Assignment_00

April 14, 2021

1 Assignment 00 - Python

As an introduction to the assignments, we will start with a few basic Python operations and concepts. If anything is unlcear or if you find errors, feel free to post in the forum set up in Ilias.

You can submit incomplete assignments that don't validate. If a test cell validates correctly, you will get the points. If not, you will receive feedback from us with some remarks about what might be wrong.

1.1 Operators (1 point)

In the lecture, we haven't talked about the operators `//`and `%`. Use the following cell to check what they do using different numbers:

[ ]:
```
a = 317
b = 17

print(a // b)
print(a % b)
```

Now that you know what these operators produce, you should see how you can use the results and the value of variable `b` to reconstruct number `a`.

Write a function that takes as input the result of operation `//` called `intdiv`, the result of operation `%` called `mod` and number `b`. It should calculate number `a` and `return` the result:

[ ]:
```
def reconstruct(intdiv, mod, b):
    # YOUR CODE HERE
    raise NotImplementedError()
```

The cell above contains a comment saying `### YOUR SOLUTION HERE` and a `NotImplemented` exception, both of which you can delete, so you can place your solution there.

You can test your function in the following cell:

[ ]:
```
a = 317
b = 17

result = reconstruct(a // b, a % b, b)
print(result)
```

The following cell contains tests. These will be used to determine if you solved the problem correctly (or creatively tricked the system). Taking a look at them might give you some hints about what is expected here, but sometimes some of the tests, or all of them, will be hidden.

When the check above yields the correct number, execute the test cell below to see if your function does the correct thing:

[ ]:
```
assert reconstruct(317 // 17, 317 % 17, 17) == 317
import random as r
import math as m
a = m.floor(100*r.random())
b = m.floor(100*r.random())
assert reconstruct(a // b, a % b, b) == a
```

If anything went wrong, the above cell will throw an `AssertionError` and show you which test failed. Take a good look at it to get some hints as to what went wrong. Always make sure your functions are as general as possible.

1.2 Modules (1 point)

Sometimes you need a combination of different functions that you imported more than once. In that case it often makes sense to write a function that wraps the functionality of this combination.

Write a function that takes a number as input and returns $\frac{1}{2}\left(\exp\left(x\right) - \exp\left(-x\right)\right)$:

[ ]:
```
# you need to import math here

def sh(x):
    # YOUR CODE HERE
    raise NotImplementedError()
```

In this case, there is a hidden test below that will check if you correctly implemented the desired functionality. Do you recognize which function is built here?

[ ]:
```
import math as m
assert sh(0) == 0.0
assert sh(m.log(0.5*(1+5**0.5))) == 0.5
```

1.3 Cancellation (1 point)

We talked about cancellation problems a little bit in the lecture. A consequence of this problem is that sometimes, when you check the result of an operation to be zero, your check will fail, since it is not exactly zero. Numerical zeros are extremely small though, so in most cases it suffices to compare the result to a very small number, the **tolerance**, that you have to define sensibly as a programmer.

Write a function that takes as input a result `res` from some calculation and a tolerance `tol`. It should return `True` if the result is close to zero within the given tolerance and return `False` otherwise:

[ ]:
```
def zero_check(res, tol):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
import numpy as np
assert zero_check(0.1 + 0.1 + 0.1 - 0.3, 1e-16) == True
assert zero_check(np.exp(np.arcsinh(0.5)) - 0.5 * (1+5.00005**0.5), 1e-6) ==␣
↪False
```

1.4 Lists (1 point)

Assume you have a set of values that you want to test whether they are zero with a fixed, given tolerance. These values are in the list `values`.

Create a list called `checks` that contains the checks for all values using your function `zero_check` from above. *Hint*: there is an easy way to do this in a single line.

[ ]:
```
tol = 5e-7
vals = [1.73275932e-7, 7.02650253e-7, 6.17275235e-6, -5.21986805e-6, 1.
↪25262623e-8, \
        3.31753765e-7, 8.36552362e-6, -6.08163535e-8, 8.01356532e-7, 3.
↪75065322e-6]

# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```

```

1.5 Classes (5 points)

Although you won't have to use object-oriented programming during the course, it makes sense to understand roughly what classes do and how they work. We will use an example from FEM here and try to think of it in an object-oriented way.

In FEM, you describe a model using **nodes** and **elements** in a **mesh**. A mesh consists of nodes, which are arranged in elements. We will constrain our analysis to elements here for brevity's sake. Two classes are needed here, `Node` and `Element`. Remember that a class has attributes (things it know) and methods (actions it can take). Additionally, elements *have* nodes associated with it, so the two need to be connected somehow. We will implement a very rudimentary version of that concept here.

Implement a `Node` class. It should have as attributes a `list` of coordinates, that describes its position in the mesh. Assume a 2d mesh, so you'll only need to save two coordinates per node. As methods, the node should be able to print its own coordinates, and to add a displacement to its coordinates.

Use the following skeleton code:

[ ]:
```
class Node():
    def __init__(self, x_coord, y_coord):
        # self.coordinates = [ ?? ]
        # YOUR CODE HERE
        raise NotImplementedError()

    def print_coordinates(self):
        # print( ?? )
        # YOUR CODE HERE
        raise NotImplementedError()

    def add_displacement(self, x_disp, y_disp):
        # self.coordinates[ ?? ] += ??
        # YOUR CODE HERE
        raise NotImplementedError()
```

Check that your class does what you expect below:

[ ]:
```
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)

node2.print_coordinates()
node2.add_displacement(0.1, 0.2)
node2.print_coordinates()
```

[ ]:
```
testnode = Node(0.5, 0.5)
assert type(testnode.coordinates) == type([])
```

[ ]:
```
testnode = Node(-0.8, 0.8)
testnode2 = Node(-0.8, 0.8)
testnode2.add_displacement(0.3, 0.7)
assert testnode2.coordinates == [testnode.coordinates[0] + 0.3, testnode.
 ↪coordinates[1] + 0.7]
```

Now write an `Element` class, that is instanciated empty, but can save a list of nodes. For adding a node, write a method that does this. Additionally, write a method that iterates over all nodes of the element and prints their coordinates. For later plotting, also write a function that returns a list of all nodal coordinates as a list called `get_nodal_coordinates()`. The

list should look like this ind the end: `[[x-coordinate_of_node1, y-coordinate_of_node1], [x-coordinate_of_node2, y-coordinate_of_node2], ...]`.

Use the following skeleton code:

[ ]:
```
class Element():
    def __init__(self):
        self.nodes = []

    def add_node(self, node):
        # YOUR CODE HERE
        raise NotImplementedError()

    def print_nodal_coordinates(self):
        # YOUR CODE HERE
        raise NotImplementedError()

    def get_nodal_coordinates(self):
        # YOUR CODE HERE
        raise NotImplementedError()
```

Check that your class does what you desire in the cell below:

[ ]:
```
# play with these lines and numbers, but
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)

element1 = Element()

element1.add_node(node1)
element1.add_node(node2)
element1.add_node(node3)
element1.add_node(node4)

element1.nodes[1].add_displacement(0.0, 0.45)
element1.nodes[2].add_displacement(0.25, 0.25)
element1.nodes[3].add_displacement(0.35, 0.1)

element1.print_nodal_coordinates()

## you might need to adjust the x- and y-limits here:
import matplotlib.patches as patches
import matplotlib.pyplot as plt

%matplotlib notebook
```

```
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-0.1, 2.0)

print(element1.get_nodal_coordinates())

ax.add_patch(patches.Polygon(xy=[*element1.get_nodal_coordinates()],␣
↪color="darkorange"))

plt.show()
```

[ ]:
```
from random import random
from math import isclose
node1 = Node(0.0, 0.0)
node2 = Node(0.0, 1.0)
node3 = Node(1.0, 1.0)
node4 = Node(1.0, 0.0)
u_r = [random(), random()]
node3.add_displacement(*u_r)
element1 = Element()
element1.add_node(node1)
element1.add_node(node2)
element1.add_node(node3)
element1.add_node(node4)
assert isclose(sum([sum(node.coordinates) for node in element1.nodes]), 4.0 +␣
↪sum(u_r))
```

[ ]:
```
assert type(element1.nodes) == type([])
for node in element1.nodes:
    assert type(node) == type(node1)
```

[ ]:
```

```