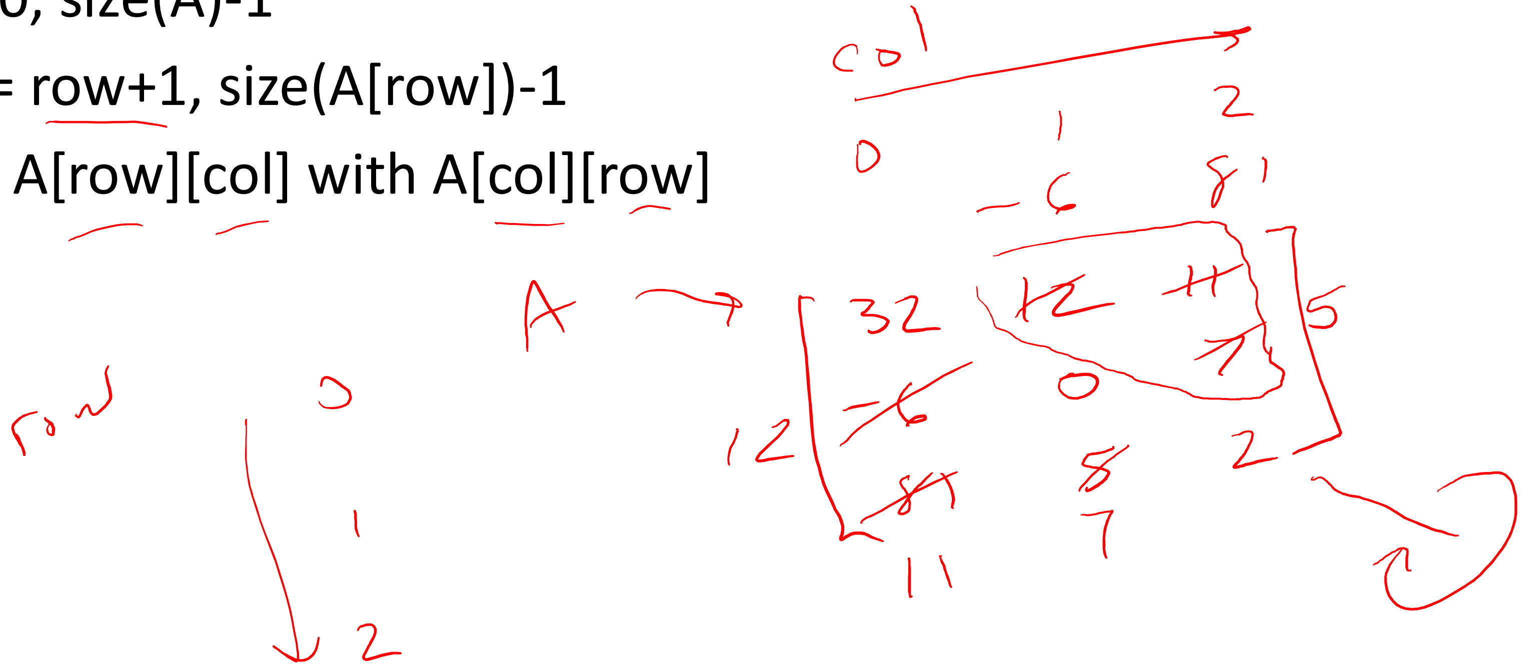
Objectives

- Be able to translate a pseudo-code function definition into Python
- Be able to construct a Python class definition
- Be able to create operator functions for a class, and to use the corresponding operators
- Be able to define the term *scope*, and to list some different kinds of scopes in Python
- Be able draw the tree form of an expression involving binary operators
- Be able to convert expressions between any two of the standard notations: infix, prefix, and postfix
- Be able to describe the public interface to the primary data structures: Stack, Queue, and Deque
- Be able to use Stacks and Queues in client code

Example: Translating an algorithm into Python

```
Algorithm transpose(A: a square matrix)
    for row = 0, size(A)-1
        for col = row+1, size(A[row])-1
            swap A[row][col] with A[col][row]
```



Local and non-local variables

- A variable is “declared” when it is first given a value, and it will in that case be considered local to the innermost scope which contains that “declaration”.
- Example:

```
def doLittle():
   x = 5
   def doLess():
       print(x) # Refers to nonlocal x
   return doLess()
doLittle() # Returns None, but causes console output.
```

Hiding of Variables

- This rule has some interesting consequences, since “hiding” of variables via overlapping scopes is permitted…

```
def doLittle():
   x = 5
   def doLess():
       print(x) # Illegal. x is referenced before it is
                # "declared."
       x = 3
   return doLess()
doLittle()
```

UnboundLocalError

The `nonlocal` declaration

- The error can be removed by declaring `x` to be nonlocal…

```
def doLittle():
   x = 5
   def doLess():
       nonlocal x
       print(x) # Now x legitimately refers to the variable
                # declared in the outer scope.
       x = 3
   return doLess()
doLittle()
```

The `global` declaration

- There is also a `global` declaration, with similar effects…

```
x = 10

def doLittle():
   x = 5
   def doLess():
       global x
       print(x)  # Now x refers to the outer declaration.
       x = 3
   return doLess()
doLittle()  # Prints 10
```

Defining Classes

```
class Point:
    def __init__(self, x=0, y=0): # The constructor.
        self.x = x
        self.y = y

    def norm(self):
        return (self.x**2 + self.y**2) ** 0.5

p = Point(3,4)
print(p.norm()) # Prints 5.0.
```

- Here `self` is not a keyword. Only its position in the parameter list identifies it as the object receiving the message!
- Lack of information hiding means “getters and setters” are alien to Python culture.

Scopes – block scope, function scope, class scope, and object scope

Operator Overloading

- Operators are overloaded as member functions, using predefined names.
- Examples:

`… = x[i]`	`x.__getitem__(i)`
`x[i] = value`	`x.__set_item__(i,value)`
`x + y`	`x.__add__(y)`
`x - y`	`x.__sub__(y)`
`x / y`	`x.__truediv__(y)`
`x // y`	`x.__floordiv__(y)`
`x ** y`	`x.__pow__(y)`
`print(x)`	`print(x.__str__())`

Some pre-Chapter 2 topics…

- For any expression involving operators, the *expression tree* for that expression is the tree whose root is the last operation done and whose children are the expression trees for its operands, in the right order

Postfix, Prefix, and Infix

- Translate to tree form first, then to the desired representation

Chapter 2 – Basic Data Structures

- Stack
- Queue
- Deque
- Unordered list
- Ordered list

Stack

- Stack()
- push(item)
- pop()
- peek()
- isEmpty()
- size()

When is a stack needed?

Queue

- Queue()
- enqueue(item)
- dequeue()
- isEmpty()
- size()

When is a queue needed?

Deque

- Deque()
- addFront(item)
- addRear(item)
- removeFront()
- removeRear()
- isEmpty()
- size()

Example Application

Evaluating a postfix expression (for example 89 77 33 + 5 * -)

```
Algorithm valueOf(pe) /* pe is a queue of operands and operators in which is stored the expression to be evaluated */
  Create a stack s
  While pe is not empty, do
      Dequeue item from pe
      if item is an operator
          pop two operands op2 and op1 from the stack s, in that order
          apply the operator to the operands, op1 being the left-hand operand
          push the resulting value on stack s
      if item is an operand, push it on stack s
  Pop the value on top of s into variable rv
  Return rv
```

The List Implementation of Stack