In [ ]:
```
BIG_NUMBER = 10**6
BIG_NUMBER_LIST = range(BIG_NUMBER)
BIG_NUMBER_ITER = xrange(BIG_NUMBER)
BIG_STR_LIST = ['sample string' for x in BIG_NUMBER_LIST]
BIG_STR_ITER = ('sample string' for x in BIG_NUMBER_ITER)


def perform_operations(operations, obj):
    '''Executes a set of operations on a particular object
    and prints the value of that object'''
    for operation in operations:
        print '>>>', operation
        exec(operation)
        print obj
        print '-' * 2
```

List Types v/s Iterator types

range v/s xrange

range

In [ ]:
```
%%timeit
# range
sum = 0
l = range(BIG_NUMBER)
for x in l:
    sum += x

# Access Operations
#print l[-1] # Pass

# Assignment operation
l[10] = -1 # Pass
#print l[10]
```

xrange

In [ ]:
```
%%timeit
# xrange
sum = 0
l = xrange(BIG_NUMBER)
for x in l:
    sum += x

# Access Operations
```

```
#print l[-1] # Pass

# Assignment operation
#l[10] = -1 # FAILS !!
#print l[10]
```

List Comprehensions v/s Generator Expressions

List Comprehensions

Syntax: `[expression for variables in iterable if condition]`

In [ ]:
```
%%timeit
# Using loops
square_list = []
for x in range(BIG_NUMBER):
    square_list.append(x**2)
```

In [ ]:
```
%%timeit
# List Comprehensions
square_list = [x**2 for x in range(BIG_NUMBER)]
```

Playing with List Comprehensions

In [ ]:
```
square_list = [x**2 for x in range(BIG_NUMBER)]
```

In [ ]:
```
# Playground
print square_list[:10]
print square_list[4]
```

Generator Expressions

Syntax: `(expression for variables in iterable if condition)`

In [ ]:
```
%%timeit
# Generator Expressions
square_list = (x**2 for x in range(BIG_NUMBER))
```

Playing with Generator Expressions

In [ ]:
```
square_list = (x**2 for x in range(BIG_NUMBER))
```

In [ ]:
```python
# Playground
print square_list.next()
```

Functions v/s Generators

Functions

```
def function(params):
    ....
    ....
    return something
```

In [ ]:
```python
import math
def get_me_factorials_till(n):
    '''Returns a list of factorials from 1 to n'''
    return_list = []
    for x in range(1, n):
        return_list.append(math.factorial(x))
    return return_list
```

In [ ]:
```python
# Playground
factorials = get_me_factorials_till(10)
print factorials
```

Generators

```
def function(params):
    ...
    ...
    yield something
```

In [ ]:
```python
import math
def generate_me_factorials_till(n):
    '''Generates factorials from 1 to n'''
    for x in range(1, n):
        yield math.factorial(x)

factorials = generate_me_factorials_till(10)
```

In [ ]:
```python
# Playground
print factorials

# Its an iterator
print factorials.next()
```

In [ ]:
```
# Playground
# So, it can be iterated on:
# Create a new generator, since the earlier one will be used up
factorials = generate_me_factorials_till(10)
for factorial in factorials:
    print factorial,
```

Set and Dictionary Comprehensions

Set Comprehension: `{value for value in expressions if conditions}`

Dict Comprehensions: `{key: value for key, value in expressionss if conditions}`

In [ ]:
```
# Playground
# Set comprehension
print {k for k in 'ABCDEFEDCBA'}

# Dictionary Comprehension
print {k: k+1 for k in range(10)}
```

import itertools

[itertools documentation]()

N-Queens Puzzle

In [ ]:
```
# N-Queens Puzzle
# Reference: http://code.activestate.com/recipes/576647-eight-queens-si
from itertools import permutations

n = 8
cols = range(n)
for vec in permutations(cols):
    if (n == len(set(vec[i]+i for i in cols))
          == len(set(vec[i]-i for i in cols))):
        print vec
```

import collections

deque

In [ ]:
```
import collections
# Creating a deque
```

```
d = collections.deque(['first', 'second', 'third', 'current last'])
print d
print '--'

# right rotation
d.rotate()
print '>>> d.rotate()'
print d
print '--'

# left rotation
d.rotate(-1)
print '>>> d.rotate(-1)'
print d
print '--'

# append from left side
d.appendleft("new first")
print '>>> d.appendleft("new first")'
print d
print '--'

# remove from the left side
d.popleft()
print '>>> d.popleft()'
print d
print '--'

d.append("new last")
print '>>> d.append("new last")'
print d
print '--'

d.pop()
print '>>> d.pop()'
print d
print '--'
```

Counter

In [ ]:

```
# Playground
from collections import Counter
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)

print c, d
print '--'

# add two counters together:  c[x] + d[x]
print '>>> c + d'
print c + d
print '--'

# subtract (keeping only positive counts)
print '>>> c - d'
```

```
print c - d
print '--'

# intersection:  min(c[x], d[x])
print '>>> c & d'
print c & d
print '--'

# union:  max(c[x], d[x])
print '>>> c | d'
print c | d
print '--'

text = "I saw Susie sitting in a shoe shine shop."

import re
c = Counter(re.split(r"\W+", text, flags=re.IGNORECASE))
print c
print '--'

new_text = "Where she sits she shines, and where she shines she sits."
print '>>> c.update(new_text)'
c.update(re.split(r"\W+", new_text, flags=re.IGNORECASE))
print c
print '--'

print '>>> c.most_common(3)'
print c.most_common(3)
print '--'
```

defaultdict

In [ ]:
```
from collections import defaultdict

# defaultdict takes a default_factory as an argument
# you can always subclass it to have it with your own
# default factory
default_dict = defaultdict(int)
print default_dict[10]
```

Memoization

Collatz Conjecture

```
The following iterative sequence is defined for the set of positive integers:

n ->  n/2 (n is even)
n -> 3n + 1 (n is odd)
```

```
Using the rule above and starting with 13, we generate the following sequence:
13 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

Which starting number, under one million, produces the longest chain?
```

In [ ]:

```python
# Iterative solution
def chain(i):
    '''Iterative approach to solve the chain problem'''
    c = [i]
    last_element = i

    while last_element > 1:
        if last_element % 2 == 0:
            last_element = last_element / 2
        else:
            last_element = 3 * last_element + 1
        c.append(last_element)
    return c

def the_simple_chain():
    start = 2
    limit = 1000000
    max_length = 1
    max_num = 1
    for i in xrange(start, limit):
        c = chain(i)
        if len(c) > max_length:
            max_num = i
            max_length = len(c)

    return max_num

%timeit the_simple_chain()
# takes around 37.7s
```

In [ ]:

```python
# Recursive solution
def chain(n):
    """
    Recursive function for the above expression
    """
    if n == 1:
        return [1]
    else:
        if n % 2 == 0:
            return [n] + chain(n / 2)
        else:
            return [n] + chain(3 * n + 1)

def the_recursive_chain():
    start = 2
    limit = 1000000
    max_length = 1
    max_num = 1
```

```python
    for i in xrange(limit, start, -1):
        chain_list = chain(i)

        if len(chain_list) > max_length:
            max_length = len(chain_list)
            max_num = i

    return max_num

%timeit the_recursive_chain()
```

In [ ]:

```python
# Recursive solution
cache = {}
def chain(n):
    """
    Memoized recursive function for the above expression
    """
    if n in cache:
        return cache[n]
    if n == 1:
        cache[1] = [1]
        return cache[1]
    else:
        if n % 2 == 0:
            cache[n] = [n] + chain(n / 2)
            return cache[n]
        else:
            cache[n] = [n] + chain(3 * n + 1)
            return cache[n]

def the_recursive_chain():
    start = 2
    limit = 1000000
    max_length = 1
    max_num = 1

    for i in xrange(limit, start, -1):
        chain_list = chain(i)

        if len(chain_list) > max_length:
            max_length = len(chain_list)
            max_num = i

    return max_num

%timeit the_recursive_chain()
```

Multiplication Problem

Naive Implementation

2 * 3 = 2 + 2 + 2

In [57]:
```python
# Naive Multiplication
def naive(a, b):
    x = a
    y = b
    z = 0
    while x > 0:
        z = z + y
        x = x - 1
    return z

%timeit naive(BIG_NUMBER, BIG_NUMBER)
# 1 loops, best of 3: 11.6 s per loop
```

```
10 loops, best of 3: 112 ms per loop
```

Russian Peasant's Algorithm

```
Algorithm:
----------
* Write each number at the head of a column.
* Double the number in the first column, and halve the number in the second column.
  * If the number in the second column is odd, divide it by two and drop the remainder.
* If the number in the second column is even, cross out that entire row.
* Keep doubling, halving, and crossing out until the number in the second column is 1.
* Add up the remaining numbers in the first column. The total is the product of your original numbers.

```
Let's multiply 57 by 86 as an example:
Write each number at the head of a column.

57   86
Double the number in the first column, and halve the number in the second column.

57   86
114   43
If the number in the second column is even, cross out that entire row.

57 XX   86 XX
114   43
Keep doubling, halving, and crossing out until the number in the second column is 1.

57 XX   86 XX
114   43
228   21
456 XX 10 XX
912   5
1824 XX 2
3648   1
Add up the remaining numbers in the first column.
```
```

```
114      43
228      21
```

In [ ]:
```
# Russian Peasant's Algorithm for Multiplication
def russian(a, b):
    x = a
    y = b
    z = 0
    while x > 0:
        if x % 2 == 1:
            z = z + y
        y = y << 1
        x = x >> 1
    return z

%timeit russian(BIG_NUMBER, BIG_NUMBER)
# 100000 loops, best of 3: 7.4 us per loop
```

In [ ]:
```
# Recursive Russian Peasant's Algorithm
def rec_russian(a, b):
    if a == 0:
        return 0
    if a % 2 == 0:
        return 2 * rec_russian(a/2, b)
    return b + 2 * rec_russian((a-1)/2, b)

%timeit rec_russian(BIG_NUMBER, BIG_NUMBER)
# 100000 loops, best of 3: 9.47 us per loop
```

this IPython notebook is a part of presentation at ***PyCon India 2012****, by* ***Dhruv Baldawa (@dhruvbaldawa)***