Homework3

February 3, 2017

1 Introduction to Python

2 Homework #3

3 Due Thurdsay, 2/9, Noon in Courseworks

- You MUST submit on Courseworks before it closes
- Email submissions are NOT accepted

4 Academic Honesty

- The computer science department has strict polices. Check the department web page for details.
- Do not look at anybody else's source code. Do not show anybody your source, or leave your source where somebody could see it. You MUST write your own code.
- For this class, feel free to discuss issues with other people, but suggest waiting an hour or two after a discussion, before writing your code.
- Cases of non original source will be refered to the Judical Committee.

4.1 Tips

In [55]:
```
# you can terminate a generator by using 'return',
# or falling off the end of the generator

def g1():
    yield(1)
    yield(2)
    return
    yield(3)

def g2():
    yield(1)
    yield(2)
```

In [83]:
```
# can get the elements from a
# FINITE length generator with 'list'
```

```
list(g1())
```

Out[83]: [1, 2]

In [82]:
```
list(g2())
```

Out[82]: [1, 2]

In [102]:
```
# if a generator calls a 2nd generator for elements,
# and the 2nd generator finishes, the 1st one will finish as well

def g3(g):
    while True:
        yield(next(g))

for i in g3(g2()):
    print(i)
```

```
1
2
```

In [100]:
```
# string replace method

'324234213foo1234324'.replace('foo','XYZ')
```

Out[100]: '324234213XYZ1234324'

In [61]:
```
# divmod gives integer quotient and remainder

divmod(13, 5)
```

Out[61]: (2, 3)

In [4]:
```
# 'bin' function yields the binary representation of a int
#  returns a string of 0/1's, plus a '0b' prefix

[bin(11), bin(31), bin(32)]
```

Out[4]: ['0b1011', '0b11111', '0b100000']

In [24]:
```
# 1 and 0 function as True and False in an 'if' statement

[True if 1 else False, True if 0 else False]
```

Out[24]: [True, False]

In [168]:
```
# the set constructor function can take a list

set([1,2,3])
```

```
Out[168]: {1, 2, 3}
```

```
In [169]: # an empty set prints as 'set()'

          set([])
```

```
Out[169]: set()
```

```
In [179]: # can check for presense of a key in a dictionary with 'in' operator
          d = {'foo':234}
          ['foo' in d, 'bar' in d]
```

```
Out[179]: [True, False]
```

5 Problem 1a - decimals

- define a 'decimals' generator function, that 'generates' the decimal digits of 1/n, where n is an integer greater than 1
- if the decimal expansion terminates, like 1/8 = .125, the generator should terminate. otherwise, like for 1/3=.333. . . , the generator should never stop
- use long division to compute the expansion - it is very simple

```
In [65]: # example: 1/8 = .125
         # the digits of the expansion are the quotients

         r = 10
         q,r = divmod(r,8)
         print(q,r)
         r *= 10
         q,r = divmod(r,8)
         print(q,r)
         r *= 10
         q,r = divmod(r,8)
         print(q,r)
         # r == 0, so done
```

```
1 2
2 4
5 0
```

```
In [67]: # 1/3 = .333...

         r = 10
         q,r = divmod(r,3)
         print(q,r)
         r *= 10
         q,r = divmod(r,3)
```

```
        print(q, r)

        # (q,r) pairs have repeated, will never terminate
```

```
3 1
3 1
```

```
In [52]: # finite generator

         list(decimals(8))
```

```
Out[52]: [1, 2, 5]
```

```
In [53]: # infinite generator
         # can't call 'list' on it

         g = decimals(3)
         [next(g), next(g), next(g), next(g)]
```

```
Out[53]: [3, 3, 3, 3]
```

6 Problem 1b - genlimit

- define 'genlimit(g, limit)', which generates at most 'limit' number of values from a generator 'g'

```
In [49]: list(genlimit(decimals(8), 5))
```

```
Out[49]: [1, 2, 5]
```

```
In [47]: list(genlimit(decimals(3), 5))
```

```
Out[47]: [3, 3, 3, 3, 3]
```

```
In [212]: # works with any interator

          list(genlimit(iter(range(30)), 3))
```

```
Out[212]: [0, 1, 2]
```

7 Problem 2 - Deal With Repeated Decimals

- genlimit is useful, but never sure what we're missing with an arbitrary limit
- since 1/n is a rational number, its decimal expansion must eventually repeat(unlike irrational numbers like PI)
- write 'decimals2', a variant of 'decimals'
- if the decimal expansion is finite, it should just return the finite set of digits

- if the decimal expansion repeats, it should return the digits up to the point it starts repeating. then the final yield should be a list of the repeating sequence of digits
- hint - keep a list, 'seen', of the [quotient, remainder] pairs as you generate digits. if you generate a new pair that is already in 'seen', you know you have started to repeat.

In [81]:
```
import textwrap

for j in range(3,30,2):
    d = list(decimals2(j))
    print('   Expansion of 1/' + str(j) + ':')
    # hack needed because lines don't wrap in pdf version
    print( textwrap.fill(str(d), 80))
```

```
   Expansion of 1/3:
[3, [3]]
   Expansion of 1/5:
[2]
   Expansion of 1/7:
[1, 4, 2, 8, 5, 7, [1, 4, 2, 8, 5, 7]]
   Expansion of 1/9:
[1, [1]]
   Expansion of 1/11:
[0, 9, [0, 9]]
   Expansion of 1/13:
[0, 7, 6, 9, 2, 3, [0, 7, 6, 9, 2, 3]]
   Expansion of 1/15:
[0, 6, [6]]
   Expansion of 1/17:
[0, 5, 8, 8, 2, 3, 5, 2, 9, 4, 1, 1, 7, 6, 4, 7, [0, 5, 8, 8, 2, 3, 5, 2, 9, 4,
1, 1, 7, 6, 4, 7]]
   Expansion of 1/19:
[0, 5, 2, 6, 3, 1, 5, 7, 8, 9, 4, 7, 3, 6, 8, 4, 2, 1, [0, 5, 2, 6, 3, 1, 5, 7,
8, 9, 4, 7, 3, 6, 8, 4, 2, 1]]
   Expansion of 1/21:
[0, 4, 7, 6, 1, 9, [0, 4, 7, 6, 1, 9]]
   Expansion of 1/23:
[0, 4, 3, 4, 7, 8, 2, 6, 0, 8, 6, 9, 5, 6, 5, 2, 1, 7, 3, 9, 1, 3, [0, 4, 3, 4,
7, 8, 2, 6, 0, 8, 6, 9, 5, 6, 5, 2, 1, 7, 3, 9, 1, 3]]
   Expansion of 1/25:
[0, 4]
   Expansion of 1/27:
[0, 3, 7, [0, 3, 7]]
   Expansion of 1/29:
[0, 3, 4, 4, 8, 2, 7, 5, 8, 6, 2, 0, 6, 8, 9, 6, 5, 5, 1, 7, 2, 4, 1, 3, 7, 9,
3, 1, [0, 3, 4, 4, 8, 2, 7, 5, 8, 6, 2, 0, 6, 8, 9, 6, 5, 5, 1, 7, 2, 4, 1, 3,
7, 9, 3, 1]]
```

8 Problem 3a - select

- define a function 'select(input, selectors)', where 'input' and 'selectors' lists are the same length
- 'select' returns a new list which consists of the elements of input that have a True value in the corresponding selectors element
- remember 'generalized booleans'

```
In [71]: select(range(7), [0, 1, '', 'foo', True, [], [1,2]])

Out[71]: [1, 3, 4, 6]

In [72]: select([x*3 for x in [4,2,1]] , [0,1,0])

Out[72]: [6]
```

9 Problem 3b - intToNDigits

- define a function 'intToNDigits(x, n)'
- returns a list of the digits(int 0 and 1, not strings) in a base 2 representation of 'x'
- list must have n digits, pad with 0 on the left if needed

```
In [2]: [intToNDigits(3, 2), intToNDigits(3, 6), intToNDigits(11, 4)]

Out[2]: [[1, 1], [0, 0, 0, 0, 1, 1], [1, 0, 1, 1]]
```

10 Problem 3c - powerSet

- using 'select' and'intToNDigits', define a function 'powerSet(x)' that returns a list of all possible subsets of the elements of input list x, including the empty set and the set of all elements
- if a set has N elements, the power set will have 2**N elements

```
In [75]: powerSet(['avery', 'math', 'butler'])

Out[75]: [set(),
          {'butler'},
          {'math'},
          {'butler', 'math'},
          {'avery'},
          {'avery', 'butler'},
          {'avery', 'math'},
          {'avery', 'butler', 'math'}]

In [76]: powerSet(['avery', 'math', 'butler', 'dodge'])

Out[76]: [set(),
          {'dodge'},
          {'butler'},
          {'butler', 'dodge'},
```

```
 {'math'},
 {'dodge', 'math'},
 {'butler', 'math'},
 {'butler', 'dodge', 'math'},
 {'avery'},
 {'avery', 'dodge'},
 {'avery', 'butler'},
 {'avery', 'butler', 'dodge'},
 {'avery', 'math'},
 {'avery', 'dodge', 'math'},
 {'avery', 'butler', 'math'},
 {'avery', 'butler', 'dodge', 'math'}]
```

In [77]:
```
len(powerSet(['avery', 'math', 'butler', 'dodge']))
```

Out[77]: 16

11 Problem 4 - generalized dot product(dotn)

- take the dot product of any number of lists and finite generators
- hints - refering to functions2
  - use the variable number of arguments format
  - you might find it convenient to 'spread a list of args' to 'zip'

In [41]:
```
def g(s, e):
    for j in range(s, e):
        yield j

dotn([5,3,9], g(10,12))
```

Out[41]: 83

In [43]:
```
# above is
# number of terms is length of shortest sequence

5*10 + 3*11
```

Out[43]: 83

In [40]:
```
dotn([2,3,7], g(0,3), g(2,8))
```

Out[40]: 65

In [37]:
```
# above is

2*0*2 + 3*1*3 + 7*2*4
```

Out[37]: 65

12 Problem 5a - countBases

- define 'countBases(dna)' - returns the number of 'A', 'C', 'G', 'T' bases in a strand of DNA in a dict

```
In [14]: # dna strings use upper case letters

         bases = 'ACGT'
         dna = 'CATCGATATCTCTGAGTGCAC'
```

```
In [16]: countBases('AACAT')
```

```
Out[16]: {'A': 3, 'C': 1, 'G': 0, 'T': 1}
```

```
In [17]: countBases(dna)
```

```
Out[17]: {'A': 5, 'C': 6, 'G': 4, 'T': 6}
```

13 Problem 5b - percentBases

- return the percentage of each base in a strand of DNA in a dict

```
In [27]: percentBases('ACG')
```

```
Out[27]: {'A': 0.3333333333333333,
          'C': 0.3333333333333333,
          'G': 0.3333333333333333,
          'T': 0.0}
```

```
In [24]: percentBases(dna)
```

```
Out[24]: {'A': 0.23809523809523808,
          'C': 0.2857142857142857,
          'G': 0.19047619047619047,
          'T': 0.2857142857142857}
```

14 Problem 5c - reverseComplement

- define 'reverseComplement(dna)'
- swaps A <-> T, C <-> G, and returns the new DNA in reverse order

```
In [196]: reverseComplement('ACGT')
```

```
Out[196]: 'ACGT'
```

```
In [197]: reverseComplement(dna)
```

```
Out[197]: 'GTGCACTCAGAGATATCGATG'
```