1.3 Real numbers in Python

Decimal module

The module decimal implements the arithmetic we usually do by hand. All the standard arithmetic can be used and, if necessary, the results will be rounded to a number of digits which can be changed by setting the context parameter prec. However, the available functions are limited and the operations may take longer than with the built in data types.
Note that the numbers have to be input as strings (as otherwise Python would automatically round to floating point numbers).

```
import decimal as dec

x = dec.Decimal('0.6')
y = dec.Decimal('0.5999999999')  # 10 significant digits
z = x - y
```

```
print("x = {0}, y = {1}, x-y = {2}\n".format(x,y,z))
print("representation of x: {!r}\n".format(x))
print("type of x: {}\n".format(type(x)))

print("timing")
%timeit(x-y)
```

```
x = 0.6, y = 0.5999999999, x-y = 1E-10

representation of x: Decimal('0.6')

type of x: <class 'decimal.Decimal'>

timing
101 ns ± 0.729 ns per loop (mean ± std. dev. of 7 runs, 10(
```

Python's default real numbers

Python uses 64 bit floating point numbers with 53 binary significant digits per default for real numbers. The arithmetic with these numbers is implemented in hardware and is very fast. However, even simple numbers like 0.6 cannot be exactly represented as floating point numbers of this type and thus one gets rounding errors. The effect is even worse when one substracts two numbers which are close and looses a significant amount of digits. This is called *cancellation*.

The usage of floating point numbers is illustrated below:

```
## Default floating point numbers

x = 0.6
y = 0.5999999999  # 10 significant decimal digits
z = x - y

print("x = {0}, y = {1}, x-y = {2}\n".format(x,y,z))
print("representation of x: {!r}\n".format(x))
print("type of x: {}\n".format(type(x)))
```

```
x = 0.6, y = 0.5999999999, x-y = 1.000000082740371e-10

representation of x: 0.6

type of x: <class 'float'>
```

```
print("printing the result with some more digits:\n")
print("   x = {0:3.100g}\n   y = {1:3.100g}\n   x-y = {2:3.

xstring = "{:3.100g}".format(x)
xmystring = "{:3.100g}".format(x-y)
print("significant decimal digits:   x: {0},     x-y: {1}\n

print("timing")
%timeit(x-y)
```

```
printing the result with some more digits:

   x = 0.59999999999999997779553950749686919152736663818359
   y = 0.59999999989999996952150240758783183991909027099609
   x-y = 1.000000082740370999090373516082763671875e-10

significant decimal digits:   x: 53,     x-y: 39

timing
40.7 ns ± 0.389 ns per loop (mean ± std. dev. of 7 runs, 10
```

Real numbers in numpy

The numerical package numpy has three major floating point number systems: float16, float32 and float64. They are useful for the development of resource critical applications using lower accuracy arithmetic whic is supported by some hardware including some graphics boards. In particular using these types is in principle a first way to save energy as the costliest operations are data transfers and using the types below on can thus cut costs by a factor of up to four.

The significant (binary) digits of the three numpy types are

- float16 has 11
- float32 has 24
- float64 has 53

Note that only 10, 23 and 52 of these bits are stored as it is assumed that the first significant bit is always 1. The accuracy and timing is illustrated below. On my laptop the timings were the same, actually, the 64 bit version was the fastest! The reason for this might be that the actual arithmetic is still done in the hardware floating point unit (using 64 or even higher accuracy) and the result is then rounded. As a direct conversion from float16 and float32 to decimal is not supported we first convert these two types to float64 (which can be done without error). Also note that the difference is less than the error for float16 and one gets then 100 percent error.
NB As we are computing with lower accuracy, we have changed the problem a bit compared to above!

```
## numpy floating point numbers -- print out using the dec

import decimal as dec
import numpy as np

# three numpy types

x16 = np.float16(0.6)
y16 = np.float16(0.59999)
z16 = x16 - y16

x32 = np.float32(0.6)
y32 = np.float32(0.59999)
z32 = x32 - y32

x64 = np.float64(0.6)
y64 = np.float64(0.59999)
z64 = x64 - y64
```

```python
# exact values
xex = dec.Decimal("0.6")
yex = dec.Decimal("0.59999")
zex = xex - yex

print("   x16 = {0}\n   x32 = {1}\n   x64 = {2}\n".format(d
```

```
   x16 = 0.60009765625
   x32 = 0.600000023841857910156250
   x64 = 0.59999999999999997779553950749686919152736663818
```

```
print("relative rounding errors:\n")
e16 = (dec.Decimal(float(x16)) - xex)/xex
e32 = (dec.Decimal(float(x32)) - xex)/xex
e64 = (dec.Decimal(x64) - xex)/xex
print("  for x:     float16 :   {:3.2g},   float32 :   {:3.

em16 = (dec.Decimal(float(z16)) -zex)/zex
em32 = (dec.Decimal(float(z32)) -zex)/zex
em64 = (dec.Decimal(float(z64)) -zex)/zex
print("  for x-y:   float16 :   {:3.2g},       float32 :
```

```
relative rounding errors:

  for x:     float16 :   0.00016,   float32 :   4.0e-8,   f
  for x-y:   float16 :    -1,       float32 :   0.0014,   f
```

```python
print("timing  - not much difference at this level")
print("* float16")
%timeit(x16-y16)
print("* float32")
%timeit(x32-y32)
print("* float64")
%timeit(x64-y64)
```

```
timing  - not much difference at this level
* float16
87.7 ns ± 0.776 ns per loop (mean ± std. dev. of 7 runs, 10
* float32
75.5 ns ± 0.608 ns per loop (mean ± std. dev. of 7 runs, 10
* float64
76.7 ns ± 0.836 ns per loop (mean ± std. dev. of 7 runs, 10
```

npmath multiple precision

This module does provide floating point with choosable (binary) accuracy. We will choose 113 binary digits which is the standard for quadruple (128 bit) arithmetic.
In order to compute the exact numbers we first convert the npmath number to a string (note that we need to use 113 bit decimal precision to get the exact result). Then we convert this string to a Decimal. Note that multiple precision operations are substantially slower than the floating point ones.
Looking at the printout it seems that only the first 30 or so decimal digits are accurate and the later ones are wrong. Thus without loosing much accuracy, one could set these later digits to zero.
However, we should remember, that the numerical approximation is binary number with 113 digits and the number is accurate to all the binary digits. If one now changes any of the later decimal digits and then rounds again to the nearest binary number on typically gets a larger error.

```
import mpmath as mpm

prec = 113
mpm.mp.prec = prec # set precision to quadruple

## Default floating point numbers -- print out using the de

x = mpm.mpf('0.6')
y = mpm.mpf('0.5999999999')  # 10 significant decimal digi
z = x - y

xex = dec.Decimal("0.6")             # exact values ..
yex = dec.Decimal("0.5999999999")
zex = xex - yex

xdec = dec.Decimal(mpm.nstr(x,prec))  # frist convert to s
ydec = dec.Decimal(mpm.nstr(y,prec))
zdec = dec.Decimal(mpm.nstr(z,prec))
```

```
print(" x = {0}\n y = {1}\n x-y = {2}\n".format(xdec, ydec,

print("errors:\n")
e = (xdec - xex)/xex
print("relative rounding error of x :   {:3.2g}".format(e))
em = (zdec -zex)/zex
print("relative rounding error of x-y:   {:3.2g}\n".format(
```

```
 x = 0.5999999999999999999999999999999999807407005561276414
 y = 0.5999999999000000000000000000000000634054841180440404
 x-y = 9.99999999999999999999999744001521443232369822629110

errors:

relative rounding error of x :   -3.2e-35
relative rounding error of x-y:   -2.6e-25
```

```
print("timing")
%timeit(x-y)
```

```
timing
1.66 µs ± 9.08 ns per loop (mean ± std. dev. of 7 runs, 10(
```

Other options

One can use other C data types and can also get access to the 80 or 128 bit accuracy of the hardware processor. Especially for running on GPUs one may also use the data types these processors use natively. For our purposes the above methods will be sufficient.