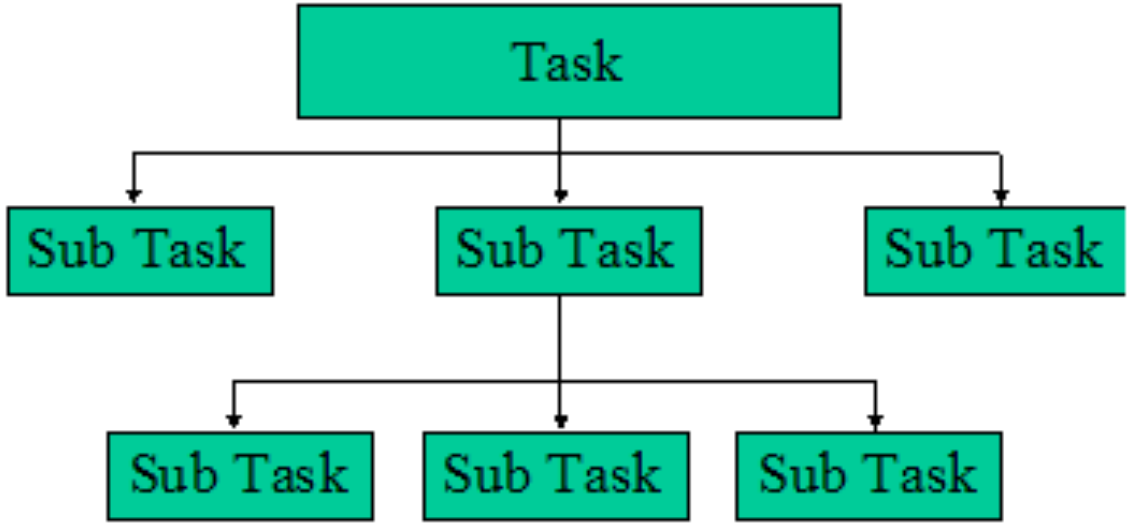
15-112
Fundamentals of Programming

Week 3 - Lecture 2:
More strings. Top-down design. Style.



January 28, 2016

Reminder1: Quiz tomorrow.

Reminder2: Homework posted.

Debugging tips:

- read error messages carefully
- find smallest/simplest input that leads to wrong output
- trace code with that input
- add print statements to your code to see variable values
- with strings, be careful about white spaces

"hello" ≠ "hello " ≠ "hello\n"

Today's Menu

String formatting

File I/O

Style

Top-down design

Example(s)

String formatting

team = “Steelers”
numSB = 6
s = “The ” + team + “ have won ” + numSB + “ Super Bowls.”

String formatting

```
team = "Steelers"
numSB = 6
s =  "The " + team + " have won " + str(numSB) + " Super Bowls."
```

```
team = "Steelers"
numSB = 6
s =  "The %s have won %d Super Bowls"  %  (team, numSB)
```

↓ string ↓ decimal

```
print(s)
```

The Steelers have won 6 Super Bowls

String formatting

```
print("Miley Cyrus gained %f pounds!" % 2**(-5))
```

↓

float

```
Miley Cyrus gained 0.03125 pounds!
```

```
print("Miley Cyrus gained %.2f pounds!" % 2**(-5))
```

```
Miley Cyrus gained 0.03 pounds!
```

```
print("Miley Cyrus gained %10.2f pounds!" % 2**(-5))
```

```
Miley Cyrus gained       0.03 pounds!
```

```
print("Miley Cyrus gained %-10.2f pounds!" % 2**(-5))
```

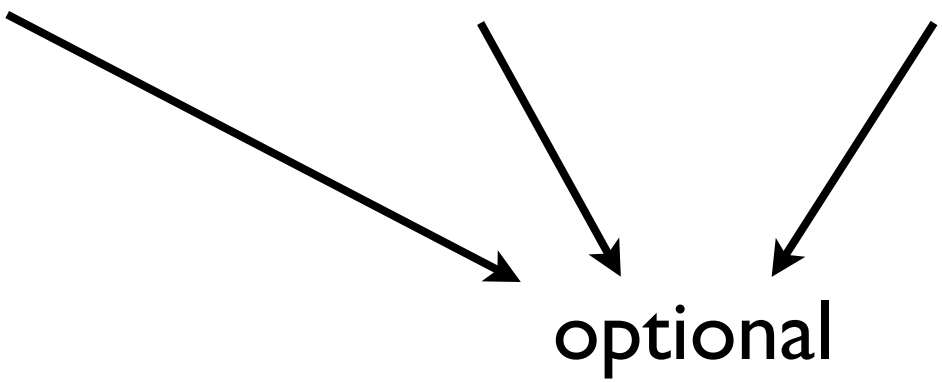
```
Miley Cyrus gained 0.03       pounds!
```

String formatting

print("Miley Cyrus gained %-10.2f pounds!" % 2**(-5))

Miley Cyrus gained 0.03 pounds!

```
% [-] [minWidth] [.precision] type
```



Today's Menu

String formatting

File I/O

Style

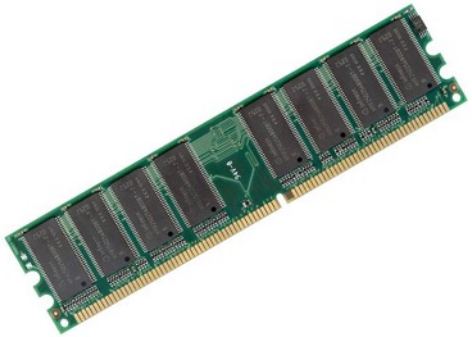
Top-down design

Example(s)

File I/O

- What happens when you run a program?

hard disk



RAM

- Should be able to interact with the files in hard disk
  > Read from a file. Write to a file.

File I/O

```
def readFile(path):
    with open(path, "rt") as f:
        return f.read()

def writeFile(path, contents):
    with open(path, "wt") as f:
        f.write(contents)

contentsToWrite = "This is a test!\nIt is only a test!"
writeFile("foo.txt", contentsToWrite)

contentsRead = readFile("foo.txt")
assert(contentsRead == contentsToWrite)
```

Today's Menu

String formatting

File I/O

Style

Top-down design

Example(s)

From lecture 1

What you will learn in this course:

1. How to think like a computer scientist.

2. Principals of good programming.

3. Programming language: Python

From lecture 1

2. Principals of good programming.

Is your code easy to read? easy to understand?

Can it be reused easily? extended easily?

Is it easy to fix errors (bugs)?

Are there redundancies in the code?

Summary

better style = better code

= a better world

Strong correlation between bad style and # bugs

Good style ---> saves money

Good style ---> saves lives

Style guides

- Official Python Style Guide

- Google Python Style Guide

- 15112 Style Guide

15112 Style Rubric

Comments

Concise, clear, informative comments when needed.

15112 Style Rubric

Comments

Ownership Good

```
# Name: Anil Ada
# Andrew id: aada
# Section: AA
```

15112 Style Rubric

Comments

Before functions (if not obvious) Good

```
# This function returns the answer to the ultimate question of life,
# the universe, and everything.
def foo():
    return 42
```

15112 Style Rubric

Comments

Before a logically connected block of code

Good

```
def foo():
    …
    …
    # Compute the distance between Earth and its moon.
    …
    …
```

15112 Style Rubric

Comments

Bad

```
x = 1      # Assign 1 to x
```

15112 Style Rubric

Comments

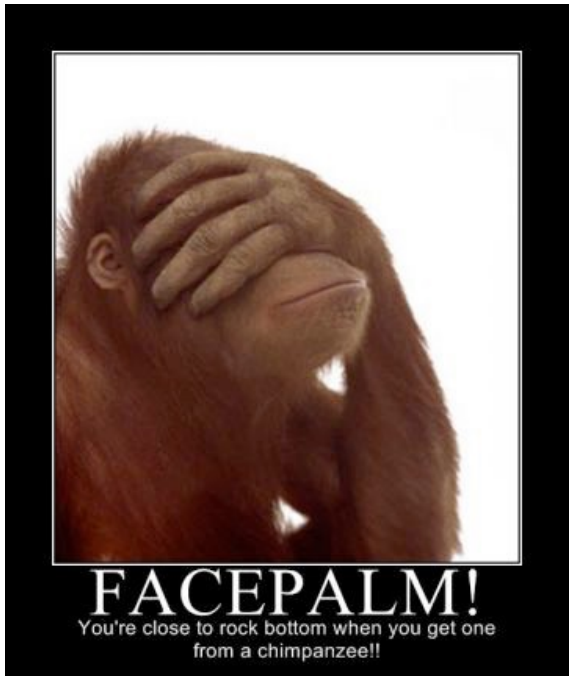
Very Bad

```
x = 1    # Assign 10 to x
```

15112 Style Rubric

Comments

```
# This method takes as input a thing that represents the
# thing that measures how long it takes to go from
# the center of a round circle to the outer edge of it. I
# learned in elementary school that.........
# The number PI does not really have anything
# to do with apple pie, although I kind of wish it did
# because it's really delicious. My grandma makes great pies.
```



15112 Style Rubric

Helper functions

Use helper functions liberally!

No function can contain more than 20 lines.
(25 lines for functions using graphics)

15112 Style Rubric

Test functions

Each function should have a corresponding test function.

exceptions: graphics, functions with no returned value

15112 Style Rubric

Clarity

```
def abs(n):
    return (n < 0)*(-n) + (n >= 0)*(n)
```

```
def abs(n):
    if(n < 0):
        return -n
    else:
        return n
```

15112 Style Rubric

Meaningful variable/function names

No more a, b, c, d, x, u, ww, pt, qr

Use mixedCase.

Bad variable names

`x`

`anonymous`

`thething`

`anilsucks`

Good variable names

`length`

`counter`

`degreesInFahrenheit`

`theMessageToTellAnilHeSucks`

15112 Style Rubric

"Numbered" variables

count0
count1
count2
count3
count4
count5
count6
count7
count8
count9

Use lists and/or loops

15112 Style Rubric

Magic numbers

Hides logic. Harder to debug.

```
def shift(c, shiftNum):
    shiftNum %= 26
    if (not c.isalpha()):
        return c
    alph = string.ascii_lower if (c.islower())  else string.ascii_upper
    shifted_alph = alph[shiftNum:] + alph[:shiftNum]
    return shifted_alph[alph.find(c)]
```

magic number

15112 Style Rubric

Magic numbers

Hides logic. Harder to debug.

```
def shift(c, shiftNum):
    alphabetSize = 26
    shiftNum %= alphabetSize
    if (not c.isalpha()):
        return c
    alph = string.ascii_lower if (c.islower())  else string.ascii_upper
    shifted_alph = alph[shiftNum:] + alph[:shiftNum]
    return shifted_alph[alph.find(c)]
```

15112 Style Rubric

Formatting

- max 80 characters per line
- proper indentation (use 4 spaces, not tab)
- one blank line between functions
- one blank line to separate logical sections

15112 Style Rubric

Others

Efficiency

Global variables

Duplicate code

Dead code

Meaningful User Interface (UI)

Other guidelines as described in course notes

Today's Menu

String formatting

File I/O

Style

Top-down design

Example(s)

Problem solving with programming

Not a good strategy:

```
write code

while (bugs exist):
    change code
```

Problem solving with programming

1. Understand the problem

2. Devise a plan

   2a. How would you solve it with paper, pencil, calc.

   2b. Write an algorithm
   - use explicit, clear, small steps
   - don't require human memory or intuition

3. Translate the algorithm into code

   3a. Write test cases

   3b. Write code → Starting here is big mistake!!!

   3c. Test code

4. Examine and review

Problem solving with programming

1. Understand the problem

2. Devise a plan

   2a. How would you solve it with paper, pencil, calc.

   2b. Write an algorithm
   - use explicit, clear, small steps
   - don't require human memory or intuition

3. Translate the algorithm into code

   3a. Write test cases

   3b. Write code

   3c. Test code

4. Examine and review

Devise a plan

Some useful strategies:

Divide and conquer
(top-down design)

Incremental layers of complexity

Solve a simplified version

Divide and conquer cinnamon rolls

For the rolls, dissolve the yeast in the warm milk in a large bowl.

Add sugar, margarine salt, eggs, and flour, mix well.

Knead the dough into a large ball, using your hands dusted lightly with flour.

Put in a bowl, cover and let rise in a warm place about 1 hour or until the dough has doubled in size.

Roll the dough out on a lightly floured surface, until it is approx 21 inches long by 16 inches wide. It should be approx 1/4 thick.

Preheat oven to 400 degrees.

To make filling, combine the brown sugar and cinnamon in a bowl.

Spread the softened margarine over the surface of the dough, then sprinkle the brown sugar and cinnamon evenly over the surface.

Working carefully, from the long edge, roll the dough down to the bottom edge.

Cut the dough into 1 3/4 inch slices, and place in a lightly greased baking pan.

Bake for 10 minutes or until light golden brown.

While the rolls are baking combine the icing ingredients.

Beat well with an electric mixer until fluffy.

When the rolls are done, spread generously with icing.

Looking closely, 3 main parts:

- Make the dough
- Make the filling
- Make the icing

Then combine the parts.

Making the dough:

- Mix the ingredients
- Knead
- Roll

Not so bad...

Divide and conquer

- Break up the problem into smaller components.

- Assume solutions to smaller parts exist.
  Combine them to get a “solution”.

- Solve each smaller component separately.

The secret to programming/computing

Many layers of *abstraction*.

- We start with electronic switches.
- We abstract away and represent data with 0s and 1s.
- We have machine language (0s and 1s) to tell the computer what to do.
- We abstract away and build/use high-level languages.
- We abstract away and build/use functions and *objects* (more on objects-oriented programming later).

This is how large, complicated programs are built!

Devise a plan

Some useful strategies:

Divide and conquer
(top-down design)

Incremental layers of complexity

Solve a simplified version

Incremental layers of complexity

- Start with basic functionality.

- Add more functionality.

- Build your program layer by layer.

Pong Game

1. Start with a ball bouncing around.

2. Add paddles.

3. Make paddles move up and down with keystrokes.

4. Make the ball interact with the paddles. How will the ball bounce?

5. Implement scoring a goal.

6. Keep track of scores.

Devise a plan

Some useful strategies:

Divide and conquer
(top-down design)

Incremental layers of complexity

Solve a simplified version

Solve a simplified version

- Identify a meaningful simplified version of the problem

- Solve it

- Sometimes the simplified version can be an important subproblem (make it a helper function)

Today's Menu

String formatting

File I/O

Style

Top-down design

Example(s)

hasBalancedParentheses(s)

<u>Input</u>: a string s

<u>Output</u>: True if s contains a "balanced parentheses".

balanced: ( ) , (( )) , (( ) ( )) , (( )) ( ) ( )

unbalanced: (,)(, (( ) ( ) , ( )) ( ) ( )

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

height: 0

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

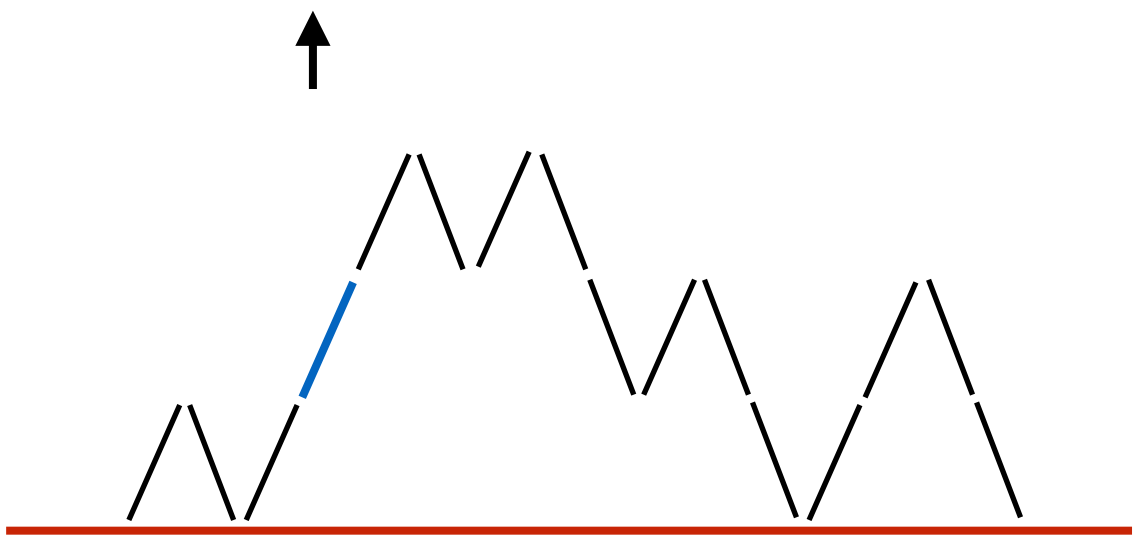
height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



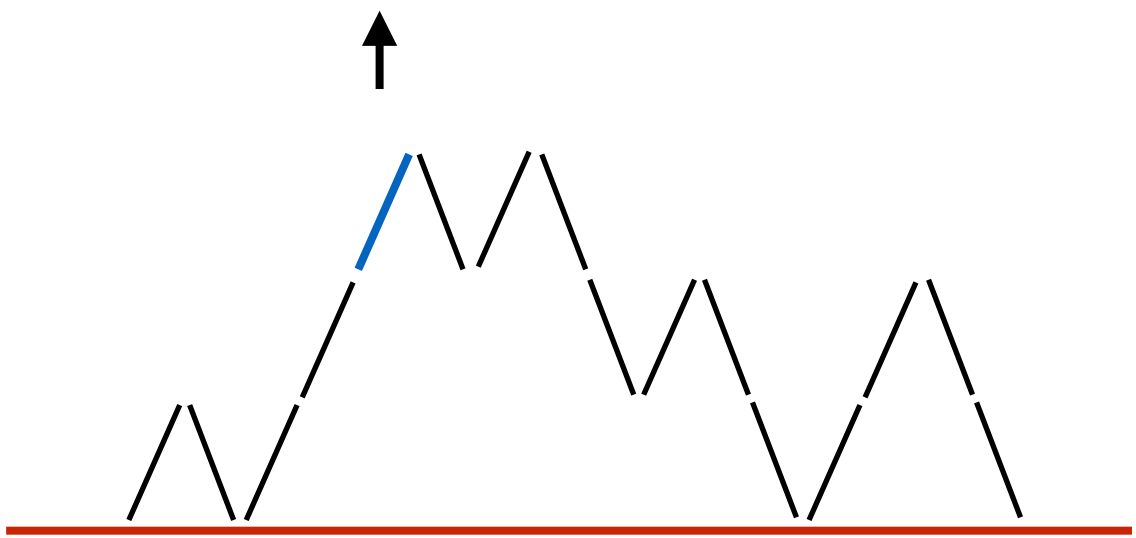
height: 2

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



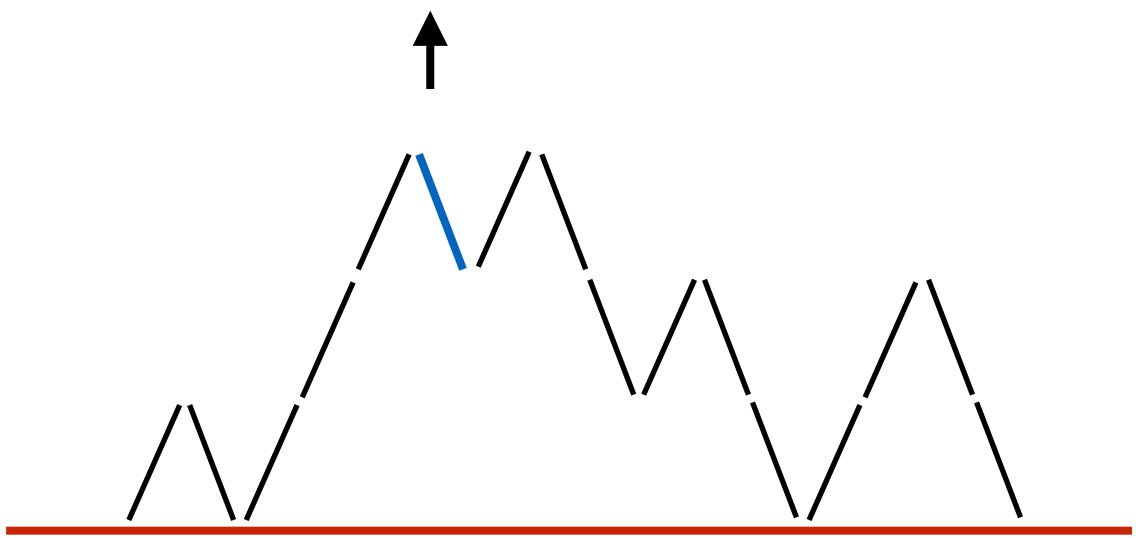
height: 3

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



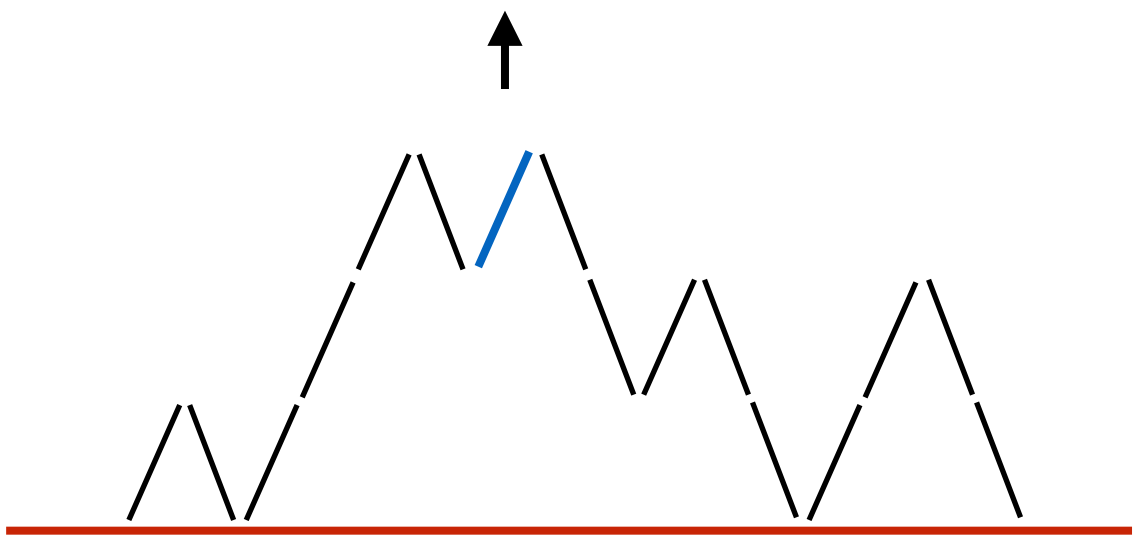
height: 2

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



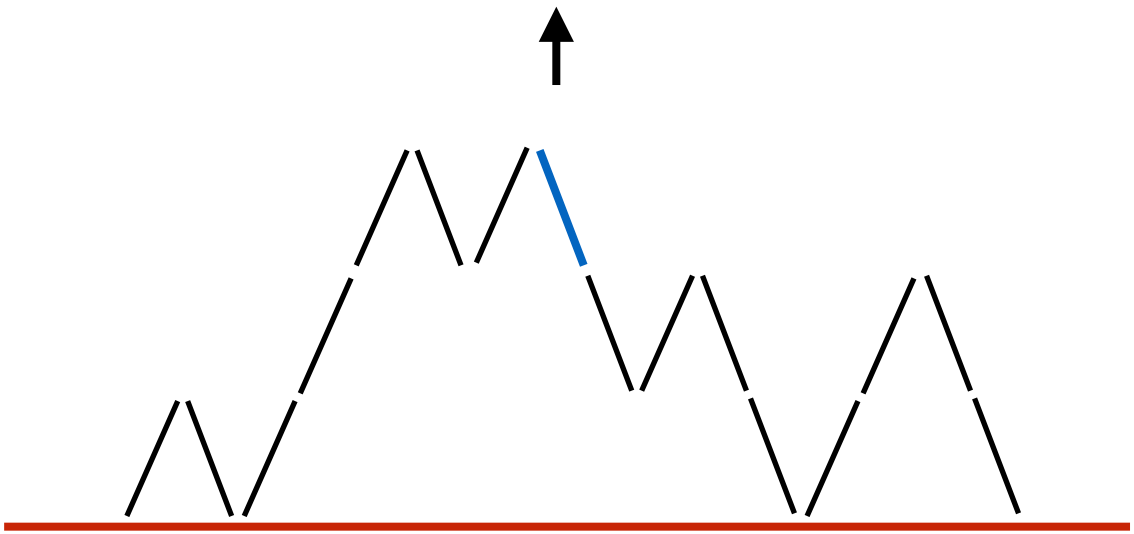
height: 3

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



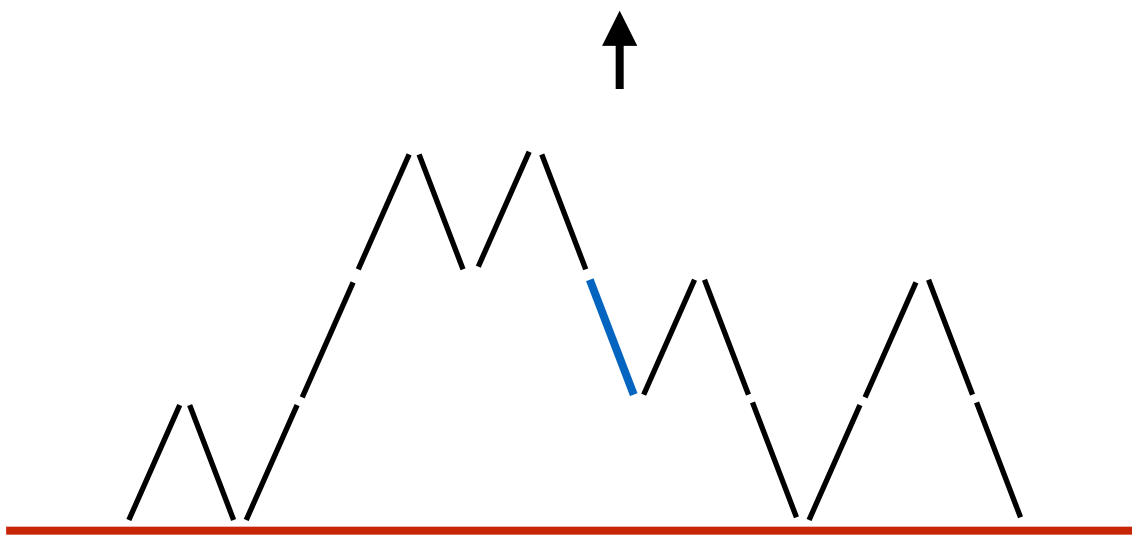
height: 2

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



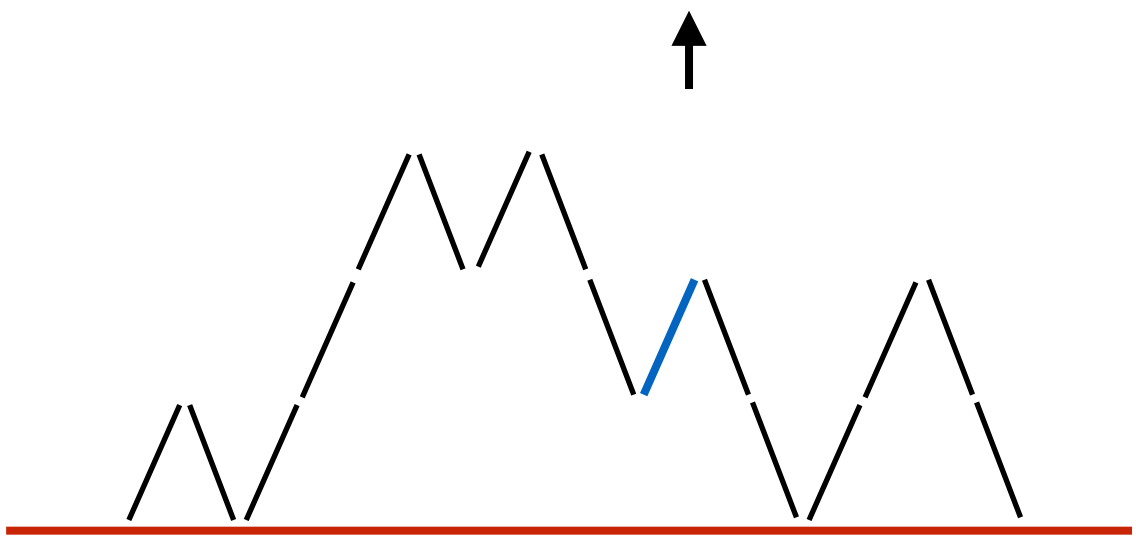
height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



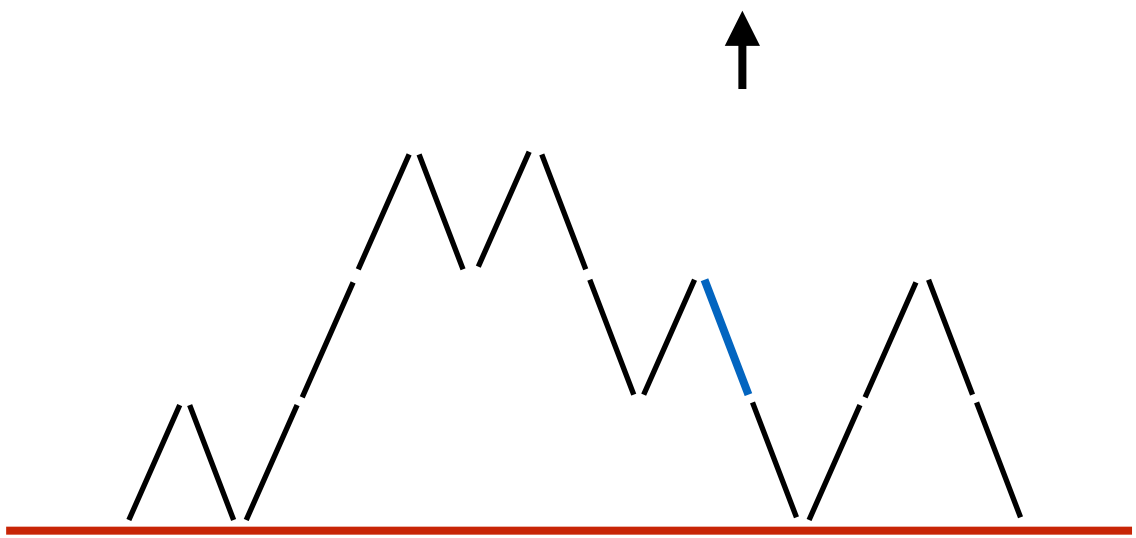
height: 2

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



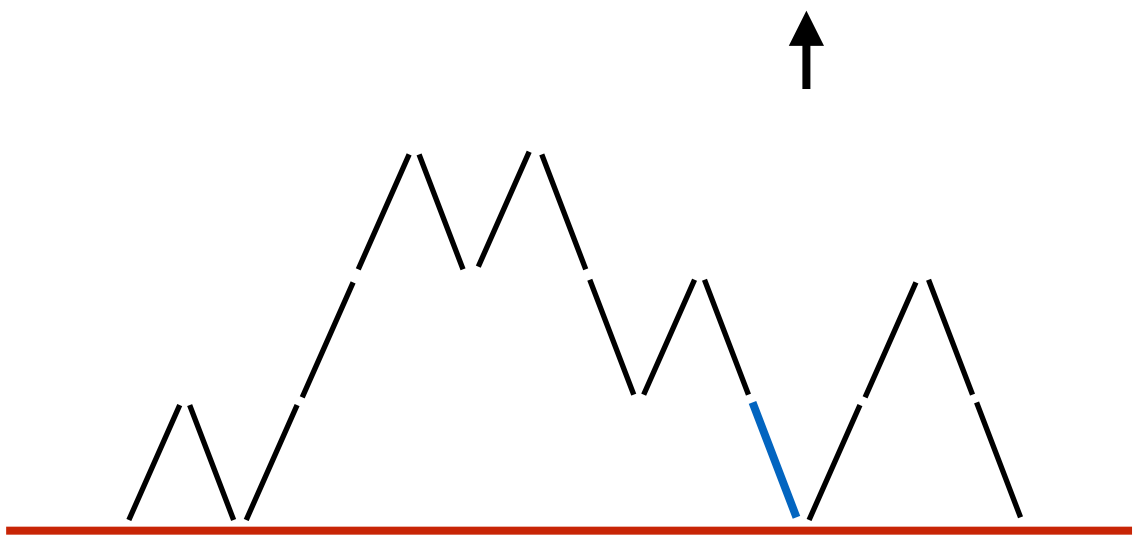
height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))



height: 0

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

height: 2

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

height: 1

hasBalancedParentheses(s)

Input: a string s

Output: True if s contains a "balanced parentheses".

( ) ((( ) ( )) ( )) (( ))

- height should never be negative
- height should end at 0

height: 0

areAnagrams(s1, s2)

Input: two strings s1 and s2

Output: True if you can reorder the characters in s1 to get s2.

For each character c in s1:

Check if (# c in s1 == # c in s2)

(make s1 and s2 lower case at the beginning)

rotateStringLeft(s, k)

("abcde", 0) → "abcde"

("abcde", 1) → "bcdea"

("abcde", 2) → "cdeab"

("abcde", 3) → "deabc"

("abcde", 4) → "eabcd"

("abcde", 5) → "abcde"

("abcde", 6) → "bcdea"

abcde 3

deabc

```
def rotateStringLeft(s, k):
    k = k % len(s)
    return s[k:] + s[:k]
```

playMastermind()

Let's do it.