CS 3430: Python & Perl
Assignment 3: Anonymous 2nd Degree Poly Dances

Vladimir Kulyukin
Department of Computer Science
Utah State University

0. Learning Objectives

1. Anonymous Functions & Code Blocks
2. Py Lambdas
3. Py Lists & Tuples
4. PL Lists
5. Mapping Functions/Subroutines over Lists

1. Anonymous Functions & Subroutines

As we discussed in Lecture 6, both Py & PL allow us to define anonymous functions. Py uses lambda forms. PL does not have the exact equivalent of lambdas. However, we can still achieve the same functionality through other means such as anonymous subroutines and code blocks. Let us briefly review these concepts before discussing the actual assignment. Suppose we would like to define a Py function that returns another function that takes a number and adds a specific constant to that number. We will call these functions adders. Here is how we can define an adder factory in Py.

```
def make_adder(n): return lambda x: x + n
```

Here is how we can test it in the Py shell.

```
>>> adder1 = make_adder(1)
>>> adder1(1)
2
>>> adder1(2)
3
>>> adder1(3)
4
>>> make_adder(2)(4)
6
>>> make_adder(3)(5)
8
```

Let us write a test function to test two adders over a range of numbers.

```
import sys

def make_adder(n): return lambda x: x + n
```

```
def test_adders():
   add3 = make_adder(3)
   add4 = make_adder(4)
   print 'Testing adders...'
   for x in xrange(1, 6):
      sys.stdout.write('add3(' + str(x) + ')=' + str(add3(x))+"\n")
      sys.stdout.write('add4(' + str(x) + ')=' + str(add4(x))+"\n")
```

Here is the output in the Py shell.

```
Testing adders...
add3(1)=4
add4(1)=5
add3(2)=5
add4(2)=6
add3(3)=6
add4(3)=7
add3(4)=7
add4(4)=8
add3(5)=8
add4(5)=9
```

We can replicate the same behavior in PL with anonymous subs. We start with the sub **make_adder** and then write a few tests on it.

```
sub make_adder { my $n = $_[0]; return sub { return $_[0] + $n;  }  }
```

Note that the PL **make_adder** does exactly what the Py **make_adder** does except that it returns a scaler reference of anonymous function.

Now we can write a test sub. We create two adders and then map them over the range from 1 upto 5.

```
use strict;
use warnings;
use 5.10.0; ## add this use pragma to ensure that anonymous subs work.

sub test_adders {
   my $add3 = make_adder(3); my $add4 = make_adder(4);
   say 'Testing adders…';
   map { say '$add3->(', $_ , ') = ', $add3->($_); say '$add4->(', $_,  ') = ', $add4->($_); } (1..5);
}
```

The output is as follows:

```
Testing adders…
```

$add3->(1) = 4
$add4->(1) = 5
$add3->(2) = 5
$add4->(2) = 6
$add3->(3) = 6
$add4->(3) = 7
$add3->(4) = 7
$add4->(4) = 8
$add3->(5) = 8
$add4->(5) = 9

2. Anonymous Polynomials

Let us get to the actual assignment. You will define anonymous second degree polynomials, map them over lists, and display the results. I recommend that you do it step by step but you are free to use your own methods: every programmer is different and has his or her own bag of problem-solving tricks.

Step 1: Start by the defining a function/subroutine that takes three coefficients **k2**, **k1**, and **k0** and returns an anonymous function/subroutine that takes one argument **x** and returns a string representation of the second degree polynomial with the given coefficients and its value at **x**, i.e., **k2*x^2 + k1*x + k0**.

```
def make_2nd_degree_poly(k2, k1, k0): // your code
sub make_2nd_degree_poly { my ($k2, $k1, $k0) = @_; // your code }
```

Testing the Py function in the Py shell should give you this:

```
>>> p1 = make_2nd_deg_poly(1, 2, 3)
>>> p1(1)
('1x^2 + 2x + 3 at x = 1 is ', 6)
>>> p2 = make_2nd_deg_poly(4, 5, 6)
>>> p2(3)
('4x^2 + 5x + 6 at x = 3 is ', 57)
```

Here is the PL test:

```
my @triplet0 = (1, 2, 3); my @triplet1 = (4, 5, 6);
my $p1 = make_2nd_deg_poly(@triplet0);
my $p2 = make_2nd_deg_poly(@triplet1);
print "@{$p1->(1)}\n";
print "@{$p2->(2)}\n";
```

The output of the PL test is as follows:

```
1x^2 + 2x + 3 at x = 1 is  6
```

4x^2 + 5x + 6 at x = 2 is 32

Step 2: Write a function/sub **gen_2nd_deg_polys** that takes a list of 3-tuples and returns a list of anonymous 2nd degree polynomials.

def gen_2nd_deg_polys(coeffs): // your code here

Below is the defined PL sub that you can plug into your code to make your PL job a little easier.

sub gen_2nd_deg_polys { return map(make_2nd_poly(@{$_}, @_);

The Py shell test of your implementation of **gen_2nd_deg_polys** should give you this.

```
>>> polys = gen_2nd_deg_polys([(1, 2, 3), (4, 5, 6)])
>>> polys[0](1)
('1x^2 + 2x + 3 at x = 1 is ', 6)
>>> polys[1](2)
('4x^2 + 5x + 6 at x = 2 is ', 32)
```

Step 3: Define a Py function and a PL **sub apply_2nd_deg_polys** that takes a list of anonymous 2nd degree polynomials and a list of numbers and applies each polynomial to each number in the list.

```
def apply_2nd_deg_polys(polys, numbers): // your code
sub apply_2nd_deg_polys2 {
   my @polys = @{$_[0]}; my @numbers = @{$_[1]};
   // your code
}
```

Below is the Py shell output of my testing **apply_2nd_deg_polys**:

```
>>> polys = gen_2nd_deg_polys(((1, 2, 3), (4, 5, 6)))
>>> poly_maps = apply_2nd_deg_polys(polys, xrange(1, 6))
>>> poly_maps
[('1x^2 + 2x + 3 at x = 1 is ', 6), ('4x^2 + 5x + 6 at x = 1 is ', 15), ('1x^2 + 2x + 3 at x = 2 is ', 11), ('4x^2 + 5x + 6 at
x = 2 is ', 32), ('1x^2 + 2x + 3 at x = 3 is ', 18), ('4x^2 + 5x + 6 at x = 3 is ', 57), ('1x^2 + 2x + 3 at x = 4 is ', 27),
('4x^2 + 5x + 6 at x = 4 is ', 90), ('1x^2 + 2x + 3 at x = 5 is ', 38), ('4x^2 + 5x + 6 at x = 5 is ', 131)]
```

You can write a PL test as follows:

```
my @polys = gen_2nd_deg_polys(@coeffs);
my @numbers = (1..5);
my @poly_maps = apply_2nd_deg_polys2(\@polys, \@numbers);
```

Step 4: Write a function/subroutine **display_poly_maps** that takes the output of **apply_2nd_deg_polys** and displays them nicely in the Py shell or the PL command line window. Here is the output, which is identical in Py and PL.

```
1x^2 + 2x + 3 at x = 1 is  6
4x^2 + 5x + 6 at x = 1 is  15
1x^2 + 2x + 3 at x = 2 is  11
4x^2 + 5x + 6 at x = 2 is  32
1x^2 + 2x + 3 at x = 3 is  18
4x^2 + 5x + 6 at x = 3 is  57
1x^2 + 2x + 3 at x = 4 is  27
4x^2 + 5x + 6 at x = 4 is  90
1x^2 + 2x + 3 at x = 5 is  38
4x^2 + 5x + 6 at x = 5 is  131
```

Step 5: Define a top-level function/subroutine **poly_dance** that combines all of the above functions/subroutines by taking a list of coefficient 3-tuples and a range of numbers, creating anonymous polynomials, applying each polynomial to each number in the range, and displaying the results. Below are the ready Py and PL definitions.

```
def poly_dance(coeffs, numbers):
    polys = gen_2nd_deg_polys(coeffs)
    poly_maps = apply_2nd_deg_polys(polys, numbers)
    display_poly_maps(poly_maps)

poly_dance(((1, 2, 3), (4, 5, 6)), xrange(1, 6))
```

```
sub poly_dance {
   my @coeffs = @{$_[0]}; my @numbers = @{$_[1]};
   my @polys = gen_2nd_deg_polys(@coeffs);
   my @poly_maps = apply_2nd_deg_polys2(\@polys, \@numbers);
   display_poly_maps(@poly_maps);
}

my @numbers = (1..5);
poly_dance(\@coeffs, \@numbers);
```

4. What to Submit

Save your Py solution in **poly_dance.py** and your PL solution in **poly_dance.pl** and submit both files in Canvas.

Happy Hacking!