Things I wish I Knew
Before Using Python for
Data Processing

Miguel Cabrera
@mfcabrera
Hello!
I am Miguel!

Data Engineer / Scientist @ TrustYou
Python ~ 2 years
Berlin

@mfcabrera
mfcabrera at gmail
http://mfcabrera.com
Priors!

- (Relatively) New to Python, mostly Scientific stack
- You have used things like Numpy, Scikit-Learn, Gensim, etc...
- Your job title includes either the word Data or “Machine Learning”.
- Not necessarily a trained Software Engineer
Who is who?

- Data Scientist?
- Data Analyst?
- Data Engineer?
- Machine Learning Developer?
- Software Developer?
- Other?
Agenda

- Basic Concepts and practices
- Some goodies of the collection module
- Iterators and Iterables
- Conclusion
David’s Story

- Recent university grad
- Mostly R and Matlab
- Writes nifty code to classify documents using Jupyter Notebooks
- Mostly NLtk and Scikit-Learn
```python
from bokeh.models import ColumnDataSource
from bokeh.plotting import *
import numpy as np

x = np.linspace(0, 2*np.pi, 2000)
y = np.sin(x)

output_notebook()

print('BokehJS successfully loaded.

source = ColumnDataSource(data=dict(x=x, y=y))

p = figure(title="simple line example", plot_height=300, plot_width=600)
p.line(x, y, color="#2222aa", line_width=3, source=source, name='foo')

Out[4]: <bokeh.plotting.Figure at 0x1a0a038600>

def update(f, w=1, A=10, phi=0):
    if f == "sin": func = np.sin
    elif f == "cos": func = np.cos
    elif f == "tan": func = np.tan
    source.data['y'] = A * func(w * x + phi)
    source.push_notebook()

show(p)

from IPython.html.widgets import interact
interact(update, f=['sin', 'cos', 'tan'], w=(0,1000), A=(1,10), phi=(0, 10, 0.1))
```
1. Back to the basics
Code vs Software

Daniel Moisset: https://www.youtube.com/watch?v=4dlWg0B4ASw
“Code is something that runs on a Computer”
Code does not necessarily...

- Have tests
- Follow conventions
- Have documentation
- Follow processes
“Software is the programming text that is part of a deliverable”
You want to build Software...

- Maintainable
- Testable
- Deployable
Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.
Object Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Source: https://en.wikipedia.org/wiki/Object-oriented_programming
How does an object look in Python?
How does an object look in Python?

Cookie Cutters & Cookies
How does an object look in Python?
class Cookie(object):
    def __init__(self, sugar=5):
        self.sugar = sugar
    def eat(self):
        pass
    def split(self):
        pass
class Cookie(object):
    def __init__(self, sugar=5):
        self.sugar = sugar
    def eat(self):
        pass
    def split(self):
        pass
class Cookie(object):
    def __init__(self, sugar=5):
        self.sugar = sugar
    def eat(self):
        pass
    def split(self):
        pass
class Cookie(object):
    def __init__(self, sugar=5):
        self.sugar = sugar
    def eat(self):
        pass
    def split(self):
        pass
class Cookie(object):
    def __init__(self, sugar=5):
        self.sugar = sugar
    def eat(self):
        pass
    def split(self):
        pass

c = Cookie(3)
class Alfajor(Cookie):
    def __init__(self, chocolate=10, sugar=10):
        super(Alfajor, self).__init__(sugar=sugar)
        self.chocolate = chocolate

a = Alfajor(chocolate=20, sugar=30)
from sklearn import svm
data = # multiple lines to load the data
X = # multiples lines extract the features
y = # ...
clf = svm.SVC()
clf.fit(X, y)
clf.predict(...)
# multiples lines store the results
How do I write good object oriented code?
How do I write good OO code?

- DRY
- KISS
- SOLID
Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Source: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
KISS: Keep it Simple Stupid

"Simplicity is the ultimate sophistication"
Be SOLID

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
Be SOLID

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Learn OOP in Python
Coding Conventions
Coding Conventions

- “Readability counts” (PEP20)
- Spaces vs. Tabs
- Indentation rules
- Code organization
- PEP-8 is the de-facto style
PEP-8

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                           var_three, var_four)

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

http://pep8.org
```python
import os, sys

def main():
    a = 1 + 2
```
Other Topics

- Project structure
- Testing (Check out py.test!)
- Versioning and branching
- Code Reviews
- Software Development Life Cycle
Books!

- The Pragmatic Programmer
- Code Complete
- The Passionate Programmer

Images of the book covers are shown.
Talks @ Europython

- Clean Code in Python by Mariano Anaya
  (Today at 15:45 Barria 2)

- What’s the point of Object Orientation?
  by Iwan Vosloo (Thursday 11:15 A2)
2. Tips & Tricks
2. Tips & Tricks
The Collection Module
Counting
First Attempt
Using *dicts*

```python
items = ["a", "b", "a", "x", "x", "y", "c", "c", "a"]

dict_counts = {}

for i in items:
    if i in items:
        dict_counts[i] = dict_counts[i] + 1
    else:
        dict_counts[i] = 1
```
Using **dicts** *(EAFP version)*

```python
items = ['a', 'b', 'a', 'x', 'x', 'y', 'c', 'c', 'a']

item_counts = {}

for i in items:
    try:
        item_counts[i] = item_counts[i] + 1
    except KeyError:
        item_counts[i] = 1
```
We can do better
Let's use defaultdict
Using defaultdict

```python
from collections import defaultdict

items = ['a', 'b', 'a', 'x', 'x', 'y', 'c', 'c', 'a']

defaultdict(int,
  {'a': 3, 'b': 1, 'c': 2, 'x': 2, 'y': 1})
```

Let’s use Counter
Using Counter

```python
from collections import Counter

items = ['a', 'b', 'a', 'x', 'x', 'y', 'c', 'c', 'a']

item_counts = Counter(items)
print(item_counts)

Counter({'a': 3, 'x': 2, 'c': 2, 'b': 1, 'y': 1})
```
Counter’s extra goodies
Extra goodies

```python
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c.most_common()
>>> c.values()
>>> c + d
>>> c - d
>>> c & d  # intersection: min(c[x], d[x])
>>> c | d  # union: max(c[x], d[x])
```
Counter is a class
Classes can be extended
from collections import Counter

class PMF(Counter):
    def normalize(self):
        total = float(sum(self.values()))
        for key in self:
            self[key] /= total
Counter based PMF

```python
from collections import Counter

class PMF(Counter):
    def normalize(self):
        total = float(sum(self.values()))
        for key in self:
            self[key] /= total

    def __init__(self, *args, **kwargs):
        super(PMF, self).__init__(...)
        self.normalize()
```
On Counting and Python

- Use the and extend Counter class
- Awesome article from @TreyHunner on Counting [1].

Named Tuples
Named Tuples

- Code around the `dict`, `tuples` or `lists`
- Never know what to expect
- Code becomes hard to read
Example

```python
from math import sqrt
pt1 = (1.0, 5.0)
pt2 = (2.5, 1.5)

line_length = sqrt((pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2)
```

Source: http://stackoverflow.com/questions/2970608/what-are-named-tuples-in-python
Example

\[
\text{line} \_ \text{length} = \sqrt{(pt1[0] - pt2[0])^2 + (pt1[1] - pt2[1])^2}
\]

Example

```python
from collections import namedtuple
from math import sqrt

Point = namedtuple('Point', 'x y')
pt1 = Point(1.0, 5.0)
pt2 = Point(2.5, 1.5)

line_length = sqrt((pt1.x - pt2.x)**2 + (pt1.y - pt2.y)**2)
```

Source: http://stackoverflow.com/questions/2970608/what-are-named-tuples-in-python
It has cool methods

```python
_asdict
```
Return a new OrderedDict which maps field names to their values

```python
_make(iterable)
```
Class method that makes a new instance from an existing sequence or iterable.
Extending NamedTuple

```python
from collections import namedtuple

_HotelBase = namedtuple(
    'HotelDescriptor',
    ['cluster_id', 'trust_score', 'reviews_count',
     'category_scores', 'intensity_factors'],
)

class HotelDescriptor(_HotelBase):
    def compute_prior(self):
        if not self.trust_score or not self.reviews_count:
            raise NotEnoughDataForRanking("...")
        return _compute_prior(self.trust_score,...)
```
2. Tips & Tricks

2.1 Iterators & Iterables
l = [1, 2, 3, 4]
for i in x:
    print(x)
A generator expression is a generator function.

A generator function is always a generator expression.

A generator is typically an iterator.

An iterator is a generator function.

Lazily produce the next value.

By Vincent Driessen - Source: http://nvie.com/posts/iterators-vs-generators/
an iterator

always is

(iter())

Lazily produce the next value

(next())

(an) iterable
an iterator

Lazily produce the next value

always is

(an) iterable

iter()

next()
an iterator

Lazily produce the next value

next()

always is iter()

produces comprehension

(an) iterable
an iterator

(an) iterable

always is iter()

next()

produces container (list, dict, etc)

Lazily produce the next value

comprehension
assert 1 in [1, 2, 3]
assert 1 in {1, 2, 3}
an iterator

() iter

next()

Lazily produce the next value

always is

iter()

typically is

(an) iterable

produces

container (list, dict, etc)

comprehension
l = [1, 2, 3, 4]
x = iter(l)
y = iter(l)
l = [1, 2, 3, 4]
x = iter(l)
y = iter(l)

type(l)
>> <class 'list'>
type(x)
>> <class 'list_iterator'>

Lazily produce the next value

an iterator
next()

always is
iter()

(typically is
(an) iterable

produces
container (list, dict, etc)

comprehension
l = [1, 2, 3, 4]
x = iter(l)
y = iter(l)

```python
type(l) >> <class 'list'>
type(x) >> <class 'list_iterator'>
next(x) >> 1
next(y) >> 1
next(y) >> 2
```
an iterator

iter() 

Lazily produce the next value 

next()

always is

(an) iterable

typically is

container (list, dict, etc)

produces

comprehension

l = [1, 2, 3, 4]

for e in l:
    print(e)
Iterables

- An **iterable** is any object that can return an **iterator**
- Containers, files, sockets, etc.
- Implement `__iter__()`.
- Some of them may be infinite
- The `itertools` contain many helper functions
class InverseReader(object):
    def __init__(self):
        with open('file.txt') as f:
            self.lines = f.readlines()
        self.index = len(self.lines) - 1

    def __iter__(self):
        return self

    def next(self):  # Python 3 __next__
        self.index -= 1
        if self.index < 0:
            raise StopIteration
        return self.lines[self.index]
ir = InverseReader()

for line in ir:
    print(line)
a generator expression is a generator

an iterator is always

Lazily produce the next value

next()

iter()

(an) iterable typically is container produces comprehension

a generator function
a generator expression

an iterator

next()

Lazily produce the next value

always is

iter()

(an) iterable
an iterator

(a) iterable

always is

an iterator

next()
a generator expression

an iterator

always is

(an) iterable

Lazily produce the next value

next()
an iterator (an iterable)

always is

next() Lazily produce the next value

iter()
A generator expression is a generator. An iterator always is a generator. An iterator is (an) iterable. The `next()` function lazily produces the next value.
a generator expression

is a generator

is a generator function
numbers = [x for x in range(1, 10)]
squares = [x * x for x in numbers]
type(squares)
# list
numbers = [x for x in range(1, 10)]
squares = [x * x for x in numbers]
type(squares)
# list

lazy_squares = (x * x for x in numbers)
lazy_squares
# <generator object <genexpr> at 0x104c6da00>
is a generator expression

numbers = [x for x in range(1, 10)]
squares = [x * x for x in numbers]
type(squares)
# list

lazy_squares = (x * x for x in numbers)
lazy_squares
# <generator object <genexpr> at 0x104c6da00>

next(lazy_squares)
# 1
next(lazy_squares)
# 4

x
```python
numbers = [x for x in range(1, 10)]
squares = [x * x for x in numbers]
type(squares)  # list

lazy_squares = (x * x for x in numbers)
lazy_squares  # <generator object <genexpr> at 0x104c6da00>

next(lazy_squares)  # 1
next(lazy_squares)  # 4

lazy_squares = (x * x for x in numbers)
for x in lazy_squares:
    print(x)
```

A generator expression is a generator.
a generator function is a generator
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

is a generator function
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

f = fib()
next(f)
# 1
next(f)
# 1
next(f)
# 2
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

for x in islice(fib(), 0, 3):
    print x
# 1
# 1
class HdfsLineSentence(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        stream = self.source.open('r')
        for line in stream:
            cid, s = line.split('	')
            # decode and do some work with s
            yield s

sentences = HdfsLineSentence(...)  # Insert the source
for s in sentences:
    print(s)
An iterator is always an iterable. A generator expression is a generator, and a generator function always is a generator. Lazily produce the next value typically is a container that produces (an) iterable.
What does it have to do with Data Processing?
Unknown amount of data
Not enough memory
Data streaming via lazy evaluation
Data processing pipelines through iterables
Chaining iterables
class HdfsLineSentence(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        stream = self.source.open('r')
        for line in stream:
            cid, s = line.split('	')
            # decode and do some work with s
            yield s

sentences = HdfsLineSentence(...)
for s in sentences:
    print s
class FilterComment(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        for s in self.source:
            if s[0] != "#":
                yield s
class FilterComment(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        for s in self.source:
            if s[0] != "#":
                yield s

sents = FilterComment(HdfsLineSentence(source))
for s in sents:
    print(s)
class FilterComment(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        for s in self.source:
            if s[0] != '#':
                yield s

sents = FilterComment(HdfsLineSentence(source))
for s in sents:
    print s
class FilterComment(object):
    def __init__(self, source):
        self.source = source

    def __iter__(self):
        for s in self.source:
            if s[0] != '#':
                yield s

sents = FilterComment(HdfsLineSentence(source))
for s in sents:
    print s
def filter_comment(source):
    for s in source:
        if s[0] != "#":
            yield s

sents = filter_comment(HdfsLineSentence(source))

for s in sents:
    print(s)
Talks @ Europython

- Iteration, iteration, iteration by John Sutherland (Friday 15:45 Barria 1)
3. Conclusions
Data Scientists / Engineers / ML Developers should learn...
collections and itertools modules
Iterables and iterators for data processing pipelines
Object oriented programming
Good software engineering practices
Credits


PMF Class based on  Vik Paruchuri’s https://www.dataquest.io/blog/python-counter-class/

Ideas from Iterables taken from RaRe Technologies blog; http://rare-technologies.com/data-streaming-in-python-generators-iterators-iterables/

Iterators and Iterables based on work of Vincent Driessen: http://nvie.com/posts/iterators-vs-generators/

Spaguetti:  https://www.flickr.com/photos/129610671@N02/16633987421/ (CC BY-NC-ND 2.0) vision.communicate

Autovification: Credit: AV Dezign
https://www.flickr.com/photos/91345457@N07/22666878846/ (CC BY-NC-ND 2.0)

Counter image: Dean Hochman Source:
https://www.flickr.com/photos/17997843@N02/24061690099/“ (CC BY-NC-ND 2.0)

Cookies: Source Wikipedia
https://en.wikipedia.org/wiki/File:R%C5%AFzn%C3%A9_druhy_cukrov%C3%AD_(2).jpg (CC BY 3.0)
We are Hiring!
Visit our table for more info!
Thanks!

Any questions?

You can find me at:
@mfcabrera
mfcabrera@gmail.com