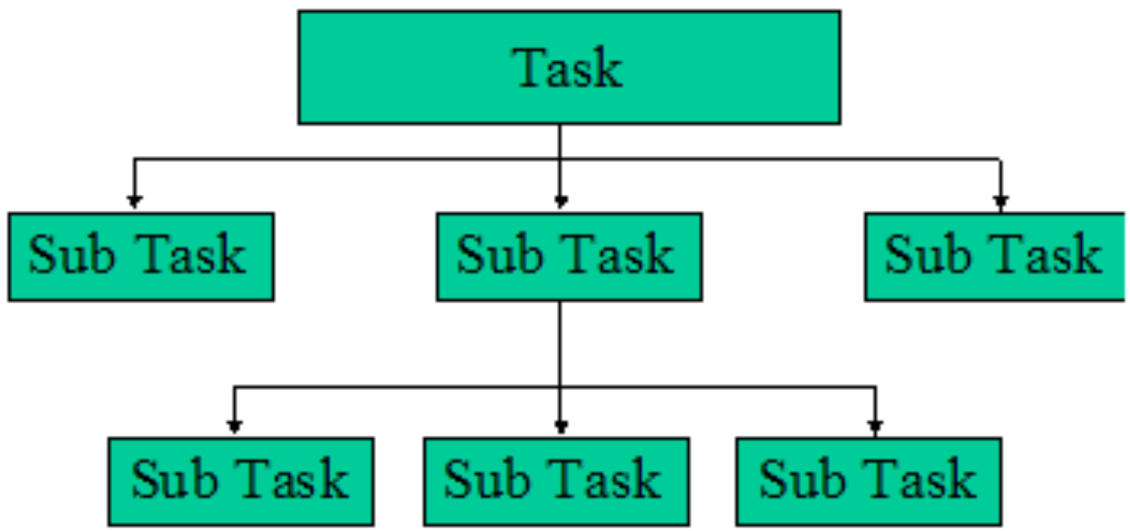
15-112
Fundamentals of Programming

Week 2 - Lecture 2:
Nested loops + Style + Top-down design



May 24, 2016

Nested Loops

My first ever program

```
************
***********
**********
*********
********
*******
******
*****
****
***
**
*
```

Nested loops

Many situations require one loop inside another loop.

```
for y in range(10):
    for x in range(8):
        # Body of the nested loop
```

Nested loops

Many situations require one loop inside another loop.

```
for y in range(10):
    for x in range(8):
        print("Hello")
```

How many times will "Hello" get printed?

Nested loops

Many situations require one loop inside another loop.

```
for y in range(4):
    for x in range(y):
        print("Hello")
```

y	# iterations of inner loop
0	0
1	1
2	2
3	3

How many times will "Hello" get printed?

Example: Draw a rectangle

Write a function that:

- Gets two integers, height and width as input
- Prints a rectangle with those dimensions

height = 4, width = 3

```
* * *
* * *
* * *
* * *
```

Repeat 4 times:

- Print a row (3 stars)

Example: Draw a rectangle

Write a function that:

- Gets two integers, height and width as input
- Prints a rectangle with those dimensions

height = 4, width = 3

```
* * *
* * *
* * *
* * *
```

Repeat 4 times:
    Repeat 3 times:
        - Print a star
    Skip a line

Example: Draw a rectangle

Write a function that:

- Gets two integers, height and width as input
- Prints a rectangle with those dimensions

height = 4, width = 3

```
* * *
* * *
* * *
* * *
```

```
for row in range(4):
    for col in range(3):
        print("*", end=" ")
    print()
```

Example: Draw a rectangle

Write a function that:

- Gets two integers, height and width as input
- Prints a rectangle with those dimensions

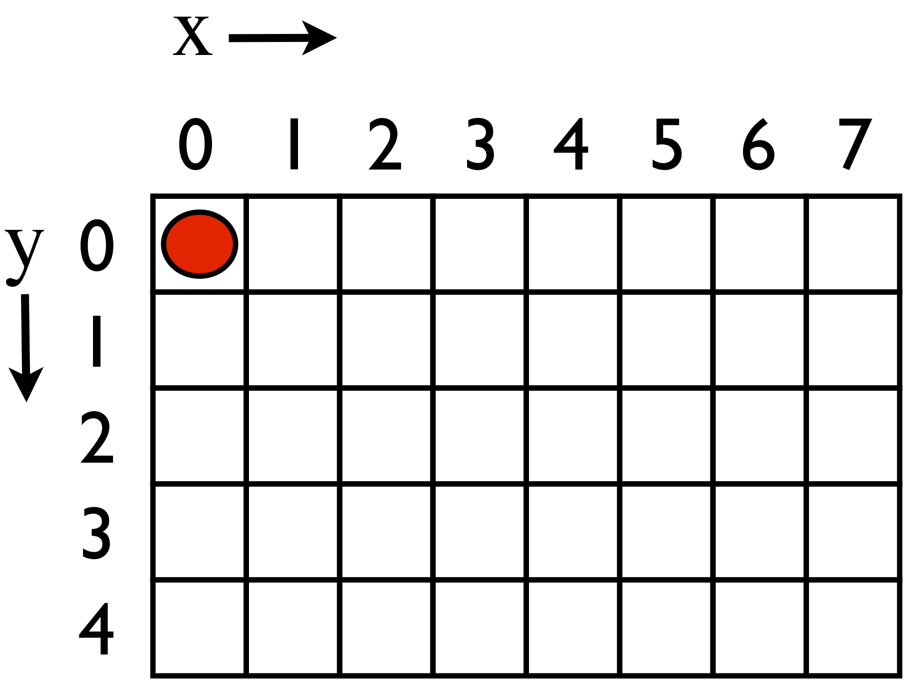
height = 4, width = 3

```
* * *
* * *
* * *
* * *
```

```
def printRectangle(height, width):
    for row in range(height):
        for col in range(width):
            print("*", end= " ")
        print()
```

Nested loops

```
for y in range(5):
    for x in range(8):
        # Body of the nested loop
```



Example

```
for y in range(4):
    for x in range(5):
        print(“(%d , %d)” % (x, y)), end=“ ”)
    print()
```

x →

y ↓

(0 , 0)	(1 , 0)	(2 , 0)	(3 , 0)	(4 , 0)
(0 , 1)	(1 , 1)	(2 , 1)	(3 , 1)	(4 , 1)
(0 , 2)	(1 , 2)	(2 , 2)	(3 , 2)	(4 , 2)
(0 , 3)	(1 , 3)	(2 , 3)	(3 , 3)	(4 , 3)

Example

```
for y in range(4):
    for x in range(y):
        print("(%d , %d)" % (x, y)), end=" ")
    print()
```

```
\n
( 0 , 1 )
( 0 , 2 ) ( 1 , 2 )
( 0 , 3 ) ( 1 , 3 ) ( 2 , 3 )
```

Example

```
for y in range(1, 10):
    for x in range(1, 10):
        print(y*x, end=“ ”)
    print()
```

Multiplication table

```
for y in range(1, 10):
    for x in range(1, 10):
        print(y*x, end=“ ”)
    print()
```

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

A trick to get rid of nested loops

Write a function for the inner loop.

Example: Write a function that:

- Gets an integer height as input
- Prints a right-angled triangle of that height

height = 5

```
*****
****
***
**
*
```

```
def printStars(n):
    for x in range(n):
        print("*", end="")

def printTriangle(height):
    for x in range(height):
        printStars(   ?   )
    print()
```

A trick to get rid of nested loops

Write a function for the inner loop.

Example: Write a function that:

- Gets an integer height as input
- Prints a right-angled triangle of that height

height = 5

```
*****
****
***
**
*
```

```
def printStars(n):
    for x in range(n):
        print("*", end="")

def printTriangle(height):
    for x in range(height):
        printStars(height - x)
    print()
```

A common nested loop

Input: a string s

Output: True if s contains a character more than once.
False otherwise.

```
def hasDuplicates(s):
    for i in range(len(s)-1):
        for j in range(i+1,len(s)):
            if(s[i] == s[j]): return True
    return False
```

Style

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: A
```

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

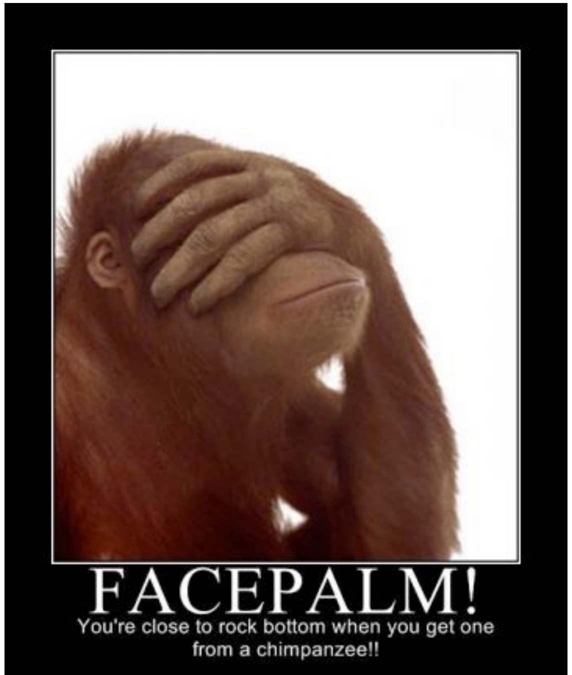
Very Bad

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

15112 Style Rubric

Comments

```
# This function 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

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, u, ww, pt, qr, abc

Use mixedCase.

Bad variable names

`a`

`thething`

`anonymous`

`anilsucks`

Good variable names

`length`

`degreesInFahrenheit`

`counter`

`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

Magic numbers

Hides logic. Harder to debug.

```
def toUpperLetter(c):
    if (“a” <= c <= “z”):
        return chr(ord(c) - 32)
    return c
```

magic number

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

Top-down Design

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 the overall 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 object-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)

Top-down Design Example

playMastermind()