Introduction to Programming Lecture 8

- 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

Partition (reverse) lexicographic order

A **partition** of an integer n is a tuple $(a_0, a_1, \dots, a_k)$ such that $a_0 \geq a_1 \geq \dots \geq a_k \geq 1$ and $a_0 + \dots + a_k = n$.

Given partitions $\alpha = (a_0, a_1, \dots, a_k)$ and $\beta = (b_0, b_1, \dots, b_s)$ of n, one can define a complete ordering as follows. We say $\alpha > \beta$ if there is an index t such that $a_i = b_i$ for all $i < t$ and $a_t > b_t$.

For example, with `n = 6`, this ordering on paritions give us :

```
(6)
(5,1)
(4,2)
(4,1,1)
(3,3)
(3,2,1)
(3,1,1,1)
(2,2,2)
(2,2,1,1)
(2,1,1,1,1)
(1,1,1,1,1,1)
```

As you may notice, it is possible to generate the next element in the list above from the previous. To do this, you keep track of the last number (and position) $a_i ! = 1$. You can then replace the end

$$a_i, a_{i+1}, \dots$$

with

$$a_i - 1, a_i - 1, \dots, r$$

where thre are k repeats of $a_i - 1$ where $a_i + a_{i+1} + \dots = k(a_i - 1) + r$. Note, you should only include r if $r > 0$.

Remark On the homework, I had a partition as an increasing sequence. However, it is actually more efficient to work with a decreating sequence because appending or modyfing the tail end of a list is more efficient than inserting at the beginning. Feel free to keep your homework as is, or adapt this ordering if you prefer.

Undoing `git add`

In git, if you have accidentally called `git add` and now there is a file you do not want in your “Changes to be committed" list, you can do the following

```
git reset file_name
```

This will remove `file_name` from the “Changes to be committed” list and it will **not** change the contents of that file.

If you want to remove all files from the “Changes to be committed” and **not** change their contents, you can do

```
git reset
```

If you actually want to revert a file to how it was before you edited it

```
git checkout file_name
```

will **overwrite** `file_name` with the last version you saved/committed on that branch.

A remark on testing

I strongly suggest you start writing functions to test your algorithms. For example, if you are working on polynomial division, and you want to check that everything runs fine for some random polynomials, you could write :

In [1]:

```
from random import randint, random

def test_remainder_with_random(num_times) :
    for _ in range(num_times) :
        m = randint(1,10)
        n = randint(1,m)
        p_scale = randint(-10,10)
        q_scale = randint(-10,10)
        p = tuple([ p_scale*random() for _ in range(m) ])
        q = tuple([ q_scale*random() for _ in range(n) ])
        r = remainder(p,q)
        print(r)

def test_remainder() :
    assert remainder((1,2,4,4,6),(2,1,2)) == (3.5, 2.25)
    print("Passed remainder value check")
```

In [2]:

```
from poly import remainder

test_remainder()
# passed the test_remainder assertion
test_remainder_with_random(3)
```

```
Passed remainder value check
(-3.6966384437507855, -2.6106221646219936)
(-5.951073464800859, -1.9772251258161524, -5.232919557471874, -2.1
215612228488023)
()
```

Some modules for working with files

Just like you can work with files in the command line, you might want to list directories, copy files, and get system information from within your programs. The classic modules for this task are the `os`, `shutil`, and `glob` modules.

In [3]:

```
import os
import shutil
import glob
```

File paths

For the `os.path` module, paths are strings and can be manipulated as follows.

In [4]:

```
# get the present (or current) working directory
cur_path = os.getcwd()
print(cur_path)
```

```
/Users/yarmola/Teaching/python-course/lectures/week-8
```

In [5]:

```
# get the abosulte path of a relative path
abs_path_up = os.path.abspath('..')
print(abs_path_up)
```

```
/Users/yarmola/Teaching/python-course/lectures
```

In [6]:

```
# get the base name of a file, including the extension
file_name = os.path.basename('../pdf/Lecture 1.pdf')
print(file_name)
```

```
Lecture 1.pdf
```

In [7]:

```
# get the directory name
dir_path = os.path.dirname('../week-1/Lecture 1.html')
# NOTE : observe that the os.path module treats paths as strings
# so the dirname is the relative directory name
print(dir_path)
```

```
../week-1
```

In [8]:

```
# get both the dir and file
dir_name, file_name = os.path.split('../../../homework/pdf/Homework 1.pdf')
print(dir_name)
print(file_name)
```

```
../../../homework/pdf
Homework 1.pdf
```

In [9]:

```
# split off the file extension
root , ext = os.path.splitext('../week-1/Lecture 1.html')
print(root)
print(ext)
```

```
../week-1/Lecture 1
.html
```

In [10]:

```
# check if a path is absolute
print(os.path.isabs('../week-1/Lecture 1.html'))
print(os.path.isabs('/Users/'))
```

```
False
True
```

In [11]:

```
# returns a canonial path for a file
print(os.path.realpath('../week-1/../week-2/Lecture 2.html'))
```

```
/Users/yarmola/Teaching/python-course/lectures/week-2/Lecture 2.ht
ml
```

In [12]:

```
# joint one path relative to another (or several)
old_path = "/Users/yarmola/"
new_path = os.path.join(old_path, 'Teaching', 'python-course/lectures/week-9',
                        'Lecture 9.doc')
print(new_path)
# Notice that os.path.join takes care of all the slashes for you
```

```
/Users/yarmola/Teaching/python-course/lectures/week-9/Lecture 9.do
c
```

The `os.path` module **does not automatically check for you if a file at a given path actually exists**. To do this, you can specifically check :

In [13]:

```
# check if there really is a file at a given path
print(os.path.isfile(new_path))
```

```
False
```

In [14]:

```
# check if there really is a dir at a path
print(os.path.isdir('/Users'))
```

```
True
```

Copy, move, delete and rename files and directories

The `shutil` and `os` modules contains all the useful tools for these operations. Here are some examples

In [15]:

```
# copy file or directory
path_of_copy = shutil.copy('json_data.json', 'new_data.json')
print(path_of_copy)
```

```
new_data.json
```

In [16]:

```
# delete
os.remove(path_of_copy)
```

In [17]:

```
# make direcory
os.mkdir('test_dir_new')
```

In [18]:

```
# remove directory
os.rmdir('test_dir_new')
```

In [19]:

```
# change current working directory
# this is the same as `cd` in the terminal app
os.chdir('..')
```

Walking a directory tree

There are two useful tools to walk a directory tree. First there is `os.walk`

In [20]:

```
for root, dirs, files in os.walk('week-8/demo') :
    print('='*40)
    print("The root directory is :", root)
    print('-'*40)
    print("The root directory has subdirectories:", *dirs, sep='\n' )
    print('-'*40)
    print("The root directory has files:", *files, sep='\n')
    print('='*40)
    print('\n')
```

```
========================================
The root directory is : week-8/demo
----------------------------------------
The root directory has subdirectories:
demo2
demo3
----------------------------------------
The root directory has files:
file1
========================================


========================================
The root directory is : week-8/demo/demo2
----------------------------------------
The root directory has subdirectories:
----------------------------------------
The root directory has files:
file2
========================================


========================================
The root directory is : week-8/demo/demo3
----------------------------------------
The root directory has subdirectories:
----------------------------------------
The root directory has files:
file3
file4
========================================
```

The second one is uses the **wildcard** character `*` **in path names** (this is **not** the python list unpacking notation!). For example, you can consider a path `../Lectures/*.pdf` which **matches** any path where you replace `*` with a file name.

- `glob.iglob(path, recursive=False)`
  - return a possibly-empty list of path names that match `path`. The argument `path` can be either absolute (like `/Users/yarmola/*`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards.
  - if you set recursive = True then `**` will match any files and zero or more directories and subdirectories.

In [21]:

```
# print all files and directories relative to ..
for x in glob.iglob("../*") :
    print(x)
```

```
../projects
../solutions
../homework
../lectures
```

In [22]:

```
# get all pdf files in the given directory
for x in glob.iglob("pdf/*.pdf") :
    print(x)
```

```
pdf/Lecture 3.pdf
pdf/Lecture 2.pdf
pdf/Lecture 1.pdf
pdf/Lecture 5.pdf
pdf/Lecture 4.pdf
pdf/Lecture 6.pdf
pdf/Lecture 7.pdf
```

In [23]:

```
# get all pdf files two directories down
for x in glob.iglob("../*/*/*.pdf") :
    print(x)
```

```
../homework/pdf/Homework 1.pdf
../homework/pdf/Homework 3.pdf
../homework/pdf/Homework 2.pdf
../homework/pdf/Homework 6.pdf
../homework/pdf/Homework 7.pdf
../homework/pdf/Homework 5.pdf
../homework/pdf/Homework 4.pdf
../lectures/pdf/Lecture 3.pdf
../lectures/pdf/Lecture 2.pdf
../lectures/pdf/Lecture 1.pdf
../lectures/pdf/Lecture 5.pdf
../lectures/pdf/Lecture 4.pdf
../lectures/pdf/Lecture 6.pdf
../lectures/pdf/Lecture 7.pdf
```

In [26]:

```
# get all directories and files in the tree below ..
for x in glob.iglob("week-8/demo/**", recursive = True) :
    print(x)
```

```
week-8/demo/
week-8/demo/demo2
week-8/demo/demo2/file2
week-8/demo/demo3
week-8/demo/demo3/file3
week-8/demo/demo3/file4
week-8/demo/file1
```

Overloading operators

You might want to write a class where you can use the operators +, -, *,/, //, %, pow, etc. For example, you might want to build a (multiplicative) cyclic group.

In [27]:

```
class CyclicGroupElement :
    group_order = 10
    def __init__(self, power = 1) :
        assert isinstance(power, int)
        self._power = power % CyclicGroupElement.group_order
    def __mul__(self, other) :
        new_power = self._power + other._power
        return CyclicGroupElement(new_power)
    def __str__(self) :
        # return a string showing a human readable description of self
        return 'x^{}'.format(self._power)
```

In [28]:

```
a = CyclicGroupElement()
b = CyclicGroupElement(3)
```

In [29]:

```
a*b
```

Out[29]:

```
<__main__.CyclicGroupElement at 0x10ce90080>
```

In [30]:

```
print(a*b*b*b*b)
```

```
x^3
```

Above, I defined functions `__mul__(self, other)` and `__str__(self)`. As you may have guessed, `__mul__` is the function that is called when I use the multiply command. Notice that since the arguments of `__mul__` are ordered, we can, if we wish, define non-commutative multiplication.

The `__str__` method is used by the `print` function. The return value should always be a string that is a human readable description of your instance.

If we try to use any other operators, we will fail :

In [31]:

```
a+b
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-31-ca730b97bf8a> in <module>()
----> 1 a+b

TypeError: unsupported operand type(s) for +: 'CyclicGroupElement'
and 'CyclicGroupElement'
```

Here is a list of **some** operators you know and their corresponding function names. Below, everywhere you see objects `a,b` in the methods you have `self == a` and `other == b`

- `a + b` corresponds to `__add__(self, other)`
- `a - b` corresponds to `__sub__(self, other)`
- `a*b` corresponds to `__mul__(self, other)`
- `a/b` corresponds to `__truediv__(self, other)`
- `a//b` corresponds to `__floordiv__(self, other)`
- `a % b` corresponds to `__mod__(self, other)`
- `divmod(a,b)` corresponds to `__divmod__(self, other)`
  - you should have that `divmod(a,b) = (a//b, a % b)` for your implementation
- `a ** b` or `pow(a,b,n)` corresponds to `__pow__(self, other[, modulo])` where `modulo == n`
- `len(a)` corresponds to `__len__(self)` (if your object has some sense of "length")
- `a < b` corresponds to `__lt__(self, other)`
- `a <= b` corresponds to `__le__(self, other)`
- `a == b` corresponds to `__eq__(self, other)`
- `a != b` corresponds to `__ne__(self, other)`
- `a > b` corresponds to `__gt__(self, other)`
- `a >= b` corresponds to `__ge__(self, other)`
- `repr(a)` corresponds to `__repr__(self)`
  - this is the string you see in your interpreter if you just type `a` followed by ENTER
- `str(a)` corresponds to `__str__(self)`
  - this is the string you see when you call `print(a)`

You can find a full list at : https://docs.python.org/3/reference/datamodel.html#special-method-names

Let's add some more of these to our `CyclicGroupElement` object.

In [32]:

```
def CyclicGroup(group_order) :
    class CyclicGroupElement :
        def __init__(self, power = 1) :
            assert isinstance(power, int)
            self._power = power % self.group_order

        @property
        def group_order(self) :
            return CyclicGroupElement._group_order

        def __mul__(self, other) :
            # so that we don't crash comparing group orders
            if not isinstance(other, CyclicGroupElement) :
                # this tells python that we can't do the operation
                return NotImplemented
            return CyclicGroupElement(self._power + other._power)

        def __truediv__(self, other) :
            if not isinstance(other, CyclicGroupElement) :
                return NotImplemented
            return CyclicGroupElement(self._power - other._power)

        def __str__(self) :
            return 'x^{}'.format(self._power)

        def __repr__(self) :
            return str(self)

    CyclicGroupElement._group_order = group_order
    return CyclicGroupElement
```

In [33]:

```
# the cyclic group class that lets us build elements
C_10 = CyclicGroup(10)

# some elements
a = C_10(4)
b = C_10(12)

print(a)
print(b)
```

```
x^4
x^2
```

In [34]:

```
a*b
```

Out[34]:

```
x^6
```

In [35]:

```
a/b
```

Out[35]:

```
x^2
```

In [36]:

```
a*2
# tries to call a.__mul__(2)
# where other == 2, since we return
# NotImplemented, we get a TypeError
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-36-269f58b28805> in <module>()
----> 1 a*2
      2 # tries to call a.__mul__(2)
      3 # where other == 2, since we return
      4 # NotImplemented, we get a TypeError

TypeError: unsupported operand type(s) for *: 'CyclicGroupElement'
and 'int'
```

There are several key things we are doing here at the same time. Let us first focus on the `class CyclicGroupElement`. This class tells us how to build, multiply, divide, and represent elements on a cyclic group. Notice that when I write the multiplication or division functions, I am careful to check the type of `other`. The keyword `self` is always guaranteed to be of the correct type, however, we don't know about `other`. The `NotImplemented` keyword tells python that a `TypeError` has occurred because we don't know how to perform the operation.

The second key point is the **nesting of a class inside a function**. Let's look at what the function `CyclicGroup(group_order)` actually returns.

In [37]:

```
repr(CyclicGroup(10))
```

Out[37]:

```
"<class '__main__.CyclicGroup.<locals>.CyclicGroupElement'>"
```

So `CyclicGroup` is the `class CyclicGroupElement`, however, it's a little more than that. In fact, every time we call `CyclicGroup(n)` we obtain a **new class (description) object** with the specific condition that `CyclicGroup(n)._group_order == n`. This is a rather complicated concept, but we can demonstrate this was follows :

In [38]:

```
C_10 = CyclicGroup(10)
C_10_again = CyclicGroup(10)
```

In [39]:

```
id(C_10)
```

Out[39]:

```
140700292636616
```

In [40]:

```
id(C_10_again)
```

Out[40]:

```
140700292638200
```

You should literaly think of `G_10` and `G_10_again` as two **different** group of order 10.

In [41]:

```
a = C_10(4)
b = C_10_again(5)
a*b
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-41-62d608b3e606> in <module>()
      1 a = C_10(4)
      2 b = C_10_again(5)
----> 3 a*b

TypeError: unsupported operand type(s) for *: 'CyclicGroupElement'
and 'CyclicGroupElement'
```

The reason we get an error here is that `a` and `b` are in different groups. Programmatically speaking, they are of different types (i.e. instances of different classes). Thus, when in multiple I check `isinstance(other, CyclicGroupElement)` where `other == b`, I get `False`.

In [42]:

```
type(a) == type(b)
```

Out[42]:

```
False
```

Thus, every time we call `CyclicGroup(n)` we get a **new recipe** on how to make `CyclicGroupElement`'s. This is why we can have different groups with different orders and nothing will collide!

In [43]:

```
C_127 = CyclicGroup(127)
```

In [44]:

```
a = C_127(23)
b = C_10(6)
```