bit.ly/clps950_2pm

picking up speed with python

this week we're going to cover a lot of ground!

the goal is to give you the basics for building to whatever you're going to need to be a successful programmer – we won't have time to go into great depth for most of these but hope to give you a taste of the power of python

we'll learn **tuples** and **dictionaries**, then the more powerful dictionary-like objects **pandas dataframes**. we'll see how to convert all the matrix operations you learned into **numpy,** and work with **matplotlib** to do plotting in python

along the way we'll also look at **IO** (input/output), **packages**, error and **exceptions**, and the all powerful **lambda** functions

tuples - one of 4 built in python data types

tuples are data structures which are used to store multiple items in a single variable

mytuple = (“apple”, “banana”) # create a tuple – see () not [] like a list!

tuple items are **ordered** and **unchangeable.** they can have **duplicate values**.

mytuple = (“apple”, “apple”) # create a tuple with the same elements

they can have multiple types of data

mytuple = (“apple”, 4, True) # create a tuple with mixed values

you can access them like lists

print(mytuple[0]) # what do you think should happen?

but! you cannot edit their contents

```
>>> mytuple[0] = 'oranges'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
 item assignment
```

Dictionaries

dictionaries are data types made of pairs of “keys” and “values” like a real paper dictionary. you can think of the key as the word, and the value as the definition. the keys must be single data types, like a string or a number, but the values can be anything you want – even other dictionaries!

dictionaries are not ordered, so you can’t access them using [numbers] like for arrays or lists, unless your keys are numbers!

dictionaries can be edited and are therefore “mutable” – unlike tuples which are “immutable”!

mydictionary = {‘myval’:5}; mytuple = (‘myval’,5); mylist = [‘myval’,5];

myset = set(‘myval’, 5); # we won’t do sets, but they are the 4th built-in datatype

```
# to make a dictionary, you use curly brackets {}
# then we add elements using square brackets []
prices = {}
prices['apples'] = 1.3
prices['oranges'] = 4.9
prices['peppers'] = {'red':2.1,'green':5.9}
print(prices)
print(prices['apples'])
```

```
{'apples': 1.3, 'oranges': 4.9, 'peppers': {'red': 2.1, 'green': 5.9}}
1.3
```

```
# what is the price of red peppers? we can chain the square brackets to go into another dictionary!
print(prices['peppers']['red'])
```

```
2.1
```

```
# apples go on sale!
prices['apples'] -= 1
print(prices['apples'])
```

```
0.30000000000000004
```

```
# what is at the store?
prices.keys()
```

```
dict_keys(['apples', 'oranges', 'peppers'])
```

```
] # and what do they cost?
  prices.values()
```

```
dict_values([0.30000000000000004, 4.9, {'red': 2.1, 'green': 5.9}])
```

```
# how about in pairs?
prices.items()
```

```
dict_items([('apples', 0.30000000000000004), ('oranges', 4.9), ('peppers', {'red': 2.1, 'green': 5.9})])
```

```
# we can iterate through a dictionary!
for item in prices.items():
    print(item)
```

```
('apples', 0.30000000000000004)
('oranges', 4.9)
('peppers', {'red': 2.1, 'green': 5.9})
```

```
# we can also use a more powerful for loop operation:
# since we know there will be 2 values in each of the items call, we can explicitly ask for them and name them
for k,v in prices.items():
    print('item is '+ k)
    print('value is '+str(v))
```

```
item is apples
value is 0.30000000000000004
item is oranges
value is 4.9
item is peppers
value is {'red': 2.1, 'green': 5.9}
```

if I ran **for (k,v) in prices.items():** for the first line, what data type would (k,v) be?

a foray into packages…

we've been saying "built in" about these data types. this means that python *on its own* comes with this functionality. but what's amazing about python is that we can use other code which people write, and add it to our own code to become more powerful. *it's like being able to download folders of other helpful functions that other people have written, but instead of just functions it will also have other data structures!* we'll introduce some of the most popular packages – numpy, pandas, and matplotlib

follow the tutorial on canvas for more details than we can cover in lecture

```
# import all of a package's functions and data types
# using the packages' name to index what you want to use
import numpy as np
print(np.mean([1,2,3,4])) # use numpy's function mean
print(np.std([1,2,3,4])) # use numpy's function std
```

```
2.5
1.118033988749895
```

```
# import a specific function/ data type
# which you can call without the package
# doesn't import anything else
from numpy import mean
print(mean([1,2,3,4]))
print(std([1,2,3,4])) # doesn't exist!
```

```
2.5
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-e917ca256336> in <module>()
      4 from numpy import mean
      5 print(mean([1,2,3,4]))
----> 6 print(std([1,2,3,4])) # doesn't exist!

NameError: name 'std' is not defined
```

pandas: series and dataframes

some of you used matlab **tables** for your projects. pandas is a python **package** which is for working with big data tables, and for importing and exporting data.

a pandas series is like an ordered group of dictionaries all using the same keys

```
import pandas as pd
students = ['abby','alejandro','isabella','alaina']
favorite_snack = ['cookie','broccoli','iced coffee','pizza']
s = pd.Series(favorite_snack,index=students)
print(s)
```

```
abby              cookie
alejandro       broccoli
isabella     iced coffee
alaina             pizza
dtype: object
```

any non-built in data type is an object!

```
# you don't have to pass an index
# it will just assume you want to number the items
s = pd.Series(favorite_snack,)
print(s)
```

```
0         cookie
1       broccoli
2    iced coffee
3          pizza
dtype: object
```

pandas series and dataframes

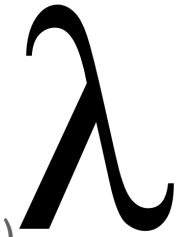
```
# we can retrieve values associated with keys
students = ['abby','alejandro','isabella','alaina']
favorite_snack = ['cookie','broccoli','iced coffee','pizza']
s = pd.Series(favorite_snack,index=students)
print('alejandros favorite is ' + s['alejandro'])
# and update them as well!
s['alejandro'] = 'popcorn'
print('~~ my updated data frame ~~')
print(s)
```

```
alejandros favorite is broccoli
~~ my updated data frame ~~
abby              cookie
alejandro        popcorn
isabella     iced coffee
alaina             pizza
dtype: object
```

```
# the real power of pandas comes from dataframes
# which are like combining multiple series with the same indices
# so if we wanted to store our TA's section time too
# we could do this in a dataframe
section_times = [2,2,10,10]
df = pd.DataFrame(index=students)
df['snacks'] = favorite_snack
df['sections'] = section_times
df
```

	snacks	sections
abby	cookie	2
alejandro	broccoli	2
isabella	iced coffee	10
alaina	pizza	10

lambda functions : power of minimal syntax



so far, we have made all of our functions using **def** syntax, but python allows for **anonymous** functions as well – these are called lambdas! (also in matlab with @s)

```
# syntax:
# function_name = lambda variable: result_using_variable
funA = lambda x: x**2
funB = lambda x: x**3

print(funA(3))
print(funB(4))
```

```
9
64
```

```
funC = lambda x: 'egg'
print(funC(10))
```

```
funD = lambda egg: egg**2
funD(2)
```

```
# why is this special? why not use def?
# we can now return a function from a function
# and save that into another variable
def raise_power(n):
    return lambda x: x**n

cubing = raise_power(3)
squaring = raise_power(2)
print(cubing(3))
print(squaring(10))
```

use case: lambdas and pandas

```
df
```

	snacks	sections
abby	cookie	2
alejandro	broccoli	2
isabella	iced coffee	10
alaina	pizza	10

```
df['newcolumn'] = df.apply(lambda x: x.name+' likes to eat '+x['snacks']+\
                ' and teaches in the '+('morning' if x['sections']==10 else 'afternoon'), axis=1)
```

```
df['newcolumn'].values
```

```
array(['abby likes to eat cookie and teaches in the afternoon',
       'alejandro likes to eat broccoli and teaches in the afternoon',
       'isabella likes to eat iced coffee and teaches in the morning',
       'alaina likes to eat pizza and teaches in the morning'],
      dtype=object)
```

bit.ly/sheep_clps950