Topics for Today

- Examples
  - Classes
  - Stacks
  - Trees
- Problems
  - Expression evaluation
  - File system searching

1

Clicker Question

How do you create an instance of a Python class named "MyClass"?

A. `x = new MyClass()`
B. `x = MyClass()`
C. `MyClass.x(new)`
D. `x.MyClass = new`
E. `x.y`

2

Clicker Question

How do you call a method named "f" inside object "x"?

A. `f(x)`
B. `x.f()`
C. `f.x()`
D. `x(f)`
E. `All of the above`

3

Problem: Expression Evaluation

- Write a program that analyzes an expression and computes its value
- Simplifications
  - Only integers
  - Only four operators: +, -, *, /
  - Use space separators (so split works)
  - Use RPN to avoid operator precedence
- Step one: convert expression to "internal form"
- Step two: evaluate internal form

4

RPN

- Reverse Polish Notation
- Used in HP calculators for many years
- Enter operands (numbers) first, then operator
  - Intermediate results kept on a "stack" until needed
  - Operator "pops" numbers off stack; "pushes" on result
- Examples
  - "1 1 +" = 2
  - "3 2 – 2 3 * +" = 7

More coming on RPN in a few slides…

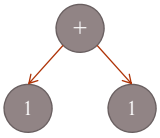
5

Expressions and "Trees"

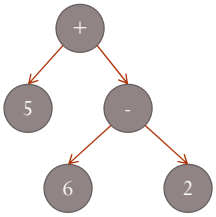
- We need a way to represent expressions
- An expression (specifically, a "binary expression") consists of three pieces:
  - an operator
  - a left operand
  - a right operand
- The left and right operands may themselves be expressions
- Thus: This is a *recursive* definition

6

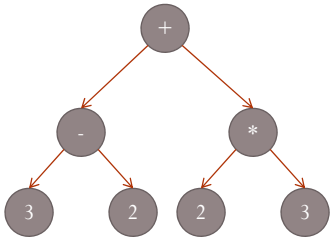
Expressions as (Upside Down) Trees



1 + 1



5 + (6 – 2)



(3 – 2) + (2 * 3)

7

Tree Representation in Python

- Use class
- Each instance represents a node in the tree
  - Base case: Integer value (a "leaf node": no child nodes)
  - Recursive case: Operator node, with child nodes for operands
- Instance variables store information about the node
  - Data value (integer or operator name, in circle)
  - Link to left node (if operator)
  - Link to right node (ditto)

8

Binary Tree Class (Minimalist)

```
class BinaryTreeNode:
    def __init__(self, data, left, right):
        self.data = data
        self.left = left
        self.right = right


v1 = BinaryTreeNode(1, None, None)
v2 = BinaryTreeNode(1, None, None)
v = BinaryTreeNode("+", v1, v2)
```

9

Binary Tree Class (Improved)

```
class BinaryTreeNode:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

v1 = BinaryTreeNode(1)
v2 = BinaryTreeNode(1)
v = BinaryTreeNode("+", v1, v2)
```

10

From RPN Expression to Tree

```
3 2 - 2 3 * +
```

- Follow the calculator process…
- Process each "token" (number or operator)
  - If number
    - create node with just number as data
    - push onto stack
  - If operator
    - pop operands off stack
    - create node with operator and two operands
    - push onto stack

11

Code Snippet

```
e = "3 2 - 2 3 * +"
stack = []
for token in e.split():
    if token.isdigit():
        node = BinaryTreeNode(token)
    else:
        right = stack.pop()
        left = stack.pop()
        node = BinaryTreeNode(token, left, right)
    stack.append(node)
root = stack[0]
```

12

Expression Parser

```
def parse(expression):
    stack = []
    for token in expression.split():
        if token.isdigit():
            node = BinaryTreeNode(token)
        else:
            right = stack.pop()
            left = stack.pop()
            node = BinaryTreeNode(token, left, right)
        stack.append(node)
    assert(len(stack) == 1)
    return stack[0]
```

13

Tree to String

- Basic (recursive) idea…
- Given a node, there are three reasonable possibilities:
  1. data + left child string + right child string
  2. left child string + data + right child string
  3. left child string + right child string + data
- For expressions, these correspond to:
  1. Prefix (or preorder): "+ 3 1"
  2. Infix (or inorder): "3 + 1"
  3. Postfix (or postorder): "3 1 +"

Python feature: object to string conversion via `__str__` *method.*

14

Implementation: String method

```
def __str__(self):
    if self.left == None:
        return self.data
    else:
        return "(%s) %s (%s)" %
            (str(self.left), self.data, str(self.right))
```

15

Evaluating the Tree

- Perform inorder (infix) traversal
  - Recursively evaluate left child
  - Recursively evaluate right child
  - Operate on two values and return
- Base case
  - A leaf node?
  - Just return data value

16

Implementation: Evaluate method

```
def evaluate(self):
  if self.data == "+":
      return self.left.evaluate() + self.right.evaluate()
  elif self.data == "-":
      return self.left.evaluate() - self.right.evaluate()
  elif self.data == "*":
      return self.left.evaluate() * self.right.evaluate()
  elif self.data == "/":
      return self.left.evaluate() / self.right.evaluate()
  else:
      return int(self.data)
```

17

More General Trees

- No need to limit tree nodes to two children
- General case: use a "list" of children
- Examples…
  - Family tree
    - parent node
    - child subnodes
  - File system
    - regular files are leaves
    - directories have child lists

18

Example: File System

```
class FileSystemNode:
    def __init__(self, path):
        self.path = path
        self.nodes = []

    def appendNode(self, node):
        self.nodes.append(node)
```

19

Read File System (simplified)

```
def readfs(path):
   node = FileSystemNode(path)
   if os.path.isdir(path):
      for f in os.listdir(path):
         node.appendNode(readfs(path + "\\" + f))
   return node
```

20

Walk Tree

```
def walktree(fs, pat=None):
    if pat:
        if fs.path.find(pat) >= 0:
            print fs.path
    else:
        print fs.path
    for f in fs.nodes:
        walktree(f, pat)
```

21