Lists and Loops

What we will cover...

1. DRY
2. Lists and Loops
3. Operations on lists: map, filter, reduce

DRY

DRY stands for **D**o not **R**epeat **Y**ourself.

DRY is a basic strategy for removing repetition in code. Almost all code should be dry!

Why do we want to remove repetition from our code?

Loops are a necessary tool when we want the computer to repeat something, but we don't want to write any redundant code.

Lists

In python, the basic loop happens over an **iterable**.

An iterable is a data structure, which consists of a set of values which can be looped (or *iterated*) over.

The most simple iterable is the **list**.

```
my_list = [0, 1, 2, 3, 4]
```

Looping

The basic loop in python is the **for loop** which consists of:

1. The keyword `for`.

```
for
```

Looping

The basic loop in python is the **for loop** which consists of:

1. The keyword `for`
2. A variable name for each item in the iterable (i.e. `number`)

```
for number
```

Looping

The basic loop in python is the **for loop** which consists of:

1. The keyword `for`
2. A variable name for each item in the iterable (i.e. `number`)
3. The keyword `in`.

```
for number in
```

Looping

The basic loop in python is the **for loop** which consists of:

1. The keyword `for`
2. A variable name for each item in the iterable (i.e. `number`)
3. The keyword `in`.
4. The iterable (`my_list`) and `:`.

```
for number in my_list:
```

Looping

The basic loop in python is the **for loop** which consists of:

1. The keyword `for`
2. A variable name for each item in the iterable (i.e. `number`)
3. The keyword `in`.
4. The iterable (`my_list`) and `:`.
5. The code **block** to be repeated, indented 4 spaces.

```
for number in my_list:
    print(number)
```

Data

Data often comes in a list (if not, it would be called datum).

There are 3 basic operations we perform on data in lists:

map transform each element
filter remove some elements
reduce aggregate the list into a single element

```
names = ['Foo', 'Bar', 'Baz']
```

Map

Map consists of transforming each element of the list into a new element. We can use functions to transform elements!

The map operation returns a new list of the same length as the old list.

Given a list and a function, we can **map** the old list into a new list with a **list comprehension**.

```
names = ['Foo', 'Bar', 'Baz']

def get_length(name):
    return len(name)

[get_length(n) for n in names]
# [3, 3, 3]
```

Writing list comprehensions

Steps to write a **list comprehension**:

```
[]
```

1. Start with square brackets `[]`.

Writing list comprehensions

Steps to write a **list comprehension**:

1. Start with square brackets `[]`.
2. Call the function with a variable that you will define later (`n`).

```
[get_length(n)]
```

Writing list comprehensions

Steps to write a **list comprehension**:

1. Start with square brackets `[]`.
2. Call the function with a variable that you will define later (`n`).
3. `for`
4. new variable(`n`)
5. `in`
6. iterable(`names`)

```
[get_length(n) for n in names]
```

Filter

Filter is used to remove certain elements from a list.

We can filter a list by adding an **if statement** to a list comprehension.

Like all if statements, the `if` keyword is followed by a boolean.

The element is included in the new list only if the boolean is true.

```
names = ['Foo', 'Bar', 'Baz']

def we_like(name):
    return name != 'Bar'

[n for n in names if we_like(n)]
```

Map + Filter

List comprehensions allow us to easily map and filter at the same time.

What will this code output?

```
names = ['Foo', 'Bar', 'Baz']

def get_length(name):
    return len(name)

def we_like(name):
    return name != 'Bar'

[get_length(n) for n in names if we_like(n)]
```

Without explicit functions

We can perform operations directly in the list comprehension instead of defining functions separately.

Note: this is often a stylistic choice and one should consider readability, modularity, and testability.

```
names = ['Foo', 'Bar', 'Baz']

[len(n) for n in names if n != 'Bar']

nums = [2,5,10,25,35]

[n**2 for n in nums]
```

Reduce

The reduce operation **aggregates** a list into a single element.

The single element that comes out of a reduce operation is called the **acumulator**.

Summing the numbers in a list is a reduction!

```
numbers = [0,1,2,3,4]

def get_sum(nums):
    total = 0
    for n in nums:
        total += n
    return total

get_sum(numbers)
```

Reduce

To write a **reduce** function:

1. Initialize the **acumulator** (`lowest`)

```
def get_lowest(nums):
    lowest = nums[0]
```

Reduce

To write a **reduce** function:

1. Initialize the **acumulator** (`lowest`)
2. Define a for loop (`for n in nums:`)

```
def get_lowest(nums):
    lowest = nums[0]
    for n in nums:
```

Reduce

To write a **reduce** function:

1. Initialize the **acumulator** (`lowest`)
2. Define a for loop (`for n in nums:`)
3. Modify the accumulator (`lowest`) in the body of the for loop

```
def get_lowest(nums):
    lowest = nums[0]
    for n in nums:
        if n < lowest:
            lowest = n
```

Reduce

To write a **reduce** function:

1. Initialize the **acumulator** (`lowest`)
2. Define a for loop (`for n in nums:`)
3. Modify the accumulator (`lowest`) in the body of the for loop
4. Return the accumulator

```
def get_lowest(nums):
    lowest = nums[0]
    for n in nums:
        if n < lowest:
            lowest = n
    return lowest
```

Map + Filter + Reduce

Together, map, filter, and reduce consist of a powerful set of abstractions that allow us to transform data.

Breaking down a data transformation task into these steps, then implementing them, is a large part of any data analysis project.

```
def get_length(name):
    return len(name)

def we_like(name):
    return name != 'Bar'

def get_sum(nums):
    total = 0
    for n in nums:
        total += n
    return total

names = ['Foo', 'Bar', 'Baz']
lengths = [get_length(n) for n in names
           if we_like(n)]
total_length = get_sum(lengths)
```