PYTHON PROGRAMMING

Notes by Michael Brothers, available on http://github.com/mikebrothers/data-science/

Content taken from the following sources (among others):

Avinash Jain's Udemy course *Introduction To Python Programming*
https://www.udemy.com/pythonforbeginnersintro/

Jose Portilla's Udemy course *Complete Python Bootcamp*
https://www.udemy.com/complete-python-bootcamp/

Bill Lubanovic's book *Introducing Python: Modern Computing in Simple Packages*
(O'Reilly Media; 1st edition, December 4, 2014)

Table of Contents

PYTHON PROGAMMING 4
- **Variable** 4
- **Multiple Declaration** 4
- **Multiple Assignment** 4
- **Data Types** 4
- **Operators** 4
- **Relational operators:** 4
- **Chained Comparison Operators:** 4
- **Strings**: 4
- **Lists**: 4
- **Tuples**: 4
- **Dictionaries**: 4
- **Sets:** 5
- **Comments:** 5

WORKING WITH STRINGS 5
- **Built-in String Functions:** 5
- **Built-in String Methods:** 5
- **Splitting Strings:** 5
- **Joining Strings:** 5
- **Turning Objects Into Strings** 6
- **Escape characters:** 6
- **Placeholders:** 6

FORMAT 7

WORKING WITH LISTS: 8
- **Built-in List Functions:** 8
- **Built-in List Methods:** 8
- **List Index Method** 9
- **Making a list of lists:** 9

LIST COMPREHENSIONS 9

WORKING WITH TUPLES: 10

WORKING WITH DICTIONARIES: 10
Dictionary Comprehensions: 10
WORKING WITH SETS: 10
Set Operators: 10
Built-in Set Methods: 11
RANGE 11
CONDITIONAL STATEMENTS & LOOPS 12
If / Elif / Else statements: 12
For Loops 12
While Loops 12
Nested For Loops 12
Loop Control Statements (Break, Continue & Pass) 13
Try and Except 13
INPUT (formerly raw_input) 13
UNPACKING 14
Tuple Unpacking 14
Dictionary Unpacking 14
FUNCTIONS 15
Default Parameter Values 15
Positional Arguments *args and **kwargs 15
Inner Functions: ..... **Error! Bookmark not defined.**
Closures: ..... **Error! Bookmark not defined.**
PRE-DEFINED FUNCTIONS 16
LAMBDA EXPRESSIONS 16
MORE USEFUL FUNCTIONS 17
MAP 17
REDUCE 17
FILTER 17
ZIP 17
ENUMERATE 18
ALL & ANY 18
COMPLEX 18
PYTHON THEORY & DEFINITIONS 19
FUNCTIONS AS OBJECTS & ASSIGNING VARIABLES 20
FUNCTIONS AS ARGUMENTS 20
DECORATORS: 21
GENERATORS & ITERATORS 22
NEXT & ITER built-in functions: 22
GENERATOR COMPREHENSIONS 22
WORKING WITH FILES 23
READING AND APPENDING FILES 23
RENAMING & COPYING FILES 23

OBJECT ORIENTED PROGRAMMING – Classes, Attributes & Methods 24

MODULES 28

COLLECTIONS Module: 28
- **Counter** 28
- **defaultdict** 29
- **OrderedDict** 29
- **namedtuple** 30

DATETIME Module 30

TIMEIT Module 30

PYTHON DEBUGGER – the pdb Module 31

REGULAR EXPRESSIONS – the re Module 31
- Searching for Patterns in Text 31
- Finding all matches 32
- Split with regular expressions 32
- Using metacharacters 32

STYLE AND READABILITY (PEP 8) 34

GOING DEEPER: 36
- **The '_' variable** 36
- **To print on the same line:** 36
- **Some more (& obscure) built-in string methods:** 37
- **Some more (& obscure) built-in set methods:** 37
- **Common Errors & Exceptions:** 38
- **For more practice:** 38

PYTHON PROGAMMING

Variable: reserved memory space. Can hold any value, assigned to a term. Case-sensitive. Can't contain spaces.

Variable names:
1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead
3. Can't use any of these symbols: ' " , < > / ? | \ ( ) ! @ # $ % ^ & * ~ - +
4. It's considered best practice (PEP8) that the names are lowercase.
5. Don't use these reserved words: and assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try while

Multiple Declaration: `var1, var2, var3 = 'apples','oranges','pears'`
Multiple Assignment: `var1 = var2 = var3 = 'apples'` (spaces/no spaces doesn't matter)
Data Types: number (integer or float), string (text), list, tuple, dictionary, set, Boolean (True, False, None)
Operators: + - * / addition, subtraction, multiplication, division
% modulo = gives the remainder after division
// floor divisor = discards the fraction without rounding
** exponentiator

`>>> 5/2` returns `2.5`
`>>> 5//2` returns `2`
`>>> 5%2` returns `1`
`>>> 5**3` returns `125`

```
NOTE: Python 2 treats '/' as 'classic division'
and truncates the decimal. Python 3 does
'true division' and always returns a float.
```

Note: 2 is an *int* type number, while 2.5 is a *float*. Division returns a float. (`6/3` returns `2.0`)

Relational operators: (aka Comparison Operators)

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	equal to	(use == when comparing objects.	
!=	not equal to	One equals sign is used to assign values to objects.)	
<>	not equal to		

Chained Comparison Operators:
`1 < 2 < 3` returns `True` (this is shorthand for `1 < 2 and 2 < 3`)

Strings: anything between two sets of quotation marks (single or double)
use `\n` in a string to insert a line-break, `\t` for a tab
NOTE: strings are immutable. You can't change elements in a string once they're created, but you can add to them

Lists: `list1 = ['apples','oranges','pears']` created using square brackets

Tuples: `tuple1 = (1,2,3)` created using parentheses
Tuple elements cannot be modified once assigned (tuples are immutable)
`max(tuple1)` returns 3, `min(tuple1)` returns 1

Strings, lists and tuples are *sequences*. Their contents are indexed (0,1,2…)
`list1[1]` returns `'oranges'`
`tuple1[1]` returns `2`

Dictionaries: contain a key and a value, using { } and colons
`dict1 = {'Tom':4, 'Dick':7, 'Harry':23}` created using curly braces
`dict1['Harry']` returns `23`
Dictionaries are mappings, not sequences. `dict1[1]` would return an error.

Sets: behave like dictionaries, but only contain unique keys. Sets are unordered (not sequenced).
`set1 = set([1,1,2,2,3])` this is called "casting a list as a set"
`set1` returns `{1,2,3}`

Comments:
`#` (hash) provides quick one-liners
`"""` (triple quotes) allow multiline full text (called docstrings) `"""`

WORKING WITH STRINGS
Slices: `var1[10]` returns the 11th character in the string (all indexing in Python starts at 0)
`var1[2:]` returns everything after the second character (ie, it chops off the first two elements)
`var1[:3]` returns everything up to the third character (ie, the first three elements)
`var1[1:-1]` returns everything between the first and last character
Steps: `var1[::2]` returns every other character starting with the first (0,2,4,6…)
`var1[::-1]` returns the string *backwards* [aka Reversing a String]
Concatenate: `var1 = var1 + ' more text'`
Multiply: `var1*10` returns the var1 string 10 times
Reverse: `var1[::-1]` (there is no built-in reverse function or method)
Shift: `var1[2:]+var1[:2]` moves the first two characters to the end

Built-in String Functions:
len(string) returns the length of the string (including spaces)
str(object) converts objects (int, float, etc.) into strings

Built-in String Methods:
.upper s.upper() returns a copy of the string converted to uppercase.
.lower s.lower() returns a copy of the string converted to lowercase.
.count s.count("string") adds up the number of times a character or sequence of characters appears in a string *(case-sensitive!)* NOTE: If `s='hahahah'` then `s.count('hah')` returns only 2.
.isupper s.isupper() returns true if all cased characters in the string are uppercase. There must be at least one cased character. It returns false otherwise.
.islower s.islower() returns true if all cased characters in the string are lowercase. There must be at least one cased character. It returns false otherwise.
.find s.find(value,start,end) finds the index position of the first occurrence of a character/phrase in a range
.replace s.replace("old","new")
In Jupyter, hit Tab to see a list of available methods for that object.
Hit Shift+Tab for more information on the selected method - equivalent to `help(s.method)`

Splitting Strings:
```
>>> greeting = 'Hello, how are you?'
>>> greeting.split() returns ['Hello,', 'how', 'are', 'you?']
```
Note that the default delimiter is a space

```
>>> fruit = 'Apple'
>>> fruit.split('p') returns ['a', '', 'le'] (note the additional null value)
>>> fruit.partition('p') returns ('a','p','ple') (head, sep, tail)
```
Note also that methods work on objects, so `'The quick brown fox'.split()` is valid

Joining Strings:
`delimeter.join(list)` joints a list of strings together, connected by a start string (delimeter)
```
>>> list1 = ['Ready', 'aim', 'fire!']
>>> ', '.join(list1) returns 'Ready, aim, fire!'
```

Turning Objects Into Strings : `str()` *aka "casting objects as strings"*

```
>>> test = 3
>>> print('You have just completed test ' + str(test) + '.')
You have just completed test 3.
```

Escape characters:

`string = 'It's a nice day'` returns an error

`string = 'It\'s a nice day'` handles the embedded apostrophe

\ can also break code up into multiline statements for clarity

Note: embedded apostrophes are also handled by changing the apostrophe type

`string = "It's a nice day"` is also valid.

Placeholders: (%s, %f et al) *Note: the .format() method is usually preferable. See below.*

Placeholders: `%s` acts as a placeholder for a string, `%d` for a number

```
>>> print('Place my variable here: %s' %(string_name))
```

Note that %s converts whatever it's given into a string.

```
print('Floating point number: %1.2f' %(13.145))
Floating point number: 13.14
```

where in 1.2, **1** is the minimum number of digits to return,
and **2** is the number of digits to return past the decimal point.

```
print('Floating point number: %11.4f' %(13.145))
Floating point number:     13.1450
```

There are 4 extra spaces (11 total characters incl decimal)

NOTE: %s replicates the str() function, %r replicates the repr() function to do the same thing.

Passing multiple objects:

```
print('First: %s, Second: %s, Third: %s' %('hi','two',3))
First: hi, Second: two, Third: 3
```

Variables are passed in the order they appear in the tuple. Not very pythonic because to pass the same variable twice means repeating it in the tuple. **Use .format instead** (see below!)

Omitting the argument at the end causes the placeholder to print explicitly:

```
print('To round 15.45 to 15.5 use %1.1f')
To round 15.45 to 15.5 use %1.1f
```

…as does using %% (python sees this as a literal %)

```
print('To round 15.45 to %1.1f use %%1.1f') %(15.45)
To round 15.45 to 15.5 use %1.1f
```

NOTE: Python 2.7 has a known issue when rounding float 5's (up/down seem arbitrary).
See http://stackoverflow.com/questions/24852052/how-to-deal-with-the-ending-5-in-a-decimal-fraction-when-round-it
For better performance, use the decimal module.

FORMAT

Double curly-brackets serve as positional placeholders and eliminate need for str()

```
print('I prefer Python version {} to {}.'.format(3.4, 2.7))
```
Note the lack of quotes

```
I prefer Python version 3.4 to 2.7.
```

You can change the order of variables inside the function:

```
print('I prefer Python version {1} to {0}.'.format(3.4, 2.7))
I prefer Python version 2.7 to 3.4.
```

You can assign local variable names to placeholders:

```
print('First: {x}, Second: {y}, Third: {z}.'.format(x=1., z='B', y=5))
First: 1.0, Second: 5, Third: B.
```

Note that variables x, y and z are not defined outside of the function, and format handles the different object types. Unlike %s placeholders, format variables may be used more than once in a string, and stored in any order.

Within the brackets you can assign field lengths, left/right alignments, rounding parameters and more

```
print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
Fruit    | Quantity
Apples   |       3.0
Oranges  |        10
```

the **0** parameter takes the first object encountered

the **8** parameter sets the minimum field length to 8 characters

By default, .format aligns text to the left, numbers to the right

```
print('{0:8} | {1:<8}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:<8.2f}'.format('Apples', 3.66667))
print('{0:8} | {1:<8.2f}'.format('Oranges', 10))
Fruit    | Quantity
Apples   | 3.67
Oranges  | 10.00
```

< sets a left-align (^ for center, > for right)

.2f converts the variable to a float with 2 decimal places

You can assign field lengths as arguments:

```
print('{:<{}} goal'.format('field', 9))
field     goal
```

With manual field specification this becomes `{0:<{1}s}`

You can choose the padding character:

```
print('{:-<9} goal'.format('field'))
field---- goal
```

You can truncate (the opposite of padding):

```
print('{:.5}'.format('xylophone'))
xylop
```

…and by argument:

```
print('{:.{}}'.format('xylophone',7))
xylopho
```

Conversion tags enable output in either str, repr, or (in python3) ascii: `{ !s}  { !r} { !a}`

Format supports named placeholders (**kwargs), signed numbers, Getitem/Getattr, Datetime and custom objects.

For more info: https://pyformat.info

WORKING WITH LISTS:

Built-in List Functions:

`del list1[1]`	removes the second item from the list
`len(list1)`	returns the number of objects in the list
`len(list1[-2])`	returns the number of *characters* in the second-to-last string in the list, including spaces)

Built-in List Methods:

.append	L.append(object) -- append object to end
.count	L.count(value) -> integer -- return number of occurrences of value
.extend	L.extend(iterable) -- extend list by appending elements from the iterable
.index	L.index(value, [start, [stop]]) -> integer -- return first index of value. Raises ValueError if the value is not present.
.insert	L.insert(index, object) -- insert object before index
.pop	L.pop([index]) -> item – remove and return item at index (default last).
.remove	L.remove(value) -- remove first occurrence of a value. Raises ValueError if the value is not present.
.reverse	L.reverse() -- reverse *IN PLACE*
.sort	L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

In Jupyter, hit Tab to see a list of available methods for that object.
Hit Shift+Tab for more information on the selected method - equivalent to `help(l.method)`

Adding objects to a list: `list1.append('rhubarb')`
Adding multiple objects to a list: `list1.extend('turnips','squash')`
Adding contents of one list to another: `list1.extend(list2)` adds the *contents* of list2 to list1
NOTE: to add a list to another list *as one object*, use append.

```
list1 = ['a', 'b']
```

`list1.extend['c', 'd']` returns `['a', 'b', 'c', 'd']`
`list1.append['c', 'd']` returns `['a', 'b', ['c', 'd']]`

Inserting items into a list: `list1.insert(3,'beets')` puts 'beets' in the fourth position

Sorting items in a list: `list1.sort()` rewrites the list in alphabetical order IN PLACE
`list2 = sorted(list1)` creates a new list while retaining the original
Reverse sorting a list: `list1.sort(reverse=True)`

Reverse items in a list: `list1.reverse()` reverses the order of items in a list IN PLACE

Remove items from a list: `list1.pop()` returns the last (-1) item and permanently removes it
`list1.pop(1)` returns the second item and removes it
You can capture the popped object:

```
list3 = [1,2,3,4]
x = list1.pop()
print(x) returns 4
print(list3) returns [1,2,3]
```

To check the existence of a value in a list: `object in list` returns True/False as appropriate (names work too)

To join items use the *string method*: `' potato'.join(['one',', two','.'])`
returns `'one potato, two potato.'`

List Index Method

`list.index(object)` returns the index position of the *first* occurrence of an object in a list.

```
list1 = ['a','p','p','l','e']
list1.index('p') returns 1
```

Making a list of lists:

```
list1=[1,2,3]    list2=[4,5,6]    list3=[7,8,9]
   matrix = [list1,list2,list3]
   matrix
   [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Note: "matrix" absorbs the *content* of the lists, not the variable names. If you later change one of the lists, matrix will not be affected.

Slicing:

`matrix[0]` returns `[1,2,3]`

`matrix[0][0]` returns `1` (the first object inside the first object)

Reversing:

`matrix[1].reverse()` returns `[[1, 2, 3], [6, 5, 4], [7, 8, 9]]`

Slicing with a list comprehension:

```
first_col = [row[0] for row in matrix]
```

`first_col` returns `[1,4,7]`

LIST COMPREHENSIONS

[**expression** for **item** in **iterable** (if **condition**)] – always return a list

allow you to perform for loops within one set of brackets

Longhand:

```
l = []
for letter in 'word':
    l.append(letter)
print(l)
['w', 'o', 'r', 'd']
```

As a comprehension:

```
l = [letter for letter in 'word']
print(l)
['w', 'o', 'r', 'd']
```

```
list_of_squares = [x**2 for x in range(6)]
```

result: `[0, 1, 4, 9, 16, 25]`

```
even_numbers = [num for num in range(7) if num%2==0]
```

result: `[0, 2, 4, 6]`

Convert Celsius to Fahrenheit:

```
celsius = [0,10,20.1,34.5]
fahrenheit = [(temp*(9/5)+32) for temp in celsius]
```

type `9/5.0` in Python 2!

result: `[32.0, 50.0, 68.18, 94.1]`

Nested list comprehensions:

```
fourth_power = [x**2 for x in [x**2 for x in range(6)]]
```

result: `[0, 1, 16, 81, 256, 625]`

WORKING WITH TUPLES:
Remember: Tuple elements cannot be modified once assigned (tuples are immutable)
`tuple1 = (1,2,3)`
`max(tuple1)` returns 3, `min(tuple1)` returns 1
Note: commas define tuples, not parentheses. `hank = 1,2` assigns the tuple `(1,2)` to hank

Built-in Tuple Methods: (there are only 2)

.count	T.count(value) -> integer -- return number of occurrences of value
.index	T.index(value, [start, [stop]]) -> integer -- return first index of value. Raises ValueError if the value is not present.

WORKING WITH DICTIONARIES:
`dict1 = {'Tom':4, 'Dick':7, 'Harry':23}`

To update a value: `dict1['Harry'] = 25`
To increase a value: `dict1['Harry'] += 100` (the pythonic way to add/subtract/etc. value)
To clear a dictionary: `dict1.clear()` (keeps the dictionary, but now it's empty of values)
To delete a dictionary: `del dict1`

`dict1.keys()` returns `['Dick', 'Tom', 'Harry']`
`dict1.values()` returns `[7,4,23]` NOTE: Dictionaries are unordered objects!
`dict1.items()` returns `[('Dick',7),('Tom',4),('Harry',23)]` a list of tuples!
To add one dictionary to another: `dict1.update(dict2)`
Nesting dictionaries: `dict3 = {'topkey':{'nestkey':{'subkey':'fred'}}}`
`dict3['topkey']['nestkey']['subkey'].upper()` returns `'FRED'`

Dictionary Comprehensions:
`{key:value for key,value in iterable}` used to create a dictionary
`{key:value for value,key in iterable}` used if x,y appear in y,x order in iterable

WORKING WITH SETS:
To declare an empty set: `set1=set()` (because set1={} creates an empty dictionary)

A list can be cast as a set to remove duplicates
`set([2,1,2,1,3,3,4])` returns `{1,2,3,4}` (items are put in order, though sets do not support indexing)

A string can be cast as a set to isolate every character (case matters!)
`set('Monday 3:00am')` returns `{' ', '0', '3', ':', 'M', 'a', 'd', 'm', 'n', 'o', 'y'}`

A dictionary may use sets to store values (example of mixed drinks and their ingredients)

Set Operators:
`a = {1,2,3}  b = {3,4,5}  c = {2,3}`

`1 in a`			returns `True` (set a contains a 1)
Intersection	`&`	`.intersection()`	`a&b` returns `{3}`
Union	`|`	`.union()`	`a|b` returns `{1,2,3,4,5}`
Difference	`-`	`.difference()`	`a-b` returns `{1,2}` items in a but not in b
Exclusive	`^`	`.symmetric_difference()`	`a^b` returns `{1,2,4,5}` items unique to each set
Subset	`<=`	`.issubset()`	`c<=a` returns `True`
Proper subset	`<`		`c<a` also returns `True` c has fewer items than a
Superset	`>=`	`.issuperset()`	`a>=a` returns `True`
Proper superset	`<`		`a>a` returns `False`

Built-in Set Methods:

S.add(x), S.remove(x), S.discard(x), S.pop(), S.clear()

(Not for frozen sets): adds an item, removes an item by value, removes an item if present, removes and returns an arbitrary item, removes all items. Note: .remove throws a KeyError if the value is not found. .discard doesn't.

RANGE is a *generator* in Python – it returns values stepwise (see the section on 'Generators')

```
range([start,] stop [,step])
range(10) outputs values from 0 up to but not including 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

range(1,10,2)          NOTE: In Python 2, range() creates list objects like those shown.
[1, 3, 5, 7, 9]        Use xrange() to invoke the generator (and save memory space) in for loops.
```

To create an actual list from a range:

```
list(range(6))
[0, 1, 2, 3, 4, 5]
```

CONDITIONAL STATEMENTS & LOOPS

If / Elif / Else statements:

```
>>> day = 'Tuesday'
>>> if day == 'Monday':
        print('Sunny')
    elif day == 'Tuesday':
        print('Cloudy')
    else:
        print('Rainy')
Cloudy
```

this step is called *initializing a variable*

this step runs only when the above step returns 'false'

this step runs if ALL above steps return 'false'

you can also nest `and` & `or` in the statement

For Loops (iterates through an entire set of data)

```
>>> for i in range(0,3):
        print(i)
returns: 0  1  2
```

here `i` is a name left up to the coder that can be used elsewhere in the code

Note that `range(a,b)` iterates from `(a)` to `(b-1)`

```
>>> for i in range(2,7,2):
        print(i)
returns: 2  4  6
```

```
>>> l = [1,2,3,4,5]
>>> list_sum = 0
>>> for num in l:
        list_sum += num
>>> print(list_sum)
15
```

note this "print" is outside the for loop

This is a clunky way to add items in a list!

While Loops

```
>>> counter = 7
>>> while counter < 10:
        print(counter)
        counter = counter + 1
returns: 7  8  9
```

BEWARE OF INFINITE LOOPS!

alternatively, `counter += 1`

Note: `while True:` is a way to run a loop forever until some criteria is satisfied.

Nested For Loops

Prime number generator:

```
>>> for i in range (2,30):
        j = 2
        counter = 0
        while j < i:
            if i % j == 0:
                counter = 1
                j = j + 1
            else:
                j = j + 1
        if counter == 0:
            print(str(i) + ' is a prime number')
        else:
            counter = 0
```

j is assigned as a "divisor variable"

– the "str" function converts integer i to a string

Loop Control Statements (Break, Continue & Pass)

```
counter = 0
while counter < 100:
    if counter == 4:
        break
    print(counter)
counter = counter + 1
```

returns 0 to 3 instead of 0 to 99

```
for i in 'Python':
    if i == 'h':
        continue
    print(i),
```

returns P y t o n (printed downward)

`Pass` is used to circumvent something you haven't written yet (like an unfinished "else")

Try and Except

<u>You can circumnavigate entire batches of code without crashing/invoking Python's error engine</u>

```
try:
   code…
except Exception:
   print('Uh oh!')
except:
   print('Uh oh!')
else:
finally:
```

if a particular exception occurs, do this

if any exception occurs, do this

if no exception occurs, then do this

this code runs regardless of exceptions

<u>To capture & log the exception:</u>

```
import logging
try:
    1/0
except Exception as e:
    logging.exception("My error message for log")
```

For more info: https://docs.python.org/2/library/logging.html

<u>Create your own Exception class:</u>

```
class MyException(Exception):
    pass
```

…and then, in the code:

```
if thing meets criteria:
    raise MyException(thing)
```

For info on particular exceptions: https://docs.python.org/3.4/library/exceptions.html

INPUT (formerly raw_input)

Asks for input from the user and converts whatever is typed into a string:

```
x = input('What would you like to input? ')
```

Python2: use `raw_input()`

```
What would you like to input?
```

If the user inputs the number 4, then x == '4'.

Use try/except and type() to handle exceptions.

UNPACKING

Tuple Unpacking

Straightforward printing of tuples in a list of tuples:

```
>>> l = [(2,4),(6,8),(10,12)]
>>> for tup in l:
        print(tup)
(2, 4)
(6, 8)
(10, 12)
```

Pulling apart tuples (technique used when working with coordinate inputs)

```
>>> coor = (3,8)
>>> x,y = coor
>>> x
3
>>> y
8
>>> y,x
(8, 3)
```

type(x), type(y) return "int", type(coor) returns "tuple"

technically, tuples are defined by commas, not by parentheses

Unpacking the first item inside tuples in a list:

```
>>> for (t1,t2) in l:
        print(t1)
2
6
10
```

Perform arithmetic on items inside tuples:

```
>>> for (t1,t2) in l:
        print(t1+t2)
6
14
22
```

Dictionary Unpacking

.items() creates a generator to separate keys & values

```
>>> d = {'k1':1,'k2':2,'k3':3}
>>> for (k,v) in d.items():
        print(k)
        print(v)
k3
3
k2
2
k1
1
```

NOTE: use `.iteritems()` in Python 2!

```
>>> d = {'k1':1,'k2':2,'k3':3}
>>> for k,v in d.iteritems():
        print(k)
        print(v)
```

Remember, for k in d.items() returns a list of (key,value) tuples. use (k,v) to treat them separately.

FUNCTIONS DRY - "Don't Repeat Yourself"

def name (parameter): body

```
def funcName(myname):
    print ('Hello, %s' %myname)
funcName('Michael')
Hello, Michael
```

NOTE: the variable 'myname' is only used within the function – we have not initialized it as a global variable (it can't be called elsewhere)

```
def funcName(fname, lname):
    """
    This function returns a simple greeting.
    """
    print ('Hello, %s %s' %(fname, lname))
funcName('John', 'Smith')
Hello, John Smith
```

It's good practice to include a docstring

```
def Amtwtax(cost):
    return cost * 1.0625
print(Amtwtax(7))
7.4375
```

NOTE: In Python, you don't need to declare variable types. With "x + y", numbers are added, strings are concatenated.

Default Parameter Values

You can set a default value that is overridden if when another argument is presented to the function

```
def funcName(fname='John', lname='Smith'):
    print ('Hello, %s %s' %(fname, lname))
```

Will print whatever is passed to the function, otherwise prints "Hello, John Smith"

NOTE: Never use mutable objects as default parameter values! (don't use a list, dictionary, set, etc.)

Positional Arguments *args and **kwargs

Use *args to pass an open number of arguments to a function:

```
def func1(*args):
    print(sum(args))
func1(2,3,4,5)
14
```

*args builds a tuple of arguments named 'args'

Use **kwargs to pass *keyworded* arguments to a function:

```
def func2(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" %(key,value))
func2(happy='smile',sad='frown')
sad == frown
happy == smile
```

dictionary unpacking!

**kwargs builds a dictionary, which is unordered!

Note: only the asterisks * and ** are needed. "args" and "kwargs" are just conventions

For more info: http://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/

PRE-DEFINED FUNCTIONS
For a list of pre-defined functions, see https://docs.python.org/3.4/library/functions.html
DON'T USE EXISTING NAMES WHEN CREATING FUNCTIONS!
`len()` returns the number of items in an iterable (list, tuple, etc) or the number of characters in a string

`bool()` returns "True" if populated, "False" if zero or empty

`abs()` returns absolute value
`pow()` is an exponentiator. `pow(2,4)` returns `16`.
`hex()` and `bin()` convert numbers to hexadecimal and binary, respectively.
`hex(43)` returns `'0x2b'`, `bin(43)` returns `'0b101011'`
`round()` rounds to a specified number of digits (default=0). Always returns a float. 5 always rounds up.

`dir()` – returns every applicable function for that object
`>>> dir([])` returns all functions that can be used with lists & arrays
`help()` – returns help on the application of a specific function against a specific object
`>>> help(list1.count)` tells you that *count* returns the number of occurrences of value
(where *list1* is a variable previously set up in our program)

NOTE: several pre-defined functions exploit *lambda expressions*, described below.

LAMBDA EXPRESSIONS – used for writing ad hoc functions, without the overhead of def

- lambda's body is a single one-line expression (not a block of statements)
- lambda is for coding simple functions (def handles the larger tasks)

Converting def to lambda:

DEF	LAMBDA

`def square(num):`
`  return num**2`
`square(num)`
	`lambda num: num**2`

With def, you have to assign a name to the function, and call it explicitly.
Lambda expressions can be assigned a name (eg. square = lambda num: num**2), but usually they're just embedded.
Lambdas work well inside of 3 main functions: map(), filter() and reduce()

Example: a lambda expression to check if a number is even
```
num = 9
lambda num: num%2==0
```
Note: lambdas need to *return* something (here it returns False)

For further reading: Iterating with Python Lambdas http://caisbalderas.com/blog/iterating-with-python-lambdas/
Describes use of the map() function with lambdas.

For further reading: Python Conquers the Universe
https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/
Does a nice job of explaining how expressions (that return something) differ from assignment statements (that don't)
Lambdas can include functions [including print()], list comprehensions, conditional expressions:
```
lambda x: 'big' if x > 100 else 'small'
```

MORE USEFUL FUNCTIONS

MAP

`map(function, sequence)` applies a function to all elements of the sequence, and returns a new sequence with the elements changed by function. NOTE: In Python 3, use `list(map(…` to see the output!

```
temp = [0, 22.5, 40, 100]
def fahrenheit(T):
    return (9.0/5)*T + 32)
map(fahrenheit, temp)
[32.0, 72.5, 104.0, 212.0]
```

9.0 insures a float return (not needed in python v3)
don't put parentheses after "fahrenheit" – you're calling the fahrenheit *function* object, not its output

```
map(lambda T: (9.0/5)*T+32, temp)
[32.0, 72.5, 104.0, 212.0]
```

use lambda in place of declared functions

```
a,b,c = [1,2,3],[4,5,6],[7,8,9]
map(lambda x,y,z: x+y+z, a,b,c)
[12, 15, 18]
```

function, sequence. map() returns a[0]+b[0]+c[0], a[1]+b[1]+c[1], etc.

REDUCE

`reduce(function, sequence)` continually applies a function to a sequence and returns a single value.

```
list1 = [47,11,42,13]
reduce(lambda x,y: x+y, list1)
113
```

the math: (47+11=58),(58+42=100),(100+13=113)

```
reduce(lambda a,b: a if (a>b) else b, list1)
```

works like max(list1)

FILTER

`filter(function, sequence)` returns only those elements for which a function returns True.

```
list1 = range(10)
filter(lambda x: x%2==0, list1)
[0, 2, 4, 6, 8]
```

ZIP

`zip()` makes an iterator that aggregates elements from each of the iterables. It stops after the shortest input iterable is exhausted. With no arguments it returns an empty iterator. Zipping two dictionaries only pairs the keys.

```
x,y = [1,2,3],[4,5,6]
zip(x,y)
[(1, 4), (2, 5), (3, 6)]
```

ENUMERATE

`enumerate(sequence,[start=])` returns a tuple in the form (position, item).

```
list1 = ['a','p','p','l','e']
for x,y in enumerate(list1):
    print x,y
0 a
1 p
2 p
3 l
4 e
```

alternatively:

```
list(enumerate(list1))
[(0, 'a'), (1, 'p'), (2, 'p'), (3, 'l'), (4, 'e')]
list(enumerate(list1, start=2))
[(2, 'a'), (3, 'p'), (4, 'p'), (5, 'l'), (6, 'e')]
```

ALL & ANY

`all(iterable)` returns True if every element is true.

`any(iterable)` returns True if any element is true.

COMPLEX

`complex()` accepts either a string or a pair of numbes, returns a complex number

`complex(2,3)` returns `(2+3j)` `complex('4+5j')` returns `(4+5j)`

PYTHON THEORY & DEFINITIONS

Variable names are stored in a **namespace**

Variable names have a **scope** that determines their visibility to other parts of code

LEGB Rule:

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Name in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module: open,range,SyntaxError,...

```
x = 25
def printer():
    x = 50
    return x
print printer()
print x
50                      x inside the function is local (50)
25                      x outside the function is global, and is unchanged by the function (25)

x = 25
def printer():
    global x            this calls global x into the function!
    x = 50
    return x
print printer()
print x
50
50                      the function changed global x to 50
```

Use `globals()` and `locals()` to see current global & local variables

Return variable names only with `globals().keys()`

In place – "Strings are immutable; you can't change a string in place." (string[0]='n' doesn't work)

String (not mutable in place)	List (mutable in place)
`a='crackerjack'`	`b=['joe', 'ted']`
`a.replace('cr','sn')`	`b.reverse()`
`'snackerjack'`	`b`
`a`	`['ted', 'joe']`
`'crackerjack'`	

Note that in this example, a.replace('cr','sn') returned 'snackerjack' without prompting, but a is unchanged.

Sequenced – object elements have an established order (offset), and can be sliced

Iterable – object contains any series of elements that can be called one-at-a-time. Sets are iterable, but not sequenced.

`del` is a python **statement**, not a function or method. It's sort of the reverse of assignment (=): it detaches a name from a python object and can free up the object's memory if that name was the last reference to it.

Stack – using `.append()` to add items to the end of a list and `.pop()` to remove them from the same end creates a data structure known as a LIFO queue. using `.pop(0)` to remove items from the starting end is a FIFO queue. These types of queues are called *stacks*.

FUNCTIONS AS OBJECTS & ASSIGNING VARIABLES

If you define a function and then assign a variable name to that function (output is in blue):

```
def hello(name='Fred'):
    return 'Hello '+name
hello()
'Hello Fred'

greet = hello
greet
<function __main__.hello>
greet()
'Hello Fred'
```

Note that the assignment is NOT attached to the original function. If we delete hello, greet still works!
It seems that greet was set up as its own new function, with the hello function stored as one of its methods.

Returning functions inside of functions – consider the following:

```
def hello(name='Fred'):
    def greet():
        print('This is inside the greet() function')
    def welcome():
        print('This is inside the welcome() function')
    if name == 'Fred':
        return greet                    note no parentheses – return the function, not its output
    else:
        return welcome

x = hello()
x()
This is inside the greet() function
x
<function __main__.greet>              x is assigned to greet because name == 'Fred'

x = hello('notFred')                   pass any name except 'Fred'
x()
This is inside the welcome() function
x
<function __main__.welcome>            x is assigned to welcome because name != 'Fred'
```

When x was assigned to hello(), Python ran hello and followed it's instructions – it said "return this function's greet function to x". As soon as hello() finished, greet, welcome & name were cleared from memory!
x remains as a global variable, and (until it's reassigned) it still has a copy of the embedded greet function as its object.

x is unchanged even if we change `hello` and run `hello()` elsewhere in our program. The only way to change x is to run `x=hello('notFred')`, run `x=hello()` on a changed `hello`, or assign x to a new object entirely.

FUNCTIONS AS ARGUMENTS

```
def hello():
    return 'Hi Fred!'                  we carefully said "return" here, to return a string when called

def other(func):                       Here other is expecting an argument before it executes
    print('Other code would go here')
    print(func())                      print whatever the executed function returns
other(hello)                           Here the hello function was passed as an argument to other
Other code would go here
Hi Fred!
```

DECORATORS:
Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic". Useful when working with web frameworks like Django and Flask with python. Refer to the file "Python Sample Code" for an explanation of how decorators work. Decorator syntax:

```
def new_decorator(func):                    new_decorator is looking for a function as an argument
    def wrap_func():
        print('Code could be here, before executing the function')
        func()
        print('Code here will execute after the function')
    return wrap_func                        returns the decorator's function, not its output

@new_decorator                              the @ symbol invokes the decorator
def func_needs_decorator():
    print(' This function is in need of a Decorator')

func_needs_decorator()
Code could be here, before executing the function
 This function is in need of a Decorator
Code here will execute after the function
```

NOTE: Above code without wrap_func:

```
def new_decorator(func):
    print('Code could be here, before executing the function')
    func()
    print('Code here will execute after the function')

@new_decorator
def func_needs_decorator():
    print(' This function is in need of a Decorator')
Code could be here, before executing the function
 This function is in need of a Decorator
Code here will execute after the function    returns the output immediately!
```

Whenever a function is assigned to a variable, the function *runs*, and whatever it returns is passed to the variable.

For further reading: http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/

GENERATORS & ITERATORS

In Python 2, range() returns a list object, while xrange() is a generator, used to save memory space in for loops.
Generator functions send back a value, and then can pick up again where they left off.
When a generator function is compiled they become an object that supports an iteration protocol.
That means when they are called in your code they don't actually return a value and then exit, rather, the generator functions will automatically suspend and resume their execution and state around the last point of value generation.
The main advantage here is that of not computing an entire series of values up front; the generator functions can be suspended. This feature is known as *state suspension*.
Functions become generators by using `yield` in place of `return`.
Example: Generate a Fibonnaci sequence up to n

GENERATOR:

```
def genfibon(n):
    a = 1
    b = 1

    for i in range(n):
        yield a
        a,b = b,a+b


for num in genfibon(10):
    print num
```

returns: `1 1 2 3 5 8 13 21 34 55`

ITERATOR:

```
def fibon(n):
    a = 1
    b = 1
    output = []
    for i in range(n):
        output.append(a)
        a,b = b,a+b
    return output

fibon(10)
```

returns: `[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of *every single result*, when in our case we actually only care about the previous result to generate the next one!

NEXT & ITER built-in functions:

`next()` is used to walk through a generator:

```
def simple_gen():           | print next(g) returns 0
    for x in range(3):      | print next(g) returns 1
        yield x             | print next(g) returns 2
g = simple_gen()            | print next(g) returns a StopIteration exception
```

`iter()` turns an iterable into a iterator:

```
s = 'hello'
s_iter = iter(s)
```

`next(s_iter)` returns `'h'` then `'e'` then `'l'` and so on

GENERATOR COMPREHENSIONS

Used just like list comprehensions, except they don't retain values. Use parentheses.

```
my_list = [1, 3, 5, 9, 2, 6]
filtered_gen = (item for item in my_list if item > 3)
filtered_gen.next()
5
filtered_gen.next()
9
filtered_gen.next()
6
```

WORKING WITH FILES

`>>> testFile = open('test.txt')` the *testFile* object now contains the contents of our text file (and the test.txt file is now open in Python)

`>>> testFile.read()` returns the contents as a single string in quotes. \n appears in place of line breaks. NOTE: the pointer is now at the END of our text file. Repeating the read function returns nothing.

`>>> testFile.tell()` returns the current character position of the pointer in our file

`>>> testFile.seek(0,0)` repositions 0 bytes of data from our pointer to the 0 position (beginning) of the file
`>>> testFile.seek(0)` does the same thing.

`>>> testFile.close()` closes the test.txt file

READING AND APPENDING FILES

The open function takes 3 parameters: filename, access mode & buffer size

`>>> testFile = open('test.txt','w')`	allows writing to the file *(does it nuke the original file??)*
`>>> testFile = write('new text')`	REPLACES the contents of the file with the words 'new text'
`>>> testFile = open('test.txt','a+')`	allows appending to the file (a plus is needed to read the file)
`>>> testFile = write('\nnew text')`	ADDS the words 'new text' to the end of the existing file (with a line break)

RENAMING & COPYING FILES

`>>> testFile = open('test.txt')`	
`>>> import os`	imports the operating system module which allows us to rename files and close files(?)
`>>> os.rename('test.txt','test2.txt')`	test.txt has been renamed to text2.txt
`>>> testFile.close()`	
`>>> testFile = open('test2.txt')`	
`>>> newFile = open('test3.txt','w')`	creates a new file test3.txt and allows writing to the file (note: 'newFile' is just an arbitrary name for our variable)
`>>> newFile.write(testFile.read())`	

OBJECT ORIENTED PROGRAMMING – Classes, Attributes & Methods

Classes (object types) and methods:

- using the `class` keyword
- creating class attributes
- creating methods in a class
- learning about Inheritance
- learning about Special Methods for classes

Built-in types: int, float, str, list, tuple, dict, set, function
Instances: the number 1 is an instance of the int class (objects of a particular type)

Create a new object type:

```
class Sample(object):
    thing1 = value
    def __init__(self,thing2)
        self.thing2 = thing2
        self.thing3 = thing3
    def method1(self):
        return self.thing2
    def _method2(self):
        return self.thing3

x = Sample()
```

by convention, class names start with a capital letter
here we set a "Class Object Attribute" called "thing1"
here we initialize new attributes, and require "thing2"
here we assign the "thing2" attribute
a thing3 attribute is available, but not required.
here we define a new public method called "method1"
Note: we can't say "return thing2" because thing2 isn't an object, it's an attribute attached to an object
here we define a *private* method (note the single underscore)
Public methods are visible by hitting Tab. Private methods aren't.

here we "instantiate" the class by giving it an instance

An **attribute** is a characteristic of an object. A **method** is an operation we can perform on the object.

Methods: Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are an essential encapsulation concept of the OOP paradigm. This is essential in dividing responsibilities in programming, especially in large applications.
You can basically think of methods as functions acting on an Object that take the Object itself into account through its self argument.
Methods that start with a single underscore are *private* methods; they can't be seen with the Tab key.

Inheritance: Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called *derived* classes, the classes that we derive from are called *base* classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Special Methods (aka Magic Methods): Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax (double underscore). They allow us to use specific functions on objects in our class.
For more info: http://www.rafekettler.com/magicmethods.html

For Further Reading:

Jeff Knupp's Post
Tutorials Point
Mozilla's Post
Official Documentation

Example 1:

```
class Dog(object):
    species = 'mammal'          here we assign a Class Object Attribute (all instances share this attribute)
    def __init__(self,breed):   here we initialize an attribute "breed"
        self.breed = breed      this calls for the attribute "breed" anytime we create a "Dog" object
    def bark(self):             here we define a method ".bark"
        print "Woof!"

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')

sam.breed    | frank.breed      Note: there aren't any parentheses after ".breed" because
'Lab'        | 'Huskie'         it is an attribute and doesn't take any arguments

sam.species | frank.species     species is also an attribute (no parentheses) shared by all intances
'mammal'    | 'mammal'         program output appears in blue

sam.bark()
Woof!
```

Example 2:

```
class Circle(object):
    pi = 3.14
    # Circle gets initialized with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
```

Note: by setting radius=1 in the __init__, we don't require an argument when creating a Circle. x=Circle() creates a Circle object with radius 1.

```
    # Area method calculates the area. Note the use of self.
    def area(self):
        return self.radius * self.radius * Circle.pi
```

Note: above, we can't return just "radius" because radius isn't an object – it's an attribute of the "self" object.
Similarly, we can't return "pi" as it's a class object attribute of "Circle".
Further: we can't do "`Circle.radius`" as radius is an individual attribute. However, `self.pi` works.

```
    # Method for resetting radius
    def setRadius(self, radius):
        self.radius = radius
```

This turns the assignment statement `x.radius=2` into a method `x.setRadius(2)`

```
    # Method for getting radius (Same as just calling .radius)
    def getRadius(self):
        return self.radius

c = Circle()
```

Since radius was assigned a default=1, it's not a required argument here

```
c.setRadius(2)
print 'Radius is: ',c.getRadius()
print 'Area is: ',c.area()
Radius is:  2
Area is:  12.56
```

output

Example 3 - Inheritance:

```
class Animal(object):
    def __init__(self):
        print "Animal created"
    def whoAmI(self):
        print "Animal"
    def eat(self):
        print "Eating"

class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print "Dog created"
    def whoAmI(self):
        print "Dog"
    def bark(self):
        print "Woof!"

d = Dog()
Animal created
Dog created
d.whoAmI()
Dog
d.eat()
Eating
d.bark()
Woof!
```

anytime an Animal is created, print "Animal created" (note that we didn't need to initialize any attributes)

here we absorb all of Animal's attributes and methods

here we say "whenever a Dog is created..

...initialize whatever Animal would have initialized...

...and print "Dog created"

here we override the Animal `.whoAmI` method

here we introduce a new method unique to Dog

output

the `.eat` method was inherited from Animal

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class (as shown by the `eat()` method).
The derived class modifies existing behaviour of the base class (as shown by the `whoAmI()` method).
Finally, the derived class extends the functionality of the base class, by defining a new `bark()` method.

Example 4 – Special Methods:

```
class Book(object):
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s " \
        %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print "A book is destroyed"

book = Book("Steal This Book", "Abbie Hoffman", 352)

#Special Methods
print book          "print" works now because the __str__ method enabled it and we told it what to return
print len(book)     note that just len(book) doesn't do anything visibly
del book            this deletes the book object, then prints something
A book is created
Title: Steal This Book, author: Abbie Hoffman, pages: 352
352
A book is destroyed
```

The `__init__()`, `__str__()`, `__len__()` and the `__del__()` methods:
These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

MODULES
A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py

To group many .py files put them in a folder. Any folder with an __init__.py is considered a module by python and you can call them a package
|-HelloModule
|_ __init__.py
|_ hellomodule.py
You can go about with the import statement on your module the usual way.
For more information: http://docs.python.org/2/tutorial/modules.html#packages

`import math` (or random, string, etc.)
In Jupyter, hit Tab after `math.` *to see a list of available functions for the module.*
To access a specific function: `math.sqrt(4)`

Alternatively, you can import specific functions from a module:
`from math import sqrt` (best practice!)
Then to access a function it's just: `sqrt(4)`

Many modules are included in Anaconda:
At the command prompt: `conda install flask` (flask is a web framework for creating websites with python)

For packages not distributed with Anaconda:
`pip` (pypi = Python Package Index) pie-pie, not pippy.

For example: To create powerpoints in python, FIRST google "python powerpoint module". First hit tells you to
`pip install python-pptx`

Resource: https://github.com/vinta/awesome-python - a curated list of awesome frameworks, libraries & software

COLLECTIONS Module:
The collections module is a built-in module that implements specialized container datatypes providing alternatives to Python's general purpose built-in containers. We've already gone over the basics: *dict, list, set,* and *tuple*.

Counter is a *dict* subclass which helps count hashable objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.
```
from collections import Counter
```

Counter() with lists:
```
l = [1,2,2,2,2,3,3,3,2,2,1,12,3,2,32,1,21,1,223,1]
Counter(l)
```
note the uppercase "C"
returns: `Counter({2: 7, 1: 5, 3: 4, 32: 1, 12: 1, 21: 1, 223: 1})`

Counter() with strings:
```
Counter('aabsbsbsbhshhbbsbs') note the uppercase "C"
```
returns: `Counter({'b': 7, 's': 6, 'h': 3, 'a': 2})`

`Counter(s.split())` counts words in a sentence (including punctuation!)

```
Counter('If you read you are well read'.split())
Counter({'read': 2, 'you': 2, 'well': 1, 'are': 1, 'If': 1})
```

Methods with Counter

```
c = Counter('abacab')
print c
Counter({'a': 3, 'b': 2, 'c': 1})
c.most_common(2)          note that methods act on objects. Counter('abacab').most_common(2) also works.
[('a', 3), ('b', 2)]      not sure how it resolves ties…
```

Common patterns when using the Counter() object:

```
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 (using string notation)
c += Counter()                  # remove zero and negative counts
```

defaultdict is a dictionary-like object which provides all methods provided by dictionary but takes a first argument (default_factory) as default data type for the dictionary. In other words, a defaultdict will never raise a KeyError.
Any key that does not exist gets the value returned by the default factory.
Using defaultdict is faster than doing the same using the dict.set_default method.

```
from collections import defaultdict
d = defaultdict(object)
d['key1']
<object at 0x405dfd0>       reserves an object spot in memory
d.keys()
['key1']
d.values()
[<object at 0x18e9ea10>]
type(d)
collections.defaultdict
```

Assigning objects to defaultdict: `d=defaultdict(0)` doesn't work (argument must be callable)

```
d = defaultdict(lambda: 0)  essentially, "for any argument return 0"
d['key2'] = 2               works as you'd expect
d
defaultdict(<function __main__.<lambda>>, {'key1': 0, 'key2': 2})
```

OrderedDict is a dictionary subclass that remembers the order in which its contents are added.
Any key that does not exist gets the value returned by the default factory.
Using defaultdict is faster than doing the same using the dict.set_default method.

```
from collections import OrderedDict
d = OrderedDict()
d['a'], d['b'], d['c'], d['d'] = 1,2,3,4
for k,v in d.items():
    print(k,v)
a 1   b 2   c 3   d 4       the order is retained
```

Note: two Ordered dictionaries that contain the same elements but in different order are no longer equal to each other.

namedtuple Standard tuples use numerical indexes to access their members: `t=(a,b,c)|t[0]` returns `'a'`
Named tuples assign names as well as a numerical index to each member of the tuple

```
from collections import namedtuple
Dog = namedtuple('Dog','age breed name')  arguments: name, attributes sep by spaces
sam = Dog(age=2,breed='Lab',name='Sammy')
sam
Dog(age=2,breed='Lab',name='Sammy')
sam.age     |  sam[0]
2           |  2        note that "age" is the name of the first member of the tuple, "2" is its value
```

You don't need to include names when creating individual namedtuples:

```
dave = Dog(3,'beagle','David')
dave
Dog(age=3, breed='beagle', name='David')
```

In Jupyter, if you hit Tab after `sam.` you see all the attributes associated with Dog (as well as `count` & `index`)
Each named tuple is like an ad hoc *class*.

DATETIME Module
Introduces a *time* class which has attributes such as hour (req'd), minute, second, microsecond, and timezone info.

```
import datetime
t = datetime.time(5,25,1)
print(t) returns 05:25:01, t.minute returns 25
t.min is 00:00:00, t.max is 23:59:59.999999, t.resolution is 0:00:00.000001
```

The *date* class:

```
today = datetime.date.today()
print(today) returns 2016-01-06
today.timetuple() returns time.struct_time(tm_year=2016, tm_mon=1, tm_mday=15,
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=4, tm_yday=15, tm_isdst=-1)
print datetime.date.resolution returns 1 day, 0:00:00
d2 = d1.replace(year=1990)  NOTE: d1.replace(…) returns a new value, but doesn't change d1.
```

Arithmetic:
`d1-d2` returns the time delta in days (date.resolution) although you can control this with additional code

Question: if `print datetime.date.min` returns `0001-01-01` how does datetime handle dates BCE?

TIMEIT Module
The timeit module has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times.

```
import timeit
timeit.timeit(CODE, number=10000)
```

returns 0.24759 (or similar) after running CODE 10,000 times

iPython's "built-in magic" %timeit function returns the best-of-three fastest times on one line of code:

```
%timeit "-".join(str(n) for n in range(100))  Works in Spyder!
10000 loops, best of 3: 23.8 µs per loop
```

Note that %timeit set the 10,000 loops limit. If code ran longer it would have adjusted downward to 1000 or 100.

PYTHON DEBUGGER – the pdb Module

The debugger module implements an interactive debugging environment for Python programs.
It allows you to pause programs, look at the values of variables, and watch program executions step-by-step.

```
import pdb
```

When you find a section of code causing an error, insert
`pdb.set_trace()` above it.
The program will execute up until `set_trace`, an then invoke the debugging environment:
`(Pdb)` ____

Here you can call variables to determine their values, try different operations on them, etc.
`(Pdb)` `continue` returns you to the program
`(Pdb)` `q` quits out of the program

For further reading: https://docs.python.org/3/library/pdb.html

REGULAR EXPRESSIONS – the re Module

Regular expressions are text matching patterns described with a formal syntax. You'll often hear regular expressions referred to as 'regex' or 'regexp' in conversation. Regular expressions can include a variety of rules, from finding repetition, to text-matching, and much more. As you advance in Python you'll see that a lot of your parsing problems can be solved with regular expressions (they're also a common interview question!).
If you're familiar with Perl, you'll notice that the syntax for regular expressions are very similar in Python. We will be using the **re** module with Python for this lecture. See https://docs.python.org/3/library/re.html

Searching for Patterns in Text

```
import re
# List of patterns to search for
patterns = ['term1', 'term2']     Use a list & for loop to conduct multiple searches at once
# Text to parse
text = 'This is a string with term1, but it does not have the other term.'
for pattern in patterns:
    print 'Searching for "%s" in: \n"%s"' %(pattern, text)
    #Check for match
    if re.search(pattern, text):
        print '\n'
        print 'Match was found. \n'
    else:
        print '\n'
        print 'No Match was found.\n'
Searching for "term1" in:
"This is a string with term1, but it does not have the other term."

Match was found.

Searching for "term2" in:
"This is a string with term1, but it does not have the other term."

No Match was found.
```

Note that re.search returns a *Match* object (or None). The Match object includes info about the start and end index of the pattern.
Note: `re.match` checks for a match only at the *beginning* of a string.

Finding all matches

Where `.search` found the first match, `.findall` returns a list of all matches.

```
re.findall('match','test phrase match is in middle')
['match']
```

Note: this is a list of ordinary text objects, *not* Match objects. Not very useful except you can count the result to determine how many matches there were.

Split with regular expressions

```
# Term to split on
split_term = '@'
phrase = "Bob's email address is: bob@gmail.com"
# Split the phrase
re.split(split_term,phrase)
["Bob's email address is: bob", 'gmail.com']
```

splits on *every occurrence* in the phrase

nice that it handled the single quote.

Using metacharacters

Repetition Syntax: there are five ways to express repetition in a pattern:

```
'sd*'      s followed by zero or more d's
'sd+'      s followed by one or more d's
'sd?'      s followed by zero or one d's
'sd{3}'    s followed by three d's
'sd{2,3}'  s followed by two to three d's
```

Character Sets: use brackets to match any one of a group of characters:

```
'[sd]'     either s or d
's[sd]+'   s followed by one or more s or d
```

NOTE: Matches don't overlap.

Exclusion: use ^ with characters in brackets to find all but those characters:

```
re.findall('[^!,.? ]+',phrase)
```

will strip all ! , . ? and spaces from a phrase including combinations (', ' is stripped) leaving a list of words

Character Ranges: use [start-end] to find occurrences of specific ranges of letters in the alphabet:

```
'[a-z]+'        sequences of lower case letters
'[A-Z]+'        sequences of upper case letters
'[a-zA-Z]+'     sequences of lower or upper case letters
'[A-Z][a-z]+'   one upper case letter followed by lower case letters
```

Escape Codes: use to find specific types of patterns:

```
\d      a digit
\D      a non-digit
\s      whitespace (tab, space, newline, etc.)
\S      non-whitespace
\w      alphanumeric
\W      non-alphanumeric
```

NOTE: both the bootcamp lecture and TutorialsPoint advise the use of *raw strings*, obtained by putting **r** ahead of a text string: `r'expression'`.

IO MODULE

The io module implements an in-memory file-like object. This object can then be used as input or output to most functions that would expect a standard file object.

```
import io

message = 'This is a normal string.'
f = io.StringIO(message)
```

use the StringIO method to set text as a file-like object

Now we have an object `f` that we can treat just like a file. For example:

```
f.read()
'This is a normal string.'
```

We can also write to it:

```
f.write(' Second line written to file-like object')
f.seek(0)
```

Resets the cursor just like you would a file

```
f.read()
'This is a normal string. Second line written to file-like object'
```

Use `io.BytesIO(data)` for data

This has various use cases, especially in web scraping where you want to read some string you scraped as a file.

For more info: https://docs.python.org/3.4/library/io.html

SEE ALSO: The cStringIO module provides a faster alternative.

NOTE: Python 2 had a StringIO module. The command above would be f=StringIO.StringIO(message). f becomes an *instance* type. For details see https://docs.python.org/2/library/stringio.html

STYLE AND READABILITY (PEP 8) See https://www.python.org/dev/peps/pep-0008/

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

```
foo = long_function_name(var_one, var_two,      it's ok to have arguments on the first line
                         var_three, var_four)   if aligned with opening delimeter

def long_function_name(                         do NOT put arguments on the first line
        var_one, var_two, var_three,            when using hanging indents, and use further
        var_four):                              indentation to distinguish from neighboring code
    print(var_one)
```

The closing brace/bracket/parenthesis on multi-line constructs may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
```

Limit lines to 79 characters

Surround top-level function and class definitions with two blank lines, method definitions inside a class by a single blank line.

Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+= , -= etc.), comparisons (== , < , > , != , <> , <= , >= , in , not in , is , is not), Booleans (and , or , not).

HOWEVER: If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

```
Yes:  x = y + z                  No:   x=y+z
      var1 += 1                        var1 +=1
      x = y*y + z*z                    x = y * y + z * z
      c = (a+b) * (a-b)                c = (a + b) * (a - b)
```

Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

```
Yes:  def complex(real, imag=0.0):     No: def complex(real, imag = 0.0):
          return magic(r=real, i=imag)         return magic(r = real, i = imag)
```

Use string methods instead of the string module.

Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone. For example:

```
Yes: if foo.startswith('bar'):
No:  if foo[:3] == 'bar':
```

Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None , and an explicit return statement should be present at the end of the function (if reachable).

```
Yes:  def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

```
No:   def foo(x):
        if x >= 0:
            return math.sqrt(x)

      def bar(x):
        if x < 0:
            return
        return math.sqrt(x)
```

Object type comparisons should always use `isinstance()` instead of comparing types directly.

```
Yes: if isinstance(obj, int):
No:  if type(obj) is type(1):
```

Refer to the PEP8 documentation for further info on naming & coding recommendations and conventions.
Check code at http://pep8online.com/

GOING DEEPER:

The '_' variable

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>>
```

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

To print on the same line:

```
print 'Hello'    print 'Hello',
print 'World'    print 'World'
Hello            Hello World
World
```

Python 2: use a comma to avoid the automatic line break (adds a space)

```
print('Hello', end='')
print('World')
HelloWorld
```

Python 3: replace the default \n with an empty string (does NOT add a space, unless you say end=' ')

`counter += 1` is equivalent to `counter = counter + 1`. This works for all operators (-=, *=, /= etc.)

the `divmod` function does // and % at once, returning a 2-item tuple: `divmod(9,5)` returns `(1,4)`

tuples that contain a single item still require a comma: `tuple1 = (item1,)`

you can convert lists to tuples and tuples to lists using `tuple()` and `list()` respectively

strings, lists and tuples are sequences - can be indexed [0,1,2…]. dictionaries are mappings, indexed by their keys.

You can assign the Boolean values "True", "False" or "None" to a variable.
"None" returns nothing if the variable is called on - it's used as a placeholder object.

you can combine *literal strings* (but not string variables) with `"abc""def"` (this is the same as `"abc"+"def"`)

Some more (& obscure) built-in string methods:

.strip	s.strip('.') removes '.' sequences from both ends of a string
.startswith	s.startswith(string) returns True or False, depending
.endswith	s.endswith(string) returns True or False, depending
.find	s.find(value,start,end) finds the index position of the first occurrence of a character/phrase in a range
.rfind	s.find(value,start,end) finds the index position of the last occurrence of a character/phrase in a range
.isalnum	s.isalnum() returns True if all characters are either letters or numbers (no punctuation)
.isalpha	s.isalpha() returns True if all characters are letters
.islower	s.islower() returns True if all cased characters are lowercase (may include punctuation)
.isupper	s.isupper() returns True as above, but uppercase
.isspace	s.isspace() returns True if all characters are whitespace
.istitle()	s.istitle() returns True if lowercase characters always follow uppercase, and uppercase follow uncased *Note that 'McDonald'.istitle() returns False*
.capitalize	s.capitalize() capitalizes the first word only
.title	s.title() capitalizes all the words
.swapcase	s.swapcase() changes uppercase to lower and vice versa
.center	s.center(30) returns a copy of the string centered in a 30 character field bounded by whitespace s.center(30,'z') does as above, but bounded by 'z's
.ljust	s.ljust(30) as above, but left-justified
.rjust	s.rjust(30) as above, but right-justified
.replace	s.replace('old', 'new', 10) replaces the first 10 occurrences of 'old' with 'new' (see *regular expressions*)
.expandtabs	'hello\thi'.expandtabs() returns 'hello hi' without requiring the print() function

Some more (& obscure) built-in set methods: (capital S used to distinguish these from string methods)

S.copy()	returns a copy of S
S1.difference(S2)	returns a set of all items in S1 that are not in S2 (but not the other way around). If S1<=S2, S1.difference(S2) returns an empty set.
S1.difference_update(S2)	removes any items from S1 that exist in S2
S1.intersection(S2)	returns items in common between the two sets.
S1.intersection_update(S2)	removes any items from S1 that *don't* exist in S2.
S1.isdisjoint(S2)	returns True if there are no items in common between the two sets
S1.issubset(S2)	returns True if every item in S1 exists in S2
S1.issuperset(S2)	returns True if every item in S2 exists in S1
S1.union(S2)	returns all items that exist in either set
S1.symmetric_difference(S2)	returns all items in either set not common to the other
S1.update(S2)	adds items from S2 not already in S1

Advantages of Python 3 over Python 2 (and other languages):

integers can be any size (not just 64-bit as Python 2 “long”s)

Unicode support (not just ASCII) allows text in any language

Need to figure out repr() vs ascii() handling of objects betwn v2 & v3

The ordering comparison operators (<, <=, >=, >) raise a TypeError exception when the operands don’t have a meaningful natural ordering. In Python 2, 'string'>100000 returned True because it was comparing the type names, not the items themselves, and 'str' is greater than 'int' alphabetically.

Map behaves differently in Python 3 – it returns an iterable object, not a list. To see the result, use list(map(…

Common Errors & Exceptions:

Error	Trigger
IndexError: list index out of range	tried to call the fifth item in a four-item list
ValueError	tried to remove a list value that wasn't present

List methods like `.sort` and `.reverse` permanently affect the objects they act on.
`list2=list1.reverse()` reverses list1, but doesn't assign anything to list2 (weird!).
`list1.sort(reverse=True)` is NOT the same as `list1.reverse()` (one sorts, the other doesn't)

`pow(x,y[,z])` accepts a third "mod" argument for efficiency cases. pow(2,4) = 16, pow(2,4,3)=1

LaTeX – the `.Latex` method provides a way for writing out mathematical equations

Python Debugger resources:

Read Steve Ferb's article "Debugging in Python"
Watch Eric Holscher's screencast "Using pdb, the Python Debugger"
Read Ayman Hourieh's article "Python Debugging Techniques"
Read the Python documentation for pdb — The Python Debugger
Read Chapter 9—When You Don't Even Know What to Log: Using Debuggers—of Karen Tracey's *Django 1.1 Testing and Debugging*.

For more practice:

Basic Practice:
http://codingbat.com/python
More Mathematical (and Harder) Practice:
https://projecteuler.net/archives
List of Practice Problems:
http://www.codeabbey.com/index/task_list
A SubReddit Devoted to Daily Practice Problems:
https://www.reddit.com/r/dailyprogrammer
A very tricky website with very few hints and tough problems (Not for beginners but still interesting)
http://www.pythonchallenge.com/

Rounding issues in Python

Python has a known issue when rounding float 5's (up/down seem arbitrary).
See http://stackoverflow.com/questions/24852052/how-to-deal-with-the-ending-5-in-a-decimal-fraction-when-round-it
PYTHON 2.7 & 3.5 gave the same output:

```
a = [1.25,1.35,1.45,1.55,1.65,1.75,1.85,1.95,2.05]
for i in a:
    print '%1.1f'%(i)
1.2
1.4
1.4
1.6
1.6
1.8
1.9
1.9
2.0
```

For better performance, use the decimal module.

Adding a username & password

From the Udemy course "Rock, Paper, Scissors – Python Tutorial" by Christopher Young

```
while True:
    username = input("Please enter your username: ")
    password = input("Please enter your password: ")
    searchfile = open("accounts.csv", "r")
    for line in searchfile:
        if username and password in line:
            print("Access Granted")
```

Picking a random rock/paper/scissors

```
import random
plays = ('rock', 'paper', 'scissors')
choice1 = random.choice(plays)
```

Referential Arrays

```
counters = [0]*8
```
creates a list of 8 references to the same 0 integer value

```
counters = [2] += 1
```
creates a new integer value 1, and cell 2 now points to it

```
counters.extend(extras)
```
adds pointers to the same list items that extras points to

Deep and shallow copies

```
new_list = first_list
```
the new list points to the first list.
Changes made to either one affect the other

```
new_list = copy(first_list)
```
creates a "shallow copy" – the new list points to the same objects as the first one, but independently of the first

```
import copy
new_list = copy.deepcopy(first_list)
```
creates a deep copy – it makes copies of the objects themselves so long as those elements were mutable

Dynamic Arrays

In Python you do not have to set a list length ahead of time.
A list instance often has greater capacity than current length
If elements keep getting appended, eventually this extra space runs out.

```
import sys
```
includes a "get size of" function that tells how many bytes python is holding in memory

```
n = 10
data = []
for i in range(n):
    a = len(data)
    b = sys.getsizeof(data)
    print('Length: {0:3d}, Size in bytes: {1:4d}'.format(a,b))
    data.append(n)
```

```
Length:   0, Size in bytes:   64
Length:   1, Size in bytes:   96
Length:   2, Size in bytes:   96
Length:   3, Size in bytes:   96
Length:   4, Size in bytes:   96
Length:   5, Size in bytes:  128
Length:   6, Size in bytes:  128
Length:   7, Size in bytes:  128
Length:   8, Size in bytes:  128
Length:   9, Size in bytes:  192
```

python sets aside a larger number of bytes than what it needs as items are being added

More ways to break out of loops (in "Goto" fashion)
From https://docs.python.org/2/faq/design.html#why-is-there-no-goto
You can use exceptions to provide a "structured goto" that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the "go" or "goto" constructs of C, Fortran, and other languages.
For example:

```
class label: pass  # declare a label
try:
     ...
     if condition: raise label()  # goto label
     ...
except label:  # where to goto
     pass
...
```

Bitwise Operators:

Code:	Meaning:	Result:
print 5 >> 4	# Right Shift	0
print 5 << 1	# Left Shift	10
print 8 & 5	# Bitwise AND	0
print 9 | 4	# Bitwise OR	13
print 12 ^ 42	# Bitwise XOR	38
print ~88	# Bitwise NOT	-89

The Exclusive Or (bitwise XOR) operator:

```
the ^ carat symbol
```

PYTHON "GOTCHA": Default Arguments
from https://developmentality.wordpress.com/2010/08/23/python-gotcha-default-arguments/)
Python permits default arguments to be passed to functions:

```
def defaulted(a, b='b', c='c'):
    print a,b,c
defaulted(1,2,3)
1 2 3
defaulted(1)
1 b c
```

Unfortunately, mutable default objects are shared between subsequent calls (they're only predefined once):

```
def f(a, L=[]):
    L.append(a)
    return L
print f(1)      Returns: [1]
print f(2)               [1, 2]
print f(3)               [1, 2, 3]
```

There are two solutions:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

```
def f(a, L=[]):
    L = L + [a]
    return L
```