Python-2

- None value in Python
- Iterators and generators
  - Using them with file input
- Exceptions
- Modules (e.g., Python libraries)
- Using file I/O
- Walking a directory structure w. os module
- Using regular expressions (re module)
  - Forming regular expressions & re patterns
  - Doing replacement for regular expressions

None

- Special constant that is a null value
  - Not the same as False or 0 or empty string.
  - Comparisons to any other value than itself will return False
  - If used in a Boolean context, None is taken to mean False

```
(def is_it_true(any) :
    if any :
      print("yes, it's true")
    else :
      print ("no it's false)
is_it_true(None) # no, it's false
is_it_true(not None) #yes, it's true
```

Python's Clean Approach to Iterators

- Provide an **iterator** as a built-in operator on certain datatypes
  - Can build more than one iterator on same collection at same time
  - Does not break encapsulation
  - Does not guarantee an order for generating constituent parts of collection
  - Becomes part of the PL, rather than part of a user-defined class

Generators and Iterators

- Iterators can be implicitly created by the for..in construct
- Iterators also have the iter(), next() fcns

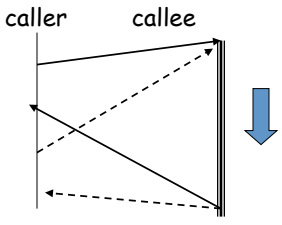
```
ss="abc"   #create string
it = iter(ss) #create iterator for this string
print (it)  #<str_iterator object at 0x11aa9e240>
print (next(it)) #a
print (next(it)) #b
print (next(it)) #c
print (next(it)) #exception raised (no more elements) - StopIteration
```

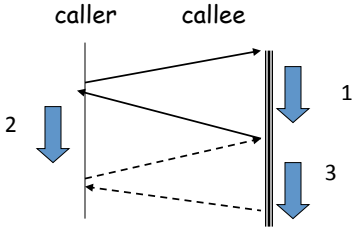
Generators & Iterators

```
def genlines (file):
  n=0
  for line in file:
        n=n+1
        yield (n, line) #yield keyword indicates a generator
# using iterators to look at lines in a file
input = open('/Users/ryder/Desktop/CS5314/intemp', 'r')
lineIter = genlines(input)
for y in lineIter: #no exception occurs because iterators
  print (y)        #stop in response to StopIteration exception!
#(1, 'This is input for my file.py program\n')#\n is newline
#(2, 'There is not much else to say about it \n')
#(3, 'Barbara Ryder\n')
#(4 '\n')
```


Coroutines: Behavior of Python Generators

- Method call from caller:
  - Caller stops, callee executes to completion, caller resumes
- Method call with a coroutine:
  - Caller stops, callee executes to yield, caller resumes to yield, callee executes to yield, etc.





Exceptions

```
def divide (x,y):
  try:
    ans = x/y
  except ZeroDivisionError:
    print "division by Zero"
  else:
    print "result is", ans
  finally:
    print "executing finally"

divide(4,2)
divide(4,0)
```

```
#result of divide(4,2)
#result is 2.0 (1st print)
#executing finally (3rd print)
#result of divide(4,0)
#division by Zero (2nd print)
#executing finally
```


Files

- Datatype in Python
  - Open operation takes a filename which can be qualified by a directory (uses / on all operating systems)

```
Afile = open("/Users/ryder/Desktop/CS5314/intemp.py", encoding="utf-8", mode="r")
print (Afile.name, Afile.encoding, Afile.mode)
# /Users/ryder/Desktop/CS5314/intemp.py utf-8 r
```

  - Close operation is sometimes done automatically by Python

Python is an OOPL

```
#can see how to define an iterator in a class
#that affects datatypes which admit sequencing
class Reverse:
    #iterator for looping over a sequence backwards,
    #works on all indexed datatypes
    def __init__(self, data):  #constructor for the iterator
        self.data = data
        self.index = len(data)
    def __iter__(self):      #returns the iterator object
        return self
    def __next__(self):   #returns next element until empty
        if self.index == 0:  #starts at last index and works
            raise StopIteration #forward
        self.index = self.index - 1
        return self.data[self.index]
```


Python is an OOPL

```
for x in Reverse((1,2,3)):  #for calls iterator implicitly
   print x           #ends on StopIteration exception
#3
#2
#1
for x in Reverse( ((4,1),(5,1),(6,1)) ):
   print x #prints 3 lines:
#(6,1)
#(5,1)
#(4,1)
z= ''
y = 'abcd'
for x in Reverse(y):
   z=z+x
print y,z
#prints abcd dcba
```

Modules

- A module is a file containing Python definitions and statements
  - Can be imported into other modules OR into main module (top-level execution)
- `Import` allows client to use a module, which is similar to a library of specialized functions
- Python program is a top-level file plus the modules it imports
  - Find the imported module (unqualified name)
  - Compile it
  - Run it

How to walk a directory and write output to a file?

```
import os
out= open("testLog", mode="w")#opens file to write
for root, dirs, files in os.walk("/Users/ryder/
  Dropbox/Desktop/CS5314/Assignments-Pgmg") :
    out.write ("root= ")
    out.write( root)
    out.write( "\n")
    for d in dirs:
      out.write(d)
      out.write("\n files = ")
      for f in files :
         out.write(f)
         out.write(" \n")
```

```
root= /Users/ryder/Dropbox/Desktop/CS5314/Assignments-
Pgmg
directory= prolog2016
files = .DS_Store dirWalk.ipy testLog testLog~
directory= Python2016
files = .DS_Store dirWalk.ipy testLog testLog~
directory= Scheme2016
files = .DS_Store dirWalk.ipy testLog testLog~
root= /Users/ryder/Dropbox/Desktop/CS5314/Assignments-
Pgmg/prolog2016
directory= NewestTest-case-data
files = .DS_Store command-and-sample-output.txt CS
5314 F14_ Prolog commenting.pdf ExplainDCG.txt
grammar-new Prolog-grading-rubric-change.txt Prolog-
grading-rubric-Rev.txt ReadMe021816.txt
staticTypeCheck.html Tokenizer-InstructionsRev
tokenizer_new021816.pl TopLevelPredicateRev warren-
paper-typos.html …
```


Actual top-level directory structure

```
drwxr-xr-x@  9 ryder  staff    306 Apr 16 15:53 .
drwxr-xr-x@ 16 ryder  staff    544 Apr 16 15:37 ..
-rw-r--r--@  1 ryder  staff  10244 Apr 15 17:53 .DS_Store
drwxr-xr-x@  6 ryder  staff    204 Apr 16 15:03 Python2016
drwxr-xr-x@ 27 ryder  staff    918 Apr 14 13:03 Scheme2016
-rw-r--r--@  1 ryder  staff    529 Apr 16 15:54 dirWalk.ipy
drwxr-xr-x@ 18 ryder  staff    612 Apr 14 13:03 prolog2016
-rw-rw-rw-@  1 ryder  staff   6266 Apr 16 15:54 testLog
-rw-rw-rw-@  1 ryder  staff   6147 Apr 16 15:54 testLog~
```

Regular Expressions in Python

- Standard pattern constructors
  - . Matches any char (except newline)
  - ^ Matches beginning of string
  - $ Matches end of string (just before \n)
  - * 0 or more repetitions of preceding RE
  - + 1 or more repetitions of preceding RE
  - ? Matches 0 or 1 of preceding RE
  - [ ] delimits a set of characters as possible matches
    - [a-zA-Z] matches any alphabetic symbol
    - [^a-z] matches any non-alphabetic symbol
  - | A|B where A,B are REs, means A or B
    - matches tried from left to right in or expression
  - \s matches any whitespace character
  - \d matches [0-9] https://docs.python.org/3/library/re.html

Ex1-simple substitution, streetConvert.py

Cf Dive into Python

Suppose you have a list of addresses and you want to convert the street address to use abbreviations for 'Avenue' (Ave.), 'Road' (Rd.), and Street (St.); how to do this with regular expressions in Python

```
import re  #regular expression module
f1=open("/Users/ryder/Dropbox/Desktop/CS5314/Assignments-Pgmg/Python2016/
   python-addrs")
f2=open("/Users/ryder/Dropbox/Desktop/CS5314/Assignments-Pgmg/Python2016/
   python-addrs")
for line in f1: #$ in pattern means end of string
  x = re.sub(r'ROAD$', 'Rd.',line)
  print (line, x)
for line in f2:
  x = re.sub('\\bROAD$', 'Rd.', line) #\b requires word to start
   at this point
  y = re.sub(r'\bROAD$', 'Rd.', line) #r is raw string mode, to
   prevent any escapes due to backslashes in string, '\t' is a tab
   char; r'\t' is a backslash followed by a t in the string.
  z = re.sub(r'\bROAD\b', 'Rd.', line) #new pattern does not
   require ROAD to be at end of string
    print (line,x,y,z)
```

Cf Dive into Python

Input:

```
100 BROAD ROAD
200 FALLS ROAD, Apt 10
3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE ROAD
```

Output (first loop)

```
100 BROAD ROAD
100 BROAD Rd  #pattern worked and didn't change ROAD in BROAD
200 FALLS ROAD, Apt 10
200 FALLS ROAD, Apt 10 #pattern failed
3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE ROAD
#pattern worked on last 'ROAD'
3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE Rd.
```


Output (second loop):

```
100 BROAD ROAD
 100 BROAD Rd.
 100 BROAD Rd.
 100 BROAD Rd.
200 FALLS ROAD, Apt 10
 200 FALLS ROAD, Apt 10
 200 FALLS ROAD, Apt 10
 200 FALLS Rd., Apt 10
3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE ROAD
 3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE Rd.
 3050 WHITE ROAD, 3051 FALLS STREET, 3052 WHITE Rd.
 3050 WHITE Rd., 3051 FALLS STREET, 3052 WHITE Rd.
```

Ex2- forming and using patterns and matches

```
import re
s = 'cabcd'
t = 'aabbaabb'
print (s,t)    #cabcd aabbaabb
patt = re.compile(r'^a+') #creates pattern
x = patt.search(s)    #searches for pattern in string s
if x != None:         #if no match, returns None
   print x.group()    #else print what matched pattern
print ((patt.search(t)).group()) #aa
patt2=re.compile(r'a+')
print ((patt2.search(s)).group()) #a
print ((patt2.search(t)).group()) #aa

for matchObj in patt2.finditer(t):
    print (matchObj.end()) #prints 2 and 6 showing that
  patt2 matched characters a position 2 (i.e, t[1]) and 6
  (i.e., t[5])
```


Ex2- forming and using patterns and matches

```
patt3=re.compile(r'(?P<x1>a+)|(?P<x2>b+)')
print (patt3.search(s)).group("x1")#a
print (patt3.search(s)).group("x2")#None

for matchObj in patt3.finditer(t):
    print matchObj.end() #prints 2,4,6,8
```