Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

Publications

Funded Projects

Research Projects

Related Events

Exam of Advance in Programming

2 September 2013

Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. The worth of the 2 exercises is 16 and 14 points respectively. To pass the exam you have to do both exercises. The submissions with only one exercise will not be evaluated at all.

Exercise 1: Asynchronous Function Call.

One of the main reasons provoking execution penalties is the call to functions that take to much time to complete their job and their result is not immediately necessary but is only used later. This is pretty sequential code that could be easily parallelized if the function call is performed by a separate thread so that the main computation goes on. The problem in this scenario is that python's function normally are called in a synchronous way and they have a return value that before or later is computed and will be used.

This exercise consists in implementing a **decorator** that once applied to a function each call to such a function is done asynchronously, i.e., a separate thread takes care of its execution and the main thread goes on with the rest of the execution. Note that the two threads need to be synchronized on the result of the computation: this is necessary in the main thread but not immediately available. So you need to implement a class (called future in the literature) wrapping the return value that i) provides a method `get_result` to get the result if available (note that this method raises an exception `NotYetDoneException` when you try to access the result and this is not ready yet), ii) provides a method `is_done` to test if the return value has been calculated or not. Any asynchronous computation is separate!

Hints: You have to use the standard module `threading`.

The following is an example of the expected computation.

```
from asynchronous import *

if __name__ == '__main__':
  import time

  @asynchronous
  def long_process(num):
    time.sleep(10)
    return num * num

  result = long_process.start(12)

  for i in range(20):
    time.sleep(1)

    if result.is_done():
        print("[{1}]: result {0}".format(result.get_result(), i))
    else: print("[{0}]: not ready yet".format(i))

  result2 = long_process.start(13)

  try:
    print("result2 {0}".format(result2.get_result()))
  except asynchronous.NotYetDoneException as ex:
```

```
[15:59]cazzola@surtur:~/pa/es1>python3 main-asynchronous.py
[0]: not ready yet
[1]: not ready yet
[2]: not ready yet
[3]: not ready yet
[4]: not ready yet
[5]: not ready yet
[6]: not ready yet
[7]: not ready yet
[8]: not ready yet
[9]: not ready yet
[10]: result 144
[11]: result 144
[12]: result 144
[13]: result 144
[14]: result 144
[15]: result 144
[16]: result 144
[17]: result 144
[18]: result 144
[19]: result 144
```

Exercise 2: Generators at Hand.

The python's generators provide a easy way to lazy set constructions but the interaction with them can be improved by composing them with other generators.

In this exercise you have to implement the following **generators** to wrap other generators.

- `even(s)`: this returns the even elements of the generator `s` at each iteration (max 2 points);
- `stopAt(s, n)`: this generator stops to iterate when the next element of `s` is greater than `n`; it only works for sorted generators (max 2 points);
- `buffer(s, n)`: this generator at each iteration returns a chunk of `n` consecutive elements of the generator `s`; note that last chunk can be shorter (max 5 points); and
- `conditional(s, p)`: this generator returns those elements of the generator `s` whose **successive** element respects the predicate `p` (max 5 points).

The following is an example of the expected computation.

```
from generators import *

def fib():
  x,y = 1,1
  while True:
    yield x
    x,y = y, x+y

if __name__ == "__main__":
  even_fib = even(fib())
  for i in range(10): print(next(even_fib), end=' ')
  print()

  for i in stopAt(even(fib()), 40000000): print(i, end=' ')
  print()

  buffered_limited_fib = buffer(stopAt(fib(),3000), 5)
  for i in buffered_limited_fib: print(i)

  condfib = conditional(fib(), lambda x: (x%2 == 0))
  for i in range(10): print(next(condfib), end=' ')
  print()

  condfib2 = conditional(fib(), lambda x: (x%2 != 0))
  for i in range(15): print(next(condfib2), end=' ')
  print()
```

```
[21:02]cazzola@surtur:~/pa/es2>python3 main-generators.py
2 8 34 144 610 2584 10946 46368 196418 832040
2 8 34 144 610 2584 10946 46368 196418 832040 3524578 14930352
[1, 1, 2, 3, 5]
[8, 13, 21, 34, 55]
[89, 144, 233, 377, 610]
[987, 1597, 2584]
1 5 21 89 377 1597 6765 28657 121393 514229
```