University of Zurich - HS 2013

Computational Science I
Exercise 1: Real numbers

Tobias Grubenmann

September 18, 2013

Exercise 1

Assuming Python uses *unsigned* 32-bit integers to expressing time then the largest number for expressing time is `0xffffffff=4294967295`. We can use the Python interpreter to evaluate the highest time expression:

```
from time import ctime
ctime(0xffffffff)
```

This evaluates to `Sun Feb 07 07:28:15 2106`.

If Python uses *signed* 32-bit integers to expressing time then teh largest number would be `0x7fffffff=2147483647` which evaluates to `Tue Jan 19 04:14:07 2038`.

Exercise 2

Assuming the floating point precision in Python follows the IEEE 754 standard, i.e. the mantissa is normalized and the highest and lowest exponent are treated different.

To get the bits of the mantissa we need to evaluate the smallest number ε such that $1 + \varepsilon > 1$

To get the bits of the exponent we need to evaluate the greatest and smallest possible exponent to calculate the range of the exponent. The range is the difference of the greatest and the smallest exponent plus 1 for the zero. To get the greatest possible exponent we need to increase the exponent until we get an *overflow exception* (plus 1 for the special treatment of the highest exponent). To get the smallest possible exponent we need to decrease the exponent until we get a loss of precision (plus 1 for the special treatment of the lowest exponent). The logarithm to the base 2 of the range gives us the number of bits for the exponent.

```
eps = 1.0
exponent = 0

while 1 + eps > 1 :
    # decrease eps as long as 1+eps>1
    exponent = exponent + 1
    eps = pow(2.0, -exponent)

mbits = exponent-1 # bits of the mantissa

print("The␣mantissa␣has␣%d␣bits␣+␣1␣bit␣for␣the␣sign" %
   mbits)

exponent = 0

while True :
    # increase exponent as long as no overflow occurs
    exponent = exponent + 1
    try:
        delta = pow(2.0, exponent)
    except (ArithmeticError):
        break

maxExponent = exponent - 1 # gratest possible exponent

exponent = 0

eps = pow(2.0, -mbits)

while delta + eps*delta > delta :
    # decrease exponent as long as there is no loss of
       precision
    exponent = exponent - 1
    delta = pow(2.0, exponent)

minExponent = exponent + 1 # smallest posssible exponent

ebits = log((maxExponent+1)-(minExponent-1)+1)/log(2) #
   bits of the exponent

print("The␣exponent␣has␣%d␣bits" % ebits)
```

The outcome of the code above is 52 bits for the mantissa (+1 for the sign) and 11

bits for the exponent.

Exercise 3

To use the Newton-Raphson method to find the n-th root of a number x we define the following function f:

$$f(y) = y^n - x$$

which has the derivative $f'(y) = n \cdot y^{n-1}$.

The function f vanishes at the same points as the function $g(y) = y - \sqrt[n]{x}$ but doesn't need the root function (which we want to implement).

I implemented also a function to calculate x^n since with the built-in `pow()` function we could directly calculate the n-th root and therefore my own function for the n-th root would be pointless.

The loop with the Newton-Raphson iteration stops when the absolute value of $\Delta x_{n+1} = \frac{f(x_n)}{f'(x_n)}$ is smaller than $x_n \cdot \varepsilon$ where ε is the smallest number s.t. $1 + \varepsilon > 1$.

```
def power(x, n) :
# returns x^n for x and an integer n
    if type(n) != type(0):
        raise TypeError("n must be an integer")

    if n == 1 :
        return x

    if n == 0 :
        return 1

    if n%2 == 0 :
        y = power(x, n/2) * power(x, n/2)
    else :
        y = x * power(x, n-1)

    return y


def nroot(x, n = 2) :
# returns n-th root of x > 0 and an integer n

    if x <=0 :
        raise ArithmeticError("The n-th root is only
           defined for positive numbers")

    if type(n) != type(0):
```

```
        raise TypeError("n must be an integer")

    eps = sys.float_info.epsilon

    root = x

    delta = float((power(root, n) - x))/(n*power(root, n-1)
       )

    while  abs(delta) > root*eps : # check if underflow
       happens
        # Newton-Raphson method
        root = root - delta
        delta = float((power(root, n) - x))/(n*power(root,
           n-1))

    return root
```

Exercise 4

The code below generates the curlicues for $\mu \approx 2.24248$, $L = 10000$ and $\mu \approx 4.12403$, $L = 4459$:

```
import pylab

# encoded continued fractions series for mu
p = reduce(lambda s, n: s + [s[-1]*2], range(101), [2])

# invert the series
p = p[::-1]

# calculate mu from the series
mu = reduce(lambda s, n : n + 1/s, p, 1.)

# f defines the series of the partial sums
f = lambda s, n: s + [s[-1]+exp(pi*1j*n*n/mu)]

# calculate the series of the  partial sums
L = 10000
s = reduce(f,range(L+1),[0])

realList = [k.real for k in s] # list of all real parts
```

```
imgList = [k.imag for k in s] # list of all imaginary parts

pylab.plot(realList, imgList)

# remove last element of the list = first term of the
    continued fraction
p2 = p[0:-1]

# calculate mu2
mu2 = reduce(lambda s, n : n + 1/s, p2, 1.)

L2 = int(L/mu)
s2 = reduce(f,range(L2+1),[0])

realList2 = [k.real for k in s2] # list of all real parts
imgList2 = [k.imag for k in s2] # list of all imaginary
    parts

# add new figure for the 2nd plot
pylab.figure()

pylab.plot(realList2, imgList2)
```

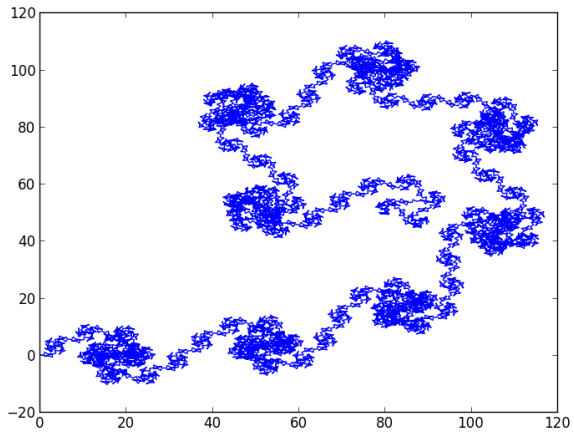

Figure 1: $\mu \approx 2.24248$, $L = 10000$

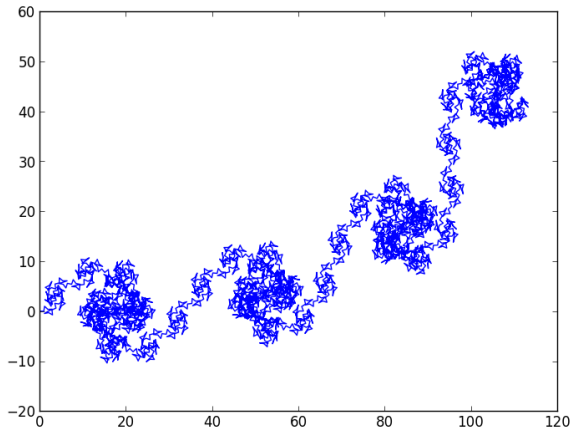

Figure 2: $\mu \approx 4.12403$, $L = 4459$

Exercise 5

`pi(n)` returns π up to n digits. To get enough accuracy the first n terms of the taylor series for the *arctan* are evaluated. The approximation of π is returned as a string.

```
class Fraction :
    def __init__(self, a, b):
    # sets the value for a fraction of the form a/b
        c = gcd(a, b)
        self.numerator = a/c
        self.denominator = b/c

def gcd(a, b) :
# returns the greatest common divisor of a and b
    if b == 0 :
        return a
    r = a%b
    return gcd(b, r)

def lcm(a, b) :
# returns the least common divisor of a and b
    return abs(a*b)/gcd(a,b)

def pi(accuracy = 1000) :
    series = []

    n = 1
```

```
    sign = 1
    # calculate all summands
    while n < 2*accuracy :
        series = series + [Fraction(sign*16,n*pow(5,n))]
        series = series + [Fraction(-sign*4,n*pow(239,n))]
        n = n + 2
        sign = sign * -1

    # get least common multiple of all denominators
    lcmDenominator = series[0].denominator
    for i in range(len(series)-1) :
        lcmDenominator = lcm(lcmDenominator, series[i+1].
           denominator)

    # calculate numerator
    numerator = 0
    for i in range(len(series)) :
        numerator = numerator + series[i].numerator*
           lcmDenominator/series[i].denominator

    result = ""

    result = result + str(numerator/lcmDenominator) + "."

    numerator = numerator%lcmDenominator

    decimal = 1
    while decimal < accuracy :
        digit = (numerator*10)/lcmDenominator
        numerator = (numerator*10)%lcmDenominator
        result = result + str(digit)
        decimal = decimal + 1

    return result
```

The output for `pi(1000)` is:

```
>>> pi(1000)
'3.14159265358979323846264338327950288419716939937510582097
7494459230781640628620899862803482534211706798214808651328
2306647093844609550582231725359408128481117450284102701938
5211055596446229489549303819644288109756659334461284756482
3378678316527120190914564856692346034861045432664821339360
```

```
726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019 8’
```