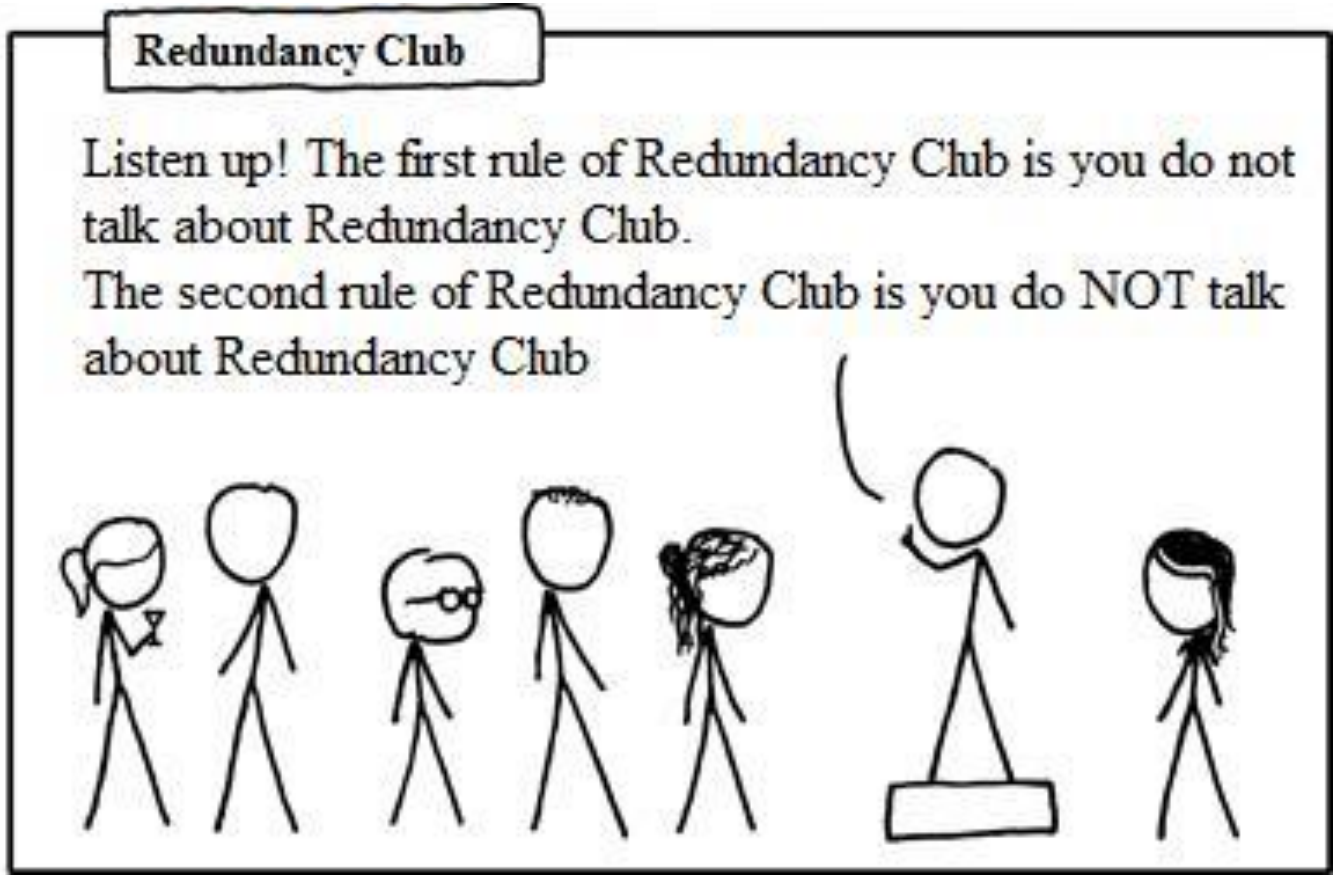
CSc 110, Spring 2018

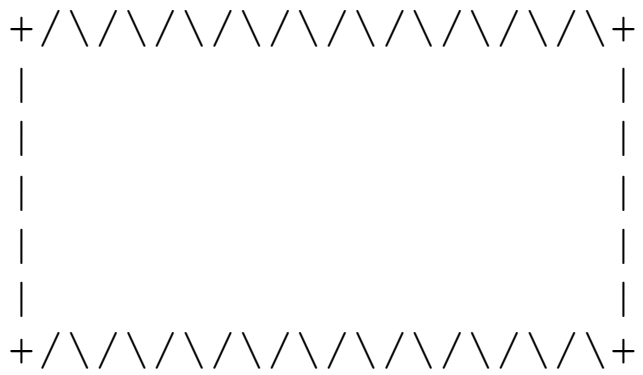
Lecture 7: `input` and Constants

Adapted from slides by Marty Stepp and Stuart Reges

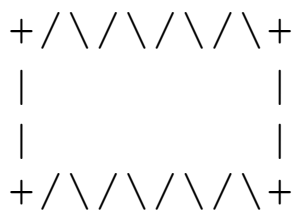


Constants and figures

- Consider the task of drawing the following scalable figure:



Multiples of 5 occur many times



The same figure at size 2

Constant tables

`SIZE = ...`

- What equation would cause the code to print:

  `2 7 12 17 22`

- To see patterns, make a table of `SIZE` and the numbers.
  - Each time `SIZE` goes up by 1, the number should go up by 5.
  - But `SIZE * 5` is too great by 3, so we subtract 3.

`SIZE`	number to print	`5 * SIZE`	`5 * SIZE - 3`
1	2	5	2
2	7	10	7
3	12	15	12
4	17	20	17
5	22	25	22

Constant tables question

- What equation would cause the code to print:

```
17 13 9 5 1
```

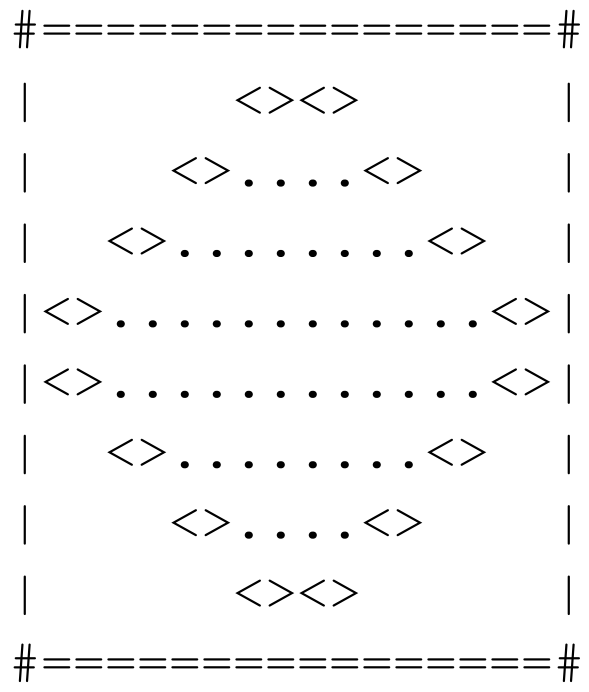
- Let's create the constant table together.
  - Each time `SIZE` goes up 1, the number printed should ...
  - But this multiple is off by a margin of ...

`SIZE`	number to print	`-4 * SIZE`	`-4 * SIZE+ 21`
1	17	-4	17
2	13	-8	13
3	9	-12	9
4	5	-16	5
5	1	-20	1

Drawing complex figures

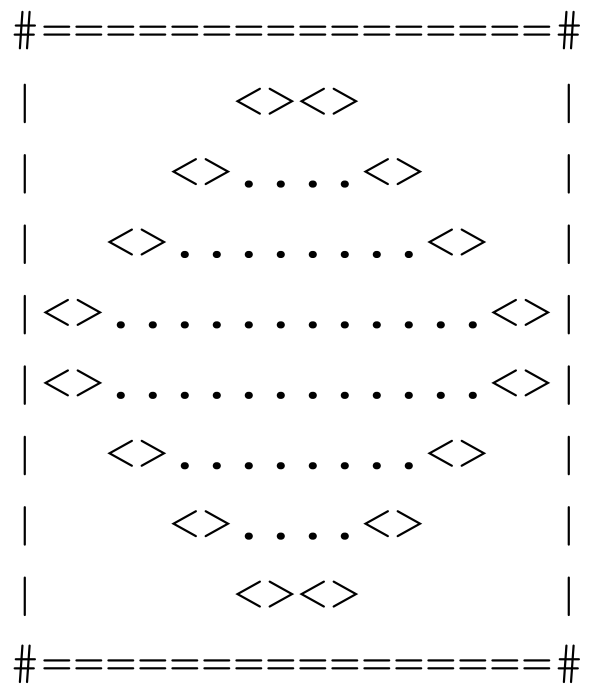
- Use nested `for` loops to produce the following output.

- Why draw ASCII art?
  - Real graphics require a lot of finesse
  - ASCII art has complex patterns
  - Can focus on the algorithms



Development strategy

- Recommendations for managing complexity:
  1. Design the program (think about steps or methods needed).
     - write an English description of steps required
     - use this description to decide the functions
  2. Create a table of patterns of characters
     - use table to write your `for` loops



1. Pseudo-code

- **pseudo-code**: An English description of an algorithm.

- Example: Drawing a 12 wide by 7 tall box of stars

```
print 12 stars.
for (each of 5 lines) :
    print a star.
    print 10 spaces.
    print a star.
print 12 stars.
```

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

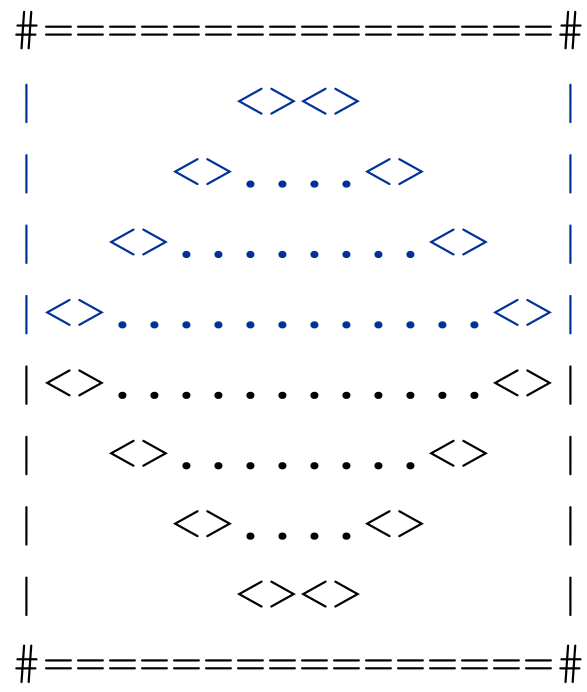
Pseudo-code algorithm

1. Line
   - # , 16 =, #

2. Top half
   - |
   - spaces (decreasing)
   - <>
   - dots (increasing)
   - <>
   - spaces (same as above)
   - |

3. Bottom half (top half upside-down)

4. Line
   - # , 16 =, #



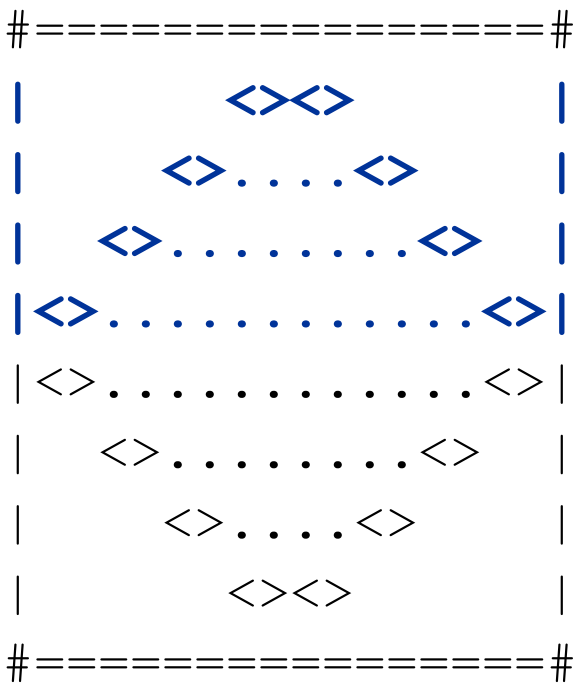
Functions from pseudocode

```
def main():
    line()
    top_half()
    bottom_half()
    line()

def top_half():
    for line in range(1, 5):
        # contents of each line

def bottom_half() {
    for line in range(1, 5):
        # contents of each line

def line():
        # ...
```

2. Tables

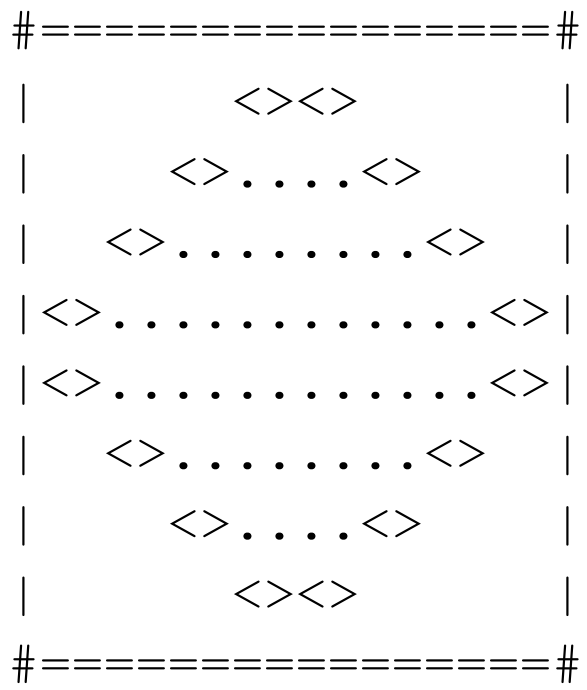
- A table for the top half:
  - Compute spaces and dots expressions from line number

line	spaces	line * -2 + 8	dots	4 * line - 4
1	6	6	0	0
2	4	4	4	4
3	2	2	8	8
4	0	0	12	12



3. Writing the code

- Useful questions about the top half:
  - Number of (nested) loops per line?



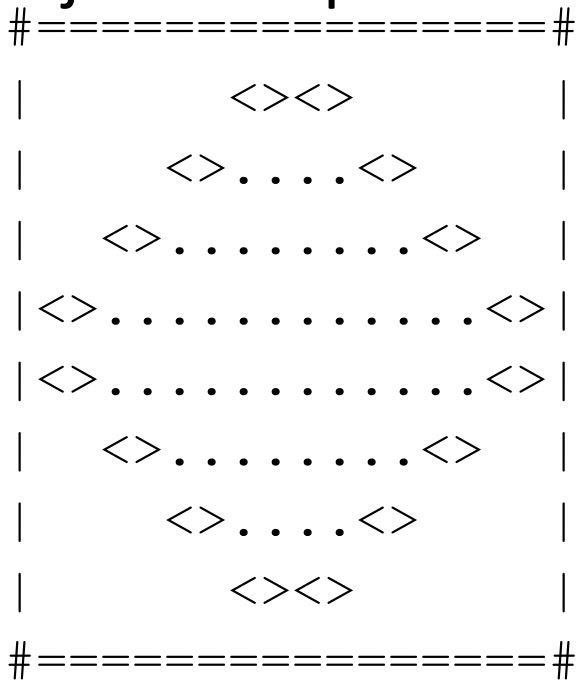
Partial solution

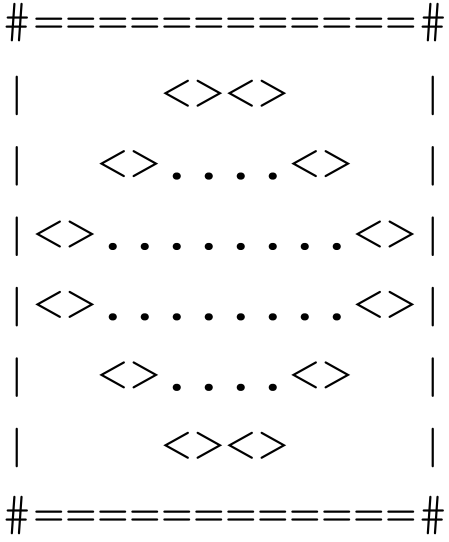
```
# Prints the expanding pattern of <> for the top half of the figure.
def top_half():
    for line in range(1, 5):
        print("|", end="")

        for space in range(1, line * -2 + 9):
            print(" ", end="")


        print("<>", end="")

        for dot in range(1, line * 4 - 3):
            print(".", end="")


        print("<>", end="")

        for space in range(1, line * -2 + 8):
            print(" ", end="")


        print("|")
```

Scaling the mirror

- Let's modify our Mirror program so that it can scale.
  - The current mirror (left) is at size 4; the right is at size 3.

- We'd like to structure the code so we can scale the figure by changing the code in just one place.

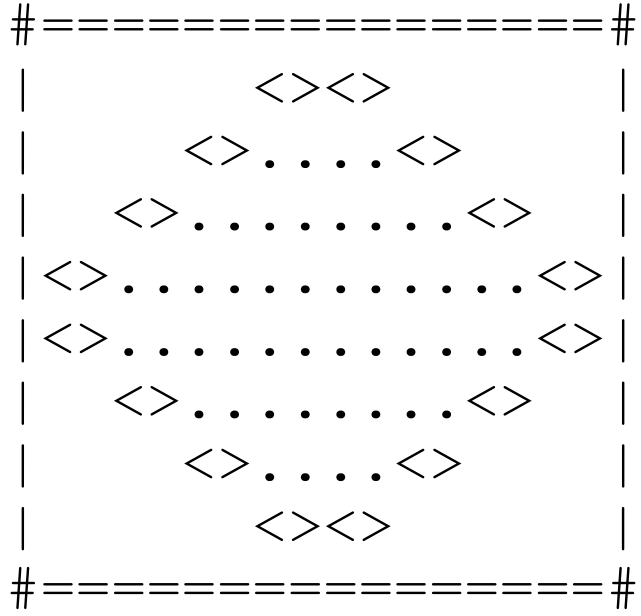




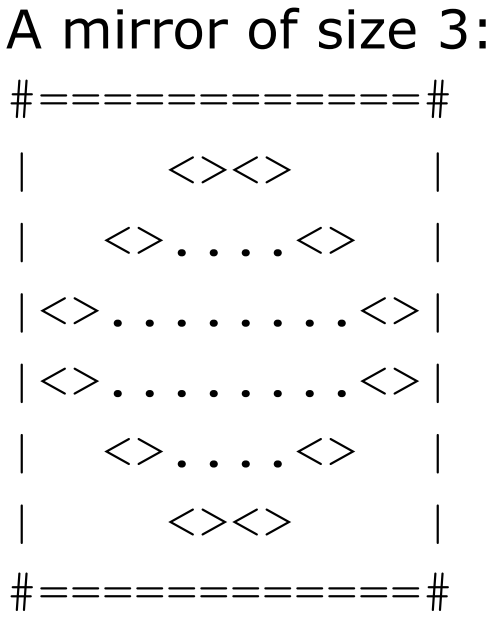
Complex figure w/ constant

- Modify the Mirror code to be resizable using a constant.

A mirror of size 4:



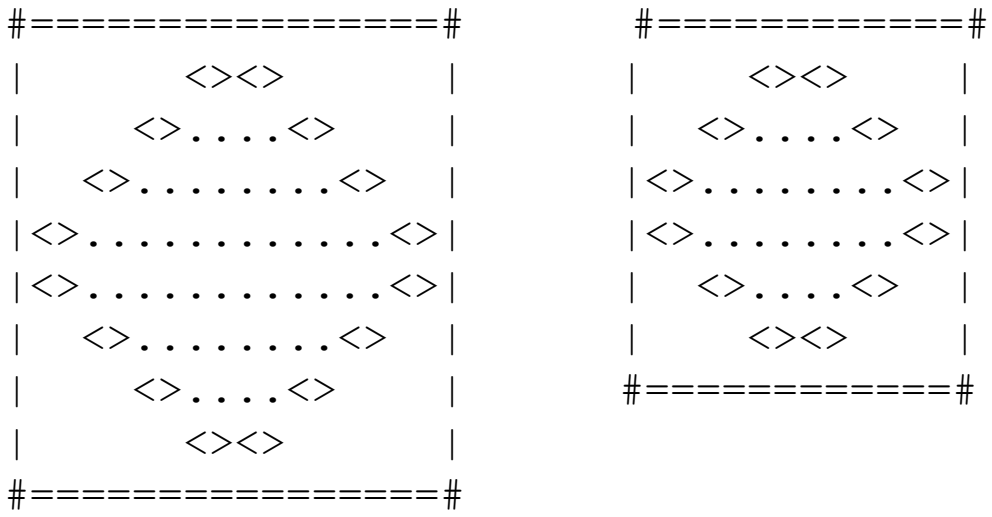
A mirror of size 3:



Loop tables and constant

- Let's modify our loop table to use `SIZE`
  - This can change the amount added in the loop expression

SIZE	line	spaces		dots	
4	1,2,3,4	6,4,2,0		0,4,8,12	
3	1,2,3	4,2,0		0,4,8	



Partial solution

```
SIZE = 4;
# Prints the expanding pattern of <> for the top half of the figure.
def top_half() {
    for line in range(1, SIZE):
        print("|", end="")
        for space in range(1, line * -2 + (2*SIZE) + 1):
            print(" ", end="")

        print("<>", end="")
        for dot in range(1, line * 4 - 3):
            print(".", end="")

        print("<>", end="")
        for space in range(1, line * -2 + (2*SIZE) + 1):
            print(" ", end="")

        print("|")
```

Observations about constant

- The constant can change the "intercept" in an expression.
  - Usually the "slope" is unchanged.

```
SIZE = 4;

for space in range(1, line * -2 + (2 * SIZE)):
    print(" ", end="")
```

- It doesn't replace *every* occurrence of the original value.

```
for dot in range(1, line * 4 - 4 + 1):
    print(".", end="")
```

Interactive programs

interactive program: Reads input from the console.

- While the program runs, it asks the user to type input.
- The input typed by the user is stored in variables in the code.

- Can be tricky; users are unpredictable and misbehave.
- But interactive programs have more interesting behavior.

`input`

- **`input`**: An function that can read input from the user.

- Using an `input` object to read console input:

  name `= input(`**prompt**`)`

  - Example:

    ```
    name = input("type your name: ")
    ```

    - The variable `name` will store the value the user typed in

input example

```
def main():
    age = input("How old are you? ")

    years = 65 - age
    print(years, " years until retirement!")
```

age | 29

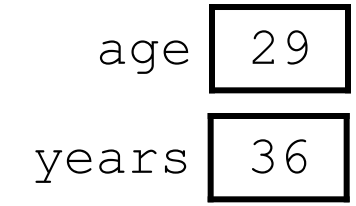
- Console (user input underlined):

```
How old are you? 29
```

```
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    print(65 - age)
TypeError: unsupported operand type(s) for -:
'int' and 'str'
```

input example

```
def main():
    age = int(input("How old are you? "))

    years = 65 - age
    print(years, "years until retirement!")
```



- Console (user input underlined):

```
How old are you? 29
36 years until retirement!
```