A word of caution about Python 2 vs. Python 3. **They are not strictly compatible.** Most of the incompatibilities you are unlikely to see for a while. The exceptions are **print** and integer division. We'll cover these shortly. Those using OSX need to be careful with Python. Why? A version of Python ships with OSX. Not only is this version 2 but it is an old version of python (2.5?) that the system depends on and which you cannot remove without destroying your system. So, you'll end up with at least two versions of Python on your system and that can make things messy if you aren't careful. If you are on the command line using the command **which python** (or variants) may help clear things up.

Anaconda should make working with Python *very* easy. Perhaps too easy. Always remember that everything you are doing in the fancy tool can be done on the command line with a text editor.

Find the Anaconda launcher and launch an iPython notebook. OSX command line ipython notebook

Jupyter notebook tour...

Objects, operators, punctuation (syntax), expressions, and statements

- Objects are the nouns of the language. You do things to them with other things. Strings, integers, variables, lists, etc. are all types of objects.
- Operators are the things you use to do things to objects. When you do something to an object with an operator we refer to that object as an operand. Things like + - ^ * ** / = == < > <= >= are all operators.
- Punctuation are the characters (including spaces) that keep the syntax together. These include spaces, : , () [] {}
- Expressions arise when we link objects together with operators and the right punctuation.
- Statements are things that the computer will do. Expressions are kinds of statements but think of them as separate things. Expressions have values, statements work with values.

Strings

- Strings are *sequences of characters* (think of them as a special kind of ordered list).

```
string1 = "hello world!"

print(string1)
```

With this the students see both the assignment operator ("=") and a function ("print()"). Take a moment to explain both of these.

- As sequences we can index and slice strings.

```
string1[1]
```

- Why do we get "e" as the return? Zero based counting.

```
string1[0:5]
```

- What values will just give us "world!"? string[6:12] and string1[6:]
- What about [0:0] and [6:3] and negative values?
- Strings are immutable. We modify them only by overwriting them.

```
string1[0:1] = "H"

string1 = "Hello World!"

string operators + *

string2 = string1 + string1

print string2
```

- How do we get a space? what about each on its own line? print string2 * 3
- string methods: len, upper, lower, replace, etc.

```
len(string1)

string1.swapcase()

string1.swapcase()
```

- Why didn't it go back to lowercase the second time? We're not modifying the string.

```
string1.replace("!", "!!!")
```

Numbers and Math

- Try all the standard operators + * / -
- What does 1 // 2 produce?
- 1. Why? Integer division.
- *Warning:* In Python 2 / is *integer division by default*. Can overcome this by adding decimal placeholders to at least one component of the formula.
- What does the ** operator do?
- ** is the exponent. Often ^ elsewhere.

```
2 ** 3
```

- What does the % operator do?
- % is the modulo operator. It returns the remainder when the first number is divided by the second.

```
7 % 2
```

- returns 1 since 2 goes into 7 three times with one left over.
- Assign an integer and a decimal to variables

Typing, variable assignments, and printing

- Have them play with type. Give them a string, an integer, a decimal. See what they can swap back and forth

```
int1 = 5

dec1 = 4.0
```

- What is int1 * 5?

```
str(int1)
```

- What is int1 * 5?
- Will still be 25 because we didn't actually change int1. Use int1 = str(int1). Now what is int1 * 5? "55" the string five-five rather than the number 55.
- type(int1) will show us the type. What type is string1 and int1?
- What type is the output of 1.0 / 2?
    - Remind them about case sensitivity and spaces.
    - Remind them that outside of interpretive mode "Silence is golden" and have them work with the print command
- What can we recast using str, int, and float?

Lists

- Ordered collection of objects. strings are a special type of list where each element is a character and the list type has its own methods.

```
list1 = ["hello","world","!"]
```

- indexing and slicing is just like strings

```
list1[0]

list1[1:2]

list1[1:1]

list1[0]="Hello"

list1[1]=list[1].capitalize()
```

- Lists can hold almost any type of object and can be heterogeneous.

```
list2=[5,4.0,"wow"]
```

- Adding

```
list1.append(list2)
```

- How will we capitalize the "*wow*" in list1?

```
list1[3][2].capitalize()

list1.insert(2,"two")
```

- What operators can we use with lists? Try some out!

```
list1 + list2

list1 * 3

list1 + string1
```

- *list1 + string1* fails. Need to recast string1 as a list. [string1]
- list methods. append, expand, and sort. Note that list methods are all void. They modify the original list and return None.

```
list1.sort()

list1.reverse

list1.index("Hello")
```

- deleting elements

```
list1.pop(2)

list1.remove("wow")
```

- *list1.remove("wow")*fails. Need to access the list in the list.
- Only removes the first instance

Dictionaries

- key : value pairs. Use when you want to index by keys rather than locations.

```
dict1 = {'one':'un','two':'deux','three':'trois'}

dict1
```

- note how the order has 9(likely) changed!
- Accessing elements

```
dict1[two]
```

- *dict1[two]* fails, no variable named "two"

```
dict1['two']

dict1['four']
```

- *dict1['four']* fails, no index named "four"
- Checking members

```
'four' in dict1

'trois' in dict1
```

- *'trois' in dict1* fails because it is a value, not a key

```
dict1Val = dict1.values()

'trois' in dict1Val
```

- Or go straight for the throat with *'trois'* in *dict1.values()*
- Adding/modifying members

```
outFile = open(‘output.txt’, ‘r’)

for line in outFile:

    print linedict1[four] =  ‘quatre’
```

- Checking for inverse members. Can’t do that yet (Dictionaries don’t work that way by default). For that we need functions, loops, and conditionals.

Functions

- Functions take arguments and return values. type() was a function. It took and object as argument and returned its type as the value.

```
def print2x():

print "hello world" * 2
```

- Values can be passed to a function

```
def duplicator(passedValue):

print passedValue * 2
```

- Multiple values can be passed

```
def multiplicator(passedValue1, passedValue2):

    print passedValue1 * passedValue2
```

- See what happens when you pass the wrong value
- Note that the name of the object that is passed (the argument) does not need to be the same as the name of the input (the parameter).
- Also do a function with two parameters that defines a variable inside and a function with no inputs. Fruitful and void functions. Show that the variable is not available outside.

```
* Good practice to pass all needed objects as parameters and to return
them if changes are made.
```

- rebuild a few of the previous functions/simple scripts and then run them as programs from the command line

Loops

- While

```
fruit = ‘banana'

index = 0

while index < len(fruit):

    letter = fruit[index]

    print(letter)

    index = index + 1
```

- For fruit = ‘banana’ for letter in fruit: print letter

Conditionals

- If

```
x = 5

y = 4

if x < y:

    print ‘x is less than y’
```

- Else

```
else:
    print ‘x is not less than y’
```

- Elif

```
elif x==y:
    print ‘x is equal to y'
```

- conjunction with a conditional

```
if 0 < x and x < 10:
    print ‘x is a positive single digit number’
```

- In

```
if ‘a’ in fruit:
    print fruit
```

- Function with a while loop with a conditional

```
def find(word, letter)
    index = 0
    while index < len(fruit)
        if word[index] == letter:
            return index
        return -1
    print find(fruit, ‘a’)
```

Reading and writing to files

- Open

```
outFile = open(‘output.txt’, ‘w’)

print outFile
```

- Write

```
outText = [‘This is a test of writing to a file’, ‘This is a test of
writing a second line’]

for item in outText:

    outFile.write(item)
```

- Close

```
outFile.close()
```

- Read

```
outFile = open(‘output.txt’, ‘r’)

for line in outFile:

    print line
```

- Appending

```
outFile = open(‘output.txt’, ‘a’)

for line in outFile:

    print line
```

- Write

```
outFile = open(‘output.txt’, ‘r’)

for line in outFile:

    print line
```

User input

- Raw Input

```
input = raw_input()

print raw_input()
```

- Help Text

```
input = raw_input(‘What is your name? ‘)

print ‘Nice to meet you ‘ + input + ‘!'
```

You now have enough to be dangerous. So, some practice (being dangerous) is in order.

- Build a program that asks the user for a number and and then prints ‘Hello World!’ that many times
- Modify the program (or write a new one) that asks a user for a number and then writes ‘Hello World!’ that many times to a filename provided by the user
- Take the previous program and have it drop one letter from the end of ‘Hello World!’ for each line it writes
- Take a word and count all the letters of that word and report them
- Write a program that asks for words and tells the user how many vowels are in the word until the user types ‘stop'