Playing With Python

"The Interactive Workshop For Kids Who Know Ruby And Wanna Learn To Do Other Languages Good Too"

March 2015, NYC

@thomaswhyyou

~~Why Python?~~

Or.. why bother?

Post-Bootcamp Advice #1:

Pick up another language & framework
(right away, if you can)

// Routes taken personally:

Ruby (on Rails) -> Python & Django -> Python & Pyramid + Objective-C -> Swift + Go?

Objectives

Not: to preach you to Python, or argue about the best language, framework, practices etc etc..;

nor advocating jumping from one technology to another w/o "mastery"

Do: want to expose you to different sets of tools, philosophies, and approaches beyond RoR:

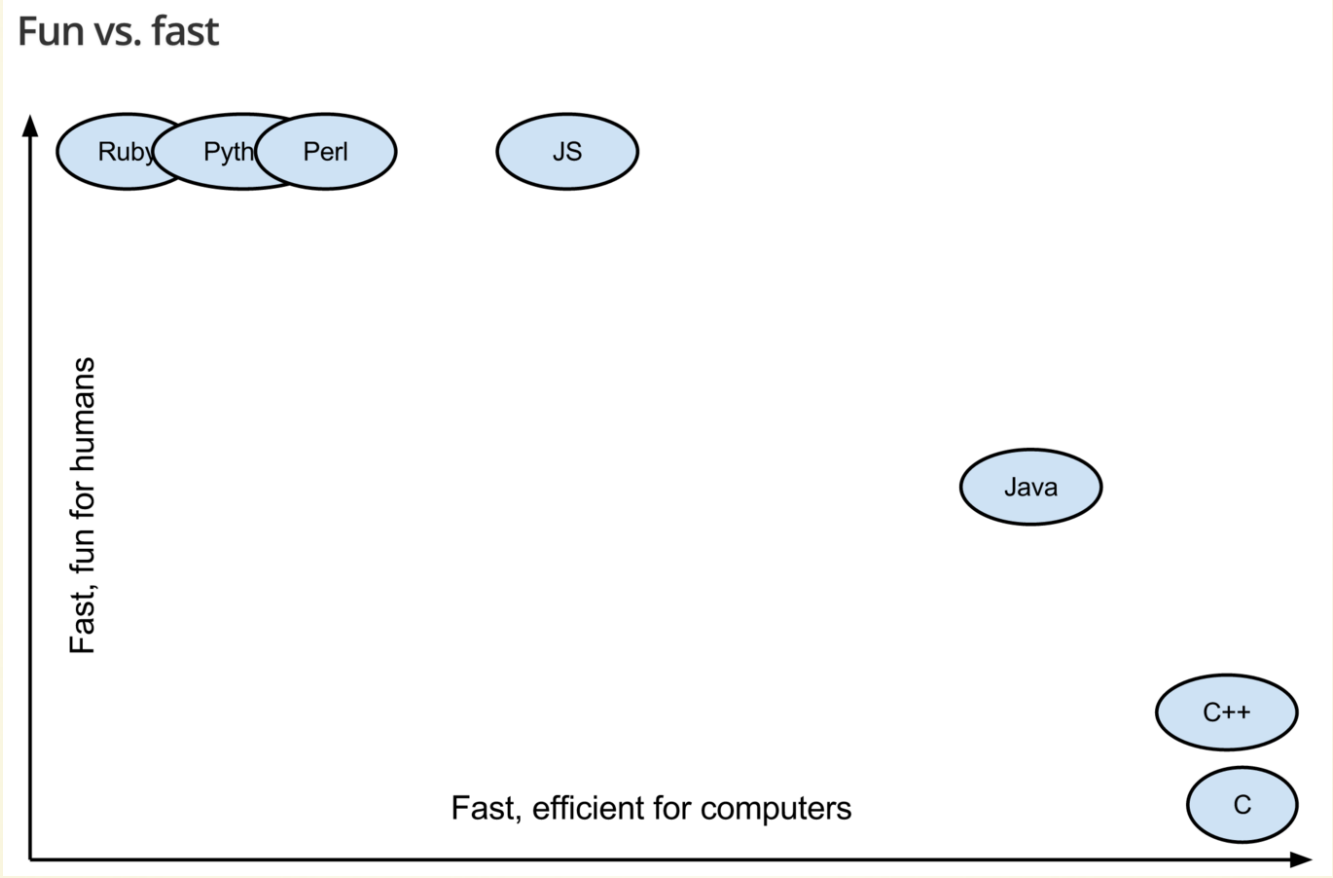
(2/1 == 100%, 3/2 == 50%, 4/3 == 33%...);

-> get you familiar enough to pick it up easily if/when needed

Maybe: it will be helpful for practical purposes, e.g. job interviews, coding test

Why Python?

Major programming language landscape



Go: 90% Perfect, 100% of the time. (http://talks.golang.org/2014/gocon-tokyo.slide)

Why Python?

Common descriptions of Python:

" *Easy to learn for beginners;*

Powerful for serious developers;

Simple, natural, and straightforward;

Flexible but sane;

Stable and mature;

Powerful standard libraries;

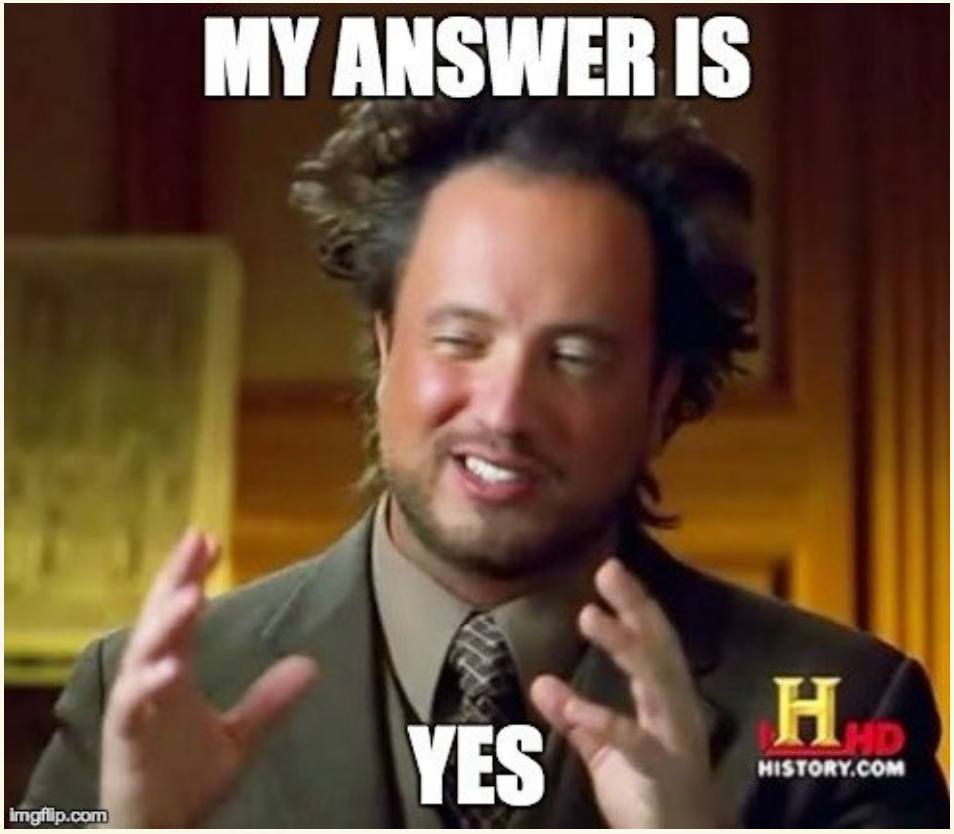
Vast third party libraries & community;

Sorta like Ruby..? -_-

Is it worth the time exploring Python?

Why Python?

Well...



Why Python?

The biggest strength..

Versatility

- General & Throwaway Scripting
- Web Development (Pinterest, Quora, Instagram, TheOnion, DropBox, Reddit etc.)
- Academia -> Scientific Computing (NumPy/SciPy stacks; BioPython (biology), Astropy (astronomy), Sage (mathematics))
- Data Science (Homogenization of Scientific Computing)
- Machine Learning (scikit-learn)
- Computer Vision (OpenCV native bindings)
- Quantitative Analysis & Finance (e.g. quant hedge funds)
- Hardware (e.g. Raspberry Pi)
- Easy integration/extension with other languages (e.g. C)

mobile...

"The second-best language for everything"

Why Python?

If it were a car... Minivan

" Python is great for everyday tasks: easy to drive, versatile, comes with all the conveniences built in. It isn't fast or sexy, but neither are your errands.

http://crashworks.org/if_programming_languages_were_vehicles/

Why Python?

Yes - similar to Ruby, language wise;

But:

Different philosophy;

Different toolsets;

Broader range of programmatic problem solving.

=

Differentiating & broader opportunities.

Revelation #1

Is learning programming hard or easy?

Programming language is the least of your battle.

The biggest battle you will be fighting is against the frameworks & external tools.

Because the hardest part is everything that stands between your code and the deployed & working app.

*(*including yourself)*

Agenda

~~Introduction + Background~~

Part 1: Tour of Python Language

Part 2: Python Tooling & Ecosystem (first)

Part 3: Building Pyramid
(Python Web Framework)

PART I

Tour of Python Language

What Is Python?

> *Python is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open.*

https://www.python.org/about/

ಠ_ಠ ?

Well.. that really doesn't explain anything.

> *Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.*

https://www.python.org/doc/essays/blurb/

What Is Python?

Brief history

- Created in 1989, by Guido van Rossum as a hobby project
- Named after "Monty Python", British sketch comedy show
- Python 2.0 released on 16 October 2000;
- Python 3.0 released on 3 December 2008
  - Broke some backward compatibilities
- Early adoption by academia
- Adoption by Google starting in 90s; initial implementation of crawlers (not so sure how much now)
- Popular web frameworks: Django (2005), Flask (2010), Pyramid(2008; 2010)

What Is Python?

TL;DR:

Interpreted*

Multi-paradigm

(Object Oriented, Imperative, Functional Flavors)

Strongly typed (vs weakly typed)

Dynamically typed (vs statically typed)

Garbage collected

.py extension

So...

pretty similar to Ruby?

Yes. Very similar.

What Is Python?

Note about different flavors of Python:

You may hear of different implementations of Python:
Jython, IronPython, Brython..

Because Python == interface specifications;
there exists multiple implementations

Most common & default implementation: CPython

Again, similar to Ruby; default implementation MRI (C based),
but has others.. JRuby, Rubinius, IronRuby etc.

What Is Python?

Note about two-step code execution

You will see .pyc files around your source directory

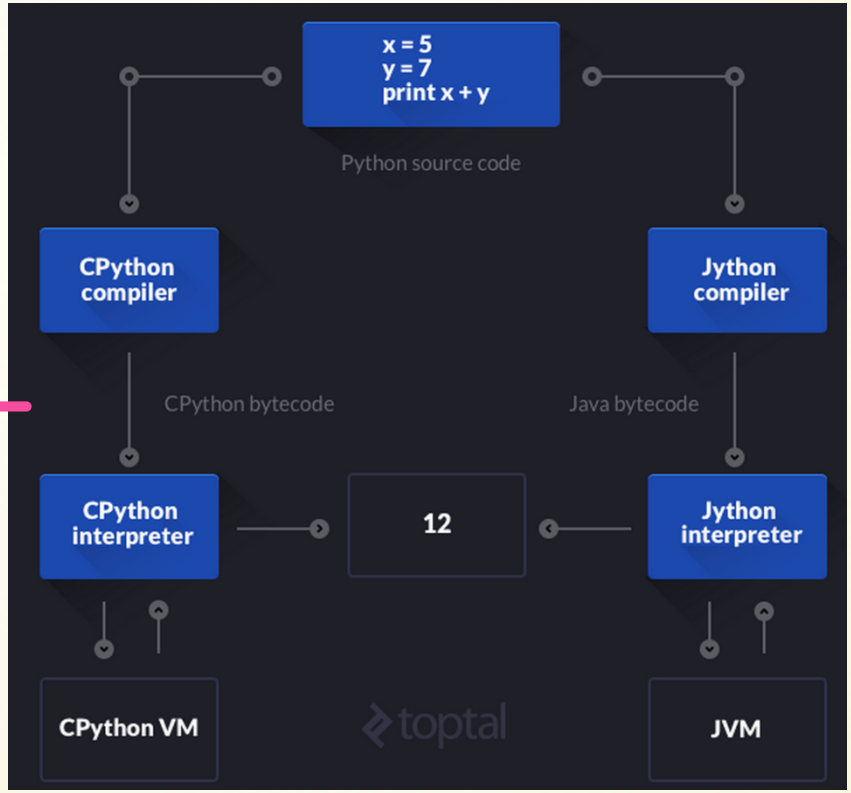
Source code (precompiled ->) Byte code (interpreted->) Machine language

.pyc

CPython puts them in disk;

MRI in memory

Usually included in .gitignore



For more info: http://stackoverflow.com/a/6210998/3479934

TL;DR Python

My Simplest Understanding

Class -> Object

Do [something], maybe with [x, y, z..].

operation () *data =*

behavior *state*

↙ ↓ ↘

- Functions & Methods
- Common operations (flow control, loop..)
- Decorators
- Exception handling

- Class definition
- Instantiation
- "everything is an object!!" ಠ_ಠ

- Values & Data types (List, Dict, Tuples, Sets..)
- "Truthy"
- Comprehensions

Also...

- Python Philosophies (?!)
- Formatting (?!)
- Code organization / namespace

@thomaswhyyou

First thing first..

"Zen of Python"

Python Philosophies

- *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 beats purity.*
- *Errors should never pass silently.*
- *Unless explicitly silenced.*
- *In the face of ambiguity, refuse the temptation to guess.*
- *There should be one - and preferably only one - obvious way to do it.*
- *Although that way may not be obvious at first unless you're Dutch.*
- *Now is better than never.*
- *Although never is often better than *right* now.*
- *If the implementation is hard to explain, 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!*

import this

@thomaswhyyou

"Zen of Python"

Abridged

Explicit is better than implicit.

Flat is better than nested; Sparse is better than dense.

Readability counts.

There should be one - and preferably only one - obvious way to do it.

Namespaces are one honking great idea - let's do more of those!

Not an absolute rule, some conflicting;
but a good measuring stick.

Example 1

// If you don't explicitly return from an operation, Python will not return anything for you.

```ruby
def add_numbers(arg1, arg2)
  arg1 + arg2
end

asdf = add_numbers(1, 1)
asdf == 2 # true
```

Any statement in ruby returns the value of the last evaluated expression

```python
def add_numbers(arg1, arg2):
    arg1 + arg2

asdf = add_numbers(1, 2)
asdf == 2 # False
```

You have to explicitly return value(s)

Example 2

// If you want to refer to itself (as in, instance of class), you must explicitly make a reference to "self"

```
class Hello
  def hello
    'hello'
  end
  def call_hello
    hello()
  end
end
```

```
class Hello(object):
    def hello(self):
        return 'hello'
    def call_hello(self):
        return self.hello()
```

"Formatting Matters"

// Whitespace is significant in Python*

(hate it or love it.. kinda like haml)

I love it.. (´∀ `)♡

// TL;DR

- Statement grouping is done by indentation instead of beginning and ending {} or "begin/end"
- Only delimiter is a colon (:) and the indentation of the code itself
- Generally, anywhere you'd expect starting and ending code block ("end" in ruby) -> you will use ":" and indentation alone.

*at indentation level of your statement;
only the relative indentation levels (not the exact amount of indentation)

"Formatting Matters"

// Example: Defining "Person" Class in Ruby

```ruby
class Person
  GOOGLE_DIRECTORS = ["Larry Page", "Sergey Brin", "Eric Schmidt"]

  def initialize(name, age)
    @name = name
    @age = age
  end

  def say_name
    puts "Hi, my name is #{@name}."
  end

  def say_age
    puts "Hi, my age is #{@age}."
  end

  def greet_google_directors
    Person::GOOGLE_DIRECTORS.each do |name|
      puts "Hello " + name + "."
    end
  end
end
```

examples/formatting_person_class_ruby.rb

Adapted from https://speakerdeck.com/mleone/python-for-ruby-programmers

"Formatting Matters"

// Example: Defining "Person" Class in Python

```
class Person(object):
    GOOGLE_DIRECTORS = ["Larry Page", "Sergey Brin", "Eric Schmidt"]

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_name(self):
        print("Hi, my name is {}.".format(self.name))

    def say_age(self):
        print("My age is {}.".format(self.age))

    def greet_google_directors(self):
        for name in Person.GOOGLE_DIRECTORS:
            print("Hello " + name + ".")
```

examples/formatting_person_class_python.py

Adapted from https://speakerdeck.com/mleone/python-for-ruby-programmers

"Formatting Matters"

// Indent responsibly (and consistently)

This doesn't work.

```
a = 12
 b = 13
```

```
File "indentation_error.py", line 2
   b = 13
   ^
IndentationError: unexpected indent
```

indentation_error.py

Below two behave completely differently.

```
numbers = [1, 2, 3]
letters = ["a", "b", "c"]
for n in numbers:
    print(n)
for l in letters:
    print(l)
```

VS

```
numbers = [1, 2, 3]
letters = ["a", "b", "c"]
for n in numbers:
    print(n)
    for l in letters:
        print(l)
```

examples/indentation_behavior.py

// "PEP 8" - the official guide

- 4 spaces per level recommended (not actual tab);
- In any case, do not want to mix them
- See for more: https://www.python.org/dev/peps/pep-0008/

"Formatting Matters"

// Commenting FYI

Ruby

```ruby
# This is a single line comment in Ruby.

=begin
This is a multiline comment,
and can spawn as many lines as
you like. But =begin and =end should
come in the first line only.
=end
```

Python

```python
# This is a single line comment in Python.

"""
This is a multiline string,
but can also be used as a multiline
comment.
"""
```

Python has a concept of "docstring", a string literal that occurs as the first statement in a module, function, class, or method definition; becomes __doc__ special attribute.

"""Useful for internal & external documentation"""

*playground/examples/*formatting_comments_docstring.py

Also: http://sphinx-doc.org/

@thomaswhyyou

Data Types & Variables

// Core Data Types Summary

- Expected & similar behavior to Ruby (+ native Set and Tuple)
- "Everything is an object"
- lowercase, snake_case for variable; CamelCase for class
- Dynamically typed -> re-assingment and different type ok

Type	Mutability	Common use case
String	Immutable	
Number (Int, Float)	Immutable	
Boolean	Immutable	
List (Array)	Mutable	Homogeneous collection, w/ order
Set (Set)	Mutable	Homogeneous collection, but unique
Dict (Hash)	Mutable	Characteristics collection, key-value
Tuple	Immutable	Heterogeneous grouping, temporary

Data Types & Structures

// Miscellaneous

- Dynamically typed, so this is perfectly legal

```
asdf = 15
asdf = "text"
asdf = None
```

- No actual constants; but ALL_CAPS by idiom
- "nil" in Ruby -> "None" in Python
  - Idiomatic to check with "is", not with "==".
  - *"Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators"*
- "is" vs. "=="
  - "reference equality" vs. "value equality"
- "true" -> "True"
- "false" -> "False"

examples/miscellaneous.py

Data Types & Structures

// Strings

- Single quote or double quote. (pretty much interchangeable)
- Tripple quote for multiline
- Unlike Ruby, strings are immutable
- String interpolation w/ format()
- Concatenate w/ "+"; repeat with "*"
- Iterable (0 indexed): len(), index lookup, slicing, looping
- Common operations:
  - replace, split, join (method of delimiter)
  - lower(), upper(), capitalize(), title()
  - "in"

examples/strings.py

Python Built-in Functions

// Btw..

```
abs()           cmp()           file()          hex()           set()
all()           compile()       filter()        id()            setattr()
any()           complex()       float()         input()         slice()
basestring()    delattr()       format()        int()           sorted()
bin()           dict()          frozenset()     isinstance()    staticmethod()
bool()          dir()           getattr()       issubclass()    str()
bytearray()     divmod()        globals()       iter()          sum()
callable()      enumerate()     hasattr()       len()           super()
chr()           eval()          hash()          list()          tuple()
classmethod()   execfile()      help()          locals()        type()
long()          next()          pow()           reduce()        unichr()
map()           object()        print()         reload()        unicode()
max()           oct()           property()      repr()          vars()
memoryview()    open()          range()         reversed()      xrange()
min()           ord()           raw_input()     round()         zip()
__import__()    apply()         buffer()        coerce()        intern()
```

They are there, and that's it.

Just FYI.

Python Keywords

```
# looping
for       while
break     continue    yield

# control flow
if        elif        else

# conditions
and       or          not
in        is

# module imports
from      import      as
```

```
# exception handling
try       except
finally   raise

# anonymous function
lambda

# noop i.e. do nothing
pass

# function/method and class
def       return
class
```

```
# context manager
with

# delete object
del

# debugging
assert
print

# miscellaneous
global
exec
```

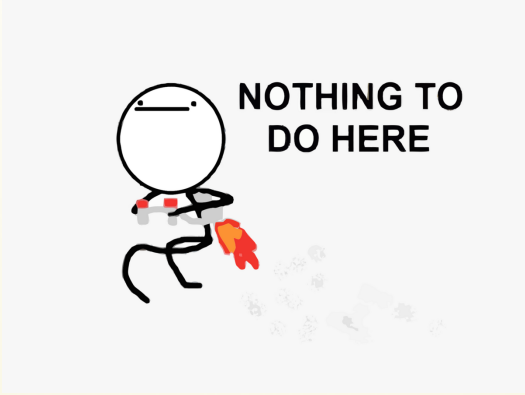
These are essential to know in order to work in Python,
and we will cover most of them.

Data Types & Structures

// Numbers

- More or less the same as in Ruby; exhibits similar, expected behavior
- int, float, long, and complex
  - "int" and "float"; personally never have used "long" and "complex"
- All usual operators:
  - + - * / % **

So, moving on..



Data Types & Structures

/ / List (Array in Ruby)

- Work pretty much the same way as in Ruby
- Get and set with [index]
- Common operations:
  - iterable: len(), index lookup, slicing, index()
  - adding element(s): append(), extend()
  - removing element: "del" keyword by index; .remove() by value
  - looping with "for ... in ..." loop; also w/ "enumerate()"

examples/lists.py

Data Types & Structures

// Dictionary (Hash in Ruby)

- Work identical to Ruby Hash for all intents and purposes
- Only one syntax {key: value} (no hash rocket)
- Get and set with [key]
- Missing key is an exception in Python (vs. nil in Ruby); use .get()
- No particular order by default (but there is OrderedDict)
- Common operations:
  - .keys(), .values(), .items(), .get()

examples/dicts.py

Looping

// Looping in Python is simple:

```
# looping
for       while
break     continue    yield
```

for ... in ... :

- Typically to iterate over known/structured collection of data
- Used very frequently and idiomatic; when in doubt, use for-in loop.

while ... :

- Useful when looping over for an unpredictable condition or unstructured data
- Used when makes sense; used more sparingly than you think?

list | dict | set comprehension

- Concise way to manipulate/filter/create collections
- Native construct in Python
- Used as frequently as for-in loop; it's super awesome

Looping

/ / Examples

for ... in ... :

Ruby

List:

```ruby
items = [1,2,3,4]
items.each do |i|
  puts i
end
```

Dict:

```ruby
menu = {"bagel": 5, "coffee": 3}
menu.each do |item, price|
  puts item, price
end
```

Python

```python
items = [1,2,3,4]
for i in items:
    print(i)
```

```python
menu = {"bagel": 5, "coffee": 3}
for item, price in menu.items():
    print(item, price)
```

while ... :

```python
while True:
    if some_condition_is_met:
        break
```

list | dict | set comprehension

```python
[x**2 for x in items if x % 2 == 0]
#[4, 16]

#[<output> for <item> in <iterable>]
# optional filter at the end
```

[loop example in python + examples/data/characters.py]

Exercise #1

/ / (fake) JSON Response Parsing

How many people live in this building?

How much total rent per month this building?

What's the average rent per sqft?

examples/data/fake_json_resp.py

Control Flows

// Control flow in Python is even simpler:

```
# control flow
if        elif      else
```

```
# conditions & comparisons
and       or
not       in        is
== != > < <= >=
```

- No "Switch"/"Case" statement. (proposal actually rejected *)
- No "Unless" statement. (use "if not")
- Just use "if-elif-else", that's it. :]

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
    x = 0
    print 'Negative changed to zero'
elif x == 0:
    print 'Zero'
elif x == 1:
    print 'Single'
else:
    print 'More'
```

examples/control_flows.py

* Note: https://www.python.org/dev/peps/pep-3103/

About "Truthy"

Ruby:

When tested for truth, only false and nil evaluate to false value

Everything else is true (including 0, 0.0, "", and [] etc)

Python:

Has more values that evaluate to falsey (including 0, 0.0, "", and [])

Rule of thumb: if something is "empty", it will evaluate to false

Very common pattern to test "Truthy"

Exercise #2

/ / FizzBuzz!

I know what you are thinking...

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"

Data Types & Structures

/ / Set

- Set is a dictionary with no values
  - It has only unique keys
  - Syntax is similar to that for a dictionary {}
  - Native data type
- In practice, sets are less often used but can be useful in certain situations
- Common operations: issubset(), issuperset(), +, -

examples/sets.py

Exercise #3

// JSON Response Parsing - Redux

How much rent needs to be collected still?

How many units are above 3000 sqft?

How many units fetch above $3000 per month rent?

How many unique names in this building?

examples/data/fake_json_resp.py

Data Types & Structures

// Tuple

- No Tuple in Ruby; Useful and widely used in Python
- Common use case:
  - Group related but heterogeneous things together, often for temporary use
- Unlike List, Tuple is immutable, and read-only.
- Created by comma separation
- Example:
  - Function return multiple values

examples/tuples

Everything Else

Class definition

```
# noop i.e. do nothing
pass

# function/method and class
class
```

Function/method definition

```
# noop i.e. do nothing
pass

# function/method and class
def       return
```

- Functions
  - No implicit return
  - Multiple return values will be in tuple
- Classes & Instances
  - __init__
  - Allows multiple inheritance
- Methods - function attached to class/object

[exercise together]

Exercise #4

// Simple card game simulator

- Only two players: "dealer" and "player"
- Dealer plays with one deck of 52 cards
- Automate plays without user input
- Dealer plays each round by handing out one card to itself and Player
- Higher card wins, same number ties
- Game ends when cards run out
- Let's create a script that simulates this card game and provide game summary.

Lambda

// Anonymous Function

```
# anonymous function
lambda
```

- Python doesn't have an equivalent of Ruby's Block
  - "foo.method do |x,y,z| ... end"
- Python has "lambda" (anonymous function) as well as "decorators" (higher order function) -> better look at decorators later
- (Intentional *) limitation of lambda: *must be expressed in one line*
- Also, mind the implicit return
- You don't _require_ lambda; but convenient in certain cases

Example:

```
# Assume "numbers" is an array/list of int.
numbers.map {|x| x*2}                    # Ruby block
map(function(x) {return x*2;}, numbers); // JS function
map(lambda x: x*2, numbers)              # Python lambda
```

[examples in ipdb]

* Multi-line lambda proposal was in fact rejected:
http://www.artima.com/weblogs/viewpost.jsp?thread=147358

Underscores in Python

/ / _ and __

_ *as a name:*

```
n = 42
for _ in range(n):
    do_something()
```

- Throw-away name by convention
- Explicit indication for no use

_ *prefix to a name:*

```
class Something(object):
    _priviate_var = 1234
    def _internal_method(self):
        pass
```

- Indicate internal use (vs public API)
- "Semi-convetion"

__ *prefix (& suffix) to a name:*

```
if __name__ == "__main__":
    app = App()
    app.run()
```

- Python special objects or attributes
- Considered Python reserved; you shouldn't play in that pool

Underscores in Python

/ / if __name__ == "__main__":

```
if __name__ == "__main__":
    app = App()
    app.run()
```

- When the Python interpreter reads a source file, it executes all of the code found in it.
- Before executing the code, it will define a few special variables
- If the python interpreter is running that module (the source file) as the main program, it sets the special __name__ variable to have a value "__main__"
- Consider it as an entry point

Namespacing

"Namespaces are one honking great idea - let's do more of those!"

```
# module imports
from      import      as
```

- A module is simply a file containing Python code.
- Module can expose classes, functions and global variables.
- Each module gets its own namespace and when imported from another Python source file, the file name is treated as a namespace.
- A Python package is simply a directory of Python module(s).
- You can be as specific as you want when you are importing modules/packages

Underscores in Python

// __init__.py

- Files named __init__.py are used to mark directories on disk as Python package directories
- Also allows you to define any variable at the package level

Exception Handling

```
# exception handling
try       except    finally     raise
```

- Exceptions work about the same despite the difference keywords
- Exceptions in Python are liberally utilized
- First on-the-job lesson: Learn to look for & catch exceptions
- "Catch-all" is generally a bad idea (sometimes your code *should* fail)

https://docs.python.org/2/library/exceptions.html#exception-hierarchy

Ruby

```ruby
begin
  # some code
rescue OneTypeOfException
  # -
rescue AnotherTypeOfException => e
  # -
rescue
  # catch all; don't do this
  raise
ensure
  # Always will be executed
end
```

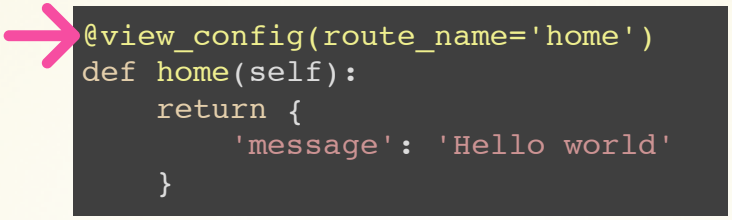
Python

```python
try:
    # some code
except OneTypeOfException:
    # -
except AnotherTypeOfException as e:
    # -
except:
    # catch all; this is bad.
    raise
finally:
    # Always will be executed
```

examples/handling_exceptions.py

Decorators

// Higher order functions



```
@view_config(route_name='home')
def home(self):
    return {
        'message': 'Hello world'
    }
```

- It looks like this *@something*
- Decorators can alter the behavior of a function/method/class without modifying the underlying callable
  - Ideal for extending existing functionalities on the fly
- Think of it as a "wrapper" that can modify the behavior of code before and/or after
- Hence the name, "decorator"

Decorators

// Getting to know functions better

- In Python, functions are "first-class" objects

1. They can be assigned to a variable

```
def greet(name):
    return "hello " + name

greet_someone = greet
print greet_someone("John")
# Outputs: hello John
```

2. Define functions inside other functions

```
def greet(name):
    def get_message():
        return "Hello "

    result = get_message() + name
    return result

print greet("John")
# Outputs: Hello John
```

3. They can be passed around as an argument

```
def greet(name):
   return "Hello " + name

def call_func(func):
    return func("John")

print call_func(greet)
# Outputs: Hello John
```

4. Function can return another function

```
def compose_greet_func(name):
    def get_message():
        return "Hello there " + name

    return get_message

greet = compose_greet_func("John")
print greet()
# Outputs: Hello there John
```

Inner function has read-only access to the enclosing scope

Original examples from: http://thecodeship.com/patterns/guide-to-python-function-

Decorators

/ / Putting it together to use

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

my_get_text = p_decorate(get_text)
print my_get_text("John")
# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

using @ syntactic sugar

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

@p_decorate
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

print my_get_text("John")
# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

Decorators

/ / Decorators with arguments

```
def tag_with(tag):
    def tags_decorator(func):
        def func_wrapper(text):
            return "<{tag}>{text}</{tag}>".format(tag=tag, text=func(text))
        return func_wrapper
    return tags_decorator

@tag_with("div")
@tag_with("p")
@tag_with("strong")
def make_text(name):
    return "Hello " + name
```

examples/understanding_decorators.py

Python 2 vs 3

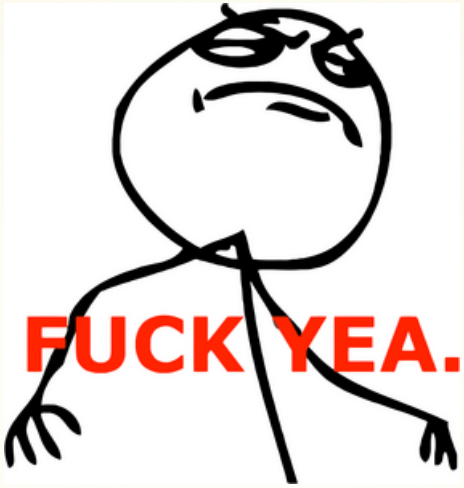
// Note about Python versions

- Python 2.0 released in 2000; Python 3.0 in 2008
  - Python 2.7 is the final 2.x version; was released in 2010
  - Extended support until 2020; bug fixes only
  - Python 3.x under active development; currently at 3.4 (as of 2015)
- So, what's the deal with this?
  - GvR decided to clean up Python 2.x properly and broke some backward compatibilities with 3.x
  - Notable changes include all text is unicode by default
- Source of some grumbling and pain in the community
  - Initial adoption and pace of porting slow, but improving continuously
- As of 2015, ~85% (169/200) of top libraries now compatible in python 3*

* Source: https://python3wos.appspot.com/
Rabbit hole: http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html

That's about it.

// Now you know Python.



Closing thoughts...

Simple, obvious and smaller grammar + similarity to Ruby

= Low learning curve to productivity

+ Versatility & broader community outside web

=> Don't "swipe left" just because it's not RoR!

PART II

Python Tooling & Ecosystem

Essential Tooling

(for getting started with Python)

Running Python

Managing Python versions

Handling Python packages

Python environment

Intro to Virtualenv

Using debugger

ST for Python

@thomaswhyyou

Running Python

/ / Two modes:

interactive via live interpreter

- Ruby: "irb" -> "pry"
- Python: "python" -> "ipython" (& "ipdb")

or

scripting via file(s)

- Ruby: $ ruby something.rb
  - require "pry"
  - binding.pry # start a REPL session
- Python: $ python something.py
  - import ipdb; ipdb.set_trace()

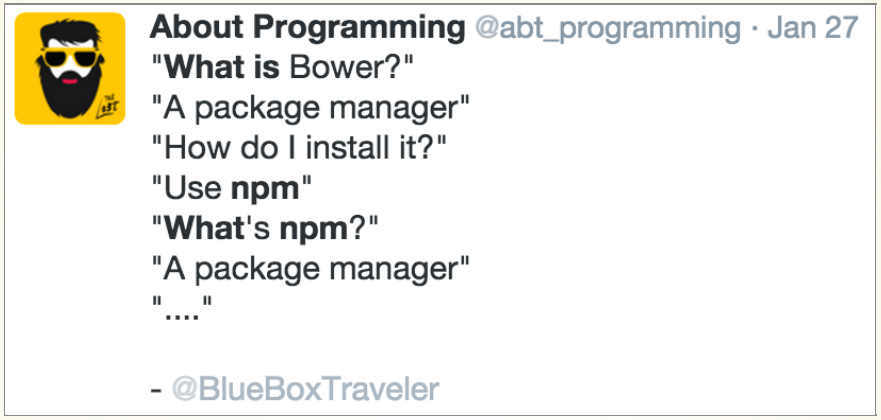
Managing Python Versions

- Sometimes you may need to switch between different Python versions
- Ruby has tools like: RVM, rbenv, chruby
- Python has: pyenv
  - Forked from rbenv and ruby-build
  - Let you change the global Python version or per-project basis
  - github.com/yyuu/pyenv

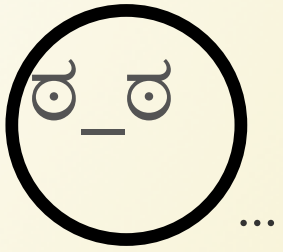
(..but we won't go into using pyenv for the workshop)

Handling Packages

Let me tell you a joke..

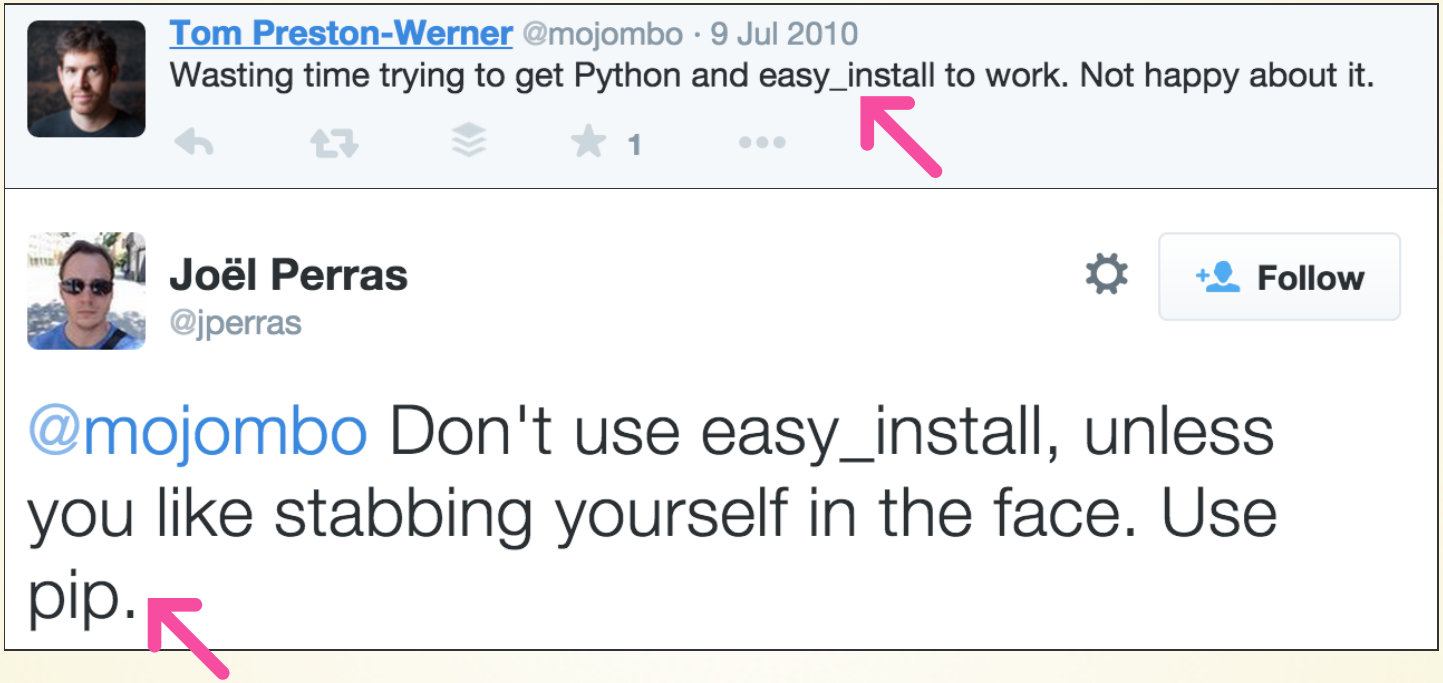


One more.. for Python



Handling Packages

easy_install? pip?



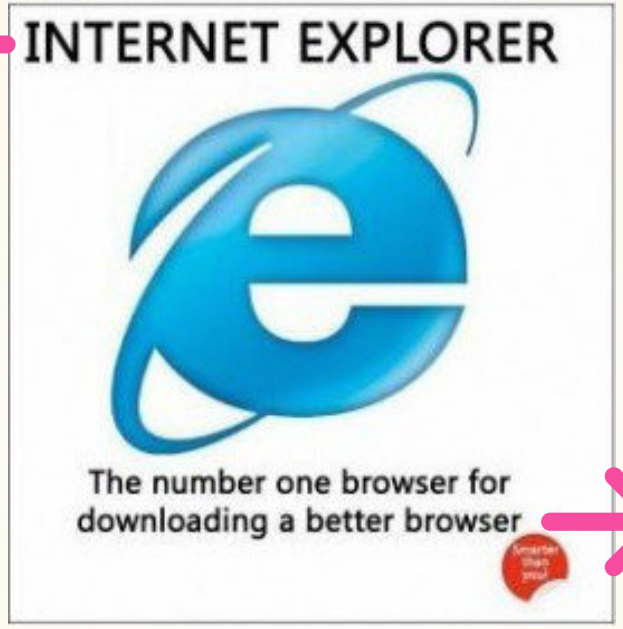
// Python has gone through some "complicated" relationships with package mangers...

For more info: https://packaging.python.org/en/latest/pip_easy_install.html

Package Manager

easy_install

Comes installed w/ Python itself



pip

Install with easy_install

// When in doubt, use PIP.

PIP will install Python packages from PyPI (Python Package Index)

(Similar to RubyGems in Ruby world, or NPM or Node.js)

```
$ easy_install pip
```

We will use PIP together shortly.

Python Environment

/ / You + Python environment

```
$ type -a python
$ pip freeze
$ pip show <package>
```

This is a global python environment.

Normally you don't want to install project dependencies directly into this global Python environment..

..Because you will have multiple Python projects as well as system dependencies

<u>The problem:</u>

"Project X depends on version 1.x but, Project Y needs 4.x"

Virtualenv

// Python level isolation

Virtualenv solves this problem by creating a completely isolated *virtual environment* for Python as needed

An environment is simply a directory that contains a complete copy of everything needed to run a Python program

- own copy of python binaries
- own copy of entire python standard libraries
- own copy of pip
- directory where libraries get installed (site-packages)

*virtualenv is one of *few* things you can install globally*

Note: virtualenv is now rolled into Python 3. (starting 3.3)

Virtualenv

// Installing & Creating a virtualenv

```
# Sudo maybe
$ pip install virtualenv
```

```
# Set up a directory where we want
# to create our virtualenvs
$ cd ~; mkdir venv; cd ~/venv

# create a virtualenv named "workshop"
$ virtualenv workshop

# activate "workshop" virtualenv
$ source ./bin/activate
-> (workshop)$ source ./bin/activate

# deactivate current virtualenv
$ deactivate

# See the difference
$ type -a python
$ type -a pip

# You can & should also have multiple
# virtualenvs, for each project
```

>> import sys
>> for path in sys.path: print(path)

Using PIP

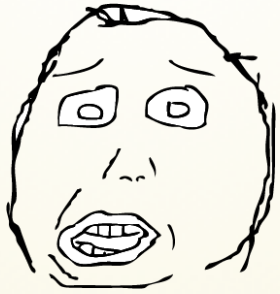
/ / Let's install a couple things:

```
# with "workshop" virtualenv activated;
(workshop) $ pip freeze

# you shouldn't need sudo
(workshop) $ pip install ipython
(workshop) $ pip install ipdb
(workshop) $ pip install requests

(workshop) $ pip freeze

# check site-packages
```



(all of this was a complete mystery to me in the beginning..)

Debugger

Python ships with native debugger, "pdb".

But there's a better debugger:

Use ipdb + ipython instead

```
import ipdb; ipdb.set_trace()
```

(memorize this line)

A few useful commands inside debugger:

n - next

c - continue

l - show where you are

q - quit

Debugger

Strategically putting breakpoints inside your code with debuggers can *really* help you, especially initially

Remember:

* Debugger is your best friend *

Seriously.

Not just for debugging;

Useful when building up code.

Sublime Text

If you are using Sublime Text to code Python:

- Anaconda (https://github.com/DamnWidget/anaconda)

Pretty much the all-in-one for Python development

<u>Some other favorite packages of mine:</u>

- SideBarEnhancements
- Bracket Highlighter
- AllAutocomplete
- SublimeLinter (doh)
- GitGutter

PART III

Building Pyramid

Python Web Framework

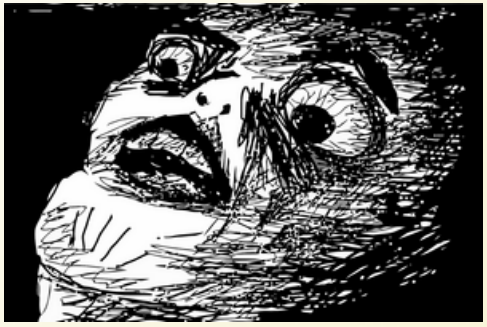
Question #1

// What is "framework"?

// What is "library"?

// What is the difference between "library" vs "framework"?

gasp



> *You call library;*
> *Framework calls you.*

Wait, but who calls the framework?

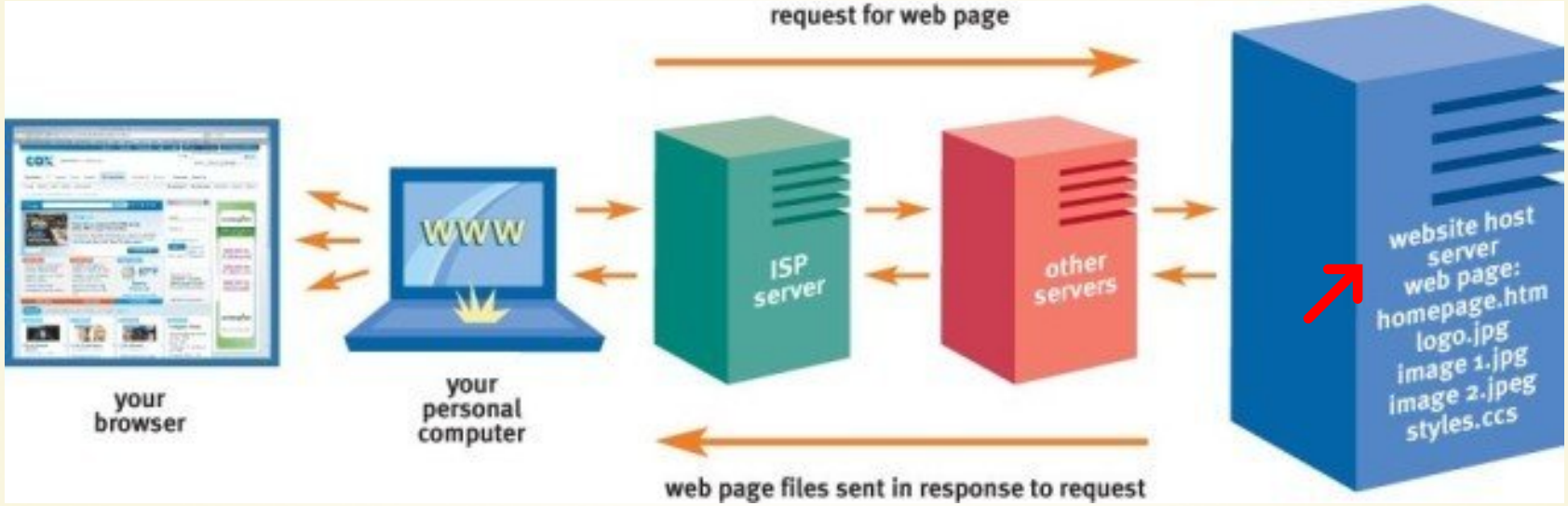
source:

http://stackoverflow.com/a/233765/3479934

Request-Response Cycle

// "How the Web Works"

"Stateless" rendering *app logic, data, etc*



HTTP (GET, POST...) *HTML/JSON etc.*

Browser sends a request <-> Your server responds with a response

So what happens in the "server"?

Servers

/ / Let's talk about servers

(hardware or software)

* Web server *

Application server

Database server

Mail server

IRC server

...

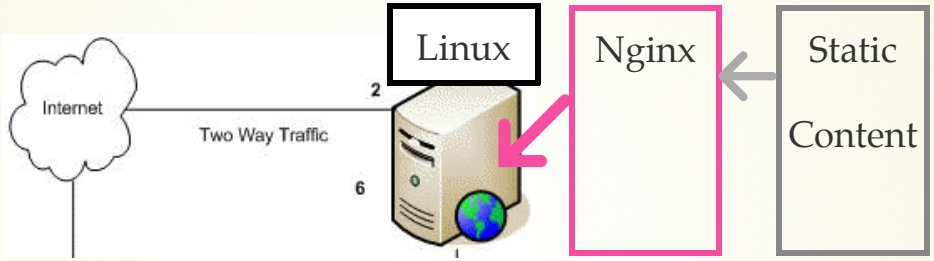
Vending machine

They are on, connected, and
ready to respond to a request
(for something)

Web Server

/ / Example

Apache, Nginx, LightSpeed, Lighttpd...



Sample Nginx configuration for serving static sites

```
server {
    listen 80;
    server_name example.com www.example.com;

    location / {
        root /data/www;
    }

    location /images/ {
        root /data;
    }
}
```

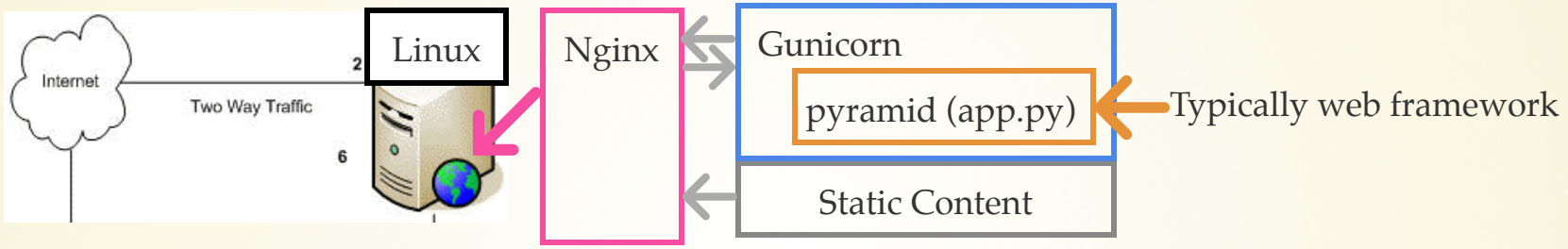
- e.g. static blog

What about dynamic contents?

Application (Web) Server

/ / Example (Python)

uWSGI, Gunicorn (Unicorn in Ruby), mod_wsgi..



Sample Nginx configuration for serving static sites + dynamic content

```
server {
    listen 80;
    server_name example.com www.example.com;

    location /         { root /data/www; }

    location /images/ { root /data; }

    location /app/ {
        proxy_pass http://127.0.0.1:8001;
    }
}
```

Example of running Gunicorn

```
$ gunicorn --bind=127.0.0.1:8001 app:main
```

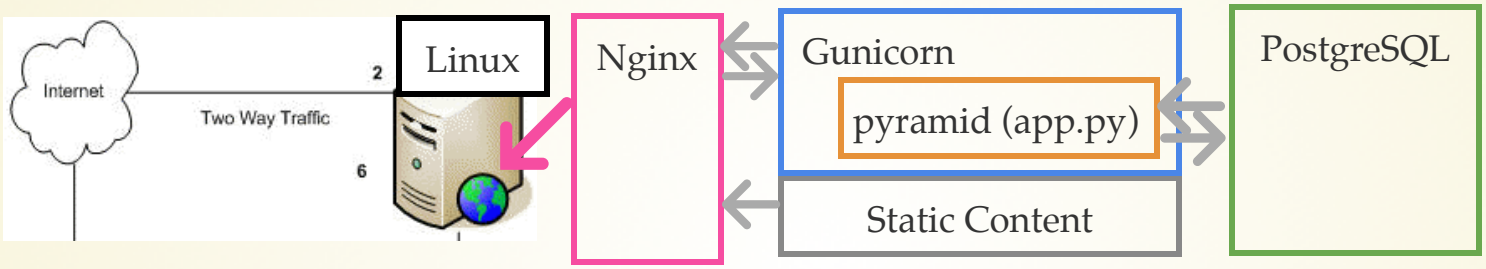
- e.g. web app
- Normally you'd want a process controller like *Supervisord* to run

What about data?

Database Server

// Example

MySQL, Oracle, PostgreSQL; MongoDB, CouchDB, Redis...



Typical three tier architecture, running on Linux

Another variation of this stack:

Linux

Apache

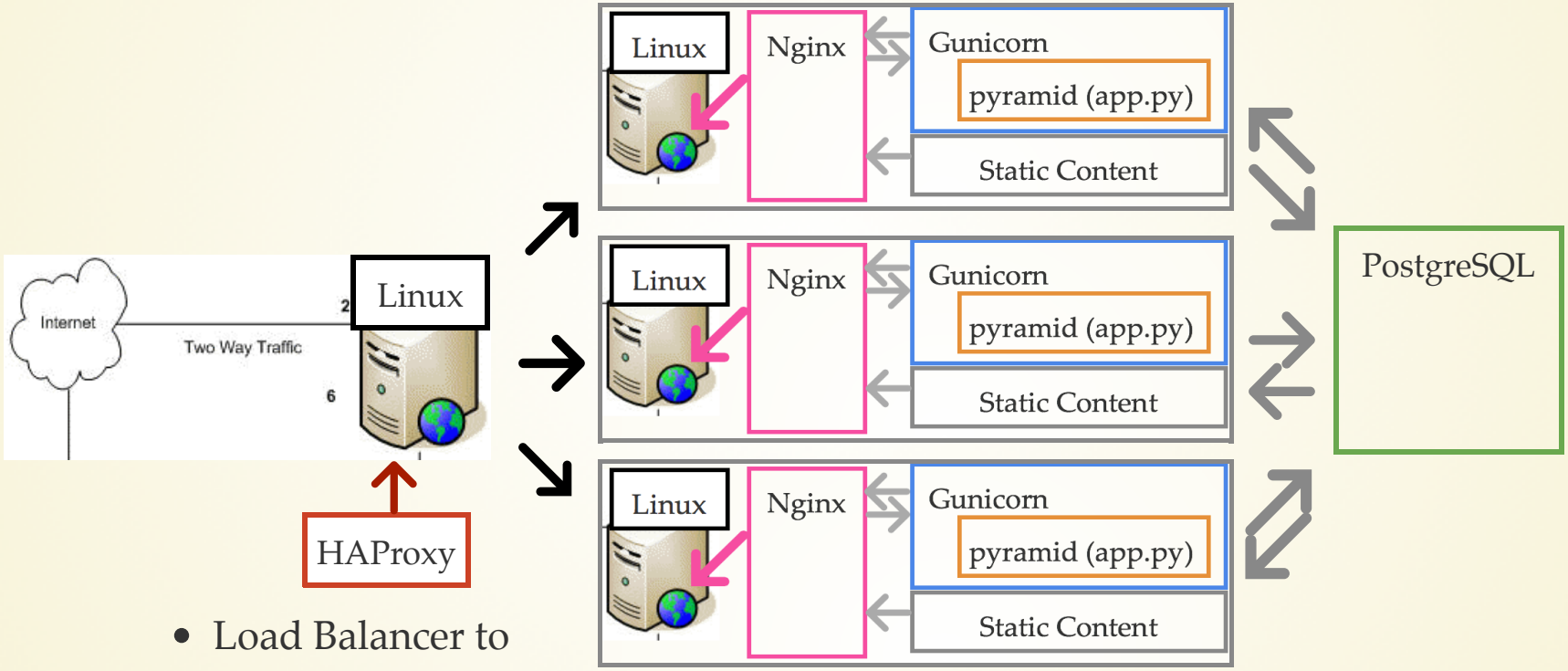
MySQL

PHP

Read more: http://serverfault.com/a/331263

Horizontal Scaling

// Example



- Load Balancer to direct traffic

Web Framework

// Let's talk about web frameworks

What do you actually need in application layer at minimum*?

Not necessarily with a framework

Something to glue together

Routing

Templating

(html, xml, json etc)

Persist / access / manipulate data

Each component doesn't need to come in the same package;
Everything else can be per "as needed".

* Of course subject to different use cases, but the core common functionalities.

Python Web Frameworks

// We have (too?) many options to choose from..

Comparison of Popular Python Web Frameworks

(As of February 28, 2015 - by twitter:@thomaswhyyou)

Framework	Note	Github Stars	Github Commits	Github Contributors	Lines of code?*		Latest update	Initial release
Django	Full-stack	13,089	20,111	786	53,792	(../django, exc. ./test)	10/22/14	2005
Flask	Micro	12,813	2,169	249	6,294	(../flask)	6/14/13	2010
Bottle	Micro	2,788	1,454	105	4,018	(exc. ./test, ./docs)	4/29/14	2009?
Pyramid	In-between	1,604	8,122	188	23,316	(../pyramid, exc. ./tests)	11/9/14	2008 (repoze.bfg)
web2py	Full-stack	774	5,841	89	110,462	(../gluon, exc. ./tests)	9/15/14	2007
TurboGears2	Full-stack-ish?	83	1,651	18	9,622	(../tg2/tg)	10/3/14	2005 (TurboGears)
CherryPy	In-between	21	2,661	38	23,185	(../cherrypy, exc. ./test, ./tutorial)	9/14/14	2001?
Rails		25,125	50,016	2,617	58,520		12/19/14	2005

* git ls-files | xargs wc -l

Top choices, by the amount of "batteries" included:

Django > Pyramid > Flask

Example:

Includes ORM	*ORM agnostic*	*ORM agnostic*
Includes templating	*Templating agnostic*	*Templating agnostic*

See more: https://wiki.python.org/moin/WebFrameworks

@thomaswhyyou

Pyramid

// Introducing Pyramid

Pyramid™

From the mind of...

Chris McDonough

Photo from Flickr: Beards of Python PyCon 2009

Opening slide stolen from: https://www.youtube.com/watch?

@thomaswhyyou

v=DBV0MsRu72M

Pyramid

/ / Some history behind Pyramid

From PyCon Aus 2011

@thomaswhyyou

Pyramid

- Non-opinionated, "micro-ish-but-more" framework
- Emphasis on simplicity & extensibility; Use only what you need "a la carte" philosophy
- Targeted for letting you start small and get big when needed
- Things Pyramid includes out of the box (but not limited to):
  - Mapping & Routing URLs (URL Dispatch or Traversal)
  - Low level template API
  - Authentication/authorization security
- Things Pyramid doesn't include or assume: Data persistence mechanism; ORM agnostic, templating engine agnostic, admin interface, form validation or model integration, mailer, job queue...

The M Word

By the way..

MVC (Model View Controller): e.g. Rails

MV*? (Model View.. *Whatever?): e.g. Backbone

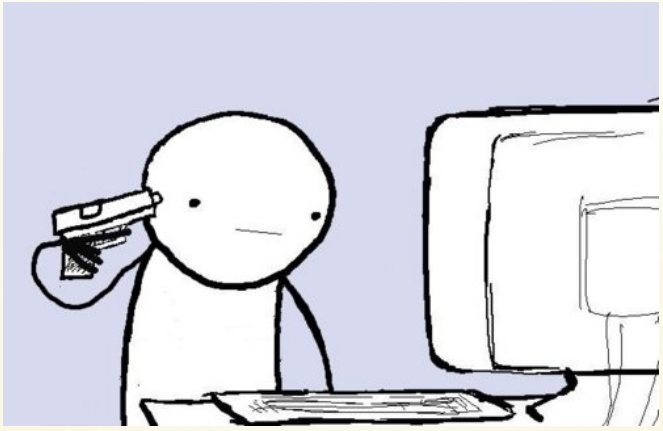
MVVM?? (Model View.. View Model?): e.g. iOS

http://www.objc.io/issue-13/mvvm.html

Now, introducing...

MTV!!! (Model Template View!): e.g. Django, Pyramid

http://bit.ly/1MTjAq7



The M Word

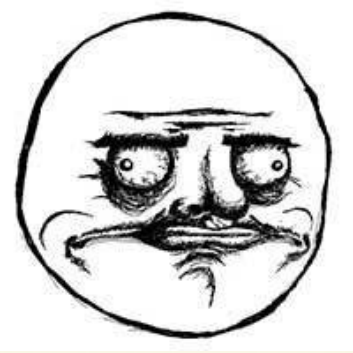
For all intents and purposes:

- one part deals with handling application data;
- another part deals w/ delivering that data via some logic
- another w/ how the data will appear to users

For the remainder of this workshop:

Let's go with MTV.

Because.. reasons.



Let's Code

// We will build something uber simple together.

[github repo]

```
# with "workshop" virtualenv activated;
(workshop) $ pip install pyramid
```

Wrap-up

Thank you.

Questions?

Feedback?