

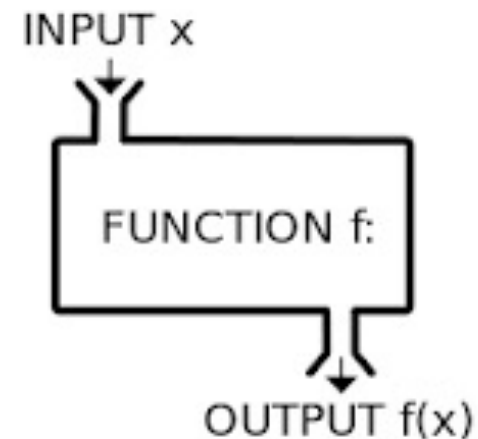
SPARK ESSENTIALS

Review of concepts

- What is Functional Programming - Python
 - Lambda
 - Map, Filter, Reduce
- List Comprehensions
- Sliced

Functional Programming

- Functional programming is a **style** whose underlying model of computation is the *function*.
- Functions take input and produce output, without any side effects.
- No state
- Immutable data
- Function as first-class citizen
- Recursion
- Purity ...



<https://marcobonzanini.com/2015/06/08/functional-programming-in-python/>



Python Functional Programming

Python is a multi paradigm programming language. As a Python programmer why uses functional programming in Python?

Python is not a functional language but have a lot of features that enables us to applies functional principles in the development, turning our code more elegant, concise, maintainable, easier to understand and test.

Lambda function

- Syntax : **lambda argument_list: expression**
 - argument_list: comma separated list of arguments
 - expression is an arithmetic expression using these arguments.
- The function can be assigned to a variable

Lambda function

Example

```
>> f = lambda x,y: x + y
```

```
>> f(1,2)
```

```
Out[1]: 3
```

- Only **one** expression in the lambda body
- Advantage of the lambda can be seen when it is used in combination with other functions (e.g. map, reduce, filter)

The map() function

- Syntax: **r = map(func, seq)**
 - *func* is the name of a function
 - Seq is a sequence (e.g. a list)
- *Map* applies the function *func* to all the elements of the sequence *seq* and returns a **new list** with the elements changed by *func*

The map() function

Example(1):

```
>> nums = [1,2,3,4]
>> squares = map(lambda x: x * x, nums)
>> print squares
Out[1]: [1, 4, 9, 16]
```

Example(2):

```
>> Celsius = [39.2, 36.5, 37.3, 37.8]
>> Fahrenheit = map(lambda x: (float(9) / 5) * x + 32, Celsius)
>> print Fahrenheit
Out[1]: [102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]
>> C = map(lambda x: (float(5) / 9) * (x - 32), Fahrenheit)
>> print C
Out[1]: [39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997] >>>
```


The filter() function

- Syntax: `f = filter(function, list)`
 - *Function* that returns true or false – applied to every element of the list
 - *List to applied the function*
- *Filter* returns a **new list** with the “True” elements returned from applying the function to the list

The filter() function

Example:

```
>> fib= [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>> result= filter(lambda x: x % 2, fib)
>> print result
Out[1]: [1, 1, 3, 5, 13, 21, 55]
```

The reduce() function

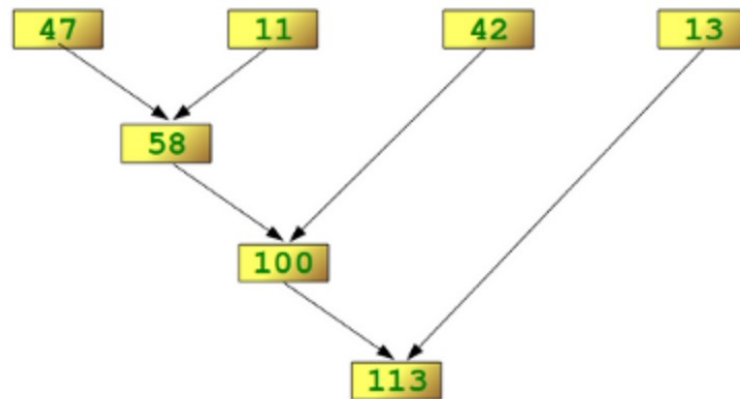
- Syntax: **r = reduce(func, seq)**
 - *func* is the name of a function
 - Seq is a sequence (e.g. a list)
- *Reduce* continually applies the function *func* to the sequence *seq*, and returns a single value.

The reduce() function

Example:

```
>> result= reduce(lambda x,y: x + y, [47, 11, 42, 13])
```

Out[1]: 113



List Comprehensions

```
doubled_odds = []  
for n in numbers:  
    if n % 2 == 1:  
        doubled_odds.append(n * 2)
```

```
doubled_odds = [n * 2 for n in numbers if n % 2 == 1]
```

We copy-paste from a `for` loop into a list comprehension by:

1. Copying the **variable assignment** for our **new empty list**
2. Copying **the expression that we've been** `append` **-ing** into this new list
3. Copying **the** `for` **loop line**, excluding the final `:`
4. Copying **the** `if` **statement line**, also without the `:`

List slicing

- Slicing: Extracting parts of list

- Syntax:

```
list[start:end]  
list[start:]  
list[end:]  
list[:]
```

- `start` inclusive and excluding `end`

- Slicing returns a new list

```
>>> colors = ['yellow', 'red', 'blue', 'green', 'black']  
>>> colors[0:]  
['yellow', 'red', 'blue', 'green', 'black']  
  
>>> colors[:4]  
['yellow', 'red', 'blue', 'green']  
  
>>> colors[1:3]  
['red', 'blue']  
  
>>> colors[:]  
['yellow', 'red', 'blue', 'green', 'black']
```

Spark Essentials

- SparkContext
- RDDs
- Operations:
 - Basic transformations
 - Basic actions
- Persistence

SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells and notebooks as variable `sc`
- In standalone programs, you'd make your own
 - (Last slide)
 - Now we are assuming that we are working either in the shell or with notebooks

Creating RDDs

Turn a local collection into an RDD

```
>> rdd_1 = sc.parallelize([1, 2, 3])
```

Load text file from local FS, HDFS, or S3

```
>> rdd_2 = sc.textFile("file.txt")
```

```
>> rdd_3 = sc.textFile("directory/*.txt")
```

```
>> rdd_4 =  
sc.textFile("hdfs://namenode:9000/path/file"  
)
```

Transforming an existing RDD

```
>> rdd_5 = rdd2.filter(function)
```

RDD operations

- *Transformations*

- lazy operation to build RDD

- *Actions*

- Computes a result based on existing RDD or write it to storage

Transformations

```
map(func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...
```

Actions

```
reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...
```



= easy



= medium

Essential Core & Intermediate Spark Operations

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

- reduce
- collect
- aggregate
- fold
- first
- take
- foreach
- top
- treeAggregate
- treeReduce
- foreachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

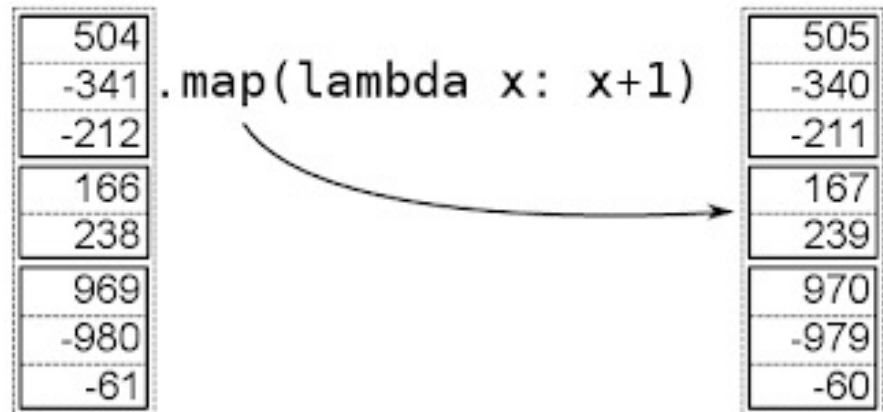
TRANSFORMATIONS

ACTIONS

pySpark map(), filter(), reduce()

Note: Different syntax than Python

- Map syntax:
 - `rdd.map(function)`
- Filter syntax:
 - `rdd.filter(function)`
- Reduce syntax:
 - `rdd.reduce(function)`



Passing functions to Spark

- With **lambda** syntax allows us to define “simple” functions inline. But we can pass defined functions.

```
def hasHadoop( line ):  
    return “Hadoop” in line
```

```
>> lines = sc.textFile( “README.txt” )
```

```
>> hadoopLines = lines.filter( hasHadoop )
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x * x)    # => [1, 4, 9]

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => [4]

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))    # => [0, 0, 1,
0, 1, 2]

# Map each element to zero or more others
nums.map(lambda x: range(0, x))        # => [[0], [0, 1],
[0, 1, 2]]
```

map() vs flatMap()

flatMap: Similar to map, it returns a new RDD by applying a function to each element of the RDD, but output is flattened.

```
>> rdd = sc.paralleliz([1, 2, 3])
```

```
>> rdd.map(lambda x: [x, x * 2])
```

```
Out[1]:  [ [1, 2],  [2, 4],  [3, 6]]
```

```
>> rdd.flatMap(lambda x: [x, x * 2])
```

```
Out[1]:  [ 1, 2,  2, 4,  3, 6]
```

Basic Actions

```
nums = sc.parallelize([5, 1, 3, 2])
```

```
# Retrieve RDD contents as a local collection
→ Results must fit in memory on the local machine
```

```
nums.collect() # => [5, 1, 3, 2]
```

```
# Return first K elements
nums.take(2) # => [5, 1]
```

```
# Return first K elements ordered
```

```
nums.takeOrdered(4) # => [1, 2, 3, 5]
```

```
# Return first K elements by applying
```

```
# a particular order
```

```
nums.takeOrdered(4, lambda n:-n)
```

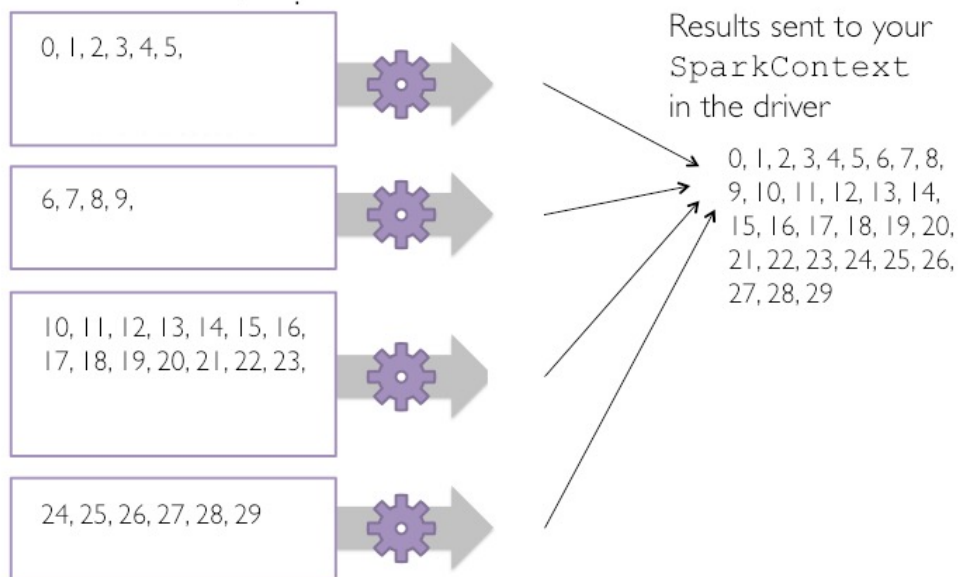
```
# => [5, 3, 2, 1]
```

```
# Count number of elements
nums.count() # => 4
```

```
# Merge elements with an associative function
nums.reduce(lambda x, y: x + y) # => 12
```

```
# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

`collect()` : Gathers the entries from all partitions into the driver



Persistence

- Spark **recomputes** the RDDs each time we call an action → expensive and can also cause data to be read from the disk again
- We can avoid this caching the data:
 - `cache()`
 - `persist()`
- Fault tolerant: In case of failure, Spark can rebuild the RDD
- Super Fast: will allow multiple operations on the same data set

Persistence

- RDDs can be **cached** using [cache](#) operation. They can also be **persisted** using [persist](#) operation.
- With `cache()`, you use only the default storage level `MEMORY_ONLY`.
- With `persist()`, you can specify which storage level you want, ([rdd-persistence](#)).
- Use `persist()` if you want to assign another storage level than `MEMORY_ONLY` to the RDD ([which storage level to choose](#))

Persistence Example

```
>> lines = sc.textFile("README.md", 4)
>> lines.count()
Out[1]: 1024
```

```
>> pythonLines = lines.filter(lambda line: "Python" in line)
>> pythonLines.count()
Out[1]: 50
```

| Causes Spark to reload **lines** from disk used.

```
>> lines = sc.textFile("README.md", 4)
>> lines.persist() # ~lines.cache()
>> lines.count()
Out[1]: 1024
```

```
>> pythonLines = lines.filter(lambda line:
"Python" in line)
>> pythonLines.count()
Out[1]: 50
```

Spark will avoid re-computing lines every time it is used.

SparkContext - Cluster execution

A **SparkContext** represents the connection to a Spark cluster.

- Interactive Shell/ Notebooks -- the spark context is created for you.
- Spark Application running in a Apache Cluster -- you have to create it

```
import sys
from pyspark import SparkContext, SparkConf

if __name__ == "__main__":
    conf = SparkConf().setAppName("Spark Count")
    sc = SparkContext(conf=conf)
    logFile = "README.md"
    textFile = sc.textFile(logFile)
    wordCounts = textFile.flatMap(lambda line: line.split())\
                           .map(lambda word: (word, 1))\
                           .reduceByKey(lambda a, b: a+b)
    wordCounts.collect()
```

