Lesson 5

Python Essentials

Overview

1. Loops
2. Map
3. filter
4. Reduce
5. List comprehensions

For loops

- For loops takes each item in an array or collection in order, and assigns it to the variable you define.
- Loop through a collection `python for name in ['David', 'James']: print(name)`
- Loop a number of times `python for index in range(0, 2): print(index)`

While loop

- While loops perform an operation as long as a condition is true.

```
names = ['Christopher', 'Susan']
index = 0
while index < len(names):
    print(names[index])
    # Change the condition!!
    index = index + 1
```

Sorting

- **sorted()** can be used on lists, tuples and sets.

- **sort()** can only be used with lists
- **sort()** returns None and modifies the values in place

```
persons = [
  {'name': 'James', 'age': 50},
  {'name': 'David', 'age': 47}
]
```

Error: TypeError: '<' not supported between instances of 'dict' and 'dict'

```
persons.sort()

print(persons)
```

Works!

```
def sorter(item):
    return item['name']

presenters.sort(key=sorter)
```

Filtering

- filter() functional programming primitive.
- Tests if each element of a sequence true or not.
- Returns an iterator that is already filtered.

```
# sequence
sequence = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

def is_vowel(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

# using filter function
filtered = filter(is_vowel, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

Reduction

- **reduce()** applies a function to an iterable and reduce it to a single cumulative value.
- Popular in functional programming.

```
from functools import reduce

def addition(a, b):
  return = a + b

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

reduce(addition, numbers)
```

Output:

```
10
```

Using map()

- Alternative approach that's based in functional programming.
- You pass in a function and an iterable, and map() will create an object.
- This object contains the output you would get from running each iterable element through the supplied function.

```
fruits = ["apple", "banana", "cherry"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x.upper())

print(newlist)

fruits = ["apple", "banana", "cherry"]
newlist = []

def copy_list(fruits):
  return fruits.upper()

newlist = map(copy_list, fruits)

print(newlist)

['APPLE', 'BANANA', 'MANGO']
```

Lambda Functions

- Alternative approach that's based in functional programming.
- You pass in a function and an iterable, and map() will create an object.
- This object contains the output you would get from running each iterable element through the supplied function.

```
fruits = ["apple", "banana", "cherry"]
newlist = []

def copy_list(fruits):
  return fruits.upper()

newlist = map(copy_list, fruits)

print(newlist)
```

```
fruits = ["apple", "banana", "cherry"]
newlist = []

def copy_list(fruits):
  return fruits.upper()

newlist = map(lamda fruits: , fruits)

print(newlist)
```

```
['APPLE', 'BANANA', 'MANGO']
```

List Comprehensions

- Shorter syntax when you want to create a new list based on the values of an existing list.
- new_list = [expression for member in iterable]

```
fruits = ["apple", "banana", "cherry"]
newlist = []

def copy_list(fruits):
  return fruits.upper()

newlist = map(copy_list, fruits)

print(newlist)
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

Notes:

Considered more Pythonic: 1. Can be used for mapping and filtering. You don't have to use a different approach for each scenario. This is the main reason why list comprehensions are considered Pythonic, as Python embraces simple, powerful tools that you can use in a wide variety of situations. 1. You don't need to remember the proper order of arguments like you would when you call map(). 1. List comprehensions are also more declarative than loops, which means they're easier to read and understand. Loops require you to focus on how the list is created.

Warning! List comprehensions might make your code run more slowly or use more memory. If your code is less performant or harder to understand, then it's probably better to choose an alternative.