Chapter 34: Defining functions with list arguments

Section 34.1: Function and Call

Lists as arguments are just another variable:

```
def func(myList):
    for item in myList:
        print(item)
```

and can be passed in the function call itself:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Or as a variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Chapter 35: Functional Programming in Python

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.below are functional techniques common to many languages: such as lambda, map, reduce.

Section 35.1: Lambda Function

An anonymous, inlined function defined with lambda. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

```
s=lambda x:x*x
s(2)    =>4
```

Section 35.2: Map Function

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

Section 35.3: Reduce Function

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

Section 35.4: Filter Function

Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Chapter 36: Partial functions

Param	details
x	the number to be raised
y	the exponent
raise	the function to be specialized

As you probably know if you came from OOP school, specializing an abstract class and use it is a practice you should keep in mind when writing your code.

What if you could define an abstract function and specialize it in order to create different versions of it? Thinks it as a sort of *function Inheritance* where you bind specific params to make them reliable for a specific scenario.

Section 36.1: Raise the power

Let's suppose we want raise *x* to a number *y*.

You'd write this as:

```
def raise_power(x, y):
    return x**y
```

What if your *y* value can assume a finite set of values?

Let's suppose *y* can be one of *[3,4,5]* and let's say you don't want offer end user the possibility to use such function since it is very computationally intensive. In fact you would check if provided *y* assumes a valid value and rewrite your function as:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Messy? Let's use the abstract form and specialize it to all three cases: let's implement them **partially**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

What happens here? We fixed the *y* params and we defined three different functions.

No need to use the abstract function defined above (you could make it *private*) but you could use **partial applied** functions to deal with raising a number to a fixed value.