# Spark:
# Spark Concepts and Python Intro

## Lecture 03

Marilson Campos

UCSC Extension - 2022

Spark

# Spark Vs MR

Map/Reducer

MR Job 1

HDFS -> MR Job 2

....

HDFS -> MR Job n

Data is written and read multiple times.

Spark

Op1 -> Op2 … -> OpN

Round trips to disk are eliminated.

# Spark
# Spark RDD

RDD → Resilient Distributed Dataset

- Why Resilient ? Because its Fault tolerant and can rebuild itself if parts of the data are removed from memory.

- Distributed because is spread across the nodes of the cluster.

- Dataset: It appears to the user as a local dataset but in reality, the data is distributed.

# Spark
# Spark Cluster

SparkContext → Is the handle to connect to a cluster.
We will use the variable named as 'sc'.

Ways to load data into RDD:

1. sc.parallelize → converts a local collection to an RDD.
   Used mostly in test code.

2. sc.textFile → reads file from HDFS or local filesystem and returns an RDD.

3. sc.read.format(<format_name>).load(<file_path>)

Spark

# Functional Programming

Anonymous functions.

```
words = input.flatMap(lambda x: x.split())
```

“lambda” is a codename to tell the interpreter that no name is needed. So, this function is only used by the enclosing function.
In this example the function “flatMap”

# Spark
# Transformations and Actions

Transformation → Operation that returns an RDD.

**Lazy** operation => Executed across the nodes.

Action → Operation that returns anything BUT an object of type RDD.

**Eager** operation => Depends on the command.

Can be Executed on the driver and/or across the nodes.

# Spark
# Word Count

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName(”WC_01")
sc = SparkContext(conf=conf)

input = sc.textFile(”article.txt")
words = input.flatMap(lambda x: x.split()) ← transformation
wordCounts = words.countByValue() ← action

for word, count in wordCounts.items():
        print(word + " " + str(count))
```

Spark

# SetMaster options

**SetMaster(str_value)**

Examples:

**local** → Runs Spark locally with one worker thread
(i.e. no parallelism at all).

**local[K]** → Runs Spark locally with K worker threads
ideally, set this to the number of cores on your machine – 1 in case of VMs.

**local[K,F]** → Runs Spark locally with K worker threads and F maxFailures
See spark.task.maxFailures for an explanation of this variable

**local[*]** → Runs Spark locally with as many worker threads as logical cores on your machine.

# Spark
# Python Lambda

```
# Regular python function
def make_tuple(x):
        return (x, 1)

# Lambdas are functions with no name. Used in place.
lambda x: (x, 1)
```

In Spark we use functions that expect functions as parameters.

For example, the map operator:

```
# Using anonymous functions
recs = words.map(lambda x: (x, 1))

# Using regular functions
recs = words.map(make_tuple)
# Notice that we are passing the function not calling it like make_tuple(1)!
```

Spark

# SetMaster options (cont)

**SetMaster(str_value)**

Examples:

**local[*,F]** → Runs Spark locally with as many worker threads as logical cores on your machine and F maxFailures.

**spark://HOST:PORT** → Connects to the given Spark standalone cluster master. The port must be whichever one your master is configured to use. (7077 is the default port)

**spark://HOST1:PORT1,HOST2:PORT2** → Connects to the given Spark standalone cluster with standby masters with Zookeeper. The list must have all the master hosts in the high availability cluster set up with Zookeeper. The port must be whichever each master is configured to use.

Spark

# Transformations

**map(f)** → 'f' is a function that transform a single element into another element.

Example:

```
counts = (
   txt_rdd.flatMap(lambda line: line.split(" ")) # word list
      .map(lambda word: (word, 1)) # list of tuples
      .reduceByKey(lambda x, y: x + y)
   )

counts.saveAsTextFile("hdfs://...")
```

Map receives each word and generates a pair (tuple) containing the 'word' and the number one. Ex: ('Apple', 1)

Spark

# Transformations

**flatMap(f)** → 'f' is a function that transform a single element into `a list of elements.

Example:

```
counts = (
  txt_rdd.flatMap(lambda line: line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda x, y: x + y)
  )

counts.saveAsTextFile("hdfs://...")
```

FlatMap splits each line into a list. Then it appends the elements of the list into the result.
Only a single list of words gets returned.

Spark

# Transformations

**filter(f)** → 'f' is a function examines a single element and returns a Boolean. (f is a predicate). Filter returns only elements for which 'f' is True.

Example:

```
counts = (
  txt_rdd.flatMap(lambda line: line.split(" "))
     .filter(lambda word: word not in ("the", "a"))
     .map(lambda word: (word, 1))
     .reduceByKey(lambda x, y: x + y)
  )
counts.saveAsTextFile("hdfs://...")
```

Filter only returns words that are not part of the "stop words". Stop works are words in NLP that are typically removed from processing.

Spark

# Transformations

**mapPartition(f)** → Converts each partition of the source RDD into many elements of the result (possibly none).
In mapPartition(), the map() function is applied on each partition simultaneously. It's a map operation over partitions and not over the elements of the partition.

**mapPartitionWithIndex(f)** → Same as MapPartitions besides that 'f' function receives an additional parameter index representing the partition number.

Spark

# Transformations

**union(other_rdd)** → returns the union of both RDDs.

Example:

```
rdd_union = rdd1.union(rdd2).union(rdd3)
```

We get the elements in any of the RDDs in new RDD.

The key rule of this function is that the input RDDs should be of the same type.

Spark

# Transformations

**intersection(other_rdd)** → returns a RDD with elements on both RDDs.

Example:

```
rdd_common = rdd1.intersection(rdd2)
```

We get only the common elements of both RDDs in new RDD.

Spark

# Transformations

**distinct()** → returns a RDD without duplicates.

Example:

```
counts = (
   txt_rdd.flatMap(lambda line: line.split(" "))
      .distinct()
      .map(lambda word: (word, 1))
      .reduceByKey(lambda x, y: x + y)
   )
counts.saveAsTextFile("hdfs://...")
```

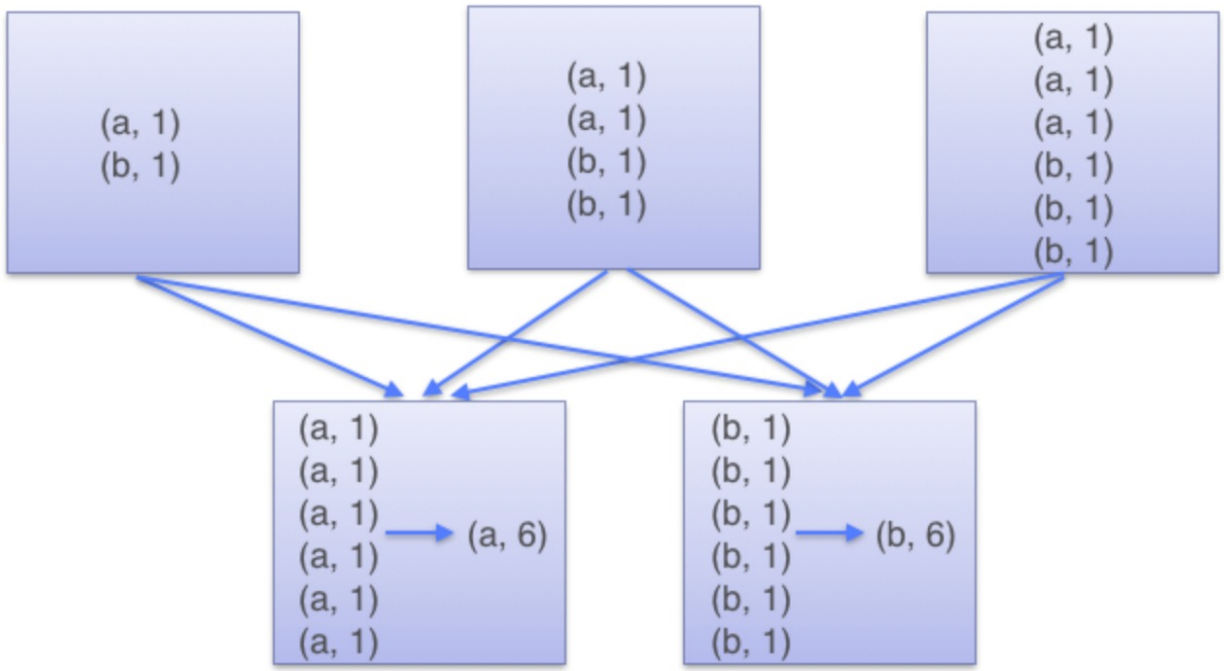
The 'distinct' operator will break the 'word count' program as all counts with be equal to one.

Spark

# Transformations

**groupByKey(f)** → Operate into pairs and uses 'f' to combine two elements but does not work will large datasets.

Example:



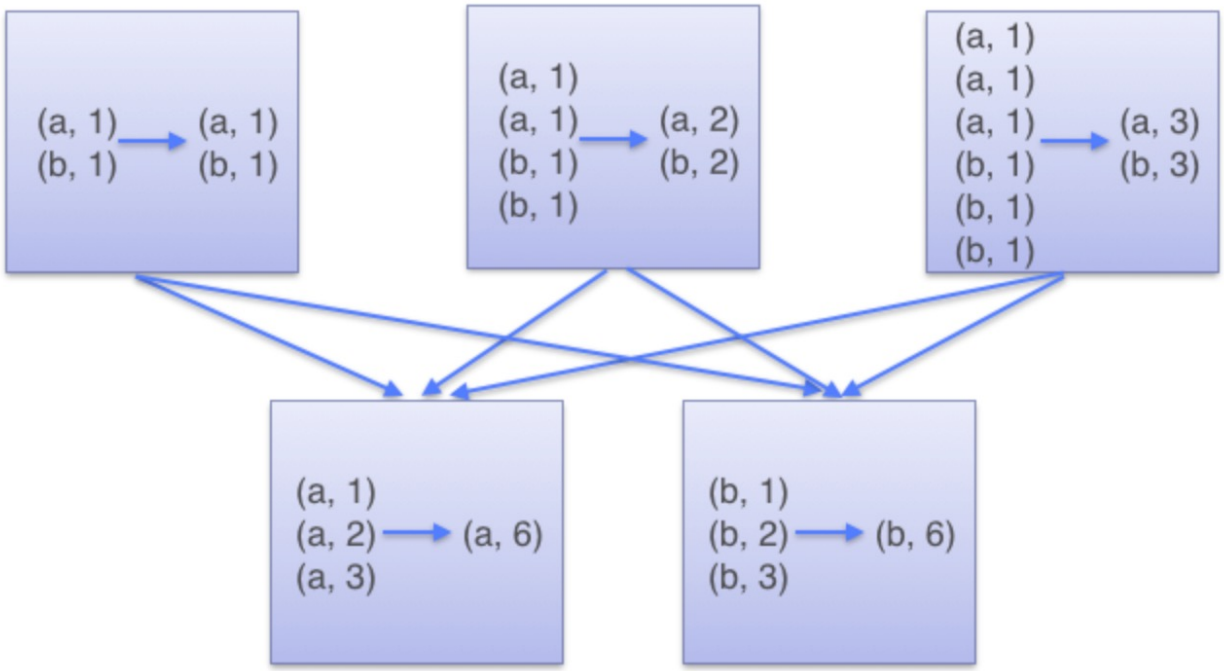

All pairs need to travel thru the network. It **should be avoided** because consumes lots of memory.

Spark

# Transformations

**reduceByKey(f, [num_tasks])** → Operate into pairs and uses 'f' to combine two elements

Example:




Its like using combiner on Map/Reduce, reduces the data moving in the network.

Spark

# Transformations

**sortByKey(ascending=True, num_parts=None, key_func)** →
Operate into pairs of (key, value) and uses ‘f’ to order the two elements using the key. If key_func is provided we can alter the key that will be used for sorting.

Example:

```
names_freq.sortByKey().collect()
[
('AADEN', 18),
('AADEN', 11),
('AADEN', 10),
('AALIYAH', 50)…
```

Spark

# Transformations

**join(other_dataset)** → Operate into pairs of (key, value) joins two RDDs using the key

Example:

```
rdd1 = sc.parallelize([("foo", 1), ("bar", 2), ("baz", 3)])
rdd2 = sc.parallelize([("foo", 4), ("bar", 5), ("bar", 6)])
rdd1.join(rdd2)

[('bar', (2, 5)), ('bar', (2, 6)), ('foo', (1, 4))]
```

Spark

# Transformations

**coalesce(num_parts)** → Return a new RDD that is reduced into num_parts partitions.

Example:

```
my_list = [1, 2, 3, 4, 5]
sc.parallelize(my_list, 3).glom().collect()
[[1], [2, 3], [4, 5]]

sc.parallelize(my_list,3).coalesce(1).glom().collect()
[[1, 2, 3, 4, 5]]
```

Spark

# Actions

**count()** →

Returns the number of elements in the RDD.

**collect()** →

Returns all the elements of RDD as local collection.
(In our case a python collection)

**take(n)** →

Returns 'n' elements from the RDD.
Cannot assume elements follow any order.

Notice that all values returned are not RDD!

Spark

# Actions

**top(n)** →

Returns the top 'n' elements from the RDD.

**countByValue()** →

Returns a collection of tuples with a value and its frequency.

**Example:**

If a rdd called 'nums' contains :{1, 2, 2, 3, 4, 5, 5, 6} then, the command "nums.countByValue()" will produce

{(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

Spark

# Actions

**reduce(combine_f)** → Combine the elements in the RDD by applying 'combine_f' continuously to return a **single value**.

**fold(zero_value, combine_f)** → Combine the elements in the RDD by applying 'combine_f' continuously.

Example:

```
sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
```

**<u>Fold</u> returns valid result even with empty input.**

Spark

# Actions

**aggregate(zero, seq_op, comb_op)** → Aggregate the elements of each partition, and then combine the results for all partitions, using a given combine functions and a neutral "zero value."

Example:

```
zero = (0, 0)
seq_op = (lambda t, y: (t[0] + y, t[1] + 1))
comb_op = (lambda t1, t2: (t1[0] + t2[0], t1[1] + t2[1]))
sc.parallelize([1, 2, 3, 4]).aggregate(zero, seq_op, comb_op)
```

Produces:

(10, 4)

Spark

# Actions

**foreach(f)** → perform the operation 'f' to each element of the RDD after its retrieved to the local computer. (not parallelized!)

Also, notice that it will not return anything.

Typically, this command is used for printing results.

Spark

# Reduce vs ReduceByKey

**Reduce** →

Is an action and returns a single value.
(action)

**ReduceByKey** →

Is a transformation and returns an RDD.
(transformation)

# Spark
# Word Count v2

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("WordCount")
sc = SparkContext(conf = conf)

input = sc.textFile(”/data/book_text.txt")
words = input.flatMap(lambda x: x.split())

wordCounts = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
wordCountsSorted = wordCounts.map(lambda x: (x[1], x[0])).sortByKey()
word_counts = wordCountsSorted.collect()

for pair in word_counts :
   count = str(pair[0])
   word = pair[1]
   print(word + ": " + count)
```

Spark

# Understanding Spark Docs in Scala

**fold() :**

fold[A](z: A)(f: (A, A) => A): A

Function fold needs 2 parameters:

- a zero value to start from.
- a function that receives two elements of type A and returns one element of type A.

Spark

# Understanding Spark Docs in Scala

## Currying

fold[A](z: A)(f: (A, A) => A): A

A function can have multiple sets of parameters.
Each time you call with one set it will return another function with the params removed

foldWithZero = fold(0)
Doc for the function
foldWithZero[A](f: (A, A) => A): A

result = foldWithZero(f)

Spark

# Parallel Reduce Operations

**fold() :**

fold[A](z:  A)(f: (A, A) => A): A

Combine pairs of elements In parallel using “f” until there is a single value and returns the value.

Difference between fold and reduce is that fold is better prepared to handle empty collections

Spark

# Parallel Reduce Operations

**aggregate() :**

Aggregate[A, B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B

Parallelizable and allows to change the return value.

Python

# Python for Big Data

# Why Python

Python

→ Clean language

→ Easy create scripts

Python vs Java (JVM)

→ Python is used to “glue” data pipelines steps.

→ Java is used for building the step as it has direct access to Hadoop M/R Api.

# Some Resources

**Start Here:**
*Think Python: How to Think Like a Computer Scientist*
by Allen B. Downey

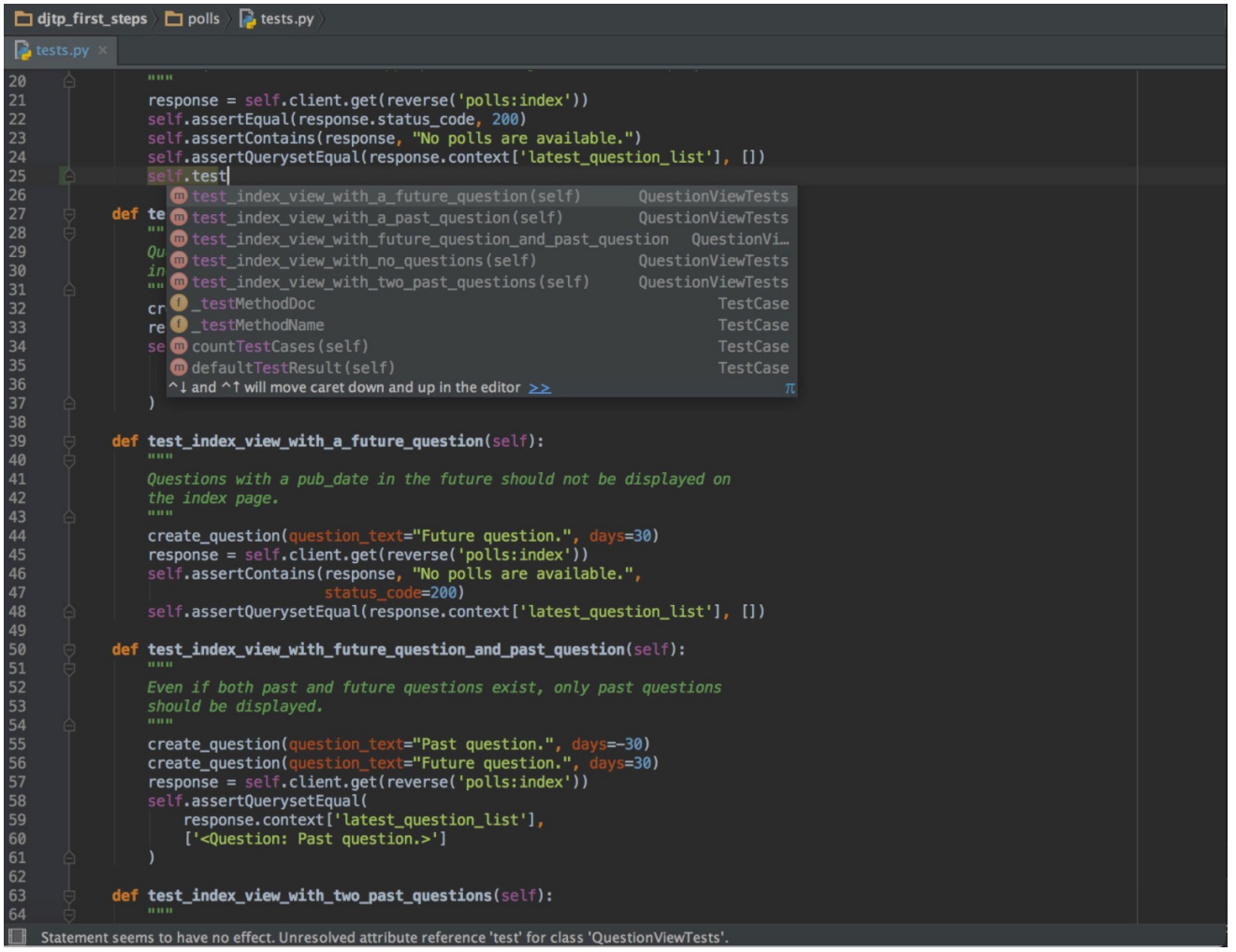
**Data Science with Python**

*Python Data Science Handbook: Essential Tools for Working with Data*
by Jake VanderPlas

*Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*
by Wes McKinney

## Python for Big Data

# Lab PyCharm install



Python

# Zen Of Python

**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!**

Python

# How to test performance

```
>>> import timeit
>>> timeit.timeit("x = 2 + 2")
0.034976959228515625
>>> timeit.timeit("x = sum(range(10))")
0.92387008666992188
```

Python

# Lists in Python

```
List_a = []

List_b = [“a”, “b”, ”c”, 1]

for elm in list_b:
    print(elm)

List_b.append(100)

List_b.extend([3,4])
```

Python

# Dictionaries in Python

```
dict_a = dict()

dict_b = {‘a’:100, ‘b’:200, ‘c’:300}

For elm in dict_b:
    print(elm)

dict_b[‘x’] = 100

for key, value in enumerate(dict_b):
    print(“{0}={1}”.format(key, value))
```

Python

# Sets in Python

```
set_a = set()

set_b = set([“a”, "b”, “c”])

for elm in set_b:
    print(elm)
```

Python

# Tuples in Python

```
rec = ('Mark’, 31, ‘male’)
rec[0]
Mark
Rec[1]
31

# --- Dataclasses - New in Python ----

from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    gender: str

p1 = Person('jon', 23, 'male’)
p1.name
p1.age
```

# Python
# Idioms

Bad

```
if x <= y and y <= z:
    return True
```

Good

```
if x <= y <= z:
    return True
```

# Python
# Idioms

Bad

```
if name: print(name)
print(address)
```

Good

```
if name:
    print(name)
print(address)
```

# Python
# Idioms

Bad

```
if should_raise_shields() == True:
    raise_shields()
    print('Shields raised')
```

Good

```
if should_raise_shields():
    raise_shields()
    print('Shields raised')
```

# Python
# Idioms

## Bad

```
is_generic_name = False
name = 'Tom'
if name == 'Tom' or name == 'Dick' or name == 'Harry':
    is_generic_name = True
```

## Good

```
name = 'Tom'
is_generic_name = name in ('Tom', 'Dick', 'Harry')
```

# Python
# Idioms

Bad

```
foo = True
value = 0

if foo:
    value = 1
```

Good

```
foo = True

value = 1 if foo else 0
```

# Python
# Idioms

Bad

```
my_container = ['Larry', 'Moe', 'Curly']
index = 0
for element in my_container:
    print ('{} {}'.format(index, element))
    index += 1
```

Good

```
my_container = ['Larry', 'Moe', 'Curly']
for index, element in enumerate(my_container):
    print ('{} {}'.format(index, element))
```

# Python
# Idioms

Bad

```
my_list = ['Larry', 'Moe', 'Curly']
index = 0
while index < len(my_list):
    print (my_list[index])
    index += 1
```

Good

```
my_list = ['Larry', 'Moe', 'Curly']
for element in my_list:
    print (element)
```

# Python
# Idioms

Bad

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Good

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

# Python
# Idioms

Bad

```
foo = 'Foo'
bar = 'Bar'
temp = foo
foo = bar
bar = temp
```

Good

```
foo = 'Foo'
bar = 'Bar'
(foo, bar) = (bar, foo)
```

# Python
# Idioms

## Bad

```
book_info = ' The Three Musketeers: Alexandre Dumas'
formatted_book_info = book_info.strip()
formatted_book_info = formatted_book_info.upper()
formatted_book_info = formatted_book_info.replace(':', ' by')
```

## Good

```
book_info = ' The Three Musketeers: Alexandre Dumas'
formatted_book_info = book_info.strip().upper().replace(':', ' by')
```

# Python
# Idioms

## Bad

```
result_list = ['True', 'False', 'File not found']
result_string = ''
for result in result_list:
    result_string += result
```

## Good

```
result_list = ['True', 'False', 'File not found']
result_string = ''.join(result_list)
```

# Python
# Idioms

Bad

```
some_other_list = range(10)
some_list = list()
for element in some_other_list:
    if is_prime(element):
        some_list.append(element + 5)
```

Good

```
some_other_list = range(10)
some_list = [element + 5
             for element in some_other_list
             if is_prime(element)]
```

# Python
# Idioms

Bad

```
log_severity = None
if 'severity' in configuration:
    log_severity = configuration['severity']
else:
    log_severity = 'Info'
```

Good

```
log_severity = configuration.get('severity', 'Info')
```

# Python
# Idioms

Bad

```
user_email = {}
for user in users_list:
    if user.email:
        user_email[user.name] = user.email
```

Good

```
user_email = {user.name: user.email
12 highlighters----------------------------------------
              for user in users_list if user.email}
              ----------------------------------------
```

# Python Idioms

Bad

```
def get_both_popular_and_active_users():
    # Assume the following two functions each return a
    # list of user names
    most_popular_users = get_list_of_most_popular_users()
    most_active_users = get_list_of_most_active_users()
    popular_and_active_users = []
    for user in most_active_users:
        if user in most_popular_users:
            popular_and_active_users.append(user)

    return popular_and_active_users
```

Good

```
def get_both_popular_and_active_users():
    # Assume the following two functions each return a
    # list of user names
    return(set(
        get_list_of_most_active_users()) & set(
            get_list_of_most_popular_users()))
```

Python

# Idioms - Bad

Bad

```
users_first_names = set()
for user in users:
    users_first_names.add(user.first_name)
```

Good

```
users_first_names = {user.first_name for user in users}
```

# Python Idioms

## Bad

```
def print_employee_information(db_connection):
    db_cursor = db_connection.cursor()
    results = db_cursor.execute('select * from employees').fetchall()
    for row in results:
        # It's basically impossible to follow what's getting printed
        print(row[1] + ', ' + row[0] + ' was hired '
        'on ' + row[5] + ' (for $' + row[4] + ' per annum)'
        ' into the ' + row[2] + ' department and reports to ' + row[3])
```

# Python Idioms

Good

```
from collections import namedtuple

EmployeeRow = namedtuple('EmployeeRow', ['first_name',
'last_name', 'department', 'manager', 'salary', 'hire_date'])

EMPLOYEE_INFO_STRING = '{last}, {first} was hired on {date} \
 ${salary} per annum) into the {department} department and reports to \
ager}'

def print_employee_information(db_connection):
    db_cursor = db_connection.cursor()
    results = db_cursor.execute('select * from employees').fetchall()
    for row in results:
        employee = EmployeeRow._make(row)

        # It's now almost impossible to print a field in the wrong place
        print(EMPLOYEE_INFO_STRING.format(
            last=employee.last_name,
            first=employee.first_name,
            date=employee.hire_date,
            salary=employee.salary,
            department=employee.department,
            manager=employee.manager))
```

# Python
# Idioms

Bad

```
(name, age, temp, temp2) = get_user_info(user)
if age > 21:
    output = '{name} can drink!'.format(name=name)
# "Wait, where are temp and temp2 being used?"
```

Good

```
(name, age, _, _) = get_user_info(user)
if age > 21:
    output = '{name} can drink!'.format(name=name)
# "Clearly, only name and age are interesting"
```

# Python
# Idioms

## Bad

```
list_from_comma_separated_value_file = ['dog', 'Fido', 10]
animal = list_from_comma_separated_value_file[0]
name = list_from_comma_separated_value_file[1]
age = list_from_comma_separated_value_file[2]
```

## Good

```
list_from_comma_separated_value_file = ['dog', 'Fido', 10]
(animal, name, age) = list_from_comma_separated_value_file
```

# Python
# Idioms

Bad

```
def calculate_mean(value_list):
    return float(sum(value_list) / len(value_list))

def calculate_median(value_list):
    return value_list[int(len(value_list) / 2)]

def calculate_mode(value_list):
    return Counter(value_list).most_common(1)[0][0]
```

Good

```
def calculate_staistics(value_list):
    mean = float(sum(value_list) / len(value_list))
    median = value_list[int(len(value_list) / 2)]
    mode = Counter(value_list).most_common(1)[0][0]
    return (mean, median, mode)

(mean, median, mode) = calculate_staistics([10, 20, 20, 30])
```

# Python
# Idioms

```
class Product(object):
    def __init__(self, name, price):
        self.name = name
        self._price = price

    @property
    def price(self):
        # now if we need to change how price is calculated, we can do it
        # here (or in the "setter" and __init__)
        return self._price * TAX_RATE

    @price.setter
    def price(self, value):
        # The "setter" function must have the same name as the property
        self._price = value
```

Use properties

# Python
# Idioms

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '{0}, {1}'.format(self.x, self.y)

p = Point(1, 2)
print (p)

# Prints '1, 2'
```

Implement __str__ functions

# Python
# Idioms

Bad

```
file_handle = open(path_to_file, 'r')
for line in file_handle.readlines():
    if raise_exception(line):
        print('No! An Exception!')
```

Good

```
with open(path_to_file, 'r') as file_handle:
    for line in file_handle:
        if raise_exception(line):
            print('No! An Exception!')
```

# Python
# Idioms

Bad

```
def is_prime(number):
    """Mod all numbers from 2 -> number and return True
    if the value is never 0"""

    for candidate in range(2, number):
        if number % candidate == 0:
            print(candidate)
            print(number % candidate)
            return False

    return number > 0
```

Good

```
def is_prime(number):
    """Return True if number is prime"""

    for candidate in range(2, number):
        if number % candidate == 0:
            return False
```

Comments are used to describe **what** you do

# Python Idioms

Bad

```
from foo import *
```

Good

```
from foo import (bar, baz, qux,
                 quux, quuux)

# or even better...
import foo
```

# Python
# Idioms

Bad

```
from gizmo.client.interface import Gizmo
from gizmo.client.contrib.utils import GizmoHelper
```

Good

```
# __init__.py:

from gizmo.client.interface import Gizmo
from gizmo.client.contrib.utils import GizmoHelper

#client code:
from gizmo import Gizmo, GizmoHelper
```

Python

# Functional Programming

## Higher order function ➤

Function that have parameters that are also functions or return a function.

In the context of Spark, most transformations have parameters that are functions.
Example: Map.

Python

# Functional Programming - Closures

```
class Adder(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, m):
        return self.n + m
add5_i = Adder(5)        # "instance" or "imperative"
```

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder
add5_f = make_adder(5)  # "functional"
```

Python

# Functional Programming

We will continue Functional programming and Spark next week.