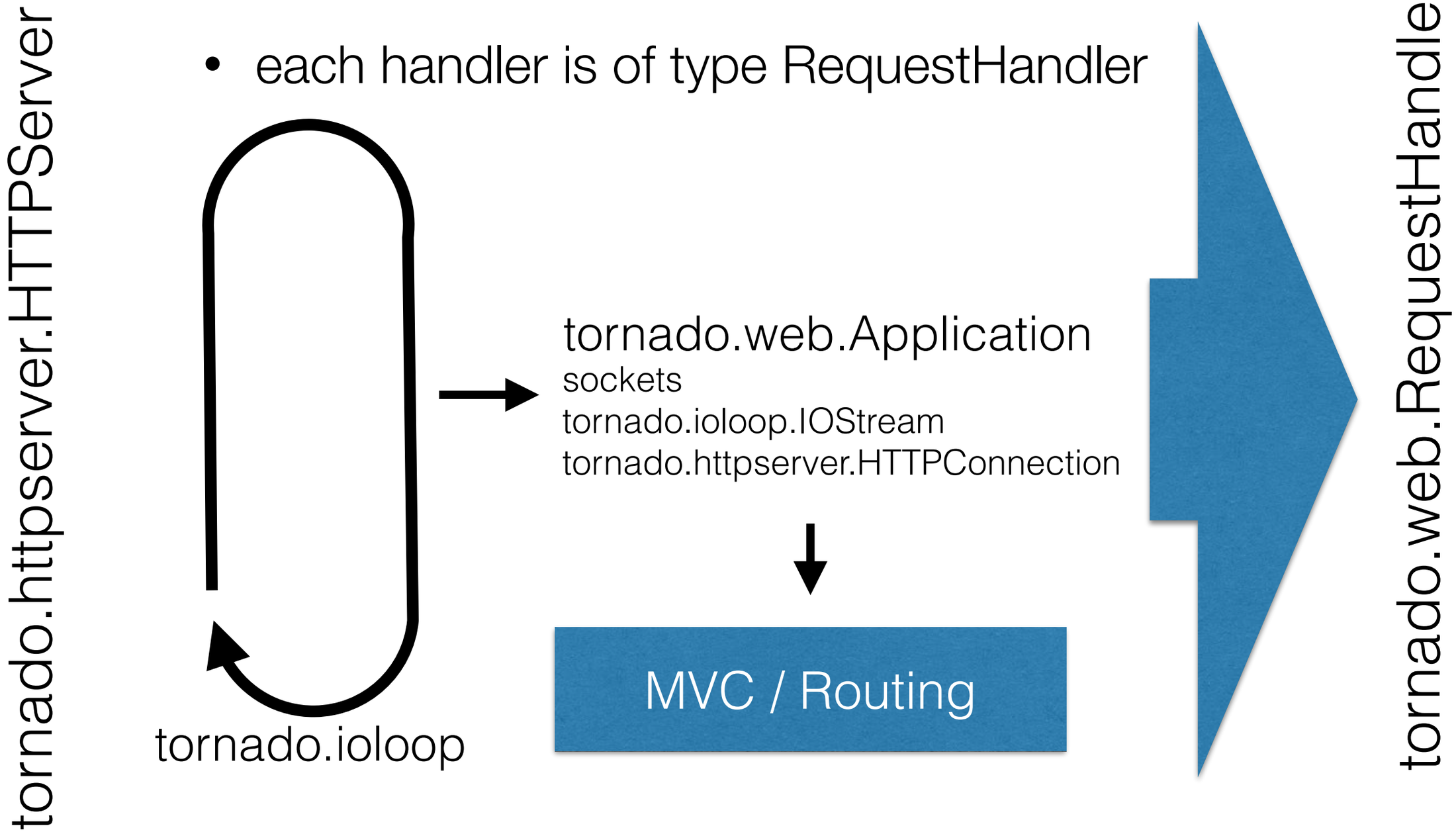
tornado



- edge triggered if possible
  - else becomes level triggered
- handles new connections
- handles new data from connection

tornado

- route URLs to different handlers
- each handler is of type RequestHandler



tornado example

- a very simple web server
- what is a get request?
  - a request for data from the server!
  - URL contains any na

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, MSLC World")

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

new class, inherit from RequestHandler

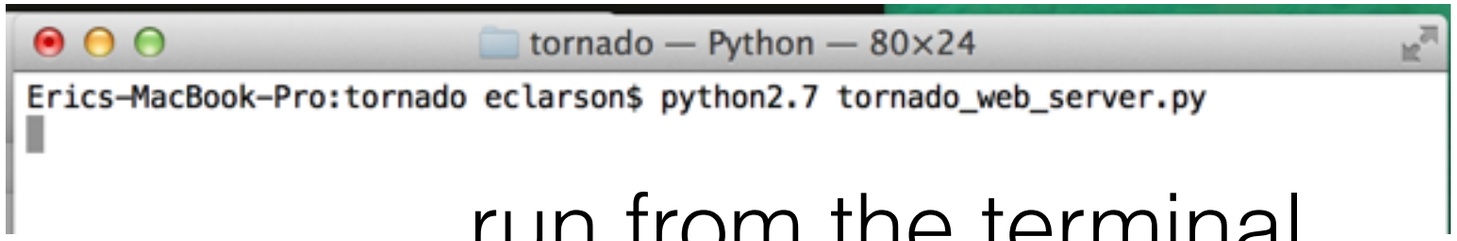
override get request handling

tuple with URL and handler

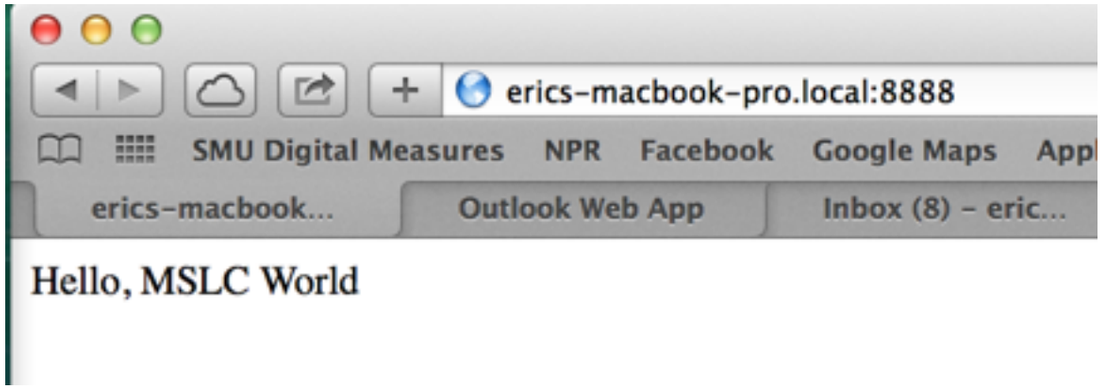
listen on 8888

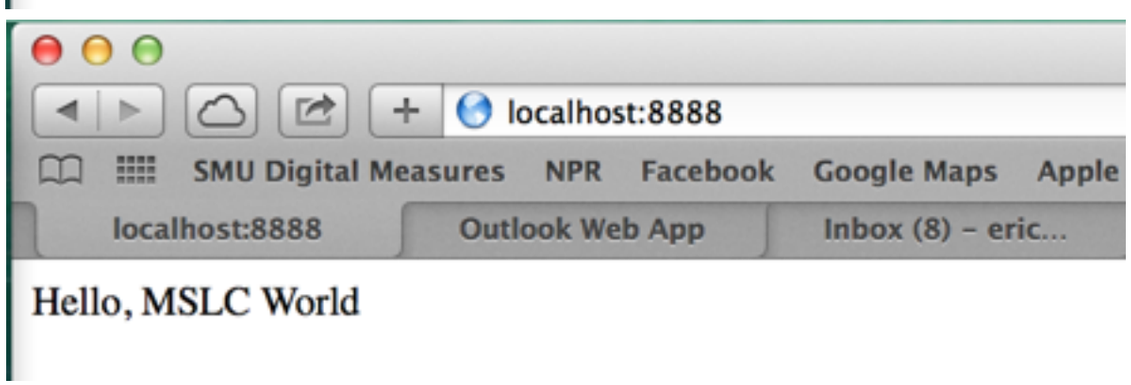
start the IO loop

tornado example



run from the terminal





for next time...

- more python examples
- tornado
- pymongo
- in preparation for
  - proper http requests in iOS
  - numpy
  - turi-create

install these on your mac!