Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

Publications

Funded Projects

Research Projects

Related Events

Exam of Advance in Programming

1 March 2012

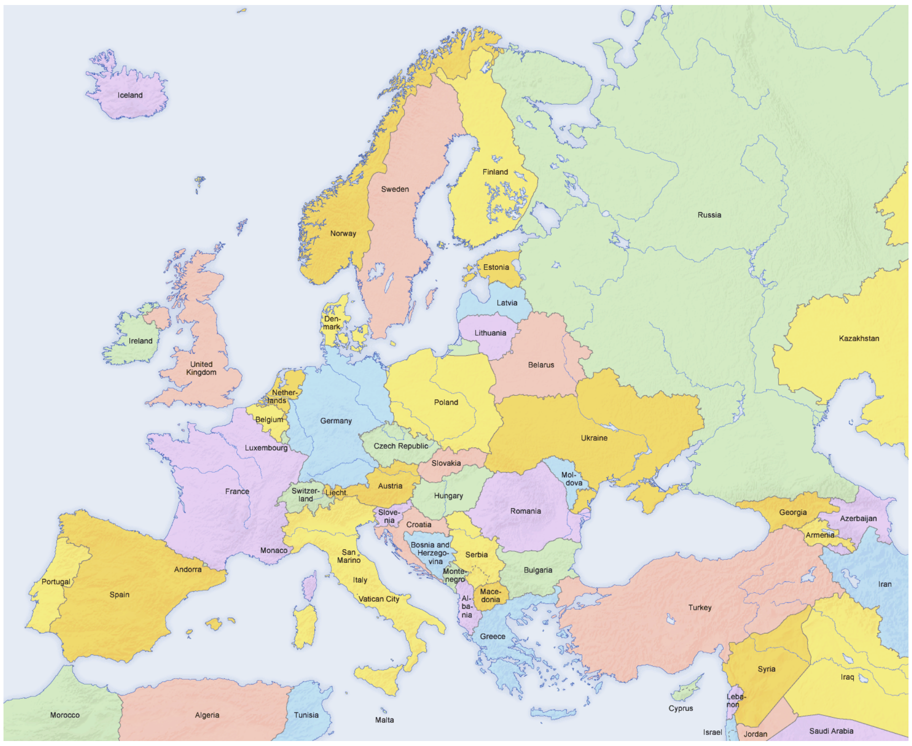
Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. All the exercises have the same value: 10; the submission with only one exercise will not be evaluated at all.

Exercise 1: The Longest Chain of Countries.

This is a variant of a game we had done in our childhood. Players take turns naming countries. Each country chosen must begin with the same letter that ended the previous element (repetition is not allowed). The game begins with any arbitrary country chosen by player 1 and ends when a player loses because he or she is unable to continue.

The exercise consists of writing a function `path` that calculates, given a country name, the longest possible path (measured in terms of number of words) yielded following the above rules. Note that when the longest path is not unique the solution must be chosen as sorted in lexicographical order.

In the exercise take in consideration European countries and since Europe is tightly coupled to Asia let exclude Kazakhstan, Georgia, Azerbaijan, Cyprus and Turkey from Europe, i.e., consider Europe as an island separated from Asia by the absence of the listed countries. To help those with problems in geography this is a political map of Europe to be used.



The following is an example of the expected behavior.

```
from geochain import *

if __name__ == "__main__":
  print("the longest chain starting from {0} is {1}".
     format('italy', path('italy')))
  print("the longest chain starting from {0} is {1}".
     format('spain', path('spain')))
  print("the longest chain starting from {0} is {1}".
     format('switzerland', path('switzerland')))
  print("the longest chain starting from {0} is {1}".
     format('luxembourg', path('luxembourg')))
  print("the longest chain starting from {0} is {1}".
     format('belarus', path('belarus')))
  print("the longest chain starting from {0} is {1}".
     format('belgium', path('belgium')))
  print("the longest chain starting from {0} is {1}".
[11:57]cazzola@surtur:~/pa>python3 main-geochain.py
the longest chain starting from italy is ['italy']
the longest chain starting from spain is
   ['spain', 'netherlands', 'serbia', 'albania', 'andorra', 'austria']
the longest chain starting from switzerland is ['switzerland', 'denmark']
the longest chain starting from luxembourg is
   ['luxembourg', 'greece', 'estonia', 'albania', 'andorra', 'austria']
the longest chain starting from belarus is
   ['belarus', 'spain', 'netherlands', 'serbia',
    'albania', 'andorra', 'austria']
the longest chain starting from belgium is
   ['belgium', 'macedonia', 'albania', 'andorra', 'austria']
the longest chain starting from portugal is
   ['portugal', 'liechtenstein', 'netherlands', 'serbia',
```

Exercise 2: Good Manners in Programming.

Traditionally python has some limitations towards the support of the object-oriented concept of data hiding. You can't define a field that is really private also the _ provides just a palliative to the privateness based on name mangling but the name is still there and can be accessed without big troubles.

Normal «good manners» in object-oriented programming establishes that an object status should be private to the object itself and can be accessed **only** through special methods called selectors (getters and setters). A getter grants the access to the content of a given field and the corresponding setter permits to modify its content.

The exercise consists of providing a mechanism based on two decorators: `@private`, `@selectors`. The `@private` decorator permits to list the names (both fields and methods) that should be made private whereas the `@selectors` decorator permits to specify which selectors to create (a dictionary of fields indexed on 'get' and 'set' is passed to the decorator). Note that a selector can be created **only** for private fields defined by the `@private` decorator. Of course selector are created dynamically and not written by the programmer and their name is after the name of the field, e.g., if `x` is the private field, the getter is named `getX` and the setter `setX` with the first letter of the attributed replaced by the corresponding capital letter.

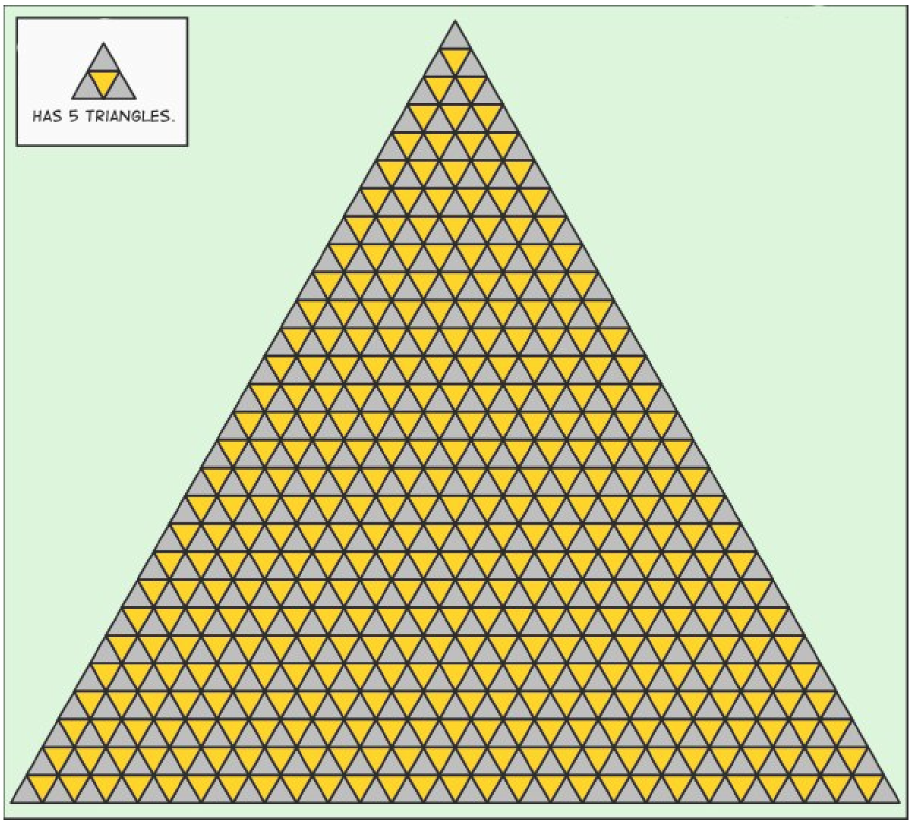
The following is an example of the expected behavior.

```python
@selectors({'get': ['data','size'], 'set':['data', 'label']})
@private('data', 'size')
class E:
   def __init__(self, label, start):
       self.label = label
       self.data = start
       self.size = len(self.data)

@selectors({'get': ['data','size'], 'set':['data', 'label']})
@private('data', 'size', 'label')
class D:
   def __init__(self, label, start):
       self.label = label
       self.data = start
       self.size = len(self.data)
   def double(self):
       for i in range(self.size):
           self.data[i] = self.data[i] * 2
   def display(self):
       print('{0} => {1}'.format(self.label, self.data))

if __name__ == "__main__":
  try:
    Y = E('Wrong Label', ['a', 'b'])
  except Exception as e: print("{0}: {1}".format(type(e).__name__, e))
  X = D('X is', [1, 2, 3])
  X.display(); X.double(); X.display()
  try:
    print(X.size)
  except Exception as e: print("{0}: {1}".format(type(e).__name__, e))
  try:
    print(X.data)
  except Exception as e: print("{0}: {1}".format(type(e).__name__, e))
  print(X.getData())
  X.setData([25])
  print(X.getSize())
  try:
    X.setSize(25)
  except Exception as e: print("{0}: {1}".format(type(e).__name__, e))
  try:
    print(X.getLabel())
  except Exception as e: print("{0}: {1}".format(type(e).__name__, e))
  X.setLabel('It has worked!!!')
  X.display()
```

```
[10:58]cazzola@surtur:~/pa>python3 main-selectors.py
TypeError: attempt to add a selector for a non private attribute: label
X is => [1, 2, 3]
X is => [2, 4, 6]
TypeError: private attribute fetch: size
TypeError: private attribute fetch: data
[2, 4, 6]
3
AttributeError: 'D' object has no attribute 'setSize'
AttributeError: 'D' object has no attribute 'getLabel'
```

Exercise 3: Magic Triangle.

The following picture shows a so called magic triangle that is recursively composed of several triangles and so on.



The exercise consists of writing a generator (`magic`) that at each step provides the number of triangles composing the whole picture considering it as composed of n stages each high as the traingle in the legend on the left.

The following is an example of the expected behavior.

```
from magic import *

if __name__ == "__main__":
  t = magic()
  print("... at step {0} you have {1} triangles".format(1, next(t)))
  next(t)
  print("... at step {0} you have {1} triangles".format(3, next(t)))
  next(t)
  next(t)
  next(t)
  print("... at step {0} you have {1} triangles".format(7, next(t)))
  next(t)
  next(t)
  next(t)
  next(t)
  next(t)
  next(t)
  print("... at step {0} (the whole picture) you have {1} triangles".
```

```
[16:40]cazzola@surtur:~/pa/es3>python3 main-magic.py
... at step 1 you have 5 triangles
... at step 3 you have 45 triangles
... at step 7 you have 245 triangles
```

Last Modified: Thu, 29 Mar 2012 10:27:11 ADAPT Lab.