Introduction to Programming Lecture 6

- 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

- Hint for `mod_pow` : keep two varaibles, a result and a value you will repeadedly square
- Hint for `probably_prime` : use the Miller-Rabin theorem that 3/4 of the numbers {1,...,n-1} are Miller-Rabin witnesses for composite numbers n
- Hint for `Node.inorder` : use recursion!

Generators

As I have mentioned before, `map` and `range` objects **generate** the next objects as required instead of computing everything first. For example, I can made huge range objects in a matter of microseconds/nanoseconds, but if I want to make them into a list, it takes much longer.

In [1]:

```
timeit range_object = range(10000000)
```

```
303 ns ± 16.2 ns per loop (mean ± std. dev. of 7 runs, 1000000 loo
ps each)
```

In [2]:

```
timeit range_list = list(range(10000000))
```

```
408 ms ± 38.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each
)
```

Generators are implemented using the very clever `yield` keyword. Here is an example.

In [3]:

```
def natural_numbers() :
    """ Generators all natural numbers starting with 1."""
    n = 1
    while True :
        yield n
        n += 1
```

The `yield` keyword is similar to `return` but with one key difference. When `return` is executed, the stack frame associated to a function is destroyed and the value returned. When `yield` is executed, the state of the function is frozen, the stack frame kept, and a value returned. When you request the `__next__` value, the execution is resumed and the function runs until the next `yield` statement, if there is one.

In [4]:

```
nat_gen = natural_numbers()
max_allowed = 5
for x in nat_gen :
    print(x)
    if x >= max_allowed :
        break
```

```
1
2
3
4
5
```

In [5]:

```
print(next(nat_gen))
print(next(nat_gen))
```

```
6
7
```

If a generators `yield` commands are exhausted, it naturally ends like any iterator (by raising the `StopIteration` exception).

In [6]:

```
def repeat_gen(val, num_times) :
    count = 0
    while count < num_times :
        print("Step :", count)
        yield val
        count += 1
```

In [7]:

```
rp_gen = repeat_gen('a',3)

print(next(rp_gen))
print(next(rp_gen))
print(rp_gen.__next__())
```

```
Step : 0
a
Step : 1
a
Step : 2
a
```

In [8]:

```
next(rp_gen)
```

```
-----------------------------------------------------------------
---------
StopIteration                             Traceback (most recent c
all last)
<ipython-input-8-a1e1c207bcc8> in <module>()
----> 1 next(rp_gen)

StopIteration:
```

Here is a generator that generates the Fibonacci sequence

In [9]:

```
def fibonacci() :
    """ A Fibonacci sequence generator. """
    prev, curr = 0, 1
    while True :
        yield prev
        prev, curr = curr, curr + prev
```

In [10]:

```
fib = fibonacci()
for i in range(10) :
    print('F_{} = {}'.format(i,next(fib)))
```

```
F_0 = 0
F_1 = 1
F_2 = 1
F_3 = 2
F_4 = 3
F_5 = 5
F_6 = 8
F_7 = 13
F_8 = 21
F_9 = 34
```

Using generators, I can also create a much faster and more efficient randomized Fermat primality test.

In [12]:

```
import prime_tests

print(prime_tests.satisfies_fermat_fast(2**107+13))
%timeit prime_tests.satisfies_fermat_fast(2**107+13)
```

```
False
54.6 µs ± 9.78 µs per loop (mean ± std. dev. of 7 runs, 10000 loop
s each)
```

In [13]:

```
print(prime_tests.satisfies_fermat_fast(2**20+7))
%timeit prime_tests.satisfies_fermat_fast(2**20+7)
```

```
True
6.74 s ± 1.56 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Exercise. Write a generator `shuffle_gen(start, stop)` that returns random numbers between `start` and `stop` without repetition. Make sure this works for very large integers. Hint: look up the Fisher-Yates shuffle.

More on modules and function

Module name aliases and reloading

So far we have been loading modules by using `import module_name`. This process **executes all lines of the code in the file `module_name.py`**, creates an object `module_name`, and attaches to this object all the functions defined in `module_name.py` as function attributes and all (module) global variables as data attributes. For example, a file called `module_example.py` with the contents :

```
a = 7

print("Loaded with name :",__name__)

def print_a() :
    print(a)

def print_hello() :
    print("Hello!")
```

can be loaded with `import` and has the data attribute `module_example.a` and function attribute `module_example.print_a`.

In [14]:

```
import module_example
print('-'*10)
print(module_example.a)
print(module_example.print_a)
module_example.print_a()
```

```
Loaded with name : module_example
----------
7
<function print_a at 0x114da1f28>
7
```

As you can see above, the module as a global `__name__` variable. The global variable `__name__` is set to the filename when a module is imported. Later, when we learn to use out modules as **scripts**, the varaible `__name__` will become more important.

One thing we can do is create an **alias** for the module object by importing as follows.

In [15]:

```
import module_example as me
print('-'*10)
print(me.a)
print(me.print_a)
me.print_a()
```

```
----------
7
<function print_a at 0x114da1f28>
7
```

Calling `import` twice on a module that is already loaded, does not reload it. You may have noticed that `print("Loaded with name :",__name__)` didn't run again in the above code. Therefore, if you have made changes to your file and want to update the module in your interpreter, you will need to use the `importlib` module.

In [16]:

```
import importlib as impl
impl.reload(me)
```

```
Loaded with name : module_example
```

Out[16]:

```
<module 'module_example' from '/Users/yarmola/Teaching/python-cour
se/lectures/week-6/module_example.py'>
```

Sometimes you only need only a few functions from a module (say you want to use `math.log` in your code). You can use the `from module_name import attribute_name` format. For example,

In [17]:

```
from math import log, sin
# We now have a function named log loaded from
# the math module
print(log(60))
print(sin(60))
```

```
4.0943445622221
-0.3048106211022167
```

Function arguments * and ** notation

In python, it is possible to have function that takes arbitrarily many arguments. For example, we can call `map` in many different ways.

In [18]:

```
def mult_2(a,b) :
    return a*b

def mult_3(a,b,c) :
    return a*b*c

a = [1,2,3,4,5]
b = [4,5,6,7,8,9]
c = [9,8,7]

map_mult_2 = map(mult_2,a,b)
map_mult_3 = map(mult_3,a,b,c)

print(list(map_mult_2))
print(list(map_mult_3))
```

```
[4, 10, 18, 28, 40]
[36, 80, 126]
```

Defining functions with `*`-notation

You can define functions with this behavior yourself using the `*`-notation. Here is an example

In [19]:

```python
def print_args(*args) :
    # Now the variable args contains a list of
    # the input arguments!
    print(args)
    for a in args :
        print(a)

print_args(1,2,3,4,5)
print('-'*10)
print_args('x','y','z')
```

```
(1, 2, 3, 4, 5)
1
2
3
4
5
----------
('x', 'y', 'z')
x
y
z
```

You can also combine this with placement and keyword/optional arguments. For example,

In [20]:

```python
def print_fancy_args(prefix, *data, suffix = None) :
    for a in data :
        print(prefix, a, suffix)


print_fancy_args('x',3,4,'a', 'k', suffix = 'y')
print('-'*10)
# Notice what happens when I don't use a keyword
print_fancy_args('x',3,4)
print('-'*10)
# Notice what happens when I don't use a keyword
print_fancy_args('x',3,4,'y')
```

```
x 3 y
x 4 y
x a y
x k y
----------
x 3 None
x 4 None
----------
x 3 None
x 4 None
x y None
```

- **Warning :** In a previous lecture I forgot to mention that keyword/optional arguments must come **after all positional arguments**. This remains true for `*`-notation.

Unpacking data using `*` notation

You can also use `*`-notation to **unpack** return values of function (or any tuple). For example,

In [21]:

```
print(tuple(map(mult_2,a,b)))
first, second, *middle, last = tuple(map(mult_2,a,b))
print(first)
print(second)
print(middle)
print(last)
```

```
(4, 10, 18, 28, 40)
4
10
[18, 28]
40
```

Calling functions using `*` notation

We can also pass an iterable object as arguments to a function by using the `*`-notation. For example,

In [22]:

```
d = [2,4]
print(mult_2(*d))
```

```
8
```

In [23]:

```
map_over = (a,b,c)
result = map(mult_3,*map_over) # same as map(mult_3,a,b,c)
print(tuple(result))
```

```
(36, 80, 126)
```

Defining functions with `**`-notation

It is also possible to group together keyword/optional arguments into a dictionary object. Here is an example,

In [24]:

```
def print_args(req1,req2,*pos_rest, kw1 = None, **kwd_rest) :
        print(req1)
        print(req2)
        print(pos_rest)
        print(kw1)
        print(kwd_rest)

print_args(1,2,3,4,5, kw1 = 'something', b = 'x', a = 'y', c = 'z')
```

```
1
2
(3, 4, 5)
something
{'b': 'x', 'a': 'y', 'c': 'z'}
```

The `**kwargs` must always come **after** `*args`, as with positional and keyword arguments in general.

Calling functions using `**` notation

We can also pass a dictionary object with **string** keys as arguments to a function by using the `**`-notation. For example,

In [25]:

```
def mult_2(a = 0, b = 0) :
    return a*b

d = { 'a' : 2, 'b' :3 }

print(mult_2(**d))
```

```
6
```

In [26]:

```
def print_stuff(x, s = None, c = None) :
    print(x,s,c)

d = { 's' : 10, 'c' : 18 }

print_stuff('something', **d)
```

```
something 10 18
```

Input and Output

One important aspect of programming that we haven't talked about is how to interact with the user and with data. At our level, user interaction will mostly consist of command line arguments and reading and writing data. We will build baisc graphical user interfaces towrads the end of the course.

Interactive input

Simple interactive input can be a fun was to get started with use input. However, it is **not a common method used by programmers and researches**.

The basic interactive input technique is

```
user_in = input("Type in a number :")
# user_in will the a *str* containing the user text up until
# a new line character (i.e. the user hits ENTER)
a = int(user_in) # if we want an integer, we need to convert
```

In [27]:

```
user_in = input("Type in a number : ")
a = int(user_in)
print("You entered the number : ", a)
```

```
Type in a number : 5
You entered the number :  5
```

If you do not want a you code to crash, you should use a `try` blocks to encapsulate interactive user input!

In [28]:

```
try :
    list_str = input("Type a sequence of integers : ")
    print(list_str)
    ints = list_str.split(',')
    print(ints)
    ints = map(int, ints)
    print(ints)
    print("You entered the list : ", list(ints))
except :
    print("Please enter a sequence of integers separated by commas.")
```

```
Type a sequence of integers : 1,2,3,4
1,2,3,4
['1', '2', '3', '4']
<map object at 0x1108eecf8>
You entered the list :  [1, 2, 3, 4]
```

Reading files

We can use the `open` command to open (text) files to read line by line. For example, say I have a file called `some_manifold_data.csv` in my current working directory, then I can read it as follows.

In [31]:

```
file_handle = open('/Users/yarmola/Teaching/python-course/lectures/week-6/some_manifold_data.csv','r')
for line in file_handle :
    print(line)

file_handle.close()
```

```
s776,GGmgMNggMGmn

s776,GGmgNggMGn

s780,GGmgMGmnGMgm

s776,GGMgNggmGn

s774,GGmnGmGmnGGn

s647,GGMGGMgNgM

s780,GMgNgMGmgMgm

s785,GGmgNgmGGmn

s774,GGMnGGnGmGn

s785,GGMgNgMgNgM

s782,GGGMgMNgMNgM
```

We will mostly focus on reading and writing **text** files. In the above command works as `open(path_to_file, mode)`, where the `path_to_file` **must include the file extension**! The `path_to_file` can be a global path or a path **relative to your current working directory**. My current working directory is `/Users/yarmola/Teaching/python-course/lectures/week-6/`, so in the future, I will open this file with just the filename.

We can also use other methods to read the text data.

- `file_handle.read()` returns the **whole** content of the file as a string
- `file_handle.read(size)` return the at most (the first) `size` characters (or bytes) of the file
- `file_handle.readline()` returns the whole next line, including the end of line character
- `list(file_handle)` or `file_handle.readlines()` returns a list of all the lines in the file

In [32]:

```
file_handle = open('some_manifold_data.csv','r')
```

In [33]:

```
file_handle.readline()
```

Out[33]:

```
's776,GGmgMNggMGmn\n'
```

In [34]:

```
# you should always close your file once you
# are done using it!
file_handle.close()
```

The `\n` above is the new line/end of line character. On windows you might also see `\r` used as a new line character. In text files, these characters are invisible and are inserted into the text whenever you hit ENTER to go to a new line. Above, when we were reading the file in a `for` loop, our output was separated by blank lines precisely because `print` always starts on a new line, but it was also printing the tailing new line character from out read `line`.

To get rid of of the new line character in a newly read string, we can use `.rstrip()` or `.rstrip('\n')`

In [35]:

```
print('-'*10)
print('\n')
print('-'*10)
```

```
----------


----------
```

In [36]:

```
file_handle = open('some_manifold_data.csv','r')
data = []
for line in file_handle :
    clean_line = line.rstrip()
    fields = clean_line.split(',')
    data.append(fields)
file_handle.close()

# prints each element of data
# separated by a new line character
print(*data, sep = '\n')
```

```
['s776', 'GGmgMNggMGmn']
['s776', 'GGmgNggMGn']
['s780', 'GGmgMGmnGMgm']
['s776', 'GGMgNggmGn']
['s774', 'GGmnGmGmnGGn']
['s647', 'GGMGGMgNgM']
['s780', 'GMgNgMGmgMgm']
['s785', 'GGmgNgmGGmn']
['s774', 'GGMnGGnGmGn']
['s785', 'GGMgNgMgNgM']
['s782', 'GGGMgMNgMNgM']
```

When you read a file, the file object returned by `open` will keep track of its position in the file. You can view and change this position by using

- `file_handle.tell()` returns an integer giving the current position in the file counted by the number of characters (or bytes) from the beginning of the file
- `file_handle.seek(offset)` changes the current position in the file by moving `offset` number of characters from the beginning

In [37]:

```
file_handle = open('some_manifold_data.csv','r')
file_handle.seek(80) # move to 80 characters from beginning
print(file_handle.read())
file_handle.close()
```

```
mnGGn
s647,GGMGGMgNgM
s780,GMgNgMGmgMgm
s785,GGmgNgmGGmn
s774,GGMnGGnGmGn
s785,GGMgNgMgNgM
s782,GGGMgMNgMNgM
```

Writing files

Writing files is a slightly more dangerous operation as you can accidentally write over a file or completely erase it. To open a file for writing, we use the `open(path_to_file, mode)` command again, but here the `mode` parameter will be different.

Here are all the `mode` parameter options.

- `'r'` open for reading (default)
- `'w'` open for writing, **ERASING** the file first
- `'x'` open for exclusive creation, failing if the file already exists
- `'a'` open for writing, appending to the end of the file if it exists
- `'b'` binary mode
- `'t'` text mode (default)
- `'+'` open a disk file for updating (reading and writing)

If you are creating a **new** file that shouldn't already exist, use the `'x'` option to write it.

In [38]:

```
import random

random_ints = random.sample(range(1000), 5)
# if I change the 'x' to a 'w' this will overwrite the
# file everytime I run this code
file_handle = open('random_numbers.txt','x')
for i in random_ints :
    file_handle.write(str(i)+'\n')
file_handle.close()

f = open('random_numbers.txt','r')
print(f.read())
```

```
699
826
908
236
609
```

In [39]:

```
random_ints = random.sample(range(1000), 5)
file_handle = open('random_numbers.txt','a')
for i in random_ints :
    file_handle.write(str(i)+'\n')
file_handle.close()

f = open('random_numbers.txt','r')
print(f.read())
```

```
699
826
908
236
609
82
176
384
191
838
```

The final three modes `'b'`, `'t'`, `'+'` are modifier modes. For example, to read a binary file, you would use the mode `'rb'`, to write a `'xb'`, etc.

The `'+'` mode modifies allows for simultaneous reading and writing with, for example, the `'r+'` mode. However, I do **not recommend** using this mode unless you really really have to. It can also behave differently on windows and unix systems.

In [40]:

```
# open the file as binary
file_handle = open('random_numbers.txt','rb')
# read the first 20 bytes and
# print them *interpreted as a string!*
print(file_handle.read(20))
file_handle.close()
```

```
b'699\n826\n908\n236\n609\n'
```

Saving python data

It may happen that you want to save some python list of dictionary for later use. This can be accomplished by using the `repr` and `eval` commands in conjunction.

- `repr(object)` (tries to) returns a string representation of an object
- `eval(string)` (tries to) evaluate `string` as a string representation of an object and returns the object

In [41]:

```
a = {float(2**65) : 'some bad example here'.split()}
x = {'1' : a }
print(repr(x))
```

```
{'1': {3.6893488147419103e+19: ['some', 'bad', 'example', 'here']}
}
```

In [42]:

```
y = eval(repr(x))
print(y == x)
```

```
True
```

You can now save python `repr` data to a file and load it later.

Writing and reading json data

A better and more portable way to store list, dictionary, or other text data is to use a file format called **json**. It is used widely in web development, server configurations, and data messaging. This file format is very similar to a python dictionary.

In [43]:

```
import json

# show the json representaion of an object
print(json.dumps(x))
print(json.dumps(a))
```

```
{"1": {"3.6893488147419103e+19": ["some", "bad", "example", "here"
]}}
{"3.6893488147419103e+19": ["some", "bad", "example", "here"]}
```

Notice the quotation marks around the data. We can also write straight to a file

In [44]:

```
file_handle = open('json_data.json', 'x')
json.dump(x, file_handle)
file_handle.close()
```

We can then `json.load(file_handle)` to get our data back!

In [45]:

```
fp = open('json_data.json','r')
data = json.load(fp)
fp.close()

print(data)
```

```
{'1': {'3.6893488147419103e+19': ['some', 'bad', 'example', 'here'
]}}
```

Note : we will talk about how to check if a file exists, move, rename, and delete files and directories in the next lecture.