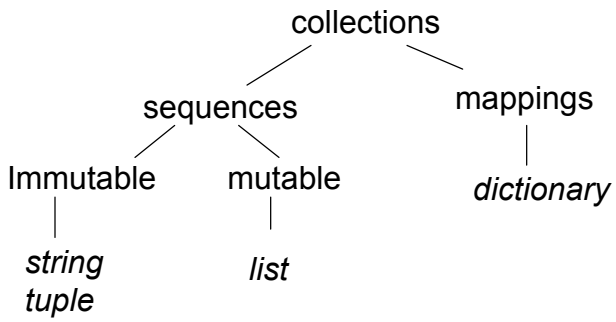
Python

- **Basic statements**
- **Functions**
- **Exceptions**
- **Iterators**
  - **How best to provide them?**
  - **How to use them?**
- **Generators - generalizing iterators**

Many examples from O'Reilly "Learning Python", 2nd Ed, 2003

Or online Python tutorial at python.org

Type Hierarchy



Basics

- **Assignments**

  X='spam' x,y='one','two' [a,b]=[1,2]

- **Variables**
  - _x2, a2, case-sensitive, can't clash with reserved words (e.g., lambda, for, not, or, if)
- **If statements**

```
if x=='Barbara':
   print 'found Barbara'
elif x=='Mary':
   print 'found Mary'
else:
   print 'missing key'
```

```
if z==1:
  a=2
  if y==2:
    print "y is 2"
  print "z is 1"
print 'done'
```


Basics

- **Boolean expressions evaluated by short-circuit**

  or, and, not, true, false

- **Looping constructs: while, for**

```
x= y/2
while x>1: #matches else below;shown by indenting
  if y%x==0:
    print y, 'has factor', x
    break  #skips the else
  x = x-1
else:    #is executed even if loop body isn't
 print y, 'is prime'
```

Basics

```
for x in ["spam", "eggs", "ham"]:
   print x #iterates over list elements
s = "lumberjack"
for y in s: print y #iterates over chars
```

- Comments delimited by # anywhere on a line
- Defining functions with `def` or `lambda`
  - *Python style: code a function to an object interface*
    - Idea is that a function should work on any datatype that supports the operations it needs
      - E.g., can use 'in' for any datatype admitting sequencing

Functions

- Two forms of function definition

```
def incr(x): return x+1 #function incr
#list of 2 functions
incrs = [lambda x: x+1, lambda x: x+2]
```

- Power of polymorphism

```
#what datatypes can this function be used on?
def intersect(seq1, seq2):
  res = [ ]
  for x in seq1:   #iterates over elements in seq1
    if x in seq2:  #checks if element is in seq2
      res.append(x) #if so, adds element to list
  return res

print intersect([1,2,3],[2,4,6])                    #[2]
print intersect( (1,2,3,4), (4,3,5))               #[3,4]
print intersect( (1,2,3), [1,2,3])  #[1, 2, 3]
print intersect({1:'a',2:'b',3:'c'}, {1:'a',4:'d'}) #[1]
print intersect({1:'a',2:'b',3:'c'}, {4:'a',5:'b'}) #[]
#clearly the intersection is on the keys, not the values!
```

Parameter Passing

- Pass by value
  - If pass a reference to a mutable object, then callee may change value seen in caller on return
  - Immutable objects used as arguments are not changed

```
def change(x,y):
  x = 2 #local change
  #shared obj change
  y[0]='spam'
```

```
x=1  #immutable
L=[1,2] #mutable
change(x,L)
print x,L
# (1,["spam"2])
```


Exceptions

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

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

```
#output
#result is 2
printed inside the try block
#executing finally
printed when finally is
executed
#None
value returned by divide call
```

```
#output
#Division by Zero
printed by exception handler
in divide
#executing finally
printed when finally is
executed
#None
value returned by divide call
```

Python is a Functional PL

- Apply(<fcn>,(list of args))
- Map(<fcn>, (list of args))
- Foldl(<fcn>,(list of args))
- List comprehension
  - Collect the results of applying an arbitrary expression to a sequence of values and returns them in a list

```
filter((lambda x: x%2 == 0), range(5))
#returns [0,2,4]
[x**2 for x in range(10) if x % 2 == 0]
#returns [0,4,16,36,64]
Res=[x+y for x in [0,1,2] for y in [100,200,300]]
#returns [100,200,300,101,201,301,102,202,302]
```


Avoid Common Beginner Errors

- Don't forget colons (:)
- Start in column 1 and indent consistently
- Use simple for loops instead of while
- Don't expect return values from functions that mutate objects; they return *None*
- Always use parentheses to call a fcn
- Don't use file extensions or diry paths in import stmts

Python Style - Use of Iterators

```
L=[1,2,4,8,16,32,64]
X = 5
I = 0
while I < len(L)
  if 2**X == L[I]:
    print "at index", I
    break
  I = I+1
else: print X, "not found"
```

```
L = [1,2,4,8,16,32,64]
X = 5
for p in L:
  if 2**X == p:
    print (2**X),
    'was found at',
    L.index(p)
    break
  else:
    print X, "not found"
```


Iterators

```
#first let's look at iterators as used in looping
#input:
   open('/Users/ryder/Rutgers/classes/314/314f07/python/intemp',
   'r');
n=0
for line in open('intemp'):
    n=n+1
    print n,"  ", line
#output
#1    this is input for my file.py program
#
#2    there is not much else to say about it
#
#3    Barbara Ryder
#
#4 1 barbaras-computer!python>
```

Iterators

```
#second example is how to create an iterator explicitly with
   'iter()'
s='abc'
it = iter(s)
print it            #<iterator object at 0x674b0>
print it.next()     #a
print it.next()     #b
print it.next()     #c
# print it.next() #causes an exception to occur
#Traceback (most recent call last):
#  File "iter.py", line 17, in ?
# print it.next() #this causes StopIteration exception to occur
#StopIteration
```


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
#(6, 1)
#(5, 1)
#(4, 1)
z= ''
y = 'abcd'
for x in Reverse(y):
   z=z+x
   print x  #prints 'd','c','b','a' one per line
print y,z #prints abcd dcba
```


Iterators in OOPL Design

- Q: How does one go through a collection of values (e.g., set, list, queue,...) if one may not need to have a look at all of them, and one needs to remain ignorant how the collection is stored?
  - Example task: there is a pile of cards, and player is looking through it top-down, for a card with a certain property; the pile is not affected by this.
- Need to consider how this is a part of the PL design and how we can preserve encapsulation
  - CLU (mid-1970s) supported this action as a built-in operation on collections
  - Store collection in a list and pass back access to representation
    - Problem: Breaks encapsulation

As an ADT method

- **Designer:** promises new Stack ADT functions

```
void startPumpStk(Stack s)
bool hasMore(Stack s)
eltType getNext(Stack s)
```

- **User:**

```
EltType v;

Stack sx;
startPumpStk(sx);
 ... //use sx
while ( hasMore(sx) ){
   v = getNext(sx);
   ... //process v here      }
```


As an ADT method

- **Implementer of ADT:**
  - Adds a field `current` to Stack structure

  (an `int` if array implementation or a `Cell*` if linked list implementation)
- Problem: can only have one iterator through a stack object at one time (i.e., no nested iteration)

Use Separate ADT for iterator

- **Designer:** creates new ADT type
  - `StackIterator`
  - `StackIterator(Stack s) //constructor`
  - `bool hasMore(StackIterator si)`
  - `EltType getNext(StackIterator si)`
- **User:**

```
EltType v_outer,v_inner;
Stack sx;
... //push stuff on sx
StackIterator *it1, *it2;
for (it1=new StackIterator(sx);
     it1->hasMore();
     v_outer = it1->getNext();)
 { for (it2=new StackIterator(sx);
        it2->hasMore();
        v_inner = it2->getNext();)
   {... //process v_inner and v_outer here}
 }
```


Use Separate ADT for Iterator

```
typedef struct{current, Stack s}* StackIterator;
```

`StackIterator createStackIter(Stack)`
allocates space for iterator and initializes `current` (e.g. sets it to top)

`bool hasMore(StackIterator)`
checks if there is more left in the collection (e.g. checks if `current >= 0`)

`EltType getNext(StackIterator)`
returns the current value and advances the cursor (e.g., `return s[current--];`)

Shown as C code

C++ Stack Iterator Specification

stack_iter.h

```
class Stack_iter {
//used to iterate over a stack from top down
 public:
  Stack_iter(Stack &goOver);
  ~Stack_iter();
  bool hasMore();
  EltType getNext();
}
```


C++ Stack Iterator

stack_iter.h

```
class Stack_iter{
  ...
 private:
   Stack& sk; //points to Stack
   int current;
```

stack.h

```
class Stack {
 private:
  eltType s[MAX];
  int top;
  static int Empty= -1;
  friend Stack_iter;
```

```
Stack_iter::Stack_iter(Stack &goOver)
   { sk = goOver; current = sk.top; }
bool Stack_iter::hasMore() {
    return (current!=sk.EMPTY);}
EltType Stack_iter::getNext()
   {if( current!=sk.EMPTY )
       return  sk.s[current--];
       else error("...");}
Stack_iter::~Stack_iter(){}
```

Using the Iterator

```
int main(){
     stack y;
     stack z(5);
     y.push(2);
     y.push(3);
     y.push(5);
     z.push(10);
     z.push(11);
     z.push(12);
     Stack_iter ity1(z);// constructor invoked implicitly
     stack_iter ity2(y);
     stack_iter ity4(y);

     while (ity4.hasMore())
          {cout << ity4.getNext() << " \n";};
     z.push(13);
     while (ity1.hasMore())
          {cout << ity1.getNext() << " \n";}
```

```
// sample run:> a.out
// 5 3 2
// 12 11 10
```
//why don't we see 13 in the output of iterator `ity1` *on* `z`*?*

Using the Iterator

```
while (ity2.hasMore()) {//nested iterators : no problem!
        stack_iter ity3(y);
        while (ity3.hasMore())
          {cout << ity3.getNext()<<" "<<
               ity2.getNext() <<"\n"
          }; }
// sample run:> a.out
// 5 3 2
// 12 11 10
// 5 2 3 2 2 2
// 5 3 3 3 2 3
// 5 5 3 5 2 5
```

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 & Iterators

```
def genlines (file):
  n=0
  for line in file:
        n=n+1
        yield (n, line)
# using iterators to look at lines in a file
input =
   open('/Users/ryder/Rutgers/classes/314/314f07/python/intemp'
   , 'r')
lineIter=genlines(input)
print lineIter  #<generator object at 0x68440>
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')
#(2, 'there is not much else to say about it \n')
#(3, 'Barbara Ryder\n')
```

Processing Strings in Files

```
import string
input3=open("/Users/ryder/Rutgers/classes/314/314f07/python/31
  4Info", 'r')
#only print lines corresponding to profs
iter=genlines(input3)
for y in iter:
    z=string.strip(y[1]) #this strips off the ending \n
    w=z.split(', ')#this splits the string on separator
    for x in w:
       if x=="professor":  #only print professor lines
           print w
#output
#['Barbara Ryder', '314', 'Core 311', 'professor']
#['Detlef Ronnenberger', '314', 'Hill 490', 'professor']
```

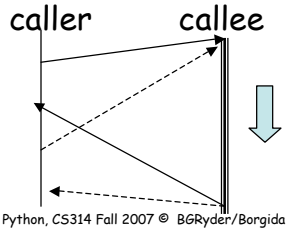

Define File Iterator as Function

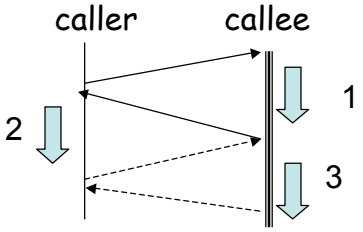
```
def filterFile(file,str):
    iter = genlines(file)
    for y in iter:
       z=string.strip(y[1]) #this strips off ending \n
       w=z.split(', ') #splits the string on separator
       for x in w:
         if x=="teaching assistant":
            print w

input4=open("/Users/ryder/Rutgers/classes/314/314f07/pyt
  hon/314Info", 'r')
filterFile(input4,'teaching assistant')
#output
#['Pavel Mahmud', '314', 'Hill 202', 'teaching assistant']
#['Peter Borosan', '314', 'Hill 355', 'teaching assistant']
#['Matt Casella', '314', 'Hill 412', 'teaching assistant']
```

Coroutines

- 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.





Modules

- `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