Advanced Strings

Strings can do operations on themselves:

.lower(), .upper(),.capitalize()

```
>>> "funKY tOwn".capitalize()
'Funky town'
>>> "funky tOwn".lower()
'funky town'
```

.split([sep [,maxsplit]])

```
>>> "funKY tOwn".split()
['funKY', 'tOwn']
>>> "funKY tOwn".capitalize().split()
['Funky', 'town']
>>> [x.capitalize() for x in "funKY tOwn".split()]
['Funky', 'Town']
>>> "I want to take you to, funKY tOwn".split("u")
['I want to take yo', ' to, f', 'nKY tOwn']
>>> "I want to take you to, funKY tOwn".split("you")
['I want to take ', ' to, funKY tOwn']
```

.strip(), .join(), .replace()

```
>>> csv_string = 'Dog,Cat,Spam,Defenestrate,1, 3.1415   \n\t'
>>> csv_string.strip()
'Dog,Cat,Spam,Defenestrate,1, 3.1415'
>>> clean_list = [x.strip() for x in csv_string.split(",")]
>>> clean_list
['Dog', 'Cat', 'Spam', 'Defenestrate', '1', '3.1415']
```

`.join()` allows you to glue a list of strings together with a certain string

```
>>> print ",".join(clean_list)
'Dog,Cat,Spam,Defenestrate,1,3.1415'
>>> print "\t".join(clean_list)
Dog  Cat  Spam Defenestrate  1    3.1415
```

`.replace()` strings in strings

```
>>> csv_string = 'Dog,Cat,Spam,Defenestrate,1, 3.1415   \n\t'
>>> alt_csv = csv_string.strip().replace(' ','')
>>> alt_csv
'Dog,Cat,Spam,Defenestrate,1,3.1415'
>>> print csv_string.strip().replace(' ','').replace(',','\t')
Dog  Cat  Spam Defenestrate  1    3.1415

'Dog,Cat,Spam,Defenestrate,1,3.1415'
```

`.find()`

incredibly useful searching,
returning the index of the search

```
>>> s = 'My Funny Valentine'
>>> s.find("y")
1
>>> s.find("y",2)
7
>>> s[s.find("Funny"):]
'Funny Valentine'
>>> s.find("z")
-1
>>> ss = [s,"Argentine","American","Quarentine"]
>>> for thestring in ss:
      if thestring.find("tine") != -1:
         print "'" + str(thestring) + "' contains 'tine'."

'My Funny Valentine' contains 'tine'.
'Argentine' contains 'tine'.
'Quarentine' contains 'tine'.
>>>
```

`string` module

exposes useful variables and functions

```
>>> import string
>>> string.swapcase("fUNKY tOWN")
'Funky Town'
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

file: checkemail.py

```
import string
## let's only allow .com, .edu, and .org email domains
allowed_domains = ["com","edu","org"]
## let's nix all the possible bad characters
disallowed = string.punctuation.replace(".","")
while True:
    res = raw_input("Enter your full email address: ")
    res = res.strip()   # get rid of extra spaces from a key-happy user
    if res.count("@") != 1:
        print "missing @ sign or too many @ signs"
        continue
    username,domain = res.split("@")

    ## let's look at the domain
    if domain.find(".") == -1:
        print "invalid domain name"
        continue
    if domain.split(".")[-1] not in allowed_domains:
        ## does this end as it should?
        print "invalid top-level domain...must be in " + ",".join(allowed_domains)
        continue
    goodtogo = True
    for s in domain:
        if s in disallowed:
            print "invalid character " + s
            ## cannot use continue here because then we only continue the for loop, not the while loop
            goodtogo = False

    ## if we're here then we're good on domain. Make sure that
    for s in username:
        if s in disallowed:
            print "invalid character " + s
            goodtogo = False

    if goodtogo:
        print "valid email. Thank you."
        break
```

example: check email address

```
BootCamp> python checkemail.py
Enter your full email address: josh.python.org
missing @ sign or too many @ signs
Enter your full email address: josh@pythonorg
invalid domain name
Enter your full email address: joshrocks!@python,.org
invalid character ,
invalid character !
Enter your full email address: joshrocks@python.org
valid email. Thank you.
BootCamp>
```

String Formatting

casting using `str()` is very limited
Python gives access to C-like string formatting

usage: “%(format)” % (variable)

```
>>> print "My favorite integer is %i and my favorite float is %f,\n" \
    " which to three decimal places is %.3f and in exponential form is %e" \
     % (3,math.pi,math.pi,math.pi)
My favorite integer is 3 and my favorite float is 3.141593,
 which to three decimal places is 3.142 and in exponential form is 3.141593e+00
```

common formats:
f (float), i (integer), s (string), g (nicely formatting floats)

http://docs.python.org/release/2.7.2/library/stdtypes.html#string-formatting-operations

String Formatting

% escapes "%"

```
>>> print "I promise to give 100%% effort whenever asked of %s." % ("me")
I promise to give 100% effort whenever asked of me.
```

+ and zero-padding

```
>>> print "%f\n%+f\n%+f\n%010f\n%10s" % (math.pi,math.pi,-1.0*math.pi,math.pi,"pi")
3.141593
+3.141593
-3.141593
003.141593
        pi
```

File I/O (read/write)

`.open()` and `.close()` are builtin functions

```
>> file_stream = open(“mydata.dat”,”r”)
>> <type 'file'>
>> file_stream.close()
```

open modes: *“r” (read), “w” (write), “r+” (read + update), “rb” (read as a binary stream, ...)*

Writing data: `.write()` or `.writelines()`

```
>>> f= open("test.dat","w")
>>> f.write("This is my first file I/O. Zing!")
>>> f.close()
>>> import os ; os.system("cat %s" % "test.dat")
This is my first file I/O. Zing!0
```

```
>>> f= open("test.dat","w")
>>> f.writelines(["This is my first file I/O.\n","Take that Dr. Zing!\n"])
>>> f.close() ; os.system("cat %s" % "test.dat")
This is my first file I/O.
Take that Dr. Zing!
0
```

Likewise, there is `.readlines()` and `.read()`

```
>>> f= open("test.dat","r")
>>> data = f.readlines()
>>> f.close() ; print data
This is my first file I/O.
Take that Dr. Zing!
>>>
```

file: tabbify_my_csv.py

```
"""
small copy program that turns a csv file into a tabbed file
"""

import os

def tabbify(infilename,outfilename,ignore_comments=True,comment_chars="#;/"):
    """
INPUT: infilename
OUTPUT: creates a file called outfilename
    """
    if not os.path.exists(infilename):
        return  # do nothing if the file isn't there
    f = open(infilename,"r")
    o = open(outfilename,"w")
    inlines = f.readlines() ; f.close()
    outlines = []
    for l in inlines:
        if ignore_comments and (l[0] in comment_chars):
            outlines.append(l)
        else:
            outlines.append(l.replace(",","\t"))
  o.writelines(outlines) ; o.close()
```

```
BootCamp> cat google_share_price.csv
# Date,Open,High,Low,Close,Volume,Adj Close
2008-10-14,393.53,394.50,357.00,362.71,7784800,362.71
...
BootCamp> cat google_share_price.tab
# Date,Open,High,Low,Close,Volume,Adj Close
2008-10-14      393.53  394.50  357.00  362.71  7784800 362.71
....
```

Functions and Input/Output

Functions

Python can be both *procedural* (using functions) and *object oriented* (using classes)

[We do objects tomorrow, but much of the function stuff now will also be applicable.]

Functions looks like:

```
def function_name(arg1,arg2,...,kw1=v1,kw2=v2,kw3=v3...)
```

argX are *arguments*

required

(and sequence is important)

kwX are *keywords*

optional

(sequence unimportant; vals act like defaults)

Functions

<u>You can name a function anything you want as long as it:</u>

- contains only numbers, letters, underscore
- does not start with a number
- is not the same name as a *built-in* function (like print)

There is no difference between *functions* and *procedures:*

unlike, say in, IDL, in Python functions
that return nothing formally, still
return None

```
>>> def addnums(x,y):
        return x + y
>>> addnums(2,3)
5
>>> print addnums(0x1f,3.3)
34.3
>>> print addnums("a","b")    # oh no!
ab
>>> print addnums("cat",23232)
TypeError: cannot concatenate 'str' and 'int' objects
```

Unlike in C, we cannot declare what type of variables are required by the function.

```
>>> def addnums(x,y):
    if (not (isinstance(x,float) or isinstance(x,int) or isinstance(x,long))) or \
       (not (isinstance(y,float) or isinstance(y,int) or isinstance(y,long))):
        print "I cannot add these types (" + str(type(x)) + "," + str(type(y)) + ")"
        return
    return x + y
>>> print addnums(2,3.0)
5.0
>>> print addnums(1,"a")
I cannot add these types (<type 'int'>,<type 'str'>) together
None
>>>
```

scope

```
>>> addnums
<function addnums at 0x103767848>
>>> type(addnums)
<type 'function'>
>>> x = 2
>>> print addnums(5,6)
11
>>> print x
2
```

Python has it's own local variables list.
x is not modified globally

```
>>> def numop(x,y):
      x *= 3.14
      return x + y
>>> x = 1
>>> print numop(x,3)
6.14
>>> print x
1
```

scope

...unless you specify that it's a global variable

```
>>> def numop(x,y):
      x *= 3.14
      global a
      a += 1
      return x + y, a
>>> a = 1
>>> numop(1,1)
(4.1400000000000006, 2)
>>> numop(1,1)
(4.1400000000000006, 3)
```

Note: we can return whatever we want (dictionary, tuple, lists, strings, etc.). This is really awesome...

keywords

```
>>> def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
...     if greetings is not None:
...          print greetings
...     return (x + y)*multiplier
>>> numop1(1,1)
Thanks for your inquiry.
2.0
>>> numop1(1,1,multiplier=-0.5,greetings=None)
-1.0
```

keywords are a natural way to grow new functionality without "breaking" old code

*arg, **kwargs captures unspecified args and keywords

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print "-" * 40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ":", keywords[kw]
```

```
>>> cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

http://docs.python.org/tutorial/controlflow.html#keyword-arguments

Documentation: Just the Right thing to Do

and Python makes it dead simple

Docstring: the first unassigned string in a function (or class, method, program, etc.)

```
def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
""" numop1 -- this does a simple operation on two numbers.
     We expect x,y are numbers and return x + y times the multiplier
     multiplier is also a number (a float is preferred) and is optional.
    It defaults to 1.0.
     You can also specify a small greeting as a string. """
       if greetings is not None:
            print greetings
       return (x + y)*multiplier
>>>
```

...accessing documentation within the interpreter

```
>>> help(numop1)   # or numop1? in ipython
Help on function numop1:

numop1(x, y, multiplier=1.0, greetings='Thank you for your inquiry.')
    Purpose: does a simple operation on two numbers.

    Input: We expect x,y are numbers
           multiplier is also a number (a float is preferred) and is optional.
           It defaults to 1.0. You can also specify a small greeting as a string.

    Output: return x + y times the multiplier
```

Modules

Organized units (written as files) which contain functions, statements and other definitions

Any file ending in .py is treated as a module
(e.g., numop1.py, which names and defines a function numop1)

Modules: own global names/functions so you can name things whatever you want there and not conflict with the names in other modules

file: numfun1.py

```
"""
small demo of modules
"""
def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
""" numop1 -- this does a simple operation on two numbers.
     We expect x,y are numbers and return x + y times the multiplier
     multiplier is also a number (a float is preferred) and is optional.
    It defaults to 1.0.
     You can also specify a small greeting as a string.
        if greetings is not None:
            print greetings
        return (x + y)*multiplier
```

import ***module_name***

gives us access to that module's functions

```
>>> import numfun1
>>> numfun1.numop1(2,3,2,greetings=None)
10
>>> numop1(2,3,2,greetings=None)
NameError: name 'numop1' is not defined
>>>
```

file: numfun2.py

```
"""
small demo of modules
"""
print "numfun2 in the house"
x    = 2
s    = "spamm"

def numop1(x,y,multiplier=1.0,greetings="Thank you for your inquiry."):
    """
Purpose: does a simple operation on two numbers.

Input: We expect x,y are numbers
       multiplier is also a number (a float is preferred) and is optional.
       It defaults to 1.0. You can also specify a small greeting as a string.

Output: return x + y times the multiplier
    """
    if greetings is not None:
          print greetings
    return (x + y)*multiplier
```

} do some stuff and set some variables

```
>>> import numfun2
numfun2 in the house
>>> import numfun2          # numfun2 is already imported...do nothing
>>>
>>> print numfun2.x, numfun2.s
2, ‘spamm’
>>> s = "eggs" ; print s, numop2.s
‘eggs’, ‘spamm’
>>> numop2.s = s
>>> print s, numop2.s
‘eggs’, ‘eggs’
>>> exit()
```

bring some of module's functions into the current namespace:

```
from module_name import function_name
from module_name import variable
from module_name import variable, function_name1,
function_name2, ...
```

```
>>> from numfun2 import x, numop1
numfun2 in the house
>>> x == 2
True
>>> numop1(2,3,2,greetings=None)
5
>>> s
NameError: name 's' is not defined
>>> numfun2.x
NameError: name 'numfun2' is not defined
```

Renaming a function (or variable) for your namespace:

`from module_name import name as my_name`

```
>>> from numfun2 import s as my_fav_food
>>> from numfun2 import numop1 as wicked_awesome_adder
>>> print my_fav_food
'spamm'
>>> wicked_awesome_adder(2,3,1)
5
```

Kitchen-Sinking It

`from module_name import *`

```
>>> from numfun2 import *
>>> print numop1(x,3,1)
5
```

This is convenient in the interpreter, but considered bad coding style. It pollutes your namespace.

Built-In Modules

give access to the full range of what Python can do

<u>For example,</u>

sys *exposes interpreter stuff & interactions (like environment and file I/O)*

os *exposes platform-specific OS functions (like file statistics, directory services)*

math *basic mathematical functions & constants*

These are super battle tested and close to the optimal way for doing things within Python

```
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://www.python.org/doc/2.7.3/lib/module-sys.html

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.

    Dynamic objects:

    argv -- command line arguments; argv[0] is the script pathname if known
    path -- module search path; path[0] is the script directory, else ''
    modules -- dictionary of loaded modules

    displayhook -- called to show results in an interactive session
    excepthook -- called to handle any uncaught exception other than SystemExit
      To customize printing in an interactive session or to install a custom
      top-level exception handler, assign other functions to replace these.
```

file: getinfo.py

```
import os
import sys

def getinfo(path="."):
    """
Purpose: make simple use of os and sys modules
Input: path (default = "."), the directory you want to list
    """
    print "You are using Python version ",
    print sys.version
    print "-" * 40
    print "Files in the directory " + str(os.path.abspath(path)) + ":"
    for f in os.listdir(path): print f
```

os.listdir() - return a dictionary of all the file names in the specified directory

sys.version() - string representation of the Python (and gcc) version

os.path.abspath() - translation of given pathname to the absolute path (operating system-specific)

Breakout Session
exploring some modules

remember: help()

A. create and edit a new file called **age.py**

B. within age.py, import the `datetime` module

- use `datetime.datetime()` to create a variable representing when you were born
- use `datetime.datetime.now()` to create a variable representing now
- subtract the two, forming a new variable, which will be a `datetime.timedelta()` object.

Print that variable.

1. how many days have you been alive? How many hours?

2. What will be the date in 1000 days from now?

C. create and edit a new file called **age1.py**

when run from the command line with 1 argument, age1.py should print out the date in days from now. If run with three arguments print the time in days since then

```
BootCamp> ./age1.py 1000
date in 1000 days 2014-10-09 07:40:49.682973
BootCamp> ./age1.py 1980 1 8
days since then... 11699
```