Real Numbers

- Many real numbers cannot be represented exactly.

```
In [1]: 1./49*49
0.9999999999999999
```

- The result should be 1.0
- Don't expect real numbers used in Python to be an exact representation of the number, it may be exact or it may be very close.
- The integer part will be exact but the fractional part may not be exact.

- Quite often a real number will be accurate to a certain number of decimal places.
- A temperature reading from a sensor may be accurate to 2 decimal places. (accuracy)
- Such a number should only be displayed with two decimal places, any more decimal places is misleading. (precision)
- In pure math, values may have many more digits of accuracy than can be stored in a real number in a program.
- The value of such a number stored in a real number is an approximation to the actual number and should be displayed with as many digits as possible.

Complex Numbers

- A complex number arises from trying to take the square root of a negative number.
- e.g. $\sqrt{-2} = ?$
- In math i represents $\sqrt{-1}$ • Therefore $\sqrt{-2} = \sqrt{2i^2} = \pm\sqrt{2}i$ • A complex number has a real part and an imaginary part typically written as **a + ib** or **a + bi**
- Suppose u = a + bi and v = c +di • Then u == v if and only if a == c and b == d • -u = -a - bi

- u^* = a - bi *(complex conjugate)*
- u + v = (a + c) + (b + d)i • u – v = (a – c) + (b – d)i
- uv = (ac – bd) + (bc + ad)i • u/v = (ac + bd)/(c^2 + d^2) + (bc – ad) /(c^2 + d^2) i • |u| = $\sqrt{a^2+b^2}$ *(magnitude)* which is abs(u)
- Solving a quadratic equation often leads to roots that are complex numbers.
- ax^2 + bx + c = 0 • The roots are: $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$
- If b^2-4ac < 0 then the roots are complex numbers.

- Python supports arithmetic computations with complex numbers but uses 1j (or j) instead of i for $\sqrt{-1}$

```
In [1]: a = 3 + 2j
In [2]: print(a)
(3+2j)
In [3]: print(type(a))
<class 'complex'>
In [4]: b = 4
In [5]: c = 3 + bj
NameError: name 'bj' is not defined
In [6]: c = 3 + b * j
NameError: name 'j' is not defined
In [7]: c = 3 + b * 1j
In [8]: print(c)
(3+4j)
```

```
In [9]: c = complex(3, b)
In [10]: print(c)
(3+4j)
In [1]: a = -2 + 3j
In [2]: b = 4.5 - 1.2j
In [3]: print(a * b)
(-5.4+15.9j)
```

- Real part is -2×4.5 + 3j×-1.2j = -9.0 + -3.6j^2 = -9.0 + -3.6×-1 = -9.0 + 3.6 = -5.4
- Imaginary part is -2×-1.2 + 3×4.5 = 2.4 + 13.5 = 15.9

```
In [4]: print(a/b)
(-0.5809128630705394+0.5117565698478561j)
In [5]: print(a.real)
```

```
-2.0
In [6]: print(a.imag)
3.0
In [1]: a = -2 + 3j
In [2]: from math import sin
In [3]: s = sin(a)
TypeError: can't convert complex to float
In [4]: from cmath import sin, asin
In [5]: c1 = sin(8j)
In [6]: print(c1)
1490.4788257895502j
In [7]: c2 = asin(c1)
In [8]: print(c2)
8j
```

- cmath functions always return complex numbers

- Would like functions that return a real number when the result is a real number and return a complex number when the result is a complex number.

```
In [1]: from numpy.lib.scimath import sqrt
In [2]: print(sqrt(4))
2.0
In [3]: print(sqrt(-4))
2j
In [4]: from numpy import sin
In [5]: print(sin(1.57))
0.9999996829318346
In [6]: print(sin(1.57j))
```

```
2.2993015057090789j
```

NumPy is the fundamental package for scientific computing with **Python**.

- Find the roots of the equation f(x) = $ax^2 + bx + c$, where a = 3, b = 5, c = 10.
- For this example $b^2 - 4ac$ is negative and the roots will be complex numbers.

```
In [1]: from numpy.lib.scimath import sqrt
In [2]: a = 3; b = 5; c = 10
In [3]: r1 = (-b + sqrt(b**2 - 4*a*c)) / (2*a)
In [4]: print(r1)
(-0.83333333333333337+1.6244657241348273j)
In [5]: r2 = (-b - sqrt(b**2 - 4*a*c)) / (2*a)
```

```
In [6]: print(r2)
(-0.8333333333333337-1.6244657241348273j)
```

- Find the roots of the equation where $f(x) = ax^2 + bx + c$, where $a = 1$, $b = 6$, $c = 5$.
- For this example $b^2 - 4ac$ is positive and the roots will be real numbers.

```
In [1]: from numpy.lib.scimath import sqrt
In [2]: a = 1; b = 6; c = 5
In [3]: r1 = (-b + sqrt(b**2 - 4*a*c)) / (2*a)
In [4]: print(r1)
-1.0
In [5]: r2 = (-b - sqrt(b**2 - 4*a*c)) / (2*a)
In [6]: print(r2)
-5.0
```

- Is there a better way to find the two roots?

- If you multiply the expressions for the two roots you get: r1 * r2 = c / a
- Let r1 be the root with the larger magnitude, i.e. abs(r1) > abs(r2), then r2 = (c / a) / r1
- NOTE: the absolute value of the complex number is the same as the magnitude of that number.
- If the smaller magnitude of the two roots is very small the error in computing it using floating point numbers may make this root very inaccurate.
- Dividing c / a by the larger (in magnitude) of the two roots involves fewer operations and is more accurate.

Trajectory of a Ball

- Compute the position of a ball thrown forward at an angle, using the formula:

- **$y(x) = x \tan(\theta) - 1/(2v_0^2)gx^2/\cos^2(\theta) + y_0$** where x is the horizontal position, g is the force of gravity, v_0 is the initial velocity, θ is the angle relative to the x axis and y_0 is the initial height of the ball when x = 0.
- As we are implementing a solution to a formula we should use the names used in the formula as variable names.
- The solution is in the script named **trajectory.py**
- In class demo

```
from time import ctime from math import pi, cos, tan
print('--------------------------------------------------------------\n'
) G = 9.81       # m/s**2 v0 = float(input('Enter the inital velocity
in m/s: ')) y0 = float(input('Enter the initial height in m: ')) theta
= float(input('Enter the initial angle in degrees: ')) x =
float(input('Enter the distance x in m: '))
# display the values of the variables
print(""" gravity = %.2f m/s^2 v0
= %.1f m/s y0      = %.1f m theta
= %g degrees x       = %.1f m
```

```
""" % (G, v0, y0, theta, x)) theta = theta * pi /
180  # convert theta to radians
# calculate the vertical position of the ball and display the
result y = x * tan(theta) - 1 / (2 * v0**2) * G * x**2 /
cos(theta)**2 + y0 print('The position of the ball is (%.2f,%.4f)'
% (x, y)) print("""
Programmed by Stew Dent.
Date: %s.
End of processing.""" % ctime())
```

- The output from the program is:

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


Enter the inital velocity in m/s: 5
Enter the initial height in m: 1
Enter the initial angle in degrees: 60
Enter the distance x in m: .5

gravity = 9.81 m/s^2
v0      = 5.0 m/s
y0      = 1.0 m
theta   = 60
```

```
degrees x       =
0.5 m

The position of the ball is (0.50,1.6698)

Programmed by Stew Dent.
Date: Wed Apr 25 07:45:41 2012.
End of processing.
```

- Suppose we wish to display a conversion table showing temperatures in degrees Celsius and Fahrenheit as shown below.

```
CelsiusFahrenheit
    -40       -40
    -35       -31
    -30       -22
    -25       -13
    -20        -4
    -15         5
    -10        14
     -5        23
      0        32
      5        41
```

```
10          50
15          59
20          68
25          77
30          86
35          95
40         104
```

- You don't want to have to type in separate statements to compute and display the values in each line of the output.
- We want to have the same statements in the program executed over and over again until every line in the table has been displayed.
- To do this we need to have a **loop** in the program.
- A **while** loop is of the form:

```
while condition:
    one or more indented statements
```

- Notice the colon after the condition, this is required.
- The solution is in the script named **toFahrenheitLoop.py** (In class demo)

```
from time import ctime
print('---------------------------------------------------\n')
celsius = float(input(
          'Enter the first celsius temperature: '))
stop = float(input(
       'Enter the last celsius temperature: '))
step = float(input(
       'Enter the difference between temperatures: '))

print("Celsius\tFahrenheit") # display a heading

while celsius <= stop:
    fahrenheit = (9/5)*celsius + 32
    print("%7.2f\t%10.2f" % (celsius, fahrenheit))
    celsius += step

print("""
```

```
Programmed by Stew Dent.
Date: %s.
End of processing.""" % ctime())
```

- **`celsius <= stop`** is the **condition** or **relational expression**, where **<=** is a **relational operator** that means less than or equal to.
- The condition **`celsius <= stop`** is either **True** or **False**
- Consider the following statements:

```
celsius = -40
DELTA_T = 5 while
celsius <= 40:
    fahrenheit = (9/5)*celsius + 32 print("%
    7g\t% 10g" % (celsius, fahrenheit)) celsius
    += DELTA_T
print('first statement after the loop')
```

- The indented statements form the **body** of the loop.

- As long as the the value of **celsius** is less than or equal to 40, the condition `celsius <= 40` is **True**, and the statements in the body of the loop are executed.
- When **celsius** becomes greater than 40 the condition is no longer **True** (it becomes **False**) and the first statement following the loop is executed, which is the print statement that is NOT indented.
- Consider the following statements:

```
celsius = 50
DELTA_T = 5 while
celsius <= 40:
    fahrenheit = (9/5)*celsius + 32
    print("% 7g\t% 10g" % (celsius,
fahrenheit)) celsius
    += DELTA_T
print('first statement after the loop')
```

- The value of **celsius** is 50 before the **while** statement.
- As 50 is greater than 40 (not less than or equal to 40) the condition `celsius <= 40` is **False** and the statements in the body of the loop are never executed.
- The statements in the body of a while loop are executed only if the condition is **True**.

Relational Operators

Operator	Meaning	Example
==	equals	a == b
!=	not equals	a != b
<	less than	a < b
<=	less than or equal to	a <= b
>	greater than	a >b

>=	greater than or equal to	a >= b

- Each relational operator requires two operands to form a relational expression.
- The value of a relational expression such as a==b is a **boolean** value, either **True** or **False**

```
In [1]: a = 10
In [2]: b = 20
In [3]: print(a == b)
False
In [4]: print(a != b)
True
In [5]: print(a < b)
True
In [6]: print(a <= b)
```

```
True
In [7]: print(a > b)
False
In [8]: print(a >= b)
False
```

Boolean Operators

Operator	Example
and	e1 and e2
or	e1 or e2
not	not e1

- e1 and e2 must evaluate to either True or False
- e1 and e2 are often relational expressions

- e1 or e2 may be a variable holding a **boolean** value, that is either **True** or **False**

Truth Table For **and**

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

- The value of **A and B** is True if and only if both A = True and B = True

Truth Table For **or**

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

- The value of **A or B** is False if and only if both A = False and B = False.

Truth Table For **not**

A	not A
False	True

True	False

```
In [1]: a = 10
In [2]: b = 20
In [3]: e1 = a == b
In [4]: print(e1)
False
In [5]: e2 = a != b
In [6]: print(e2)
True
In [7]: print(e1 and e2)
False
In [8]: print(e1 or e2)
True
In [1]: a = 10
In [2]: b = 20
In [3]: e1 = a == b
In [4]: e2 = a != b
In [5]: print(e1)
False
In [6]: print(e2)
```

```
True
In [7]: print(not e1)
True
In [8]: print(not e2)
False
In [9]: print(a <= b and a == 10)
True
In [10]: print(a <= b and b <= 10)
False
In [11]: print(0 <= a <= 25) # same as 0<=a and a<=25
True
```

Summing a Series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

- $n! = 1 \times 2 \times 3 \times \ldots \times n\text{-}1 \times n$
- $X = X^1/1!$

- $term_0 = X^1/1!$
- $term_1 = -1 \times term_0 \times X^2 / (3 \times 2) = -X^3 / 3!$
- $term_2 = -1 \text{ x } term_1 \times X^2 / (5 \times 4) = +X^5 / 5!$
- etc.
- Notice that the divisors increase by 2 for each successive term.
- X must be expressed in radians not degrees.
- In class demo (script sinSumLoop.py)

```
from math import pi, sin
print('---------------------------------------------\n')
TOLERANCE = 1.0e-17 degrees = float(input("Enter an
angle in degrees: ")) x = degrees * pi / 180 factor = 2
sine = 0.0 term = x xSquared = x * x while abs(term) >
TOLERANCE:
    sine += term term = -term * xSquared / (factor *
(factor + 1)) factor += 2 # end while print("""
```

```
Angle in radians = %g, angle in degrees = %.2f
   Python's value of sin(%.2f) = %.15f
Approximate value of sin(%.2f) = %.15f
Number of terms = %d""" % (x,  degrees, degrees, sin(x),
                             degrees, sine, count))
```

Representing Real Numbers Using Integers

- Some computers do not have real numbers.
- Real numbers can be represented using large integers.
- For example suppose 10000000000 is used to represent 1.0 to 10 decimal places.

Number	Representation
1.0	10000000000
0.5	5000000000
0.25	2500000000
0.125	1250000000
0.0625	625000000

12.0987654321	120987654321

- Fractions are represented by integers smaller than the integer that represents 1.

Displaying Integers That Represent Real Numbers

- Suppose we have the number `120987654321` that represents a real number to 10 decimals places.
- To get the integer part of the number divide by 10^{10}.

```
120987654321 // 10000000000 = 12
```

- To get the fractional part take the remainder when dividing by 10^{10}.

```
120987654321 % 10000000000 = 0987654321
```

- To display the number as a real number print the integer part a decimal point and then the fractional part.
- If we do this in python we get:

```
12.987654321
```

- Notice that the leading zero of the fractional part is missing.
- How do we get the leading zero of the fractional part to print?
- We have to compute the number of digits in the fractional part and subtract it from the number of decimal places.
- In our example `987654321` contains 9 digits. To determine this we need to use the *log10* function.

```
In [1]: from math import log10
In [2]: print(log10(987654321))
8.994604968118722
In [3]: print(int(log10(987654321)) + 1)
9    # number of digits in 987654321
In [4]: print(10 - (int(log10(987654321)) + 1))
1    # number of missing leading zeros
```

```
In [5]: print('0' * (10 - (int(log10(987654321)) + 1)))
```

```
0  # the missing zero
```

- Using variables this is done as follows:

```
In [1]: places = 10
In [2]: num = 120987654321
In [3]: intPart = 120987654321 // 10**places
In [4]: fractPart = 120987654321 %
10**places
In [5]: result = str(intPart) + '.' + '0' *\
   (places - (int(log10(fractPart)) + 1)) +\
   str(fractPart)
In [6]: print(result)
```

12.0987654321

Compute the value of $\boxed{e}$

- Compute the value of e to any number of decimal places.

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

- $term_0$ = 1
- $term_1$ = $term_0$ / 1 = 1 /1 = 1 / 1!
- $term_2$ = $term_1$ / 2 = 1 / (1 * 2) = 1 / 2!
- $term_3$ = $term_2$ / 3 = 1 / (1 * 2 * 3) = 1 / 3!
- The divisor is incremented by 1 for each new term.

- In class demo of computeE.py

```
from time import ctime from math import log10, exp

print('\n--------------------------------------\n')

places = int(input(
         'Enter the number of decimal places: '))
one = 10**places
extra = 10**4 n =
1 term = one *
extra eee = 0
count = 0

while term > 0: eee
    += term count +=
    1 term = term //
    n n += 1
eee = eee // extra
intPart = eee // one
fracPart = eee % one
```

```python
eee = str(intPart) + '.' + '0' * \
      (places - (int(log10(fracPart)) + 1)) \
     + str(fracPart)
print("""
Python's value of e is:\n%.15f\n
e to %d decimal places is:\n%s\n
The number of terms in the series is %d""" \
    % (exp(1), places, eee, count))
```

Output from the program:

```
--------------------------------------------Enter

the number of decimal places: 50

Python's value of e is:
2.718281828459045

e to 50 decimal places is:
```

```
2.71828182845904523536028747135266249775724709369995

The number of terms in the series is 44

Programmed by Stew Dent Date:
Wed May  2 21:38:12 2018 End
of processing.
```

Demo remaining programs.