CSE2520 Big Data Processing Labs

Oct 21, 2021

Unix (bash)

Intro

```bash
#!/usr/bin/env bash

# This is a bash script file.
# To run this file use ./intro.sh in your terminal while in the correct directory.


# Echo prints lines to the output, if run using ./intro.sh then the output will be the terminal
echo "Running intro.sh"

# The command cd (change directory) can be used to change the current direcory.
# Update the cd command below to make sure that the rest of this script is executed in the data folder.
cd ../data || exit
# The '||' on this line is like a or. If the left part gives an exit code of non-zero (error/false) the right part is evaluated.
# We also have the '&&' sign. this only evaluates the right part if the left part has a zero exit code (no error/true).
# They are "lazy"

echo "Moved from intro to the data folder"

# The following line will execute the 'ls' command and save it in the variable lsOutput
lsOutput=$(ls)
# Then we can use echo to print the result
echo "Result of ls"
echo "$lsOutput"

# Implement a command that displays all files and their details in the current directory
# Do this by typing it between the brackets below.
# You can first test individual commands by pasting them in the terminal.
# (Make sure that you are in the correct directory if you want to test it)
detailedLsOutput=$(ls -l -R) # or (find . -type f | xargs wc)
# Prints the lsOutput
echo "Files in directory:"
echo "$detailedLsOutput"

# Implement a command that gets all the lines that contain "#" from the 'exoplanets' file
lines=$(grep "#" planets/exoplanets)
# Print result
echo "Lines from exoplanets with '#'"
echo "$lines"

# Now that you have used the basics of finding files and getting text from files it's time to move on.
# The real power of these commands are in the ability to combine them.
# Using the output of the first command  as data for the second  is called piping and is denoted by the symbol |
# A silly example is "ls | lolcat"
echo "The ls  command but now with some more color"
ls | lolcat

# Lets create an pipeline where we get the names of the last 5 exoplanets that where discovered in the year 2001.
# The first part should take all the lines containing "2001" from the 'exoplanets' file.
# The second part should only keeps the last 5 lines.
# lastly we need to get the data that is before the first comma.
firstPipeline=$(grep ",*,*,*,*,2001,*" planets/exoplanets | tail -n5 | grep ^[^,]* -o)
echo "First pipeline results:"
echo "$firstPipeline"


#end on start path
# This is needed for the automated testing
cd ../intro || exit
```

Pipelines

```bash
#!/usr/bin/env bash
# ---- MY AMAZING BOOK ----
cd ./../data/myBook/ || exit

# -- Q1 --
echo "-- Q1 --"
# Write a pipeline that for the current directory prints the 10 most common letters in all text files.
# Your pipeline should be case insensitive and ignore punctuation and space characters.
# Example output:
# 309 e
# 229 t
#tr ' ' '\n' | sed -e 's/[^ a-zA-Z\x27]//g' | sed 's/./&\n/g' |
mostCommonLetters=$(find -name '*.txt' | xargs cat | egrep [a-zA-Z] -i -o | tr '[:upper:]' '[:lower:]' | sort | uniq -c | sort -nr | head -n 10)
# Prints the mostCommonLetters
echo "Most common letters in my book:"
echo "$mostCommonLetters"

# -- Q2 --
# Write a pipeline that finds the number of words in the book that have been repeated only once.
# Your pipeline should be case insensitive
echo "-- Q2 --"
onlyOnce=$(find -name '*.txt' | xargs cat | tr ' ' '\n' | tr -d [:punct:] | sort -f | uniq -i -u | wc -l)
echo "Words repeated only once:"
echo "$onlyOnce"

cd ../../pipelines/ || exit
```

```bash
#!/usr/bin/env bash
echo "Running exoplanetProcessing.sh"

# ---- APACHE ----
cd ./../data/planets || exit

# -- Q1 --
echo "-- Q1 --"
# Write a pipline that for all planets that were discovered using Pulsar Timing displays the discovery year, name  and facility separated by spaces.
# Example: 2017 PSR B0329+54 b Multiple Facilities
pulsarTiming=$(grep ",*,*,*,Pulsar Timing,*,*" exoplanets | cut -f6,1,7 -d, | tr ',' ' ')
# Print pulsarTiming
echo "Pulsar Timing:"
echo "$pulsarTiming"


# -- Q2 --
echo "-- Q2 --"
# Write a pipeline that finds the year in which most exoplanets were discovered. Also provide the number of planets discovered that year.
# Example: 1505 2016
highestYear=$(cat exoplanets | cut -f6 -d, -s  | sed -e '1d' | sort | uniq -c | sort -nr | head -n1)
echo "Highest year:"
echo "$highestYear"

# -- Q3 --
echo "-- Q3 --"
# Write a pipeline that counts the number of exoplanets discovered between 1997 and 2006
intervalPlanets=$(cat exoplanets | cut -f6 -d, -s | egrep "199[7-9]|200[0-6]" | wc -l)
echo "Planets in interval:"
echo "$intervalPlanets"

# -- Q4 --
echo "-- Q4 --"
# Write a pipeline that outputs the names of two exoplanets that have the highest number of starts in the planetary system.
highestStars=$(cat exoplanets | cut -f1,3 -d, -s | sort -nr -k2 -t ',' | head -n2 | cut -d, -f1)
echo "Highest stars:"
echo "$highestStars"
```

```
cd ../../pipelines/ || exit
```

Script creation

```
#!/usr/bin/env bash
# This program should take an fileIn as first parameter
# It takes the input text file and outputs the 5 most common word bigrams (https://en.wikipedia.org/wiki/Bigram) in the file.
# Your solution should be case insensitive.
#
# example: when `./bigrams.sh ../data/myBook/01-chapter1.txt' is ran the output should look like this:
#   3 the     little
#   3 little  blind
#   3 blind   text
#   2 the     word
#   2 the     cop

words=$(tr [:punct:] ' ' < $1 | tr [:space:] ' ')

lines=$(
    echo $words | wc --words
)

sequence=$(
    seq 1 $(($lines - 1))
)

bigrams=$(
    for i in $sequence ; do
        j=$(($i + 1))
        echo $words | cut -f$i,$j -d' '
    done
)

cat <(echo "$bigrams") | tr [:upper:] [:lower:] | sort | uniq -ic | cut -f7,8,9 -d' ' | sort -nr -k1,2 -t' ' | awk -F ' ' '{print "   "$1,$2"\t"$3}' | head -n 5
```

```
#!/usr/bin/env bash
# The Unix assignment is almost over, time to create a submission.
# You could create a zip folder by hand. Just place the '.sh' files in there, but where's the fun in that.
# Let's create a script that does this for us.
# This script should take an output name as first parameter
# If called in a directory it should recursively find all the .sh files and add them to a zip
# So the zip should only contain .sh files and no folders.
find -name '*.sh' | zip -r -j -@ $1
```

Functional Programming (Scala)

Basics

```
/**
 * PART 1 - BASICS - FUNCTIONS AND LAMBDAS
 *
 * This part teaches you about writing (anonymous) functions in Scala.
 * It is worth 10 points.
 */
object Part1 {

  /**
   * An example to show what function definitions look like in Scala.
   *
   * @param i a number
   * @return i * 5 + 2
   */
  def increaseGrade(i: Int): Int = {
    val temp = i * 5
    return temp + 2
  }

  def main(args: Array[String]) = {
    println(leapYear(1990))
    println(question2())
  }

  /**
   * Scala supports both val/var variables (There is actually one more type,
   * but that is outside of the scope of this course).
   * The difference is that once you assign a value to a `val` it cannot be
   * reassigned. If you would use `var` you can reassign its value though.
   * Functional Programming purists will consider var as blasphemy, but Scala
   * is a bit more pragmatic supporting both imperative programming with vars
   * and mutability as well as the functional style with vals and
   * immutability. Within this lab assignment you are *NOT* allowed to use
   * var to help speed up the functional programming learning process.
   *
   * The `return` keyword in Scala is optional:
   * the last value produced is returned automatically, in this case:
   * temp + 2
   *
   * Instead of the above method body between braces you could also write more
   * minimalistically:
   *
   * def increaseGrade(i: Int) = i*5 + 2
   *
   * The above expression can also be defined as an anonymous function and
   * assigned to a variable, i.e. the anonymous function (or lambda or lambda
   * expression as it is also called) can be written as follows:
   *
   * (i: Int) => i*5+2
   *
   * This can be assigned to variable called increasGrade by:
   *
   * val increaseGrade = (i: Int) => i*5 + 2
   *
   * So from this example you see that defining an anonymous function is done
   * by first supplying the input parameters between () separated by commas,
   * followed by a '=>', ending with the method body.
   *
   * You can apply such a function for example to a list by supplying it to
   * methods that take a function as input parameter. Be careful of the
   * correct function signature or else your compiler/interpreter/IDE
   * will complain.
   *
   * Example usage:
   * List(22,34,48).map(increaseGrade) // results in List(112, 172, 242)
   *
   * This example takes our lambda and applies it to each element of the list
   * It works because:
   *             1. List.map takes a function as input parameter
   *       2. final def map[B](f: (A) => B): List[B] is the function signature
   *       of map. It takes a function (A) => B, a function which takes a
   *       type A element, does something to it, and returns a type Be
   *       Our anonymous function is of type Int => Int, so this works.
   */

  /** Q1 (3p)
   *
   * Write the method body for determining whether a year is a leap year
   */
  def leapYear(i: Int): Boolean = {
    if (i % 400 == 0)
      return true
    if (i % 100 == 0)
      return false
    i % 4 == 0
  } // TODO: change this method
```

```scala
  /** Q2 (4p)
   * You've seen an example of anonymous functions in Scala.
   * Given the following two higher order functions (HOFs), i.e.
   * functions that take functions as input parameter, replace the ??? of the
   * anonymous functions with methods bodies and provide the correct
   * function signature for myFirstHof/mySecondHof in the method body of
   * question2 such that running `question2()` results in the Lists given in
   * Note1
   */

  // TODO: uncomment the following 3 vals, replace the ??? with lambda expressions (anonymous functions)
  val isEven = (i: Int) => i % 2 == 0 // tells whether given Int is even or not
  val isOdd = (i: Int) => i % 2 == 1 // same but this time if it's odd
  val timesTwo = (i: Int) => i * 2   // takes an Int and doubles it
  //
  def question2(): List[List[Int]] = //List(List(1), List(2), List(3))
  // TODO change this method below:

  {
    // TODO:
    // - Look up the List functions used in the Scala API
    // - change the parameter type of f in both HOFS below
    //   from ...... to the correct type
    // - then uncomment this block and remove the `???`
    def myFirstHOF(xs: List[Int], f: Int => Boolean ) : List[Int] =
      xs.filter(f)
    def mySecondHOF(xs: List[Int], f: Int => Int ) : List[Int] =
      xs.map(f)

    val first = myFirstHOF(List(1,2,3,4), isEven)
    val second = myFirstHOF(List(1,2,3,4), isOdd)
    val third = mySecondHOF(List(10,11,12,13,14,15), timesTwo)

    val finalResult = List(first, second, third)
    finalResult
    // Scala returns the final expression, but if you just assigned it to
    // a variable the return type is Unit
  }

  /* Note 1
   * running println(question2()) results in
   * List(List(2,4), List(1,3), List(20,22,24,26,28,30))
   *
   * It is also good for you to see that you can define methods within methods
   * (if you ever need helper methods but do not want the outer world to see,
   * use this technique, i.e. it does more or less the same as defining a
   * private helper function and calling that from with the method body)
   */

  /* Note 2:
   * As you can see from inspecting the function signature of question2
   * it takes no input parameters and returns a List of List[Int]
   *
   * You don't need to specify the return type, except for recursive
   * functions; Scala will infer most return types for you. It is however good
   * practice to specify the return type when defining the function signature.
   * If you read another person's code without reading the whole method body,
   * having the complete function signature including return type is a hint to
   * its inner workings. Also, when debugging, it can really help you to view
   * the function as a blackbox with input/output types, so supplying the
   * return type has good benefits.
   */

  /** Q3 (3p)
   * given two Ints, generate a list of integers (List[Int]) containing the numbers in between and with both numbers inclusive
   *
   * Examples:
   * intList(2,7) // List(2,3,4,5,6,7)
   * intList(3,0) // List()
   *
   * Hint: take a peek at part 2 since this question bridges part 1 and part 2
   */
  // TODO: implement the function body
  def intList(a: Int, b: Int): List[Int] = // TODO change this method
  {
    if (a > b)
      return Nil
    val list = a :: intList(a + 1, b)
    list
  }
  // END OF PART 1

}
```

```scala
package basics

/**
 * PART 2 - LISTS AND PATTERN MATCHING
 *
 * It is worth 18 points.
 */
object Part2 {

  // Run this main function to see the result of the `println` calls.
  def main(args: Array[String]): Unit = {
    // In Scala, lists are defined recursively. A list consisting only of
    // the item 1 looks like this:
    val one = 1 :: Nil // can also be created by calling List(1)

    /**
     * The `::` operator concatenates an item and a list.
     * The `::` prepends an item to a list in O(1); There is also an
     * append, but we highly discourage its use since it's O(n)
     *
     * As the right side always needs to be a list, the terminating value
     * `Nil` is added. `Nil` is the same as an empty List: List()
     *
     * You could read the following list as `1 :: (2 :: (3 :: (Nil)))`,
     * every part in brackets is a list and the method `::` is called on
     * that list, i.e. `::` is called on its right operand.
     */
    val three = 1 :: 2 :: 3 :: Nil
    // Nil can be replaced by List(), but using `::` Nil seems tradition

    /**
     * For 1 chain in the list `x :: y`, `x` is usually referred to as the
     * head of the list, `y` is called the tail. They can be accessed
     * as follows:
     */
    val h = three.head
    val t = three.tail
    println(s"Head of $three is $h")
    println(s"Tail of $three is $t")

    /* Note 1
     * the two println statements above use something called String
     * interpolation in Scala: You start with s"..." and between the quotes
     * whenever you need to supply a value, use $variable_name to get it
     */

    /**
     * Since the lists are defined recursively, accessing a certain index
     * is not O(1) but O(n). Traversing the full list however is still O(n)
     * and this is what is used most often in functional programming.
     *
     * Scala's lists are immutable. Any operation that should change a
     * value will return a new list. For example, increasing the value of
```

```scala
   * every item with 1 looks like this:
   */

  val plusOne = three.map(x => x + 1)
  println(three) // The original list is not changed
  println(plusOne) // This is a new list


  /**
   * Lists can also be created with the `List` function. This is a
   * shorthand for the recursive way given earlier. When printing lists
   * to the console, they are displayed like this as well.
   */
  val four = List(1, 2, 3, 4)

  // calling `println(sum(four))` displays the result of the function sum
  // which is defined using pattern matching below
  println(sum(four)) // 10

  val drivers = List(
    ("green", 14),
    ("green", 3),
    ("orange", 2),
    ("orange", 6),
    ("red", 1),
    ("bananas", 1100123))

  /*
   * the output of calling the traffic light function on drivers
   * N.B. the type signature of the function and how the input is mapped
   */
  println(drivers.map(arg => trafficLightPenalty(arg._1, arg._2)))

}

/**
 * Recursively sums all values in the list.
 *
 * @param xs the list to process.
 * @return the sum of all values in xs.
 *
 */
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case i :: tail => i + sum(tail)
}

/* Note 1
 * Using `xs match` we "match" the value of the list. This is called pattern
 * matching. For matching the whole list without any special conditions
 * there are only two cases: Nil (or List()) and head :: tail, where head
 * is one element, tail is a list
 *
 * For the above `sum` it holds that the base case is the empty list,
 * which should return 0. If the list has a number, add the value of the
 * head to the sum of the tail
 *
 * N.B. Pay special attention to the following:
 * case matching can have a large number of cases; the FIRST one that
 * matches the argument mentioned before `match` is the one that will
 * `fire`. So if you have condition checks (guards, as they are called
 * in Scala) or a long list of cases to match against. beware of the order.
 *
 * Again:
 *             !!!!! THE ORDER MATTERS IN CASE MATCHING !!!!!
 */

/*
 * The following method shows a bit more challenging example. It is supposed
 * to represent the method to determine whether a driver will receive a fine
 * when crossing a traffic light. When the light is green, or when it is
 * orange and the time passed since crossing is less than or equal to three
 * seconds passed the driver will not receive a fine. When it is red or
 * more than three seconds after the light changed to orange, the driver
 * will receive a fine.
 *
 * The return type will indicate whether the driver gets a fine, and what
 * the reason is, as a String. This tuple, i.e. (true or false, reasoning)
 * will also indicate if the traffic light is malfunctioning. Take a look
 * below for the complete example.
 */
def trafficLightPenalty(status: String, time: Int): (Boolean, String) =
  (status, time) match {
    case ("green", _) => (false, "ok: nothing wrong here")
    case ("orange", x) if x <= 3 => (false, "ok: we allow it")
    case ("orange", y) if y > 3 => (true, "ok: too late")
    case ("red", _) => (true, "ok: waaay too late")
    case other => (false, "ERROR: " + other)
  }

/**
 * Scala also has immutable classes, called case classes. You can use
 * pattern matching on them as well. Note that `OptionalNum` itself cannot
 * be instantiated (as it is abstract). It's either `Nothing()`, indicating
 * no value, or `Num(i)`, indicating a value.
 * the keyword `sealed` indicates that ONLY the two case classes below can
 * be an OptionalNum. This can be used as a safety mechanism when checking
 * pattern matching. With `sealed` forgetting a case in the pattern match
 * will result in the compiler producing a non-exhaustive matching warning
 * without the keyword, this will not happen.
 *
 * Scala has similar built-in classes:
 * `Option`, extended by `None()` and `Some(v)`.
 * This is a very useful one, since in stead of `null` in Java (and the
 * Exceptions that come with it, when something is not found for example
 * you can just return None without throwing an exception.
 */
sealed abstract class OptionalNum()

case class Nothing() extends OptionalNum

case class Num(i: Int) extends OptionalNum

/**
 * Returns the sum of all defined numbers in a list of optional values.
 *
 * @param xs list of optional numeric values.
 * @return the sum of all defined numbers in `xs`.
 */
def optionalSum(xs: List[OptionalNum]): Int = xs match {
  case Nil => 0 // the base case, List()
  case Num(x) :: t => x + optionalSum(t) // h :: t case 1
  case Nothing() :: t => optionalSum(t) // h :: t case 2
}

/* Note that the `xs match` can be placed right after the function
 * definition. Case classes allow for pattern atching. Instead of an if
 * statement to check. Remember the sum method? Lists have normally only
 * two cases, `Nil` or `head :: tail`, but in this case we have three!
 */


// PART 2: EXERCISES - LISTS AND PATTERN MATCHING


/* We hope you've seen enough examples to repeat the technique on your own
 * The point to take away is that pattern matching is really if else on
 * steroids
 * For the exercises in this part you are _not_ allowed to use library
```

```scala
   * functions. Do not use iteration, write recursive functions instead.
   */

  /** Q4 (2p)
   * Twice takes a list and duplicates each element
   *
   * @param xs list to map
   *
   *            Example: twice(List.range(0,4)) // List(0, 0, 1, 1, 2, 2, 3, 3)
   */
  //TODO: supply the method body of twice, using pattern matching
  def twice[A](xs: List[A]): List[A] = xs match {
    case Nil => Nil // the base case, List()
    case None :: t => twice(t) // not necessary ??
    case h :: t => h :: h :: twice(t) // h :: t case 2
  }

  /** Q5 (2p)
   * You had a few drinks too much after a party and recorded a message for
   * someone, but due to pushing a few buttons on your app now not only the
   * list of words that you speak is reversed, but the words itself are as
   * well
   *
   * Example:
   * drunkWords(List("Hey","you,","how","are","you","doing?"))
   * turns into
   * List("?gniod","ouy","era","woh",",ouy","yeH")
   */
  // TODO: supply the method body of drunkWords using pattern matching
  def drunkWords(xs: List[String]): List[String] = xs match {
    case Nil => Nil
    case h :: t => drunkWords(t) ::: List(h.reverse)
  }

  /** Q6 (3p)
   * MyForAll takes a list of elements, and applies a function to it, to
   * check if some condition holds. An empty list evaluates to true.
   *
   * Examples:
   * val startsWithS = (s: String) => s.startsWith("s") // lambda expression
   *
   * myForAll(List("abc", "def"), startsWithS)                                      // false
   * myForAll(List("start", "strong", "system"), startsWithS)   // true
   */
  // TODO: supply the method body of myForAll using pattern matching.
  def myForAll[A](xs: List[A], f: A => Boolean): Boolean = xs match {
    case Nil => true
    case h :: t => f(h) && myForAll(t,f)
    case None :: t => false
  }

  /** Q7 (3p)
   * This is the first question where you encounter the Option[T] type
   * Use this type in the method body of lastElem, which returns an Option[A]
   * of the last element of the given List[A]
   *
   * @param xs the list to map over
   * @return None if the list is empty or Some( .. : A), the last element of
   *         the list
   *
   *         Examples:
   *         lastElem(List()) // None
   *         lastElem(List.range(0,3)) // Some(2) (range has exclusive ceiling)
   */
  // TODO implement using pattern matching
  def lastElem[A](xs: List[A]): Option[A] = xs match {
    case x :: Nil => Option(x)
    case Nil => None
    case h :: t => lastElem(t)
  }

  /** Q8 (3p)
   * Take two lists and concatenate them, returning the result
   *
   * @param xs , ys, the list to concatenate
   * @return the result of first all elements from xs with all elements
   *         from ys appended
   *
   *         Examples:
   *         append(List(), List())                      // List()
   *         append(List(1,3,5), List(2,4))     // List(1,3,5,2,4)
   */
  // TODO implement using pattern matching
  def append[A](xs: List[A], ys: List[A]): List[A] = (xs,ys) match {
//    case (Nil, Nil) => Nil
//    case (xs, Nil) => xs
//    case (Nil, ys) => ys
    case (xs, ys) => xs ::: ys
  }

  /**
   * Q9 (5p)
   * This question is a variant on a filter function. Given a List[A] and a
   * function f: A => Boolean, `myFilter` should retain all elements from
   * the list which satisfy 'f' and throw out all other elements, but...
   * ... it has a twist: It should also throw out each even indexed list
   * element which satisfy 'f'
   *
   * Take a look at the examples to see more directly what it needs to do
   * if you find this description vague.
   *
   * You are required to solved this using pattern matching on lists.
   * HINT: define a 'helper' method within myFilter which uses case matching.
   *
   * Examples:
   * val nrs = List.range(0,11) // List(0,1,2,3,...,10)
   * myFilter(nrs, (i: Int) => i % 2 == 0) // List(0,4,8)
   *
   * so although 2, 6 and 10 satisfy the function, they are thrown out.
   */
  // TODO implement method using pattern matching
  // a helper method which you've written yourself
  def myFilter[A](xs: List[A], f: A => Boolean): List[A] = {
    def helperFunction[A](xs: List[A], f: A => Boolean, i: Int): List[A] = xs match {
      case Nil => Nil
      case h :: t if f(h) && i % 2 == 0 => h :: helperFunction(t, f, i + 1)
      case h :: t if f(h) && i% 2 == 1 => helperFunction(t, f, i + 1)
      case h :: t if !f(h) => helperFunction(t, f, i)
    }
    val list = helperFunction(xs, f, 0)
    list
  }

  // END OF PART 2
}
```

```scala
package basics

/*
 * PART 3 - RINSE AND REPEAT - THE POWER OF THE API
 *
 * Ok, good, you've finished part two and now it's on to the next thing,
 * but no good practice without seeing there are many ways to slice a cake
 *
 * You just finished implementing myFilter in part 2. Below are two ways you
 * could do the same using methods from the standard library (i.e. API methods)
 * There are developers who strive for oneliners, i.e.
 *
 * val specialEvenNumbers1 = List.range(0,11).filter(isEven(_)).zipWithIndex.filter(x => x._2 %2 == 0).map(tuple => tuple._1)
 *   or...
 * val specialEvenNumbers2 = (List.range(0,11).groupBy(x => x % 2 == 0) get true).get.zipWithIndex.filter(_._2 % 2 == 0).map(_._1)
 *
 * WAIT ... WHUT?
 * While correct for beginning Scala developers this is torture...
 *
 * After some time you'll get used to this, but there will be many times you
 * will want to pull your hair out debugging the incomprehensible messages
 * from the Scala compiler.
 *
 * How do you combat this/undertake questions to end up with a one line result?
 *
 * Well, that's easy: it takes a bit of practice, but remember the
 * Unix pipelines? We're going to do the same.
 *
 * If you use IntelliJ you'll have the advantage that types will be placed in
 * comments popups at the right side of your code, but if that's not working for
 * some reason, we're going to add it in comments by hand. I'll go over the 2
 * examples from above and show what I mean.
 *
 * val specialEvenNumbers1 = List   // put the next method call on the next line with a .
 *   .range(0, 11)                       // range(0,11) returns List(1,2,3,4,...,10) List.range 2nd arg is exclusive
 *   .filter(isEven(_))                  // provide a lambda expression to retain all list elements which are even, i.e. List(0,2,4,6,8,10)
 *   .zipWithIndex                       // creates Tuple2[Int, Int] of each list element with its index starting at 0, i.e. List((0,0),(2,1), (4,2), (6,3), (8,4), (10,5))
 *   .filter(tup => tup._2 %2 == 0) // filter the tuples: retain all elements for which the 2nd element of the tuple, the index, is even: List((0,0), (4,2), (8,4))
 *   .map(tuple => tuple._1)             // map over the tuple and just retain the first element of each tuple List(0,4,8)
 *
 *
 * val specialEvenNumbers2 = List
 *   .range(0,11)                                       // again get List(0,1,2,...,10)
 *   .groupBy(x => x % 2 == 0) get true)
 *                                                              // groupBy(lambda) transforms into
 *                                                                              HashMap(
 *                                                                                      false -> List(1, 3, 5, 7, 9),
 *                                                                                      true -> List(0, 2, 4, 6, 8, 10)
 *                                                                              )
 *   // then calling `get true` on that HashMap gets you an Option of the key
 *       // so if the map has results for key 'true' you get Some(List(0,2,4,6,8,10)
 *   .get                           // unwraps the Option. If you had a None and called .get you get an exception!!
 *                                                              // read more on Option in the Scala Option API
 *   .zipWithIndex                                      // like above
 *   .filter(_._2 % 2 == 0)         // shorthand lambda which functions just like above
 *   .map(_._1)                     // shorthand lambda/tuple notation
 *
 * so the idea is to put every method call after the `.` on a next line
 * and type in comments the result type or sometimes even just the result on the
 * right in comments, i.e.
 *
 * someVal.someMethod           // List[(a:Int, b:String)]
 *  .map(lambda)                         // List[Int]
 *  .sum                                         // Int
 *
 * By writing your code like this, you can go over it step by step, and check
 * each transformation, just like a Unix pipeline. This will be beneficial
 * if you reach the Spark/Flink assignments
 *
 * Now enough of this... Time to get your hands dirty.
 *
 * You are asked to repeat the exact same functions from part 2
 * (except myFilter), and you are also asked for intlist from part 1,
 * but this time ONLY using standard scala API methods (mostly from List)
 * And NO... still no `vars` allowed!!! :)
 *
 * This part is worth 12 points
 */
object Part3 {
  /** Q10 (2p)
   * TODO: implement `twice` only using API calls
   */
  def twiceAPI[A](xs: List[A]): List[A] = {
    val list = xs.flatMap(x => List.fill(2)(x))
    list
  }

  /** Q11 (2p)
   * TODO: implement `drunkWords` only using API calls
   */
  def drunkWordsAPI(xs: List[String]): List[String] = {
    val list = xs.reverse.map(s => s.reverse)
    list
  }

  /** Q12 (2p)
   * TODO: implement `myForAll` only using API calls
   */
  def myForAllAPI[A](xs: List[A], f: A => Boolean): Boolean = {
    val result =xs.forall(f)
    result
  }

  /** Q13 (2p)
   * implement `lastElem` only using API calls
   */
  def lastElemAPI[A](xs: List[A]): Option[A] = {
    val result = xs.lastOption
    result
  }

  /** Q14 (2p)
   * implement `append` only using API calls
   */
  def appendAPI[A](xs: List[A], ys: List[A]): List[A] = {
    val list = xs ::: ys
    list
  }

  /** Q15 (2p)
   * implement `intList` (from part 1) only using API calls
   */
  def intListAPI(a: Int, b: Int): List[Int] = {
    val list = List.range(a, b + 1)
    list
  }
  // END OF PART 3
}
```

Functions

```
package functions

/* PART 4 - MORE FUNCTIONS
 *
 * This part is worth 35 points
 *
 * In summary, what you've seen so far is:
 * - basic function and lambda definitions
 * - pattern matching on Lists and case classes
 * - the use of API methods
 * - daisy chaining method calls just like Unix pipelines
 *
 * Next up is a few more examples I'd like to show you before you start
 * answering the more advanced questions
 *
 * 1. SUM
 *
 * Remember the pattern matching on sum?
 *
 *              def sum(xs: List[Int]) : Int = xs match {
 *              case List() => 0
 *                      case h :: t => h + sum(t)
 *              }
 *
 * Here are a few other ways of writing it:
 *
 * def sum2(xs: List[Int]) : Int = if (xs.isEmpty) 0 else xs.head + sum2(xs.tail)
 * def sumAPI1(xs: List[Int]) = xs.foldLeft(0)(_ + _)
 * def sumAPI2(xs: List[Int]) = xs.foldRight(0)(_ + _)
 * def sumAPI3(xs: List[Int]) = xs.sum
 *
 * def sumIterative(xs: List[Int]) : Int = {
 *   var sum  = 0;
 *       for (x <- xs) {
 *     sum = sum + x
 *   }
 *   sum
 * }
 * STOP!!! STOP!!! STOP!!! Remember no var?, the for expression is fine, it's
 * just that this method mutates the `sum` var and we don't allow that.
 *
 * 2. Folds
 *
 * the foldLeft/foldRight are an interesting bunch. They can be very powerful
 * what the foldLeft does in this example is take from left two right, two
 * elements from the list and feed it as parameters into the function.
 * In this case `_ + _` is the shorthand for ( (x: Int, y:Int) => x + y)
 * Some people call these reduce, but that is not correct. While similar
 * please see the following outputs (I've left out the `println`):
 *
 * // foldLeft
 * List.range(0,5).foldLeft(0)(_ + _)                                     // 10 with a shorthand lambda
 * List.range(0,5).foldLeft(0)((x:Int, y:Int) => x+y)   // 10
 * List(1).foldLeft(0)((x:Int, y:Int) => x+y)                   // 1
 * List().foldLeft(0)((x:Int, y:Int) => x+y)                    // 0
 *
 * // compiler will warn about overloaded method and compiler doesn't know
 * // what to pick, so use the fully written lambda to prevent such warnings
 * List().foldLeft(0)(_ + _))
 *
 * // reduceLeft
 * // type is needed or else warning, but with type still an exception
 * List[Int]().reduceLeft(_+_)
 *
 * List(7).reduceLeft(_+_)                                                         // 7
 * List(1,4,5).reduceLeft(_+_)                                             // 10
 *
 * As you can see there are some minor, but tricky differences
 * Another one to note is if your function isn't associative, then
 * foldLeft (going from left to right), will give you another result than
 * foldRight (going from right to left). You're highly recommended to look up
 * some more examples.
 *
 * 3. Flattening/Options
 *
 * Next I'd like to show you a few things that can happen when working with
 * Options, some of which aren't quite straightforward at first.
 *
 * Say you have a variable called results of type List[Option[Int]] and want
 * an easy way to filter out the Nones:
 *
 * val results = List(Some(6), Some(8), None, None, Some(9))
 *
 * results.flatten                       // List(6,8,9) the Options are 'unwrapped', i.e. Some peeled off and None thrown out
 * results.flatMap(x => x)       // does the same (first calls map, then flattens the result)
 * results.flatMap(identity) // syntactic sugar for `x => x`
 *
 * while this is very cool to see Option unwrapped, below are a few more examples
 * which show that working with Option sometimes can be a bit challenging.
 *
 * Let's say you have a competition and with some different rounds.
 * The administration kept track of contestants name age and hobby as a
 * Tuple3(String, Int, String). The finale is aired on tv and the host
 * only cares about the name and age of the contestants. He has a list of
 * the persons who qualify for the final. If a contestant did not make it through
 * the last round, `None` is registered. The final contestants look like this:
 *
 * val contestants = List(
 *                   Some(("Richard", 26, "windsurfing")),
 *                   None,
 *                   Some(("Amy", 22, "bmx"))
 *                       )
 *
 * If the tv show host only wants their name and age:
 *
 *      contestants.flatMap(x => if (x.isDefined) Some(x.get._1, x.get._2) else None)// List((Richard,26), (Amy,22))
 *
 * other things the host could do:
 *
 * val richard = contestants(0).get                                                                                      // (Richard, 26, windsurfing)
 * //val oops = contestants(1).get                                                                                               // exception, because of None.get
 * val contestant = if (contestants(1).isDefined) contestants(1).get else None  // None
 *
 * BEWARE OF TRANSFORMATIONS!!
 *
 * // if contestants failed the earlier rounds, they get transformed from `None`
 * // to "failedContestant"
 *
 *      println(contestants.flatMap(y => y match {
 *      case None => "failedContestant"
 *      case Some((name, age, hobby)) => Some(name, age)
 *      }))
 *      // results in List((Richard,26), f, a, i, l, e, d, C, o, n, t, e, s, t, a, n, t, (Amy,22))
 *
 * Forgetting Some in case Some results in a compiler warning; Scala can't
 * flatMap the (name, age), but the trickier issue is the String in the current
 * implementation. Strings can be flatmapped resulting in the characters
 * printed out separately. Beware of this and try to build your solution
 * step by step!
 *
 * 4. Case Classes
 *
 * There is one last thing I'd like to show you before getting to the questions
 * Simple definition of a case class and one instance:
 *
 * case class Book(isbn13 : String, translations: List[String], title: String)
 * val lotr = Book("9780395647394", List("EN", "NL", "FR"), "Lord of the Rings, Part 2: The Two Towers")
 *
 * if you want the list of languages into which it was translated call:
 * val translatedTo = lotr.translations
 *
 * i.e. you can use the .fieldName of the case class to get that specific field!
 *
 * You can read more about case classes here:
 * - https://docs.scala-lang.org/tour/case-classes.html
 * - https://docs.scala-lang.org/overviews/scala-book/case-classes.html
 *
 * 5. Tuples and indexing
 *
 * Last thing for me to tell you is an example with tuples. Scala supports up
 * to Tuple22 and you can index each element of a tuple with ._indexnr
 *
 * val ingredient = ("Sugar", 25)
 * println(ingredient._2) // outputs 25
 *
 * You can also nest these, for example:
 *
 * val recipe = (("tangerine", 2.0),("celery", 0.25),("cucumber", 1.0))
```

```scala
 *
 * println(recipe._1)           // (tangerine,2.0)
 * println(recipe._2._2)        // 0.25
 *
 * so far so good, but...    are you ready? ... the type of recipe is...
 *
 * Tuple3[Tuple2[String, Double], Tuple2[String, Double],Tuple2[String, Double]]
 *
 * yes, that is mindboggling for something so simple...
 * So if you ever need to work with tuples, sometimes it's good to just take
 * one element, write it out or sort it out and see how to get to the element
 * you are interested in. Scastie (online Scala interpreter) is a good scratchpad
 * to try your scribbling out on...
 *
 * That's about the end of me ranting and trying to show you some nice Scala
 * tips and tricks. Good luck with part 4 and part 5!
 */
object Part4 {
  // PART 4 - MORE FUNCTIONS - EXERCISES

  /** Q16 (2p)
   * Reverse a list using the List method foldLeft
   *
   * Example:
   * reverseUsingFold(List.range(-3,2)) // List(1, 0, -1, -2, -3)
   */
  def reverseUsingFold[A](nrs: List[A]): List[A] = nrs.foldLeft[List[A]](Nil)((x, y) => y :: x)

  /** Q17 (5p)
   * find the second largest element in the list of Ints, if it exists
   *
   * @param xs : the List[Int] to search
   * @return an Option[Int], i.e. Some(x: Int) that is the next largest element in the list
   *         or None if it does not exist
   *
   *         Examples:
   *         secondLargest(List(1,2,3,4,5,6))   // Some(5)
   *         secondLargest(List(1,1,1,1))                // None
   */
  // TODO: implement method using Pattern Matching, API calls to standard API or a combination (Hint Hint..)
  // no imports are necessary
  def secondLargest(xs: List[Int]): Option[Int] = xs match {
    case Nil => None
    case _ => xs
      .distinct
      .sorted
      .reverse
      .lift(1) //"get" (apply) for option type
  }

  /** Q18 (5p)
   * Count the number of occurrences of each distinct number
   * Example:
   * countNumbers(List(1,2,1,2,1,2,3,4,4,4,4,4,4,4,4)) // Map(1 -> 3, 2 -> 3, 3 -> 1, 4 -> 8)
   */
  def countNumbers(xs: List[Int]): Map[Int, Int] = xs.groupBy(x => x).mapValues(_.size)

  /** Q19 (2p)
   * Ai... 13 is always a bad number... A hacker infiltrated the Scala Deployment server and hacked the API
   * The hacker removed the function `distinct` from the API. You have been asked to create a workaround
   * using the standard API... and only for two points... cheapskates...
   *
   * @param xs the list to process
   * @return a list with all the duplicates filtered out
   *
   *         Example:
   *         distinctAPI(List(1,2,1,2,1,2,3,4,4,4,4,4,4,4,4)) // List(1,2,3,4)
   */
  // TODO: implement `distinct` without using .distinct on the list
  def distinctAPI[A](xs: List[A]): List[A] = xs.foldLeft[List[A]](Nil)((x, y) => if (!x.contains(y)) x ::: List(y) else x)

  /** Q20 (3p)
   * Special partition takes all numbers, adds one to the negative numbers and checks
   * if all those together have the same absolute value as all the positive numbers
   *
   * Example:
   * specialPartition(List(List(-4,1,2), List(-4,-4,-6,1,2,4,4), List(-4,-4, -6, 1,2,8))) // true
   */
  // TODO: implement using API methods
  def specialPartition(xs: List[List[Int]]): Boolean = xs.foldLeft(0)((x, y) => x + y.foldLeft(0)(
    (x, y) => if (y < 0) x + y + 1 else x + y)) == 0

  val tudScrabblePlayers: List[(String, List[(String, Int)])] = List(
    ("Ola", List(("tree", 4), ("plant", 12), ("water", 10), ("earth", 9), ("heaven", 1))),
    ("Liudas", List(("hop", 3), ("editor", 7), ("mileage", 8), ("load", 4), ("quintuple", 16))),
    ("Roald", List(("language", 8), ("three", 5), ("hip", 3))),
    ("Burçu", List(("beverage", 11), ("institution", 14), ("player", 8), ("feedback", 7), ("positive", 9), ("login", 5))),
    ("Georgios", List(("functional", 9), ("program", 7), ("facebook", 6)))
  )

  /** Q21 (5p)
   * It's finally Friday and you and your friends are going to play a game of Scrabble
   * from the given `scrabblePlayers` determine the good players.
   * A good player has put down only words of length larger than 3.
   *
   * in `tudScrabblePlayers` Ola, Burçu, and Georgios are good players.
   */
  // TODO: implement this method
  def determineGoodPlayers(scrabblePlayers: List[(String, List[(String, Int)])]): List[String] = scrabblePlayers.foldLeft[List[String]](Nil)(
    (x, y) => if(!y._2.isEmpty && y._2.forall(w => w._1.length > 3)) x ::: List(y._1) else x)

  /** Q22 (3p)
   * determine from good Scrabble players who has an average value per word of over 7
   * return their name and the average value of words put down
   * for the above example we expect:
   * List((Ola,7.2), (Burçu,9.0), (Georgios,7.333333333333333))
   */
  def avgOver7(scrabblePlayers: List[(String, List[(String, Int)])]): List[(String, Double)] = scrabblePlayers
    .filter(x => determineGoodPlayers(scrabblePlayers).indexOf(x._1) != -1)
    .filter(x => x._2.map(ab => ab._2).sum.toDouble / x._2.length.toDouble > 7)
    .map(x => (x._1, 0d + x._2.map(ab => ab._2).sum.toDouble / x._2.length.toDouble))

  /** Q23 (5p)
   * Given an anonymised list of students who sport and another list of
   * all student results where the anonymised names relate, i.e. student "a"
   * on the sport list is the same student as student "a" in the list with
   * results from all students....
   *
   * find the students who sport, who showed up to ALL their exams AND passed
   * all of them (grade >= 6)
   * val asr = List(// list of student tuples with (name, course, Option[result])
   * ("a", "bdp", Some(8)), ("a", "ci", Some(9)), ("a", "dm", None),
   * ("b", "bdp", Some(7)), ("b", "ci", Some(7)), ("b", "dm", Some(7)),
   * ("c", "bdp", Some(6)), ("c", "ci", Some(3)), ("c", "dm", Some(9)),
   * ("d", "bdp", Some(4)), ("d", "ci", Some(9)), ("d", "dm", Some(3)),
   * ("e", "bdp", Some(7)), ("e", "ci", Some(8)), ("e", "dm", Some(8)),
   * ("f", "bdp", Some(6)), ("f", "ci", Some(6)), ("f", "dm", Some(8)),
   * ("g", "bdp", Some(6)), ("g", "ci", Some(6)), ("g", "dm", Some(9))
   * )
   *
   * val ss = List("a", "d", "f","g") // only "f"/"g" would be considered 'good'
   */
  def goodSportStudents(studentResults: List[(String, String, Option[Int])], sportStudents: List[String]): List[String] = sportStudents.foldLeft[List[String]](Nil)(
    (x, y)  => if (studentResults.filter(s => s._1.eq(y)).foldLeft(true)((z, w) => z && w._3.isDefined && w._3.get >= 6)) List(y) ::: x else x)

  /** Q24 (5p)
   *
   * The institution of the sporting students from the previous questions
   * would like to know what the average grade of each of the good sporting
   * students is.
   *
   * The expected output for this question with the previous students (from Q22)
   * would be: List((f,6.666666666666667), (g,7.0))
   */
  def avgGradeGoodSportStudents(studentResults: List[(String, String, Option[Int])], sportStudents: List[String]): List[(String, Double)] =
    sportStudents.foldLeft[List[(String, Double)]](Nil)(
      (x, y)  => if (studentResults.filter(s => s._1.eq(y)).foldLeft(true)((z, w) => z && w._3.isDefined && w._3.get >= 6))
        List((y, studentResults.filter(s => s._1.eq(y)).foldLeft(0d)((z, w) => z + w._3.get) / studentResults.count(s => s._1.eq(y)))) ::: x else x)

  // END OF PART 4
}
```

Dataset

```scala
package dataset

/**
 * PART 5A - DATASET1 / REAL LIFE APPLICATION
 *
 * In the following questions you will solve realistic problems with the
 * techniques you learned in this assignment. You will be working with data of
 * San Francisco Library patrons.
 *
 * Below you can find what each field means.
 * Id: Id of patron
 * Patron Type Definition: Description of patron (adult, teen, child, senior,
 * etc.).
 * Total Checkouts: Total number of items the patron has checked out from the
 * library since the record was created.
 * Total Renewals: Total number of times the patron has renewed checked-out
 * items.
 * Age Range: Age ranges: 0 to 9 years, 10 to 19 years, 20 to 24 years,
 * 25 to 34 years, 35 to 44 years, 45 to 54 years, 55 to 59 years,
 * 60 to 64 years 65 to 74 years, 75 years and over. Some blank.
 * Home Library Definition: Description of the branch library where the
 * patron was originally registered.
 * Circulation Active Month: Month the patron last checked out library
 * materials, or last logged into the library’s subscription databases
 * from a computer outside the library.
 * Circulation Active Year: Year the patron last checked out library materials,
 * or last logged into the library’s subscription databases from a computer
 * outside the library.
 * Notice Preference Definition: Description of the patron’s preferred method of
 * receiving library notices.
 * Provided Email Address: Indicates whether the patron provided an email
 * address.
 * Year Patron Registered: Year patron registered with library system.
 * No dates prior to 2003 due to system migration.
 * Outside of County: If a patron's home address is not in San Francisco, then
 * flagged as true, otherwise false.
 * Supervisor District: Based on patron address: San Francisco Supervisor
 * District. Note: This is an automated field, please note that if
 * "Outside of County" is true, then there will be no supervisor district.
 * Also, if the input address was not well-formed, the supervisor district
 * will be blank.
 *
 * Solve the following questions using functional programming.
 * The code for reading the file is already given.
 *
 * This part is worth 9 points
 */
object Part5A {

  def main(args: Array[String]): Unit = {
    val patrons = scala.io.Source.fromResource("library-first6000.csv").getLines().toList.drop(1).map(line => line.split(","))

    // use this if you want nice output to fiddle around a bit
    println(patrons.head.toList)
    println(exampleOutput(patrons))
  }

  /** gets you the ids of the patrons and their age */
  def exampleOutput(xs: List[Array[String]]): List[Array[String]] = xs.filter(x => x(8).eq("email") && !x(9).toBoolean)

  /** Q25 (3p)
   * Count the number of patrons who didn't provide an age to the library at all
   *
   * @param xs the list of patrons
   * @return the number of people who didn't supply their age
   */
  def noAge(xs: List[Array[String]]): Int = xs.map((x: Array[String]) => x(4)).count(x => x.isEmpty)

  /** Q26 (3p)
   * Output the id, age range and notice preference as a tuple of the patron with the highest number of checkouts
   */
  def highestCheckouts(xs: List[Array[String]]): Tuple3[Int, String, String] = xs.sortBy(x => x(2).toInt).reverse.map(x => (x(0).toInt, x(4), x(8))).apply(0)

  /** Q27 (3p)
   * Output the patron type and number of patrons as a map and make sure the patrons satisfy the following condition:
   * - Patrons who indicated to be contacted by email but have not provided that email
   */
  def missing_email(xs: List[Array[String]]): Map[String, Int] =
    xs
      .map((x: Array[String]) => (x(1), x(8),  x(9)))
      .filter(abc => abc._2 == "email" && abc._3 == "false")
      .map(x => (x._1))
      .groupBy(identity)
      .map(x => (x._1, x._2.length))

  // END OF PART 5A
}
```

```scala
package dataset

import dataset.util.XMLDatafile.Badge
import scala.io.Source

/**
 * PART 5A - DATASET2 / StackOverFlow Badges
 *
 * In this assignment you will be asked to finish reading in a not quite xml file
 * The file is one big list of lines such asked
 *
 * <row Id="1" UserId="3" Name="Autobiographer" Date="2012-03-06T18:53:16.300" Class="3" TagBased="False" />
 *
 * For this assignment you first have to prep your data a bit, and then it's
 * on to answering questions. This part is worth 9 points
 */
object Part5B {

  /** Q28 (3p)
   * Included is a first example of reading in a file:
   * `sourceAsListString` generates a `List[String]`
   *
   * We would like you to finish `source` converting this into a List of
   * case class Badge, i.e. make sure the return type for source is
   * `List[Badge], you can find Badge in the util folder
   */
  val sourceAsListString = Source.fromResource("First6200Badges.xml").getLines.toList.drop(2).dropRight(1)

  // TODO remove the 'val source = ???', uncomment the lines below and
  //  finish the method to read in the file and return List[Badge]
        val source = Source.fromResource("First6200Badges.xml").getLines().toList.drop(2).dropRight(1)
        .map(line => {
      val words = line.split("\"")
      val badge = Badge(words.apply(1).toInt, words(3).toInt, words(5), words(7), words(9).toInt, words(11).toBoolean)
      badge
    })

  /**
   * Again you can use this to get some output
   */
  def main(args: Array[String]): Unit = {
    //println(showResults(sourceAsListString))
    println(source)
  }

  def showResults(input: List[String]): Unit = input.foreach(println)


  /** Q29 (3p)
   *
   * What is the easiest attainable badge? Output a tuple of its name and nr
   */
  def easiestAttainableBadge(input: List[Badge]): (String, Int) = input.map(x => x.name).groupBy(identity).map(x => (x._1, x._2.length)).toList.sortBy(x => x._2).reverse.apply(0)

  /** Q30 (3p)
   *
   * Return a tuple of tuples of the least productive and most productive
   * year, together with the nr of badges earned
   */
  def yearOverview(input: List[Badge]): ((Int, Int), (Int, Int)) = Tuple2(input.map(x => x.badgeDate.split("-").apply(0)).groupBy(identity).map(x => (x._1.toInt, x._2.length)).minBy(x => x._2),
  input.map(x => x.badgeDate.split("-").apply(0)).groupBy(identity).map(x => (x._1.toInt, x._2.length)).maxBy(x => x._2))

  // END OF PART 5B

}
```

```scala
package dataset

import java.text.SimpleDateFormat
import java.util.SimpleTimeZone
import dataset.util.Commit.Commit
import org.json4s.native.Serialization
import org.json4s.{Formats, NoTypeHints}

import scala.io.Source
import scala.math.Ordering.Implicits._

/**
 * PART 5C - Mining Software Repositories
 *
 * Use your knowledge of functional programming to complete the following function.
 * You are recommended to use library functions when possible.
 *
 * The data is provided as a list of `Commit`s. This case class can be found in util/Commit.scala.
 * When asked for dates, use the `commit.commit.committer.date` field.
 *
 * This part is worth 7 points.
 */
object Part5C {
  implicit val formats: AnyRef with Formats = Serialization.formats(NoTypeHints)
  val source: List[Commit] = Source.fromResource("1000_commits.json").getLines().map(Serialization.read[Commit]).toList

  /** Q31 (7p)
   *
   * A day has different parts:
   * Morning 5 am to 12 pm (noon)
   * Afternoon 12 pm to 5 pm.
   * Evening 5 pm to 9 pm.
   * Night 9 pm to 4 am.
   *
   * Which part of the day was the most productive in terms of commits ?
   * Return a tuple with the part of the day and the number of commits
   *
   * Hint: for the time, use `SimpleDateFormat` and `SimpleTimeZone`.
   */
  def mostProductivePart(input: List[Commit]): (String, Int) = input.map(x => x.commit.committer.date.toString.split(" ")(3).split(":")(0).toInt)
    .map(x =>
      if (x >= 5 && x <= 12) "morning" else
        if (x >= 12 && x <= 17) "afternoon" else
          if (x >= 17 && x <= 21) "evening" else
            if (x >= 21 || x <= 5) "night" else
              ""
    ).groupBy(identity).map(x => (x._1, x._2.length)).maxBy(x => x._2)

  // END OF PART 5C & END OF THE LAB ^_^
  // Hope you enjoyed it and good luck with the next assignment!
}
```

Akka

MapReduce

```scala
package cse2520.mapreduce

import akka.actor.typed.scaladsl.{AbstractBehavior, ActorContext, Behaviors}
import akka.actor.typed.{ActorRef, Behavior, PostStop, Signal}
import cse2520.mapreduce.MapperNode.{InputSetState, MapperCommand, MapperEvent}

import scala.util.{Failure, Success}

object MapperNode {

  sealed trait MapperCommand
  case class StartMapper(taskId: Int, inFile: String, supervisor: ActorRef[MapperEvent]) extends MapperCommand
  case object ProcessNextBatch extends MapperCommand

  sealed trait MapperEvent
  case class MapperStarted(id: Int, taskId: Int) extends MapperEvent
  case class MapperFinished(id: Int, taskId: Int, intSets: Map[Int, String]) extends MapperEvent

  final case class InputSetState(name: String, lines: Iterator[String], index: Int)

  def apply(id: Int, mapperClass: Class[Mapper],
            partitions: Partitioner, fileSystem: FileSystem): Behavior[MapperCommand] =
    Behaviors.setup(context => {
      val mapper = mapperClass.getConstructor().newInstance()
      context.log.info(s"Mapper ${id} is available.", id)

      new MapperIdle(context, new MapperNode(id, fileSystem, mapper, partitions))
    })
}

class MapperNode(val id: Int, val fileSystem: FileSystem, val mapper: Mapper, val partitions: Partitioner)


class MapperIdle(context: ActorContext[MapperCommand], node: MapperNode)
  extends AbstractBehavior[MapperCommand](context) {

  import MapperNode._

  // TODO
  override def onMessage(msg: MapperCommand): Behavior[MapperCommand] = msg match {
    case StartMapper(taskId, inputSet, supervisor) =>
      val iterator = node.fileSystem.readInputSet(inputSet).iterator
      supervisor.tell(MapperStarted.apply(node.id, taskId))
      context.self ! ProcessNextBatch
      new MapperInProgress(context, node, taskId, InputSetState(inputSet, iterator, 0), new PartitioningBufferedEmitter(node.partitions), supervisor)
    case _ => Behaviors.unhandled
  }
  override def onSignal: PartialFunction[Signal, Behavior[MapperCommand]] = {
    case PostStop =>
      context.log.info(s"Mapper ${node.id} stopped [from idle]", node.id)
      Behaviors.stopped
  }
}

class MapperInProgress(context: ActorContext[MapperCommand], node: MapperNode, taskId: Int, inputSet: InputSetState,
                        emitter: PartitioningBufferedEmitter, supervisor: ActorRef[MapperEvent])
  extends AbstractBehavior[MapperCommand](context) {

  import MapperNode._

  context.log.info(s"Mapper ${node.id} is starting task ${taskId} ...", node.id, taskId)

  // TODO
  override def onMessage(msg: MapperCommand): Behavior[MapperCommand] = msg match {
    case ProcessNextBatch if inputSet.lines.hasNext =>
      context.log.info(s"Mapper ${node.id} is processing task ${taskId} [line=${inputSet.index}] ...", node.id, taskId, inputSet.index)
      // do the task! - (emitter, key, value)

      val mapResult = node.mapper.map(emitter, inputSet.name, inputSet.lines.next())
      mapResult match {
        case Success(_) =>
          context.self ! ProcessNextBatch
          new MapperInProgress(context, node, taskId, inputSet.copy(index = inputSet.index + 1), emitter, supervisor)
        case Failure(e) =>
          context.log.info(s"Mapper ${node.id} error processing ${e}.", node.id, inputSet, e)
          //Behaviors.stopped
          throw e
      }
    // TODO
    case ProcessNextBatch =>
      context.log.info(s"Mapper ${node.id} has completed processing task ${taskId}.", node.id, taskId)

      // maps partition indices to file names
      val intermediates = (0 until node.partitions.count)
        // take buffered data
        .map(i => (i, emitter.getData(i)))
        .filter(_._2.nonEmpty)
        .map(p => p._1 -> s"partition-${p._1}/task-$taskId.txt")
        .toMap

      intermediates.foreach {
        //  for each partition index, write data to corresponding file
        case (i, outFileName) => node.fileSystem.writeLocalSet(outFileName, emitter.getData(i))
      }
      supervisor.tell(MapperFinished(node.id, taskId, intermediates))
      new MapperIdle(context, node)
    case _ => Behaviors.unhandled
  }
}
```

Spark

RDD

```
package RDDAssignment

import java.math.BigInteger
import java.security.MessageDigest
import java.sql.Timestamp
import java.util.UUID

import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import utils.{Commit, File, Stats}

/**
 * Hint regarding the exercises: it is sometimes specified that the assignment asks about the committer or the
 * commit author. Those are two different entities, as per the Commit.scala file. Inspect it thoroughly and make
 * sure to always refer to the proper entity!
 */
object RDDAssignment {


  /**
    * Reductions are often used in data processing in order to gather more useful data out of raw data. In this case
    * we want to know how many commits a given RDD contains.
    *
    * * You should be able to complete this assignment with using only one function. If in doubt, read the Spark RDD
    * documentation in detail: https://spark.apache.org/docs/2.4.3/api/scala/index.html#org.apache.spark.rdd.RDD
    *
    * @param commits RDD containing commit data.
    * @return Long indicating the number of commits in the given RDD.
    */
  def assignment_1(commits: RDD[Commit]): Long = commits.count()

  /**
    * We want to know what is the most popular email domain. We require a RDD containing tuples
    * of the used email domain, combined with the number of occurrences.
    *
    * Hint: Use email of author
    *
    * @param commits RDD containing commit data.
    * @return RDD containing tuples indicating the email domain (extension) and number of occurrences.
    */
  def assignment_2(commits: RDD[Commit]): RDD[(String, Long)] = commits
    .map(c => c.commit.author.email)
    .distinct()
    .map(e => e.split("@"))
    .map(e => (e(e.length - 1), 1))
    .groupByKey()
    .map(row => (row._1, row._2.sum))

  /**
    * Return a Tuple with filename and the number of changes of the most frequently changed file.
    * If there is no filename, use 'unknown'.
    *
    * Files in a directory must have unique names but can have the same name in different directories.
    * Files can be refactored to different directories. To simplify things, you may assume that when a file
    * is refactored you *DO NOT* need to add or subtract commits from filepath(s). To further simplify this.
    * use absolute filepath as filename.
    *
    * @param commits RDD containing commit data.
    * @return RDD containing tuples indicating the filename and number of changes.
    */
  def assignment_3(commits: RDD[Commit]): (String, Long) = commits
    .flatMap(c => c.files)
    .map(f => if(f.filename.isDefined) (f.filename.get, f.additions + f.deletions) else ("unknown", f.additions + f.deletions))
    .groupByKey()
    .map(row => (row._1, 1L * row._2.sum))
    .sortBy(x => x._2, false)
    .take(1)(0)

  /**
    * Some users on Github might be interested in their ranking in number of comments. Return a
    * RDD containing tuples of the rank (zero indexed) of a commit author, a commit author's name and the sum of comment
    * counts made by the commit author. As in general with performance rankings, a higher performance means a better
    * ranking (0 = best). In case of a tie, the lexicographical ordering of the usernames should be used to break the
    * tie.
    *
    * @param commits RDD containing commit data.
    * @return RDD containing the rank number, commit author names and number of comments an author in order.
    */
  def assignment_4(commits: RDD[Commit]): RDD[(Long, String, Long)] = commits
    .map(c => (c.commit.author.name, c.commit.comment_count))
```

```scala
      .groupByKey()
      .map(row => (row._1, row._2.sum))
      .sortBy(x => x._1.toLowerCase)
      .sortBy(x => x._2, false)
      .zipWithIndex()
      .map(x => (x._2, x._1._1, x._1._2))

    /**
      * We want to know how stable and how widely used some programming languages are. There are many ways to achieve that,
      * but for the purpose of this exercise, the measure we choose is how many additions, deletions and changes
      * occur in each file extension. We will provide a list of file extensions.
      *
      * We want an RDD of tuples containing the file extension name, the number of such files
      * and the Stats object. As the Stats object is only used for commits and single files only have additions,
      * deletions and changes value, we want you to compose the Stats object for each file with those values.
      *
      * Hint: the value of "changes" is the sum of additions and deletions, so it is an equivalent of the
      * "total" value in stats.
      *
      * @param commits RDD containing commit data.
     *  @param fileExtensions List of String containing file extensions
      * @return RDD containing file extension and an aggregation of the committers Stats.
      */
    def assignment_5(commits: RDD[Commit], fileExtensions: List[String]): RDD[(String, Stats)] = commits
      .flatMap(c => c.files)
      .filter(c => c.filename.isDefined && c.filename.get.split('.').length > 0 && fileExtensions.contains(c.filename.get.split('.')(c.filename.get.split('.').length - 1)))
      .map(f => (f.filename.get.split('.')(f.filename.get.split('.').length - 1), Stats(f.changes, f.additions, f.deletions)))
      .groupByKey()
      .map(t => (t._1, t._2.fold(Stats(0,0,0))((f, s) => Stats(f.total + s.total, f.additions + s.additions, f.deletions + s.deletions))))


    /**
      * There are different types of people, those who own repositories, and those who make commits. There are also
      * people who do both. We require as output an RDD containing the names of commit authors and repository owners
      * that have both committed to repositories and own repositories in the given RDD.
      * Note that the repository owner is contained within Github urls.
      *
      * @param commits RDD containing commit data.
      * @return RDD of Strings representing the author names that have both committed to and own repositories.
      */
    def assignment_6(commits: RDD[Commit]): RDD[String] = commits
      .map(c => c.url.split("/")(4))
      .intersection(commits
        .map(c => c.commit.author.name))

    /**
     *                                              Description
     * Sometimes developers make mistakes, sometimes they make many. One way of observing mistakes in commits is by
     * looking at so-called revert commits. We define a 'revert streak' as the number of times `Revert` occurs
     * in a commit.
     *
     * Note that for a commit to be eligible for a 'commit streak', its message must start with `Revert`.
     * As an example: `Revert "Revert ...` would be a revert streak of 2, whilst `Oops, Revert Revert little mistake`
     * is not a 'revert streak'.
     *
     * Return a RDD containing tuples of
     *       - repository name (can be derived from the url)
     *       - average streak length computed over all commits.
     *
     * @param commits RDD containing commit data.
     * @return RDD of Tuple type containing a repository name and a double representing the average streak length.
     */
    def assignment_7(commits: RDD[Commit]): RDD[(String, Double)] = commits
      .map(c => (c.url.split("/")(5), 1))
        .reduceByKey(_ + _)
        .map(t => (t._1, 1D * t._2))
        .leftOuterJoin(commits
          .map(c => (c.url.split("/")(5), c.commit.message.split("[ ,?!:.\"\n]+")))
          .filter(t => t._2.length > 0 && t._2(0) == "Revert")
          .map(t => (t._1, t._2.count(s => s == "Revert")))
          .reduceByKey(_ + _))
      .map(t => (t._1, (1D * t._2._2.getOrElse(0)) / t._2._1))

    /**
      * We want to know the number of commits that are made by unique committers (represented by the field committer
      * in CommitData) in the given RDD. Besides the number of commits, we also want to know how many different
      * repositories the committers committed to. The repository name can be found in url.
      *
      * @param commits RDD containing commit data.
      * @return RDD of tuple containing committer name, list of repositories and
      * total number of commits committed to that repository.
      */
    def assignment_8(commits: RDD[Commit]): RDD[(String, Iterable[String], Long)] = commits
      .map(x => (x.commit.committer.name, ("(?=(([\\w-]+).commits))[\\w-]+".r.findAllIn(x.url).mkString, 1)))
      .reduceByKey((acc, next) => (acc._1 + " " + next._1, acc._2 + next._2))
      .map(ab => (ab._1, ab._2._1.split(" ").toList.distinct, ab._2._2))

    /**
     *                                         Description
     * Return RDD of tuples containing
     *  - repository names
     *  - list of all commit authors of that repository (commit.author.name), with date of first commit.
     *                                            Hint
     * Use commit.author.date
     *
     * @param commits RDD containing commit data.
     * @return RDD containing the repository names, list of tuples of Timestamps and commit author names
     */
    def assignment_9(commits: RDD[Commit]): RDD[(String, Iterable[(Timestamp, String)])] = {
      val m = commits
        .map(c => ((c.url.split("/")(5),c.commit.author.name), c.commit.author.date))
        .reduceByKey((acc, next) => if (acc.after(next)) next else acc)
        .collect()

      commits
        .map(c => ((c.url.split("/")(5),c.commit.author.name), c.commit.author.date))
        .reduceByKey((acc, next) => if (acc.after(next)) next else acc)
        .map(ab => (ab._1._1, "!" + ab._2 + ";" + ab._1._2 + "!"))
        .reduceByKey((acc, next) => acc + "%" + next)
        .map(ab => (ab._1, ab._2.split("%")
          .toIterable
          .map(x => (Timestamp.valueOf("[^!,].+(?=;)".r.findAllMatchIn(x).mkString
            .replace(".0", "")),
            "(?<=;)[^!]+".r.findAllMatchIn(x).mkString))))
    }


    /**
     *                                               Description
     * We want to know the committers that worked on a certain file to make an overview of every file in a repository.
     *
     * Create a tuple containing
     *  - file name
     *  - set of tuples with name of committers
     *  - Stat object representing the changes made to the file by each committer.
     *
     * @param commits RDD containing commit data.
     * @param repository String name of repository
     * @return RDD containing tuples representing a file name and a list of tuples of committer names and Stats object.
     */
    def assignment_10(commits: RDD[Commit], repository: String): RDD[(String, List[(String, Stats)])] = ???


    /**
      * BONUS ASSIGNMENT STARTS HERE
      *
      * Hashing function that computes the md5 hash from a String, which in terms returns a Long to act as a hashing
      * function for repository name and username.
      *
      * @param s String to be hashed, consecutively mapped to a Long.
      * @return Long representing the MSB from the inputted String.
      */
    def md5HashString(s: String): Long = {
      val md = MessageDigest.getInstance("MD5")
      val digest = md.digest(s.getBytes)
      val bigInt = new BigInteger(1, digest)
      val hashedString = bigInt.toString(16)
      UUID.nameUUIDFromBytes(hashedString.getBytes()).getMostSignificantBits
    }

    /**
      * OPTIONAL EXERCISE - do not expect help from TA
      *
      * Create a bi-directional graph from committer to repositories. Use md5HashString function above to create unique
      * identifiers for creating a graph.
      *
      * As the real usage Sparks GraphX library is out of the scope of this course, we will not go further into this, but
      * this can be used for algorithms like PageRank, Hubs and Authorities, clique finding, etc.
      *
      * We expect a node for each repository and each committer (based on committer name), an edge from each
      * committer to repositories the committer has committed to.
      *
      * Look at documentation of Graph and Edge before starting this complementary exercise.
      * Your vertices should contain information about the type of node, a 'developer' or a 'repository' node.
      * Edges should only exist between repositories and committers.
      *
      * @param commits RDD containing commit data.
      * @return Graph representation of the commits as described above.
      */
    def bonus_assignment(commits: RDD[Commit]): Graph[(String, String), String] = ???
  }
```

SQL

```scala
package DataFrameAssignment

import java.sql.Timestamp

import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import utils.File

/**
 * Note read the comments carefully, as they describe the expected result and may contain hints in how
 * to tackle the exercises. Note that the data that is given in the examples in the comments does
 * reflect the format of the data, but not the result the graders expect (unless stated otherwise).
 */
object DFAssignment {

  /**
   * To get a better overview of the data, we want to see only a few columns out of the data. We want to know
   * the committer name, the timestamp of the commit and the length of the message in that commit
   *
   * | committer      | timestamp            | message_length |
   * |----------------|----------------------|----------------|
   * | Harbar-Inbound | 2019-03-10T15:24:16Z | 1              |
   * | ...            | ...                  | ...            |
   *
   * Hint: try to work out the individual stages of the exercises, which makes it easier to track bugs, and figure out
   * how Spark Dataframes and their operations work. You can also use the `printSchema()` function and `show()`
   * function to take a look at the structure and contents of the Dataframes.
   *
   * Hint: for mapping values of a single column, look into user defined functions (udf)
   *
   * @param commits Commit Dataframe, created from the data_raw.json file.
   *@return DataFrame of commits from the requested committers, including the commit timestamp and the length of the
   *         message in that commit.
   */
  def assignment_1(commits: DataFrame): DataFrame = {
    val getLength = udf { s : String => s.length }
    commits
      .select("commit.committer.name", "commit.committer.date", "commit.message")
      .withColumn("committer", col("name"))
      .withColumn("timestamp", col("date"))
      .withColumn("message_length", getLength(col("message")))
      .drop("name")
      .drop("date")
      .drop("message")
  }

  /**
   * In this exercise we want to know all the commit SHA's from a list of commit committers. We want to order them
   * by the committer names alphabetically:
   *
   * | committer      | sha                                      |
   * |----------------|------------------------------------------|
   * | Harbar-Inbound | 1d8e15a834a2157fe7af04421c42a893e8a1f23a |
   * | ...            | ...                                      |
   *
   * @param commits Commit Dataframe, created from the data_raw.json file.
   * @param committers Sequence of String representing the committers from which we want to know their respective commit
   *                 SHA's.
   * @return DataFrame of commits from the requested commiters, including the commit SHA.
   */
  def assignment_2(commits: DataFrame, committers: Seq[String]): DataFrame = commits
      .select("commit.committer.name", "sha")
      .withColumnRenamed("name", "committer")
      .filter(row => committers.contains(row.get(0)))
      .orderBy("committer")

  /**
   * We want to generate yearly dashboards for all users, per each project they contribute to.
   * In order to achieve that, we need the data to be partitioned by years.
   * The returned DataFrame that is expected is in the following format:
   *
   * | repository | committer        | year | count   |
   * |------------|------------------|------|---------|
   * | Maven      | magnifer         | 2019 | 21      |
   * | .....      | ..               | .... | ..      |
   *
   * @param commits Commit Dataframe, created from the data_raw.json file.
   * @return Dataframe containing 4 columns, Repository name, committer name, year and the number of commits for that
   *         week.
   */
  def assignment_3(commits: DataFrame): DataFrame = {
    val getRepoName = udf { s : String => s.split("/")(5) }
    val getYear = udf { s: String => s.split("-")(0).toInt }
    commits
      .withColumn("repository", getRepoName(col("url")))
      .withColumn("year", getYear(col("commit.committer.date")))
      .select("repository", "commit.committer.name", "year", "_id")
      .withColumnRenamed("name", "committer")
      .groupBy(col("repository"), col("committer"), col("year"))
      .count()
  }

  /**
   * A developer is interested in what day of the week some commits are pushed, although this is something that
   * can always be calculated during runtime, this would require us to pass a Timestamp along with the computation.
   * Therefore we require you to append the inputted DataFrame with the day of the week by names: Mon, Tue, Wed, Thu,
   * Fri, Sat, Sun.
   *
   * Hint: Look into SQL functions in for Spark SQL.
   *
   * Expected Dataframe (column) example that is expected:
   *
   * | day    |
   * |--------|
   * | Mon    |
   * | Fri    |
   * | ...    |
   *
   * @param commits Commit Dataframe, created from the data_raw.json file.
   * @return the inputted DataFrame appended with a day column.
   */
  def assignment_4(commits: DataFrame): DataFrame = commits
      .withColumn("date", col("commit.committer.date"))
      .withColumn("timestamp", to_timestamp(col("date")))
      .drop("date")
      .withColumn("day", date_format(col("timestamp"), "E"))
      .drop("timestamp")

  /**
   * To perform analysis on commit behavior of some committers, you want to compare the dates of their commits.
   * We require that the DataFrame that is put in is appended with two additional columns: tha dates of the
   * previous and the following commits of the user, independent of the branch or repository. You should
   * order the commits from oldest to newest.
   *
   * Hint: Look into Spark sql's Window to have more expressive power in custom aggregations
   *
   * Expected Dataframe example:
   *
   * | $oid                                | prev_date                | date                     | next_date                    |
   * |--------------------------           |--------------------------|--------------------------|--------------------------|
   * | 5ce6929e6480fd0d91d3106a          | 2019-01-03T09:11:26.000Z | 2019-01-27T07:09:13.000Z | 2019-03-04T15:21:52.000Z |
   * | 5ce693156480fd0d5edbd708          | 2019-01-27T07:09:13.000Z | 2019-03-04T15:21:52.000Z | 2019-03-06T13:55:25.000Z |
   * | 5ce691b06480fd0fe0972350          | 2019-03-04T15:21:52.000Z | 2019-03-06T13:55:25.000Z | 2019-04-14T14:17:23.000Z |
   * | ...                                 | ...                            | ...                            | ...                              |
   *
   * @param commits Commit DataFrame, see commit.json and data_raw.json for the structure of the file, or run
   *                 `println(commits.schema)`.
   * @param committerName Name of the author for which the result must be generated.
   * @return DataFrame with a columns `$oid` and `prev_date`, `date`, `next_date`
   */
  def assignment_5(commits: DataFrame, committerName: String): DataFrame = {
    val windowSpec = Window.orderBy("date")
    commits
      .select("_id.$oid", "commit.committer.date", "commit.committer.name")
      .filter(row => row.get(2) == committerName)
      .drop("name")
      .orderBy("date")
      .withColumn("prev_date", lag("date", 1).over(windowSpec))
      .withColumn("test_date", col("date"))
      .drop("date")
      .withColumnRenamed("test_date", "date")
      .withColumn("next_date", lead("date", 1).over(windowSpec))
  }


  /**
   *
   *                                            Description
   *
   * After looking at the results of assignment 5, you realise that the timestamps are somewhat hard to read
   * and analyze easily. Therefore, you now want to change the format of the list.
   * Instead of the timestamps of previous, current and next commit, output:
   *      - Timestamp of the current commit  (date)
   *      - Difference in days between current commit and the previous commit (days_diff)
   *      - Difference in minutes between the current commit (minutes_diff)
   *      - Previous commit (Oid)
   *
   * For both fields, i.e. the difference in days and difference in minutes, if the value is null
   * replace it with 0. When there is no previous commit, the value should be 0.
   *                                              Output
   *
   *
   * | $oid                         | date                      | days_diff       | minutes_diff |
   * |--------------------------        |--------------------------      |-----------    |--------------|
   * | 5ce6929e6480fd0d91d3106a         | 2019-01-27T07:09:13.000Z      | 0             | 3            |
   * | 5ce693156480fd0d5edbd708         | 2019-03-04T15:21:52.000Z      | 36            | 158          |
   * | 5ce691b06480fd0fe0972350         | 2019-03-06T13:55:25.000Z      | 2             | 22           |
   * | ...                              | ...                           | ...           | ...          |
   *
   *                                               Hints
   *
   * Look into Spark sql functions. Days difference is easier to calculate than minutes difference.
   *
   * @param commits Commit DataFrame, see commit.json and data_raw.json for the structure of the file, or run
   *                 `println(commits.schema)`.
   * @param committerName Name of the author for which the result must be generated.
   * @return DataFrame with columns as described above.
   */
  def assignment_6(commits: DataFrame, committerName: String): DataFrame = {
    val windowSpec = Window.orderBy("date")
    val removeNulls = udf { s : String => if (s == null) "0" else s }
    val stringToInt = udf { s : String => if (!s.contains('.')) s.toInt else s.split('.')(0).toInt}
    commits
      .select("_id.$oid", "commit.committer.date", "commit.committer.name")
      .filter(row => row.get(2) == committerName)
      .drop("name")
      .orderBy("date")
      .withColumn("test_days_diff", datediff(col("date"), lag("date", 1).over(windowSpec)))
      .withColumn("days_diff", stringToInt(removeNulls(col("test_days_diff"))))
      .drop("test_days_diff")
      .withColumn("minutes_diff", stringToInt(floor(removeNulls(unix_timestamp(to_timestamp(col("date"))) -
        unix_timestamp(to_timestamp(lag(col("date"), 1).over(windowSpec)))) / 60)))
  }

  /**
   *                                         Description
   *
   * To get a bit of insight in the spark SQL, and its aggregation functions, you will have to
   * implement a function that returns a DataFrame containing columns:
   *         - repository
   *         - month
   *         - commits_per_month(how many commits were done to the given repository in the given month)
   *
   *                                           Output
   * | repository   | month | commits_per_month |
   * |--------------|-------|-------------------|
   * | OffloadBuddy | 1     | 32                |
   * | ...          | ...   | ...               |
   *
   * @param commits Commit DataFrame, see commit.json and data_raw.json for the structure of the file, or run
   *                 `println(commits.schema)`.
   * @return DataFrame containing a `repository` column, a `month` column and a `commits_per_month`
   *         representing a count of the total number of commits that that were ever made during that month.
   */
  def assignment_7(commits: DataFrame): DataFrame = {
    val getRepoName = udf { s : String => s.split("/")(5) }
    val getMonth = udf { s : String => s.split("-")(1).toInt }
    commits
      .select("url", "commit.committer.date", "_id.$oid")
      .withColumn("repository", getRepoName(col("url")))
      .drop("url")
      .withColumn("month", getMonth(col("date")))
      .drop("date")
      .groupBy(col("repository"), col("month"))
      .count()
      .withColumnRenamed("count", "commits_per_month")
  }

  /**
   *                                         Description
   *
   * In a repository, the general order of commits can be deduced from  timestamps. However, that does not say
   * anything about branches, as work can be done in multiple branches simultaneously. To trace the actual order
   * of commits, using commits SHAs and Parent SHAs is necessary. We are interested in commits were a parent commit
   * has a different committer than the child commit.
   *
   * Output a list of committer names, and the number of times this happened.
   *
   *                                          Output
   *
   *
   * | parent_name | times_parent |
   * |-------------|--------------|
   * | Emeric      | 2            |
   * | ...         | ...          |
   *
   * @param commits Commit DataFrame, see commit.json and data_raw.json for the structure of the file, or run
   *                `println(commits.schema)`.
   * @return DataFrame containing the parent name and the count for the parent.
   */
  def assignment_8(commits: DataFrame): DataFrame = {
    val contains = udf { (a : Seq[String], s : String) => a.contains(s) }
    val filterParents = udf { s : Seq[String] => s.distinct }
    val temp1 = commits
      .select(col("commit.committer.name").as("parent_name"), col("sha").as("parent_sha"))
    val temp2 = commits
      .select(col("commit.committer.name").as("child_name"), col("sha").as("current_sha"), col("parents").as("parents"))
      .withColumn("parents_shas", filterParents(col("parents.sha")))
      .drop("parents")
    temp1
      .join(temp2, contains(col("parents_shas"), col("parent_sha")))
      .filter(row => row.get(0).toString != row.get(2).toString)
      .drop("parents_shas")
      .drop("child_sha")
      .groupBy("parent_name")
      .count()
      .withColumnRenamed("count", "times_parent")
  }
}
```

Flink

```
import java.text.SimpleDateFormat
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment}
import org.apache.flink.api.scala._
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import util.Protocol.{Commit, CommitGeo, CommitSummary}
import util.{CommitGeoParser, CommitParser}

/** Do NOT rename this class, otherwise autograding will fail. **/
object FlinkAssignment {

  val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

  def main(args: Array[String]): Unit = {

    /**
      * Setups the streaming environment including loading and parsing of the datasets.
      *
      * DO NOT TOUCH!
      */
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)

    // Read and parses commit stream.
    val commitStream =
      env
        .readTextFile("data/flink_commits.json")
        .map(new CommitParser)

    // Read and parses commit geo stream.
    val commitGeoStream =
      env
        .readTextFile("data/flink_commits_geo.json")
        .map(new CommitGeoParser)

    /** Use the space below to print and test your questions. */
    question_six(commitStream).print()

    /** Start the streaming environment. **/
    env.execute()
  }

  /** Dummy question which maps each commits to its SHA. */
  def dummy_question(input: DataStream[Commit]): DataStream[String] = {
    input.map(_.sha)
  }

  /**
    * Write a Flink application which outputs the sha of commits with at least 20 additions.
    * Output format: sha
    */
  def question_one(input: DataStream[Commit]): DataStream[String] = input
    .filter(c => c.stats.isDefined && c.stats.get.additions >= 20)
    .map(c => c.sha)

  /**
    * Write a Flink application which outputs the names of the files with more than 30 deletions.
    * Output format:  fileName
    */
  def question_two(input: DataStream[Commit]): DataStream[String] = input
    .flatMap(c => c.files)
    .filter(f => f.deletions > 30 && f.filename.isDefined)
    .map(f => f.filename.get)

  /**
    * Count the occurrences of Java and Scala files. I.e. files ending with either .scala or .java.
    * Output format: (fileExtension, #occurrences)
    */
  def question_three(input: DataStream[Commit]): DataStream[(String, Int)] = input
    .flatMap(c => c.files)
    .filter(f => f.filename.isDefined)
    .map(f => {
      val fn = f.filename.get.split('.')
      fn(fn.length - 1)
    })
    .filter(fn => fn == "java" || fn == "scala")
    .map(fn => (fn, 1))
    .keyBy(t => t._1)
    .reduce((a, b) => (a._1, a._2 + b._2))

  /**
    * Count the total amount of changes for each file status (e.g. modified, removed or added) for the following extensions: .js and .py.
    * Output format: (extension, status, count)
    */
  def question_four(input: DataStream[Commit]): DataStream[(String, String, Int)] = input
    .flatMap(c => c.files)
    .filter(f => f.filename.isDefined && f.status.isDefined)
    .map(f => ({
      val fn = f.filename.get.split('.')
      fn(fn.length - 1)
    }, f.status.get, f.changes))
    .filter(t => t._1 == "js" || t._1 == "py")
    .keyBy(t => (t._1, t._2))
    .reduce((a, b) => (a._1,a._2, a._3 + b._3))
```

```scala
  /**
    * For every day output the amount of commits. Include the timestamp in the following format dd-MM-yyyy; e.g. (26-06-2019, 4) meaning on the 26th of June 2019 there were 4 commits.
    * Make use of a non-keyed window.
    * Output format: (date, count)
    */
  def question_five(input: DataStream[Commit]): DataStream[(String, Int)] = input
      .map(c => (c.commit.committer.date, 1))
      .assignAscendingTimestamps(t => t._1.getTime)
      .windowAll(TumblingEventTimeWindows.of(Time.days(1)))
      .reduce((a,b) => (a._1, a._2 + b._2))
      .map(t => ({"" + t._1.toString.substring(8,10) + "-" + (t._1.toString.substring(4, 7) match {
        case "Jan" => "01"
        case "Feb" => "02"
        case "Mar" => "03"
        case "Apr" => "04"
        case "May" => "05"
        case "Jun" => "06"
        case "Jul" => "07"
        case "Aug" => "08"
        case "Sep" => "09"
        case "Oct" => "10"
        case "Nov" => "11"
        case "Dec" => "12"
        case _ => 1
      }) + "-" + t._1.toString.substring(t._1.toString.length-4)},t._2))

  /**
    * Consider two types of commits; small commits and large commits whereas small: 0 <= x <= 20 and large: x > 20 where x = total amount of changes.
    * Compute every 12 hours the amount of small and large commits in the last 48 hours.
    * Output format: (type, count)
    */
    def question_six(input: DataStream[Commit]): DataStream[(String, Int)] = input
    .filter(c => c.stats.isDefined)
    .map(c => ({if (c.stats.get.total >= 0 && c.stats.get.total <= 20 ) "small" else "large"}, c.commit.committer.date, 1))
    .assignAscendingTimestamps(t => t._2.getTime)
    .keyBy(t => t._1)
    .window(SlidingEventTimeWindows.of(Time.hours(48), Time.hours(12)))
    .reduce((a,b) => (a._1, b._2, a._3 + b._3))
    .map(t => (t._1, t._3))
```