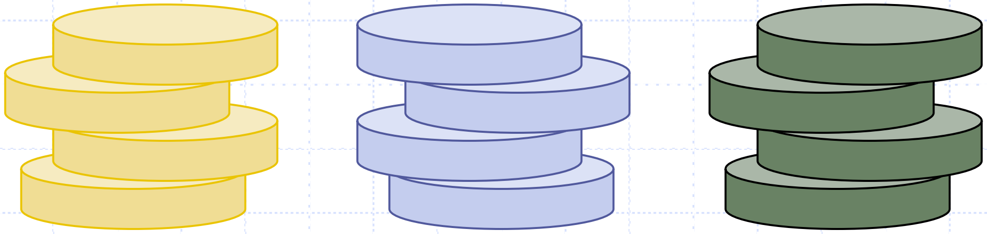
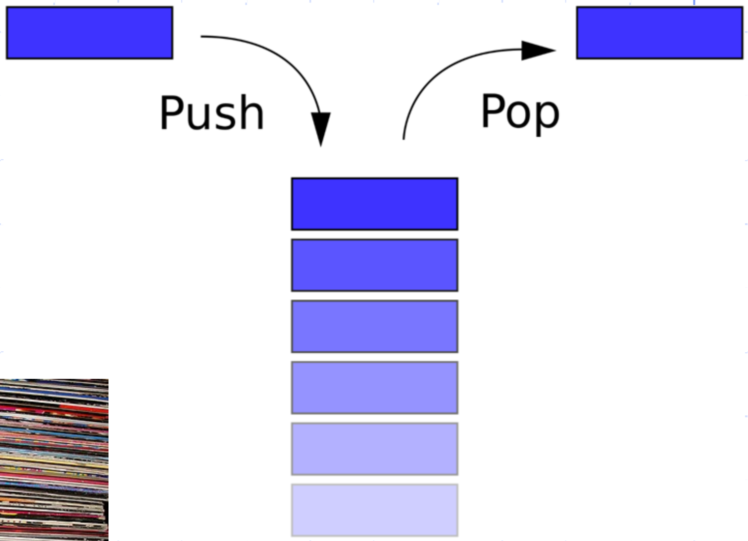
Stacks

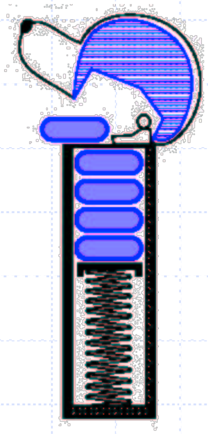


Le type abstrait `Pile` (Stack) est caractérisé par deux opérations : `push` (empiler) et `pop` (dépiler)



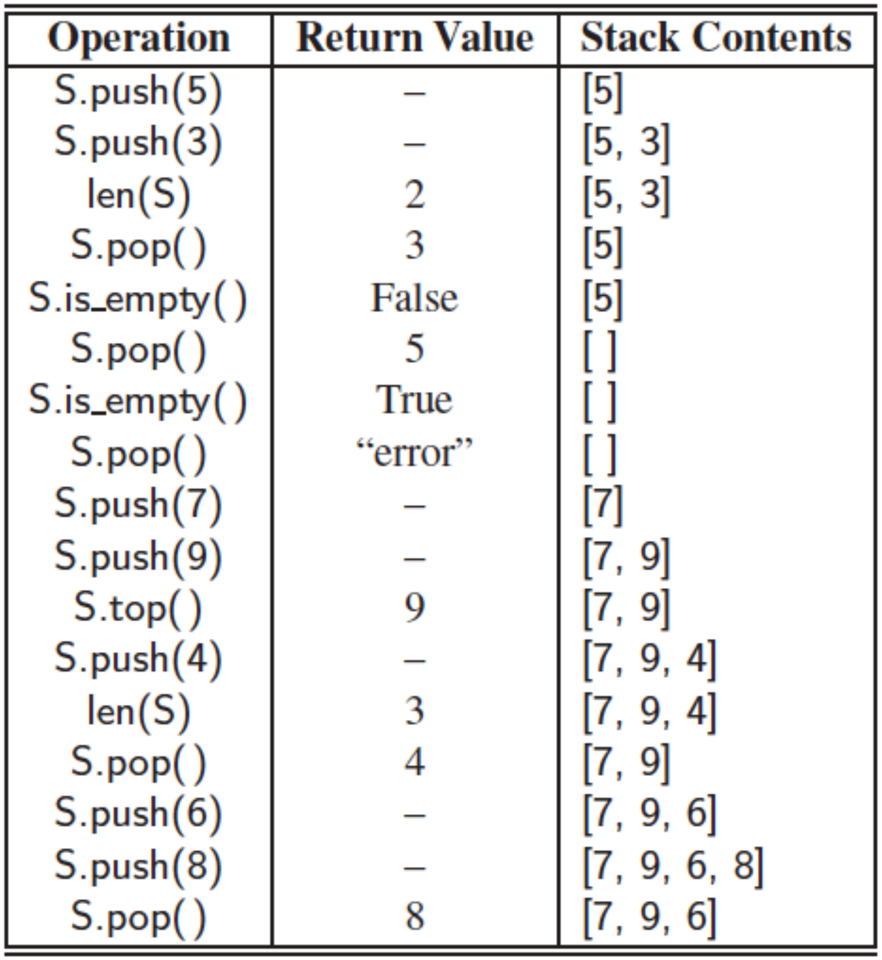
- ❑ La pile est caractérisée aussi par sa politique de dernier entré premier sorti (last-in-first-out; LIFO).

The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

Example



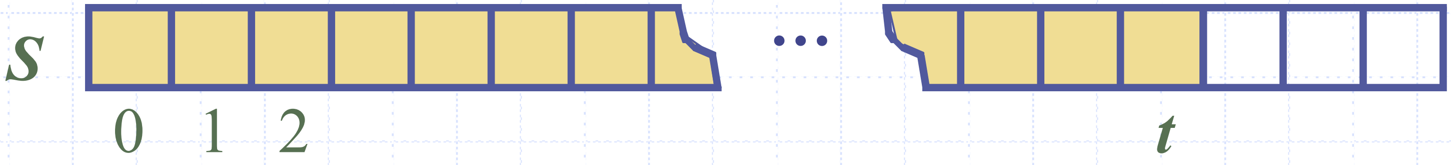
Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]
S.is_empty()	True	[ ]
S.pop()	“error”	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language that supports recursion
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

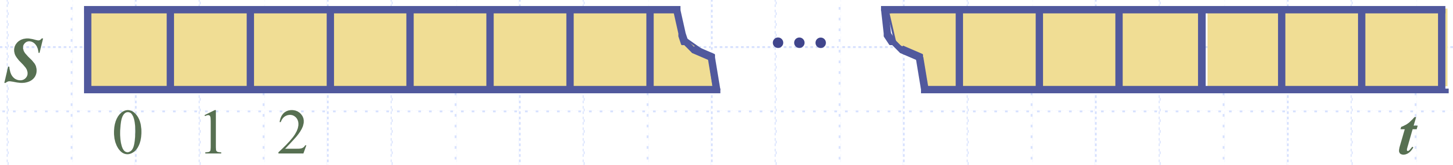
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$ (amortized in the case of a push)

Stack interface…

```
#ADT Stack "interface"
class Stack:

def __init__( self ):
    pass

#return the number of
elements in Stack
def __len__( self ):
    pass
```

Stack interface…

```
#convert a Stack into a string:
# elements listed between brackets
# separated by commas
# top element highlighted
# size and capacity of the data structure
# indicated when relevant
def __str__( self ):
    pass

#indicate whether no element are
#stored in the Stack
def is_empty( self ):
    pass
```

Stack interface

```
#add element on the Stack
def push( self, element ):
    pass

#remove an element from the Stack
def pop( self ):
    pass

#return the last inserted element
#without removing it
def top( self ):
    pass
```

ListStack…

```
class ListStack:

    #implements the ADT Stack (Stack.py)
    #uses the python default List
    def __init__( self ):
        self._A = []

    def __len__( self ):
        return len( self._A )

    def is_empty( self ):
        return len( self._A ) == 0

    def __str__( self ):
        pp = str( self._A )
        pp += "(size = “+str( len( self._A ) )+")[top = “+str( len( self._A ) - 1 )+"]"
        return pp
```

ListStack

```
#push obj
def push( self, obj ):
    self._A.append( obj )

#pop
def pop( self ):
    try:
        return self._A.pop( )
    except IndexError:
        return False

#top
def top( self ):
    try:
        idx = len( self._A ) - 1
        return self._A[idx]
    except IndexError:
        return False
```

ArrayStack...

```python
from DynamicArray import DynamicArray

class ArrayStack:

    #implements the ADT Stack (Stack.py)
    #uses the DynamicArray class (DynamicArray.py)
    def __init__( self ):
        self._A = DynamicArray()

    def __len__( self ):
        return len( self._A )

    def is_empty( self ):
        return len( self._A ) == 0
```

ArrayStack

```
def __str__( self ):
    pp = str( self._A )
    pp += "[top = " + str( len( self._A ) - 1 ) + ")"
    return pp

#push obj
def push( self, obj ):
    self._A.append( obj )

#pop
def pop( self ):
    idx = len( self._A )
    return self._A.remove( idx )

#top
def top( self ):
    try:
        idx = len( self._A ) - 1
        return self._A[idx]
    except IndexError:
        return False
```

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in *X* match

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
            S.push(X[i])
    else if X[i] is a closing grouping symbol then
            if S.is_empty() then
                        return false {nothing to match with}
            if S.pop() does not match the type of X[i] then
                        return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else return false {some symbols were never matched}
```

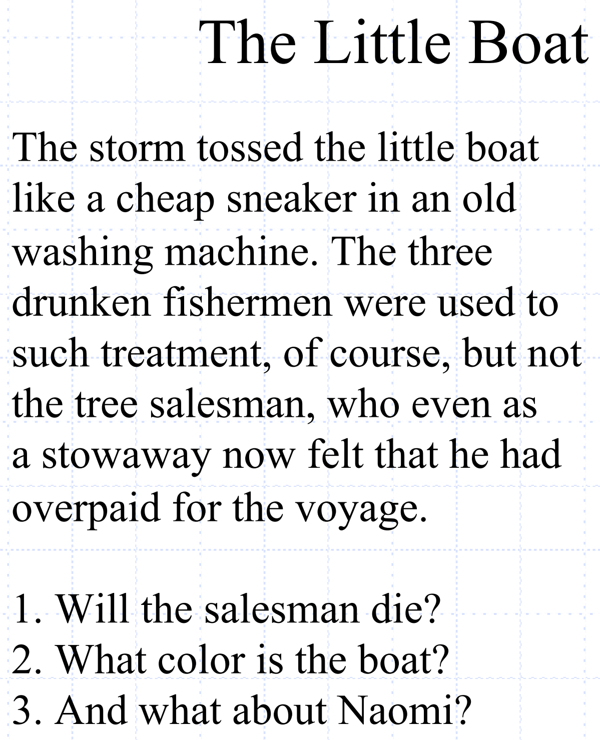
Parentheses Matching in Python

```
"""Fonction parenMatch
"""
def parenMatch( expr ):
    aGauche = "({["
    aDroite = ")}]"
    #on choisit l'implantation de la pile désirée
    S = ListStack()
    for c in expr:
        if c in aGauche:
            #si à symbole ouvrant, on empile
            S.push( c )
        elif c in aDroite:
            #si à symbole fermant...
            if S.is_empty():
                #si pile vide pas de match
                return False
            if aDroite.index( c ) != aGauche.index( S.pop() ):
                #si symbole fermant ne match pas symbole ouvrant
                return False
    #match si pile vide, sinon symbole(s) non matché(s)
    return S.is_empty()
```

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

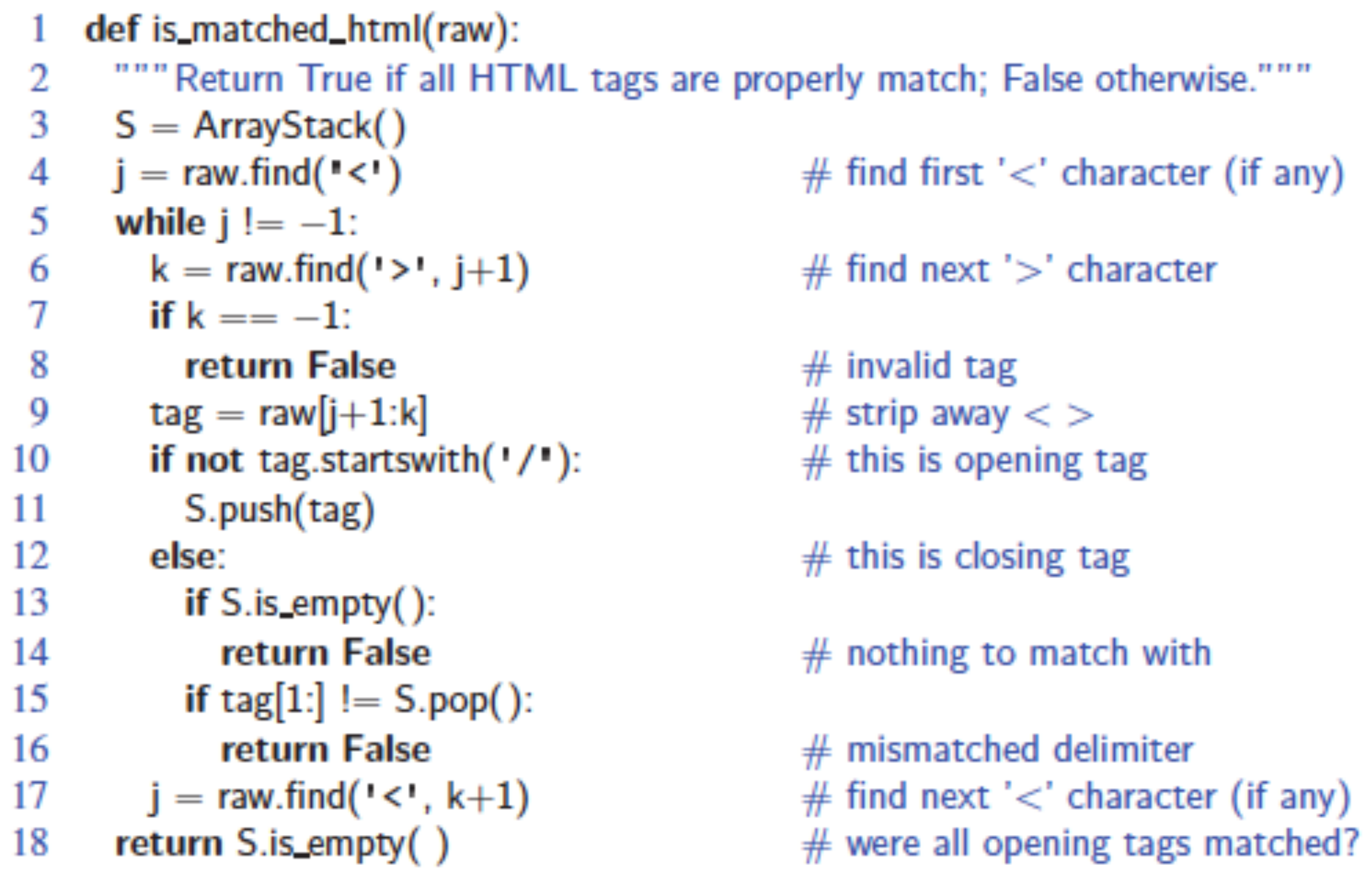


The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm in Python



```
def is_matched_html(raw):
  """Return True if all HTML tags are properly match; False otherwise."""
  S = ArrayStack()
  j = raw.find('<')                       # find first '<' character (if any)
  while j != −1:
    k = raw.find('>', j+1)                # find next '>' character
    if k == −1:
      return False                        # invalid tag
    tag = raw[j+1:k]                      # strip away < >
    if not tag.startswith('/'):           # this is opening tag
      S.push(tag)
    else:                                 # this is closing tag
      if S.is_empty():
        return False                      # nothing to match with
      if tag[1:] != S.pop():
        return False                      # mismatched delimiter
    j = raw.find('<', k+1)                # find next '<' character (if any)
  return S.is_empty()                     # were all opening tags matched?
```

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x = valStk.pop();
y = valStk.pop();
op = opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 and
        prec(refOp) ≤
        prec(opStk.top())
   doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

evalExpr…

```
"""Fonction principale"""
def main():
    # Lire en input une expression
    expr = input( 'Entrez une expression: ' )
    print( "L'expression ", expr, "=", evalExp( expr ) )
```

evalExpr...

```
def evalExp( expr ):
    valStk = ArrayStack()
    opStk = ListStack()
    for z in expr:
        if z.isdigit():
            valStk.push( z )
        elif z in "+-*/":
            repeatOps( z, valStk, opStk )
            opStk.push( z )
    repeatOps( '$', valStk, opStk )
    return valStk.top()
```

evalExpr…

```
def doOp( valStk, opStk ):
    x = valStk.pop()
    y = valStk.pop()
    op = opStk.pop()
    if op == '+':
        z = int(y) + int(x)
    elif op == '-':
        z = int(y) - int(x)
    elif op == '*':
        z = int(y) * int(x)
    elif op == "/":
        z = int(y) / int(x)
    valStk.push( z )
```

evalExpr…

```
def prec( op ):
    if op == '*/':
        return 2
    elif op in "+-":
        return 1
    #this is '$'
    return 0

def repeatOps( refOp, valStk, opStk ):
    while len( valStk )>1 and prec( refOp )<=prec( opStk.top() ):
        doOp( valStk, opStk )
```