25. Odds and Ends

Topics
- Some numpy details
- More plotting with pylab
- Tuples
- Operator Overloading

More on Numpy Arrays

1-dim Arrays: Subtleties

```python
>>> x = zeros(3)
>>> x
array([ 0.,  0.,  0.])
>>> x[1]
0.0

X is a 1d array
It has 3 entries
The entries here are floats
x[1] refers to the 1st entry

>>> x = zeros((3,1))
>>> x
array([[ 0.],
       [ 0.],
       [ 0.]])
>>> x[1] = 0.0

X is a 2d array
It has 3 rows and 1 column
The rows of a 2darray are 1d arrays.
x[1] refers to row 1 of x
```

1-dim Arrays: Subtleties

```python
>>> x = zeros(3)
>>> x
array([ 0.,  0.,  0.])
>>> x[1] = 0.0

X is a 1d array
It has 3 entries
The entries here are floats
x[1] refers to the 1st entry

>>> x = zeros((1,3))
>>> x
array([[ 0.,  0.,  0.]])
>>> x[1] = 5.3
>>> x
array([[ 0.,  0.,  5.3]])
>>> x[1] = '12'
>>> x
array([[ 0.,  0., 12.]])

IndexError: index 1 is out of bounds for axis 0 with size 1
```

int Arrays

```python
>>> A = array([[1, 2],[3, 4]], dtype=int)
>>> A
array([[1, 2],
       [3, 4]])
>>> A[1,1] = 5.3
>>> A
array([[1, 2],
       [3, 5]])
>>> A[1,1] = '12'
>>> A
array([[1, 2],
       [3, 12]])
>>> A[1,0] = 'x'
ValueError: invalid literal for long()
```

Plotting a Continuous Function With Pylab

```
Assume:
from numpy import *
from pylab import *
```
The numpy `linspace` function

```
x = linspace(1,3,5)
x : 1.0 1.5 2.0 2.5 3.0
```

`linspace(a,b,n)` is a length-\(n\) list of values that are equally spaced from \(x = a\) to \(x = b\).

### Generating Tables and Plots

```
x = linspace(0,2*pi,9)
y = sin(x)
plot(x,y)
show()
```

```
x,y 1-dim arrays of numbers
That have the same length
plot(x,y) "connects the dots":
(x[0],y[0]),..., (x[n-1],y[n-1])
```
for k in range(6,20):
    # Draw horizontal line from (0,k) to (365,k)
    plot(array([0,365]),array([k,k]),
         color='red',linestyle=':')

for k in range(6,20):
    # Draw horizontal line from (0,k) to (365,k)
    plot(array([0,365]),array([k,k]),
         color='red',linestyle=':')

A Note on subplot

subplot(2,1,1)
<code>
subplot(2,1,2)
<code>
Show()
Returning Multiple Values

A function can only return "one thing". But that is not restrictive if we use tuples:

```python
def dailyBill(self):
    return (E,S,C)
```

Functions with Variable Calling Sequences

Tuples are also handy when you have a function that can have an arbitrary number of input arguments.

```python
A = zeros(10)
B = zeros((10,20))
```

Operator Overloading

Tuples are also handy when you have a function that can have an arbitrary number of input arguments.

Consider Addition

```python
s = 'dogs' + 'and' + 'cats'
x = 100 + 200 + 300
y = 1.2 + 3.4 + 5.6
```

What "+" signals depends on the operands. Python figures it out. The "+" operation is overloaded.

Let's Define a Type and a "+" Operation for that Type

```python
class Fraction:
    Attributes:
        num: the numerator [int]
        den: the denominator [positive int]

    Invariant: num/den is reduced to lowest terms
```

A class that support operations with fractions

Review!

```python
2/3 + 3/4 = (2*4+3*3)/(3*4) = 20/12 = 5/3
2/3 * 3/4 = (2*3)/(3*4) = 6/8 = 3/4
```
A Note About Greatest Common Divisors

\[
\begin{array}{ccc}
p & q & \text{gcd}(p, q) \\
16 & 24 & 8 \\
2 & 5 & 1 \\
10 & 95 & 5 \\
\end{array}
\]

Reducing a fraction to lowest terms involves finding the gcd of the numerator and denominator and dividing.

Computing the Greatest Common Divisor

```python
def gcd(a, b):
    a = abs(a)
    b = abs(b)
    r = a % b
    while r > 0:
        a = b
        b = r
        r = a % b
    return b
```

Euclid's Algorithm 300BC

The Constructor

```python
def __init__(self, p, q=1):
    d = gcd(p, q)
    self.num = p / d
    self.den = q / d
    if self.den < 0:
        self.den = -self.den
        self.num = -self.num
```

>>> A = Fraction(10, 4)
>>> print A
5/2

The gcd of 4 and 10 is 2

>>> A = Fraction(10)
>>> print A
10/1

The default denominator is 1.

Define “+” For Fractions

```python
def __add__(self, f):
    N = self.num * f.den + self.den * f.num
    D = self.den * f.den
    return Fraction(N, D)
```

>>> A = Fraction(2, 3)
>>> B = Fraction(1, 4)
>>> C = A + B
>>> print C
11/12

By defining __add__ this way we can say A + B instead of A.__add__(B)

If the fraction is negative, make the numerator negative.
Likewise for Multiplication

```python
def __mul__(self, f):
    N = self.num * f.num
    D = self.den * f.den
    return Fraction(N, D)
```

```
>>> A = Fraction(2, 3)
>>> B = Fraction(1, 4)
>>> C = A * B
>>> print C
1/6
```

By defining `__mul__` this way we can say `A * B` instead of `A.__mul__(B)`

Would Like Some Flexibility

Would like to be able to add an int to a fraction:

\[
\frac{2}{3} + 5 = \frac{17}{3}
\]

Python needs to know the type of the operands

A More Flexible `__add__`

```python
def __add__(self, f):
    if isinstance(f, Fraction):
        N = self.num * f.den + self.den * f.num
        D = self.den * f.den
    else:
        N = self.num + self.den * f
        D = self.den
    return Fraction(N, D)
```

If `f` is a Fraction, use \((a/b + c/d) = (ad + bc)/(bd)\)

If `f` is an integer, use \((a/b + f) = (a + bf)/b\)

A More Flexible `__mul__`

```python
def __mul__(self, f):
    if isinstance(f, Fraction):
        N = self.num * f.num
        D = self.den * f.den
    else:
        N = self.num * f
        D = self.den
    return Fraction(N, D)
```

If `f` is a Fraction, use \((a/b)(c/d) = (ac)/(bd)\)

If `f` is an int, use \((a/b)(f) = (af)/b\)
An Example

Let's compute $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{30}$

```python
n = 30
s = Fraction(0)
for k in range(1,n+1):
    s = s + Fraction(1,k)
print s
```

9304682830147/2329089562800