Distributed Systems

Preassignment report

Answers to questions

Fredrik Ek
ekfr@student.chalmers.se

901109-0959, D

November 5, 2014

What is Seattle?

An open-source and free programmer community for testing distributed systems using computers all over the world, making cloud computing available for everyone.

What is a vessel in the Seattle framework?

A resource or computer somwhere in the world. Via Seattle you can browse for vessels locally or globally to use as your own distributed network. By adding Vessels to your account you can thereby run code on several machines at once without actually owning them.

Which programming language is used to write programs to run on Seattle? How does it relate to Python? Explain briefly why it is used, instead of Python.

The language used is called repy, which is a kind of python-based api. As I understand it, repy is more or less exactly like python but with some removed functionality. Which makes sense becouse of varios security reasons.

Write the full command to run locally a program written in the language in the previous question?

```
python <path to repy.py>
       <path to a restrictions file>
       <path to source file>
```

So basically assuming that I am in the folder where I put repy.py and have some kind of restriction file, I can use the following command to run the repy program test.repy locally:

```
python repy.py restrictions.allowallports
     "C:\Users\Fredrik\Documents\TDA596\Pre-Ass\test.repy"
```

List all steps to run a program remotely in vessels.

First of all you need to register an account on the seattle homepage, browse the tab My Vessels and ”get” some vessels. When that is done you have to run the python file seash.py to start the interface.

```
python seash.py
```

Next step is to load your keys from your account, which can be done by providing your account username.

```
!> loadkeys FredrikEk
```

and by using the following command you can see your username in the prompt:

```
!> as FredrikEk
```

Now you want to locate and add the resources/computers that you browsed for on your account on the seattle homepage which can be done using the command:

```
FredrikEk@ !> browse
```

All of your resources will then be added to a group with name browsegood. To use all the resources in this group you can type:

```
FredrikEk@ !> on browsegood
```

You can now run .repy code on the target resources in the added group by typing:

```
FredrikEk@browsegood !> run infloop.repy
```

And to check the status of the used resources you can use:

```
FredrikEk@browsegood !> show log
```

or

```
FredrikEk@browsegood !> list
```

In the Hello World Example in the Repy tutorial, what is this line ”listencommhandle = waitforconn(ip,port,hello)” for?

What it does is that its waiting for a connection on the specified ip and port, in this example the local ip of the computer, and when it gets it, it calls the function hello().

Python Code

string1.py

string1.py

```python
def donuts(count):
  s = str(count)
  if count>9:
    s = 'many'
  return 'Number of donuts: ' + s

def both_ends(s):
  newString = '';
  if len(s) > 2:
    newString = s[:2] + s[-2:]
  return newString

def fix_start(s):
  firstChar = s[0]
  restOfString = s[1:]
  return firstChar + restOfString.replace(firstChar,'*')

def mix_up(a, b):
  a1 = b[:2] + a[2:]
  b1 = a[:2] + b[2:]
  return a1 + ' ' + b1


# Provided simple test() function used in main() to print
# what each function returns vs. what it's supposed to return.
def test(got, expected):
  if got == expected:
    prefix = ' OK '
  else:
    prefix = '  X '
  print '%s got: %s expected: %s' % (prefix, repr(got), repr(expected))


# Provided main() calls the above functions with interesting inputs,
# using test() to check if each result is correct or not.
def main():
  print 'donuts'
  # Each line calls donuts, compares its result to the expected for that call.
  test(donuts(4), 'Number of donuts: 4')
  test(donuts(9), 'Number of donuts: 9')
  test(donuts(10), 'Number of donuts: many')
  test(donuts(99), 'Number of donuts: many')

  print
  print 'both_ends'
  test(both_ends('spring'), 'spng')
  test(both_ends('Hello'), 'Helo')
  test(both_ends('a'), '')
  test(both_ends('xyz'), 'xyyz')


  print
  print 'fix_start'
  test(fix_start('babble'), 'ba**le')
  test(fix_start('aardvark'), 'a*rdv*rk')
  test(fix_start('google'), 'goo*le')
  test(fix_start('donut'), 'donut')

  print
  print 'mix_up'
  test(mix_up('mix', 'pod'), 'pox mid')
  test(mix_up('dog', 'dinner'), 'dig donner')
  test(mix_up('gnash', 'sport'), 'spash gnort')
  test(mix_up('pezzy', 'firm'), 'fizzy perm')


# Standard boilerplate to call the main() function.
if __name__ == '__main__':
  main()
```

list1.py

list1.py

```
def match_ends(words):
  words2 = []
  for s in words:
    if len(s) > 1 and s[0] == s[len(s)-1]:
    words2.append(s)
  return len(words2)

def front_x(words):
  xWords = []
  nonXWords = []

  for s in words:
    if s[0] == 'x':
      xWords.append(s)
    else:
      nonXWords.append(s)

```

```python
  xWords.sort()
  nonXWords.sort()
  wordsFinal = xWords + nonXWords
  return wordsFinal

def sort_last(tuples):
  sortedTuples = sorted(tuples, key = lambda tuple: tuple[len(tuple)-1])

  return sortedTuples

# Provided simple test() function used in main() to print
# what each function returns vs. what it's supposed to return.
def test(got, expected):
  if got == expected:
    prefix = ' OK '
  else:
    prefix = '  X '
  print '%s got: %s expected: %s' % (prefix, repr(got), repr(expected))


# Calls the above functions with interesting inputs.
def main():
  print 'match_ends'
  test(match_ends(['aba', 'xyz', 'aa', 'x', 'bbb']), 3)
  test(match_ends(['', 'x', 'xy', 'xyx', 'xx']), 2)
  test(match_ends(['aaa', 'be', 'abc', 'hello']), 1)

  print
  print 'front_x'
  test(front_x(['bbb', 'ccc', 'axx', 'xzz', 'xaa']),
       ['xaa', 'xzz', 'axx', 'bbb', 'ccc'])
  test(front_x(['ccc', 'bbb', 'aaa', 'xcc', 'xaa']),
       ['xaa', 'xcc', 'aaa', 'bbb', 'ccc'])
  test(front_x(['mix', 'xyz', 'apple', 'xanadu', 'aardvark']),
       ['xanadu', 'xyz', 'aardvark', 'apple', 'mix'])


  print
  print 'sort_last'
  test(sort_last([(1, 3), (3, 2), (2, 1)]),
       [(2, 1), (3, 2), (1, 3)])
  test(sort_last([(2, 3), (1, 2), (3, 1)]),
       [(3, 1), (1, 2), (2, 3)])
  test(sort_last([(1, 7), (1, 3), (3, 4, 5), (2, 2)]),
       [(2, 2), (1, 3), (3, 4, 5), (1, 7)])

```

```

if __name__ == '__main__':
  main()
```

wordcount.py

wordcount.py

```
import sys
import operator

def sort_helper(filename):
  words = []
  f = open(filename, 'rU')

  for line in f:
    words += line.split()

  f.close()
  words = map(lambda e:e.lower(), words)
  wordCountDictionary = dict((w,words.count(w)) for w in
    words)
  return wordCountDictionary

def print_words(filename):
  wordCountDictionary = sort_helper(filename)

  sortedByWord = sorted(wordCountDictionary.items(), key
    = operator.itemgetter(0))
  for x in sortedByWord:
    print x[0] + ' : ' + str(x[1])
  return

def print_top(filename):
  wordCountDictionary = sort_helper(filename)

  sortedByOccurence = sorted(wordCountDictionary.items(),
     key = operator.itemgetter(1))
  sortedByOccurence.reverse()
  del sortedByOccurence[20:]
  for x in sortedByOccurence:
    print x[0] + ' : ' + str(x[1])
  return

###

# This basic command line argument parsing code is
  provided and
# calls the print_words() and print_top() functions which
   you must define.
def main():
```

```python
  if len(sys.argv) != 3:
    print 'usage: ./wordcount.py {--count | --topcount} file'
    sys.exit(1)

  option = sys.argv[1]
  filename = sys.argv[2]
  if option == '--count':
    print_words(filename)
  elif option == '--topcount':
    print_top(filename)
  else:
    print 'unknown option: ' + option
    sys.exit(1)

if __name__ == '__main__':
  main()
```