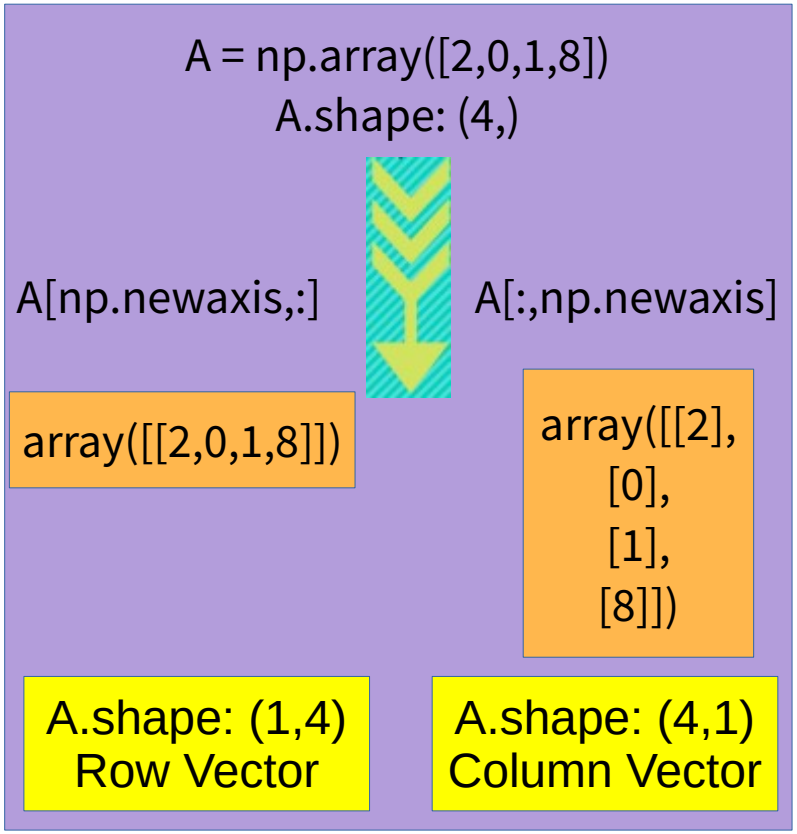
Not So Basic Python

np.newaxis is None

- When to use np.newaxis?
- In several contexts adding dimensions is useful:
  - If the data should have a specified number of dimensions. For example if you want to use matplotlib.pyplot.imshow to display a 1D array.
  - If you want NumPy to broadcast arrays. By adding a dimension you could for example get the difference between all elements of one array: a - a[:, np.newaxis]. This works because NumPy operations broadcast starting with the last dimension 1.
- To add a necessary dimension so that NumPy can broadcast arrays. This works because each length-1 dimension is simply broadcast to the length of the corresponding1 dimension of the other array.



map(func, args)

- `def fahrenheit(T):`
- `...  return ((float(9)/5)*T + 32)`
- `def celsius(T):`
- `...  return (float(5)/9)*(T-32)`
- `temperatures = (36.5, 37, 37.5, 38, 39)`
- `F = map(fahrenheit, temperatures)`
- `C = map(celsius, F)`
- `temps_in_Fahrenheit = list(map(fahrenheit, temperatures))`
  `# [97.7, 98.60000000000001, 99.5, 100.4, 102.2]`
- `temps_in_Celsius = list(map(celsius, temps_in_Fahrenheit))`
  `# [36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]`

lambda and map

```
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
print(F)
[102.56, 97.7, 99.14, 100.4, 100.03999999999999]
C = list(map(lambda x: (float(5)/9)*(x-32), F))
print(C)
[39.2, 36.5, 37.300000000000004, 38.00000000000001, 37.8]
```

.

map

- map() can be applied to more than one list

```
a = [1, 2, 3, 4]
b = [17, 12, 11, 10]
c = [-1, -4, 5]

list(map(lambda x, y : x+y, a, b))
[18, 14, 14, 14]

list(map(lambda x, y, z : x+y+z, a, b, c))
[17, 10, 19, 23]

list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
[37.5, 33.0, 24.5] # maps up to shortest length
```

filter(func, args)

- filter out all the elements of a sequence "sequence", for which the function function returns True

fibs = [0,1,1,2,3,5,8,13,21,34,55]

```
even_numbers = list(filter(lambda x:
x % 2 -1, fibs))
```

print(even_numbers)

[0, 2, 8, 34]

reduce in functools

- `functools.reduce(function, iterable[, initializer])`
- Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value.
- Eg., `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates ((((1+2)+3)+4)+5).
- The left argument, x, is the accumulated value and the right argument, y, is the update value from the sequence.
- If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

reduce, usage

```
from functools import reduce
f = lambda a,b: a if (a > b) else b
reduce(f, [47,11,42,102,13])
102
reduce(lambda x, y: x+y, range(1,101))
5050
reduce(lambda x, y: x*y,
range(44,50))/reduce(lambda x, y: x*y,
range(1,7))
```

problem

- Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Order	Book Title and Author	Quantity	Price per Item
34587	Learning Python, Mark Lutz	4	40.95
98762	Programming Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95
88112	Einführung in Python3, Bernd Klein	3	24.99

- Write a Python program using lambda and map, which returns a list of 2-tuples.
- Each tuple consists of the order number and the product of the price per items and the quantity. The product should be increased by 10, if the value of the order is smaller than 100,00.

solution

```
orders = [ ["34587", "Learning Python, Mark Lutz", 4, 40.95],
           ["98762", "Programming Python, Mark Lutz", 5, 56.80],
           ["77226", "Head First Python, Paul Barry", 3,32.95],
           ["88112", "Einführung in Python3, Bernd Klein", 3, 24.99]]

min_order = 100
invoice_totals =
     list(map(lambda x: x if x[1] >= min_order else (x[0], x[1] + 10),
              map(lambda x: (x[0],x[2] * x[3]), orders)))

print(invoice_totals)

.
```

problem

- The same bookshop, but this time we work on a different list. The sublists of our lists look like this:
  - [ordernumber, (article number, quantity, price per unit), ... (article number, quantity, price per unit)]
- Write a program using reduce which returns a list of two tuples with (order number, total amount of order).

solution

```
from functools import reduce

orders = [ [1, ("5464", 4, 9.99), ("8274",18,12.99), ("9744", 9, 44.95)],
       [2, ("5464", 9, 9.99), ("9744", 9, 44.95)],
       [3, ("5464", 9, 9.99), ("88112", 11, 24.99)],
       [4, ("8732", 7, 11.99), ("7733",11,18.99), ("88112", 5, 39.95)] ]

min_order = 100
invoice_totals = list(map(lambda x: [x[0]] +
                     list(map(lambda y: y[1]*y[2], x[1:])), orders))
invoice_totals = list(map(lambda x: [x[0]] +
                     [reduce(lambda a,b: a + b, x[1:])], invoice_totals))
invoice_totals = list(map(lambda x: x if x[1] >= min_order else (x[0], x[1] +
10), invoice_totals))
```

Closure

- A Python3 closure is when some data gets attached to the code.
- So, this value is remembered even when the variable goes out of scope, or the function is removed from the namespace.

```
>>> def outerfunc(x):
        def innerfunc():
            print(x)
        return innerfunc #Return function object
>>> myfunc=outerfunc(7)
>>> myfunc()  # 7
```

- Even if outerfunc is deleted, myfunc still gives us 7.

Closure: Another Eg.

```
def outer(x):
        result=0
        def inner(n):
                nonlocal result
                while n>0:
                        result+=x*n
                        n-=1
                return result
        return inner
myfunc=outer(7)
myfunc(3)
```

```
def outer(func):
    def inner(msg):
        func(msg)
    return inner

def sayhi(msg):
    print(msg)

myfunc=outer(sayhi)
myfunc("Hello")
```

Serialization: Pickle

- Pickling, is the act of converting a Python object into a byte stream.
- We also call this ‘serialization’, or ‘flattening’.
- Pickle is better than marshal, as pickle tracts the serialized objects. Same object not serialized twice.
- Unpickling is its inverse, ie., converting a byte stream from a binary file or bytes-like object into an object.

Pickle: Example

```
x=7
import os
os.chdir('C:\\Users\\lifei\\Desktop')
import pickle
f=open('abcde.txt','r+b')  //opened it in
binary mode to pickle
pickle.dump(x,f)
f.seek(0)
pickle.load(f)
```

Collections

Namedtuple() factory function for creating tuple subclasses with named fields

Deque list-like container with fast appends and pops on either end

ChainMap dict-like class for creating a single view of multiple mappings

Counter dict subclass for counting hashable objects

OrderedDict dict subclass that remembers the order entries were added

Defaultdict dict subclass that calls a factory function to supply missing values

UserDict wrapper around dictionary objects for easier dict subclassing

UserList wrapper around list objects for easier list subclassing

UserString wrapper around string objects for easier string subclassing

.We will discuss a few of them….

Counter collections for tally

```
from collections import Counter as C
C(‘Hello’) # ({‘l’: 2, ‘H’: 1, ‘e’: 1, ‘o’: 1})
c=C(a=3,b=2,c=1) # {‘a’: 3, ‘b’: 2, ‘c’: 1}
c.update(‘cabbage’) # {‘a’: 5, ‘b’: 4, ‘c’: 2, ‘e’:1, ‘g’:1}
for i in c.elements():
    print(f"{i}: {c[i]}")

# Find the ten most common words in Hamlet
import re
words = re.findall(r'\w+', open('hamlet.txt').read().lower())
C(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

How to get the least common?

Counter basics

```
c = Counter()                           # a new, empty counter
c = Counter('gallahad')                 # a new counter from an
iterable
c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
c = Counter(cats=4, dogs=8)             # a new counter from keyword
args
c = Counter(['eggs', 'ham'])
c['bacon']                              # count of a missing element is
zero 0
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)
C  # Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Counter: Common patterns

- `sum(c.values())  # total of all counts`
- `c.clear()  # reset all counts`
- `list(c)  # list unique elements`
- `set(c)  # convert to a set`
- `dict(c)  # convert to a regular dictionary`
- `c.items()  # convert to a list of (elem, cnt) pairs`
- `Counter(dict(list_of_pairs))  # convert from a list of (elem, cnt) pairs`
- `c.most_common()[:-n-1:-1]  # n least common elements`
- `c += Counter()  # remove zero and negative counts`

Counters and arithmetic

```
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
c + d                     # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
c - d                     # subtract (keeping only positive counts)
Counter({'a': 2})
c & d                     # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
c | d                     # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

Compute Mode

```
# mode.py

from collections import Counter

def mode(data):
    counter = Counter(data)
    _, top_count = counter.most_common(1)[0]
    return [point for point, count in counter.items() if count == top_count]
```

Deque from collections

```
from collections import deque
d = deque('ghi')                 # make a new deque with three items
for elem in d:                   # iterate over the deque's elements
...     print elem.upper()       # G\n H\n I

d.append('j')                  # add a new entry to the right side
d.appendleft('f')              # add a new entry to the left side
d                              # deque(['f', 'g', 'h', 'i', 'j'])
d.pop()                        # return and remove the rightmost item, 'j'
d.popleft()                    # return and remove the leftmost item, 'f'
list(d)                        # list the contents of the deque ['g', 'h', 'i']
d[0]                           # peek at leftmost item, 'g'
d[-1]                          # peek at rightmost item, 'i'
```

Deque from collections

```
list(reversed(d))    # list the contents of a deque in reverse['i', 'h', 'g']
'h' in d             # search the deque, True
d.extend('jkl')      # add multiple elements at once deque(['g', 'h', 'i', 'j',
'k', 'l'])
d.rotate(1)          # right rotation deque(['l', 'g', 'h', 'i', 'j', 'k'])
d.rotate(-1)         # left rotation deque(['g', 'h', 'i', 'j', 'k', 'l'])
deque(reversed(d))   # make a new deque in reverse order deque(['l', 'k', 'j',
'i', 'h', 'g'])
d.clear()            # empty the deque
d.pop()              # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque
```

deque Recipes

```
deque(open(filename), n) #Linux tail
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / float(n)
```

namedtuple

- Factory Function for Tuples with Named Fields

```
Point = namedtuple('Point', ['x', 'y'], verbose=True)
p = Point(11, y=22)# instantiate with positional or keyword args
p[0] + p[1]         # indexable like the plain tuple (11, 22), 33
x, y = p            # unpack like a regular tuple
x, y                # (11, 22)
p.x + p.y           # fields also accessible by name, 33
p                   # readable __repr__ with a name=value style
Point(x=11, y=22)
```

namedtuple

- Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules

```
EmployeeRecord = namedtuple('EmployeeRecord',
'name, age, title, department, paygrade')


import csv

for emp in map(EmployeeRecord._make,
csv.reader(open("employees.csv", "rb"))):

    print emp.name, emp.title
```

namedtuple

```
import sqlite3

conn = sqlite3.connect('/companydata')

cursor = conn.cursor()

cursor.execute('SELECT name, age, title,
department, paygrade FROM employees')

for emp in map(EmployeeRecord._make,
cursor.fetchall()):

    print emp.name, emp.title
```

OrderedDict: Collections

```
from collections import OrderedDict as od
d={'a':3, 'c':1, 'b':4}
o= od(d.items)
o.popitem() # (‘b’,4)
o.popitem(last=False) # (‘a’,1)
o.pop(‘c’) # (‘c’,1) o is empty
# regular unsorted dictionary
d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}
# dictionary sorted by key
od(sorted(d.items(), key=lambda t: t[0])) # ([('apple', 4), ('banana', 3), ('orange', 2),
('pear', 1)])
# dictionary sorted by value
od(sorted(d.items(), key=lambda t: t[1])) # ([('pear', 1), ('orange', 2), ('banana', 3),
('apple', 4)]
# dictionary sorted by length of the key string
od(sorted(d.items(), key=lambda t: len(t[0]))) [('pear', 1), ('apple', 4), ('orange', 2),
('banana', 3)]
```

Itertools

- Functions creating iterators for efficient looping

Projects

- https://www.python-course.eu/text_classification_python.php