Introduction to Programming Lecture 5

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

Remarks on homework and questions

I think we should spend a little time working on the homework in class to see if anyone needs any help. Those of you that have finished the homework can help others. If you are done and have no one to help, here is an exercise.

Exercise. You may have noticed that `a**n % m` is very very slow for large `a` and `n`. Write a function `mod_power(a,n,m)` that does this operation much faster.

Collatz

In [1]:

```
def collatz_max(n) :
    """ Returns a pair (a,s) where s in the maximum length
    of a Collatz sequence starting with 0 < x < n and a is
    a starting point of such a sequence. """
    # we use a dictionary to keep track of all of out known
    # Collatz sequence lengths
    known = {1 : 0}
    max_steps = 0
    max_start = 1
    for m in range(2, n) :
        m_steps = 0
        # we will save the seqeunce to save the steps later
        seq = []
        curr = m
        while True :
            # we see if we have already recorded steps for curr
            # note that dict.get(key) will return None if the key is
            # not in the dictionary
            curr_steps = known.get(curr)
            if curr_steps is not None :
                # we now know there are curr_steps more to go
                # so we don't need to waste our time computing
                m_steps += curr_steps
                # record all new steps counts for the sequence
                for idx,val in enumerate(seq) :
                    known[val] = m_steps - idx
                break
            # we will not save curr and procede with the
            # generation of the next sequence element
            seq.append(curr)
            if curr % 2 == 0 :
                curr = curr // 2
            else :
                curr = 3 * curr + 1
            m_steps += 1
        # we have left the while loop, let's see if more steps
        # were taken
        if m_steps > max_steps :
            max_steps = m_steps
            max_start = m

    return (max_start, max_steps)
```

Tower of Hanoi

Be sure to write auxiliary functions to make your code more readable!

In [2]:

```
def num_steps(n) :
    """ Returns the number of steps needed to solve
    the Tower of Hanoi puzzle with n disks. """
    assert n > 0
```

```python
    if n == 1 :
        return 1
    else :
        return 2 * num_steps(n-1) + 1

import copy

def append_to_peg(state, peg, value) :
    """ Appends value to the end of the list state[peg].
    Does *not* check if value > max(state[peg])! """
    state[peg].append(value)
    # we don't return anything because we modified state

def switch_pegs(state, some_peg, other_peg) :
    """ Switches values of some_peg and other_peg in state. """
    switched = { some_peg : state[other_peg],
                 other_peg : state[some_peg] }
    # recall that .update merges/overwrites the contents
    state.update(switched)
    # we don't return anything because we modified state

def solution_states(n) :
    """ Prints a list of solution states (which are dictionaries)
        to solve the Tower of Hanoi problem with n disks """
    if n == 1 :
        sol_steps = [ { 'a' : [1], 'b' : [ ], 'c' : [ ] } ,
                      { 'a' : [ ], 'b' : [ ], 'c' : [1] } ]
        return sol_steps
    else :
        # we will use the solution from n - 1 to first
        # move the top n - 1 to peg 'b', then move disk n
        # to peg 'c', and finally reuse the n - 1 solution
        # to move everything to peg 'c' from peg 'b'
        start = solution_states(n-1)
        # we make a deep copy so that when we modify start's
        # elements, we don't change what's in finish !
        finish = copy.deepcopy(start)

        for i in range(len(start)) :
            # we add disk n to the 'a' peg and switch 'b', 'c' pegs
            # so that we are moving disks from 'a' to 'b'
            switch_pegs(start[i], 'b', 'c')
            append_to_peg(start[i], 'a', n)
            # we must move disk n to peg 'c' and then use the n - 1
            # solution to move everything from peg 'b' to 'c'.
            # we accomplish this by making finish move disks from 'b'
            # to 'c' and making sure to add disk n onto peg 'c'
            switch_pegs(finish[i], 'a', 'b')
            append_to_peg(finish[i], 'c', n)

        return start + finish
```

Classes

So far, we have been mostly using built-in objects available to us from within python itself. Now, we will learn how to use classes to build our own objects. Let's start with a simple example.

In [3]:

```
class Dog :
    # Global class values
    breed = 'husky'
    tricks = ['sit']

    # instance initialization method
    def __init__(self, dog_name) :
        self.name = dog_name

    # instance method
    def say_name(self) :
        print("My name is", self.name)
```

The code above gives a description of how to build an object of type `Dog`. An object that conforms to this description is called an **instance** of type `Dog`.

Let's explore what we have built. We have defined a general object called `Dog` that describes how to build an **instance** of a dog.

In [4]:

```
# create a Dog instance
my_dog = Dog('Sam')

print(type(my_dog))
print(my_dog.name)
print(my_dog.breed)

my_dog.say_name()
```

```
<class '__main__.Dog'>
Sam
husky
My name is Sam
```

Above, when we call `Dog('sam')`, we create a new object of type `Dog`. During this creation, python calls the `__init__()`, which stands for **initialization**.

Notice the presence of the `self` variable in the function declaration of `__init__`. The keyword `self` references the object **instance** we have just created and **must be included as the first argument of any instance method**.

For example, when in python you write

```
a = int('123')
```

the interpreter creates an type `int` object. At first this object has no value, but right after creation, the `__init__(self, '123')` method for `int` is called. There, `self` points to this new integer object. Inside the `__init__` method, the code decides how to interpret the string `'123'` as an integer and sets the value. After all this is done, the variable name `a` is set to point to the newly created `int` type object with value `123`.

In defining the `Dog` class, we have created a `class` object called `Dog` that describes how to build an **instance** of a dog. Just like how `int` describes how to build integers.

Instanced have local attributes

The `name` attribute is an example of a local instance attribute. That is, we can create two different objects of type `Dog` with different names.

In [5]:

```
other_dog = Dog('jack')
print("My dog is named", my_dog.name,
      'while my other dog is named', other_dog.name)

# We should probably capitalize the name
other_dog.name = 'Jack'
other_dog.say_name()

# By the way, other_dog knows the same tricks as *all* dogs!
print(other_dog.tricks)
```

```
My dog is named Sam while my other dog is named jack
My name is Jack
['sit']
```

Notice that above we have **direct access** to the name attribute! Even though the `__init__` method set it to `'jack'`, I redefined it without the object `other_dog` knowing about it.

Class global attributes are shared by everyone

The attributes `breed` and `tricks` are **class** variables, which means that if we change them, they changes everywhere.

In [6]:

```
print(my_dog.breed)
Dog.breed = 'terrier' # I change the class!
print(my_dog.breed) # but the instance chagnes too!

# Similarly, ticks can be modified
my_dog.tricks.append('down')
print(Dog.tricks)
print(other_dog.tricks)
```

```
husky
terrier
['sit', 'down']
['sit', 'down']
```

Getters, setters, and @property

One way to "hide" instance variables is to use an _ before the variable name. Additionally, we will always use the `@property` decorator along with getter and setter methods. This also allows you to make certain checks and control what you return. Here is an example,

In [7]:

```
class Dog :
    # instance initialization method
    def __init__(self, dog_name) :
        self.name = dog_name

    # Properties
    # will be called whenever .name is used
    @property
    def name(self) :
        return self._name

    # will be called whenever .name = value is used
    @name.setter
    def name(self, dog_name) :
        if type(dog_name) is not str :
            raise ValueError("Dog name must be a string")
        self._name = dog_name.title()

    # instance method
    def say_name(self) :
        print("My name is", self.name)
```

In the new `Dog` class, whenever we call `.name` we are now using the special function right after the `@property` decorator. As you can see, when setting the name, I am checking the type and case. Let's see how this works.

In [8]:

```
my_dog = Dog('sam')
my_dog.say_name()
```

```
My name is Sam
```

In [9]:

```
my_dog.name = 1234
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-1a837d6bc8ca> in <module>()
----> 1 my_dog.name = 1234

<ipython-input-7-24d9da9b7c38> in name(self, dog_name)
     14     def name(self, dog_name) :
     15         if type(dog_name) is not str :
---> 16             raise ValueError("Dog name must be a string")
     17         self._name = dog_name.title()
     18 

ValueError: Dog name must be a string
```

As you see above, when I try to set the name `my_dog.name = 1234`, my `@name.setter` method is called instead of a direct access to a data attribute!

Let's look at a slightly more complicated examples. A class that stores the data of a graph.

In [10]:

```
class Graph :
    def __init__(self, verts, edges) :
        self.vertices = verts
        self.edges = edges

    @property
    def vertices(self) :
        # return a *copy* of your internal data
        return set(self._vertices)

    @vertices.setter
    def vertices(self, verts) :
        self._vertices = set(verts)

    @property
    def edges(self) :
        # return a *copy* of your internal data
        return set(self._edges)

    @edges.setter
    def edges(self, edges) :
        # let's check that edge endpoints are
        # in vertices
        endpts = set()
        for e in edges :
            if len(e) != 2 :
                raise ValueError("Edges must be pairs")
            endpts.update(e)
        if not endpts.issubset(self.vertices) :
            raise ValueError("All edge edpoints must be in vertices")
        self._edges = set(edges)

    def num_components(self) :
        """ Returns the number of connected components.
        Note : NOT efficient """
        # we will start with giving each vertex it's own *cluster*
        # as we go through the edges, we will merge clusters
        vert_to_clust = { v : {v}    for v in self.vertices }
        for v1,v2 in self.edges :
            c1 = vert_to_clust[v1]
            c2 = vert_to_clust[v2]
            if c1 != c2 :
                c1.update(c2)
                for v in c2 :
                    vert_to_clust[v] = c1
        # we must now count the distict sets we have
        # in vert_to_clust.values().
        clusters = frozenset(map(frozenset,vert_to_clust.values()))
        return len(clusters)
```

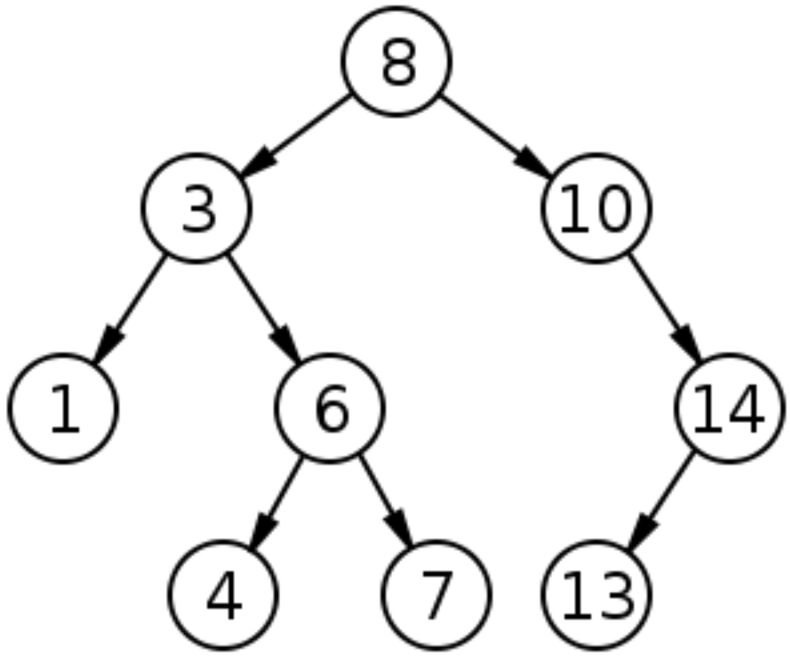
In [11]:

```
my_graph = Graph((0,1,2,3), { (0,1), (2, 1) })
print(my_graph.edges)
print(my_graph.vertices)

print('-'*20)

print("""We can still access the internal data, but good
        programmers will not accidentally do that!\n""")
print(my_graph._vertices)

print('-'*20)

print("""Notice that we are really returning a copy !\n""")
print(id(my_graph.vertices))
print(id(my_graph._vertices))

print('-'*20)

print("""We can see how many connected components there are.\n""")
print("My graph has" , my_graph.num_components(), "connected components.")
```

```
{(0, 1), (2, 1)}
{0, 1, 2, 3}
--------------------
We can still access the internal data, but good
        programmers will not accidentally do that!

{0, 1, 2, 3}
--------------------
Notice that we are really returning a copy !

4563691560
4563691784
--------------------
We can see how many connected components there are.

My graph has 2 connected components.
```

Binary Trees

A binary tree is a structure that looks like this :



Every circle is called a **node** of the tree. Each node as some data stored inside (a number in the above case). Most nodes also have a left and right child node. A nodes that don't have children are called **terminal** nodes. The top node in the image above is called the **root** node. As you can see, the **root** node does not have a **parent**.

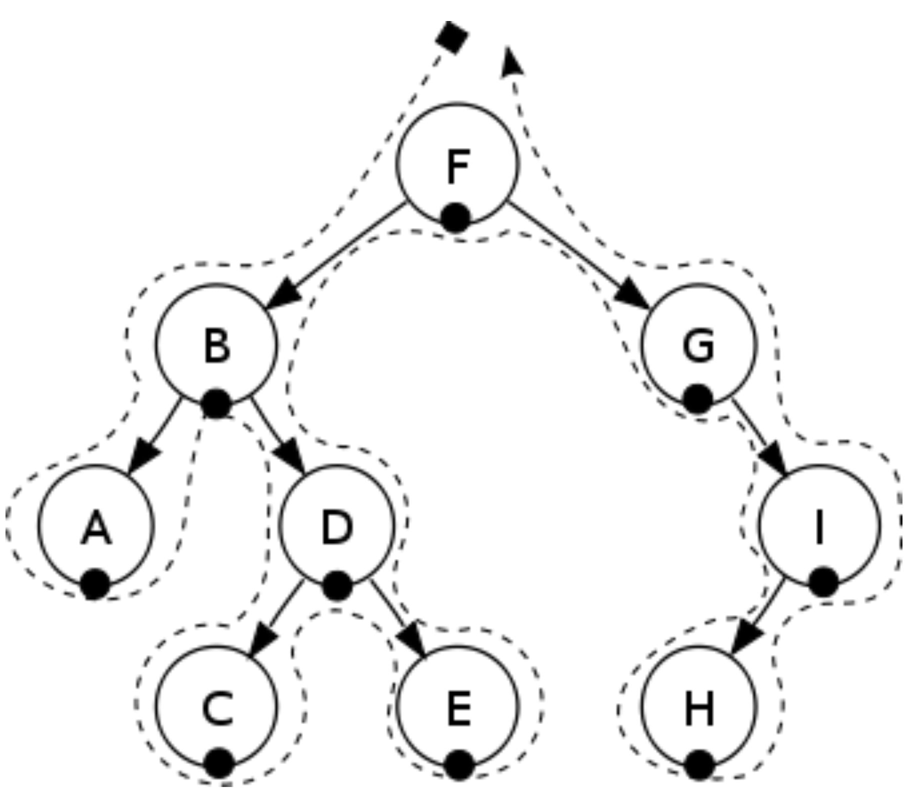
- **a.** Create a class called `Node`. It should have properties `.parent`, `.left`, `.right`, and `.data`. The `.parent`, `.left` and `.right` properties should again be `Node` instances or `None`. To simulate a node not having children (or a parent) we can set those properties to `None`.

Your `__init__` method should be of type

```
def __init__(self, data = None, left = None, right = None)
```

When writing the setter for `.parent`,`.left` and `.right`, be sure to check that the object you are setting are instances of class `Node` or that they are `None`.

- **b.** A binary tree can be represented by its starting root node. In fact, given any node, you can read off the (sub)tree below it by looking at its children. Write a module global function called `test_tree` which returns the root node of the binary tree in the above picture.
- **c.** Frequently, it is useful to read the data of the tree is a specific oder. Create a **recursive** instance method called `.inorder` which returns a list containing the data of the tree in the following order :



So, if `my_tree` is the tree in the above image, `my_tree.inorder() = ['A','B',...,'H','I']`