CS1010S Programming Methodology

Lecture 2

Functional Abstraction

20 Aug 2018

Expectations

Tutorial Allocation

Coursemology Survey

- Choose your preferred slots
- As many slots as possible
- Updated with number of classes

Recitation

Appeal on CORS

classes starts

on Thursday/Friday

Late Policy

- < 10 min: OK
- < 30 min: -10%
- < 12 hours: -20%
- < 24 hours: -50%
- > 24 hours: -100%

Ask early for extensions

Submission is Final

But please remember to click

Finalize Submission

Don’t Stress

But please do your work

Try NOT to submit at 23:58

Operators

Assignment

`a = 5`

Equality testing

`a == 5`

Not equal

`a != 5`

Backslash \

Escape character

```
print('That's')
print('That\'s')
```

#Comments

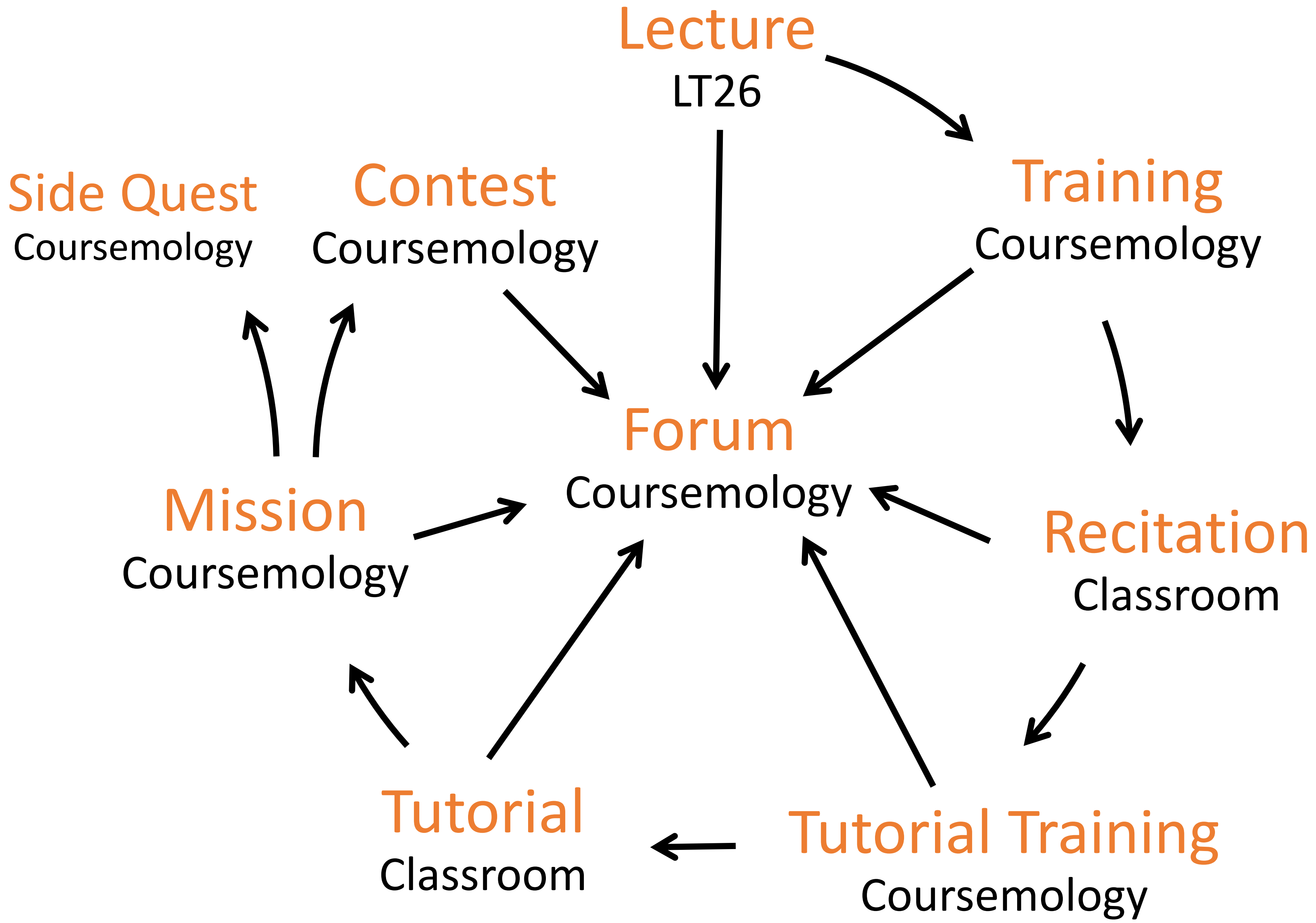
```python
# this is not a hashtag!

print("Good to go")

#print("Good to go")
# whatever is after the # is ignored

if light == "red": # Check state of light
```

What's this?

Python Imaging Library

```
from PIL import *
```

(Misison 0)


Lecture
LT26
Side Quest
Coursemology
Contest
Coursemology
Training
Coursemology
Forum
Coursemology
Mission
Coursemology
Recitation
Classroom
Tutorial
Classroom
Tutorial Training
Coursemology

Forums

Post reflections for EXP

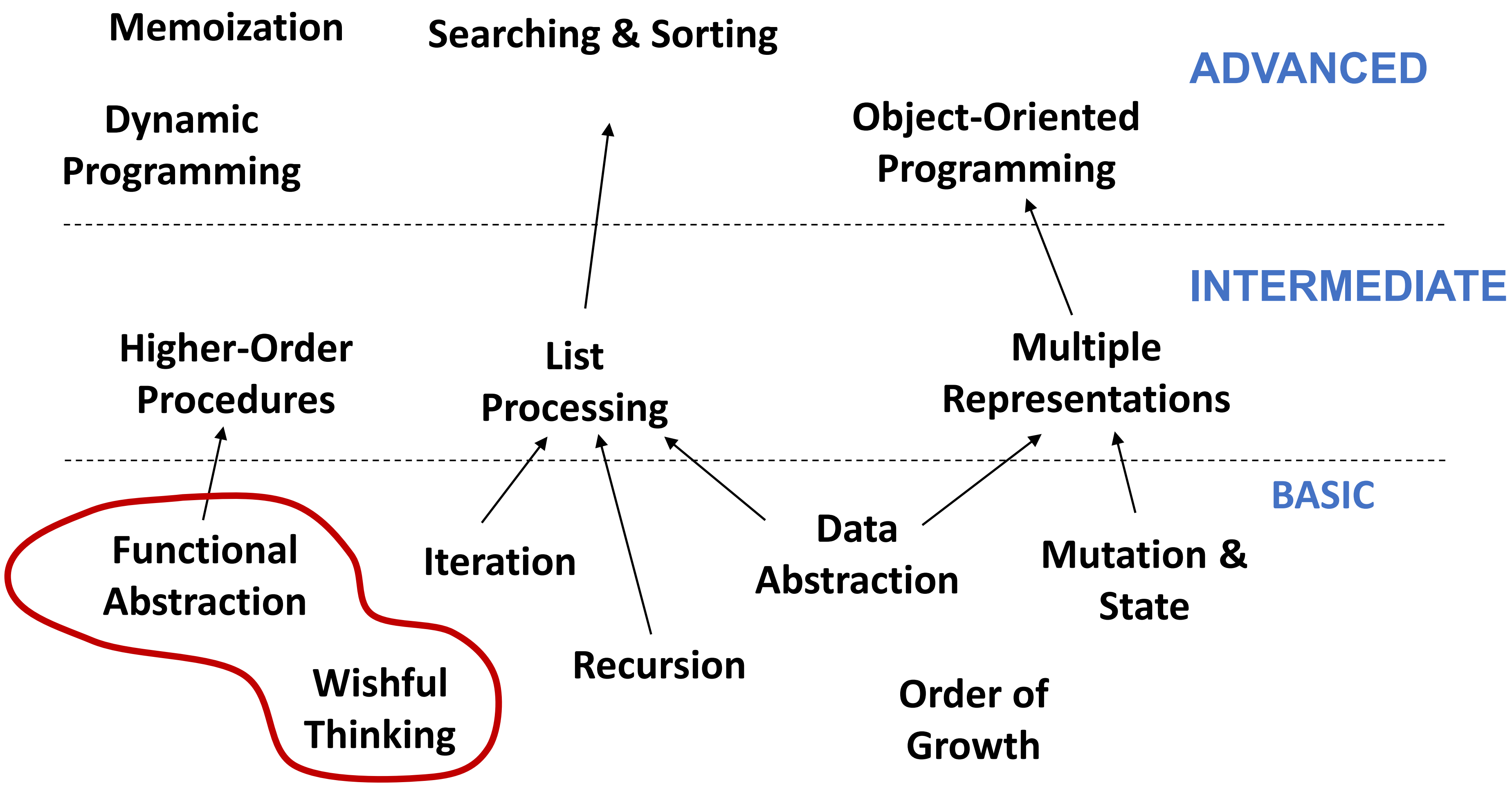
Trainings

Please don't anyhow hantam

Computational Thinking

Fasten your seatbelt

CS1010S Road Map



Fundamental **concepts** of computer programming

Functional Abstraction

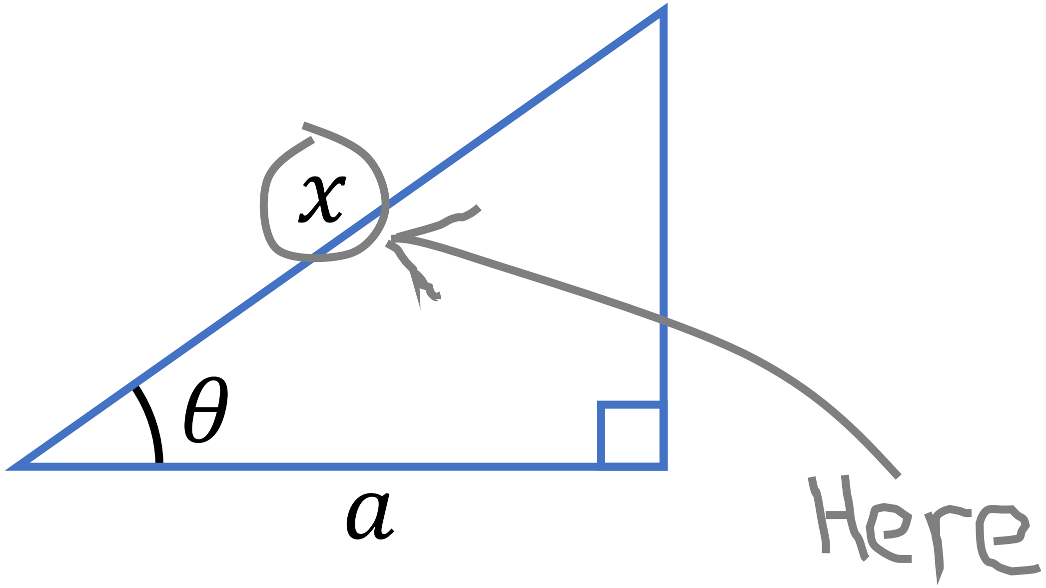
WHAT

HOW

WHY

What is a function?

Functions are nothing new



Find x?

input

$$x = \frac{\cos(\theta)}{a}$$

function

Let’s start with something easier

Question

How do we square a number?

The square function

Define Name Input

```python
def square(x):
    return x * x
```

Return Output

```
square(21)          441

square(2 + 5)       49

square(square(3))   81
```

Another function

```
def sum_of_squares(x, y):
    return square(x) + square(y)

sum_of_squares(3, 4)
```

25

And another

```
from math import sqrt        square root

def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))


hypotenuse(5, 12)            13
```

General Form

```
def <name> (<formal parameters>):
    <body>
```

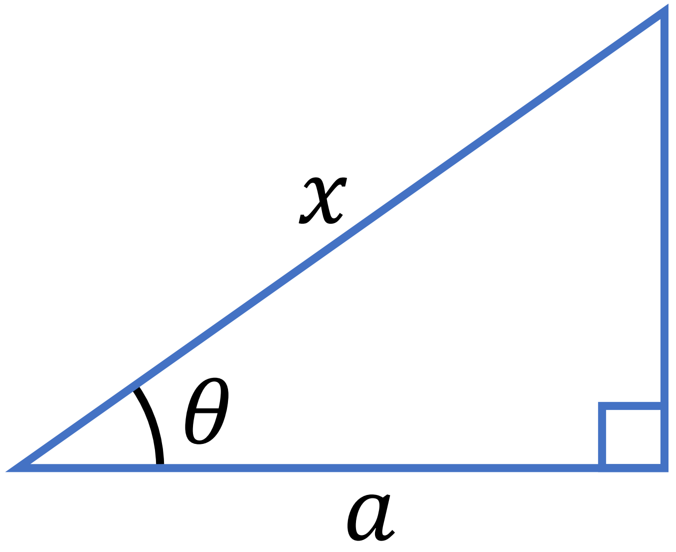
- name
  - Symbol associated with the function
- formal parameters
  - Names used in the body to refer to the arguments of the function
- body
  - The statement(s) to be evaluated
  - Has to be indented (standard is 4 spaces)
  - Can return values as output

Black Box

Don’t need to know how it works

Just know what it does

Black Box



$$x = \frac{a}{\cos(\theta)}$$

Do you know how $\cos$ work?

Black Box

As long as we know what it does,
we can use it.

(the inputs
and output)

Return Type

Output is returned with `return`

Return type can be None

Abstract Environment

Picture Language

(runes.py)

Also graphics.py + PyGif.py

Elements of Programming

1. Primitives
2. Means of Combination
3. Means of Abstraction
4. Controlling Logic

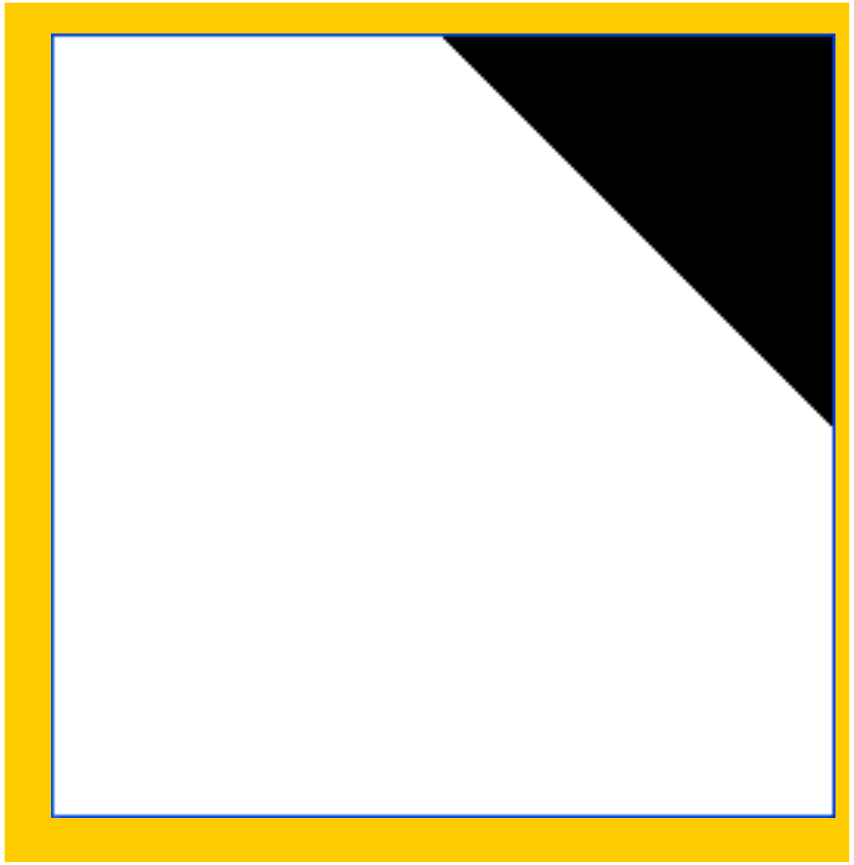
Primitives building block

`show(rcross_bb)`

Picture object

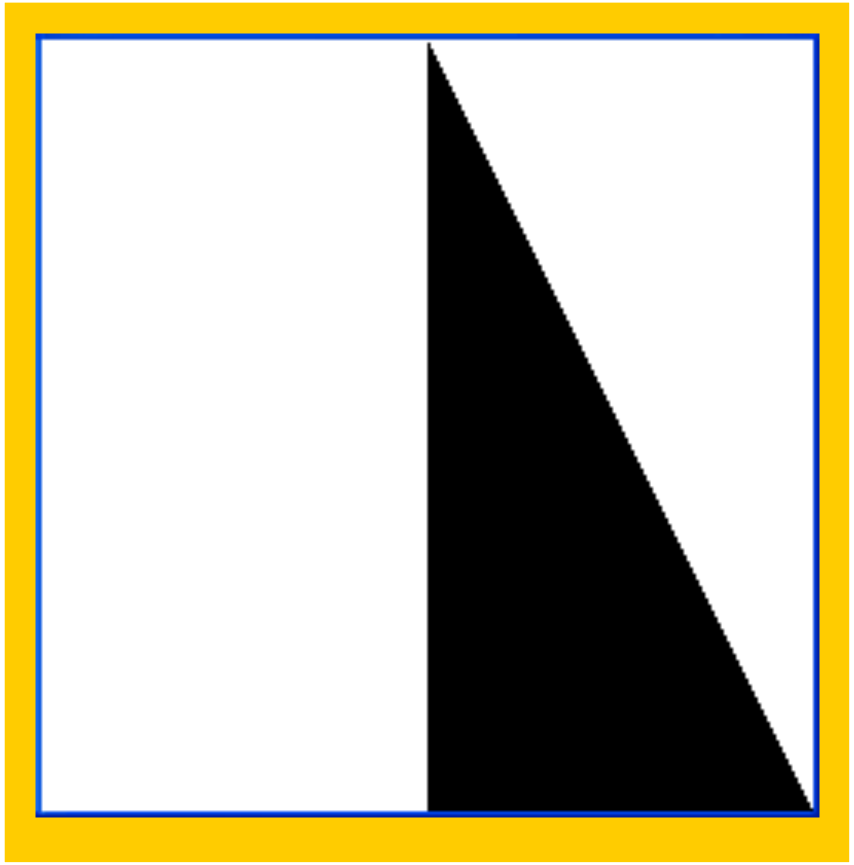
Primitives building block

```
show(corner_bb)
```



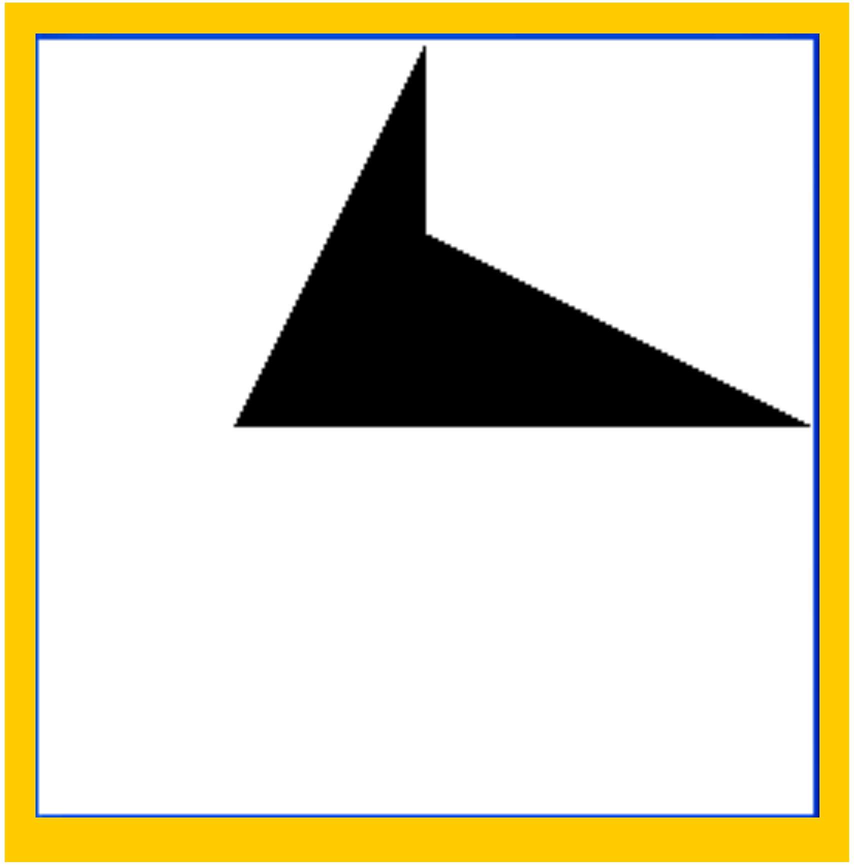
Primitives building block

```
show(sail_bb)
```



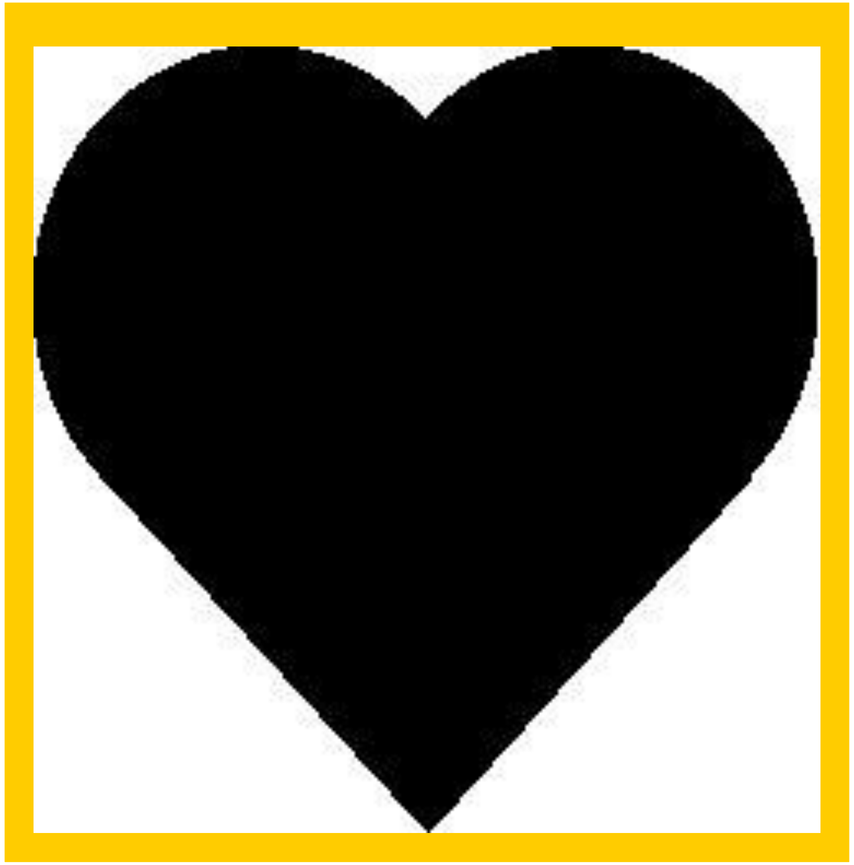
Primitives building block

```
show(nova_bb)
```



Primitives building block

```
show(heart_bb)
```



Applying operations

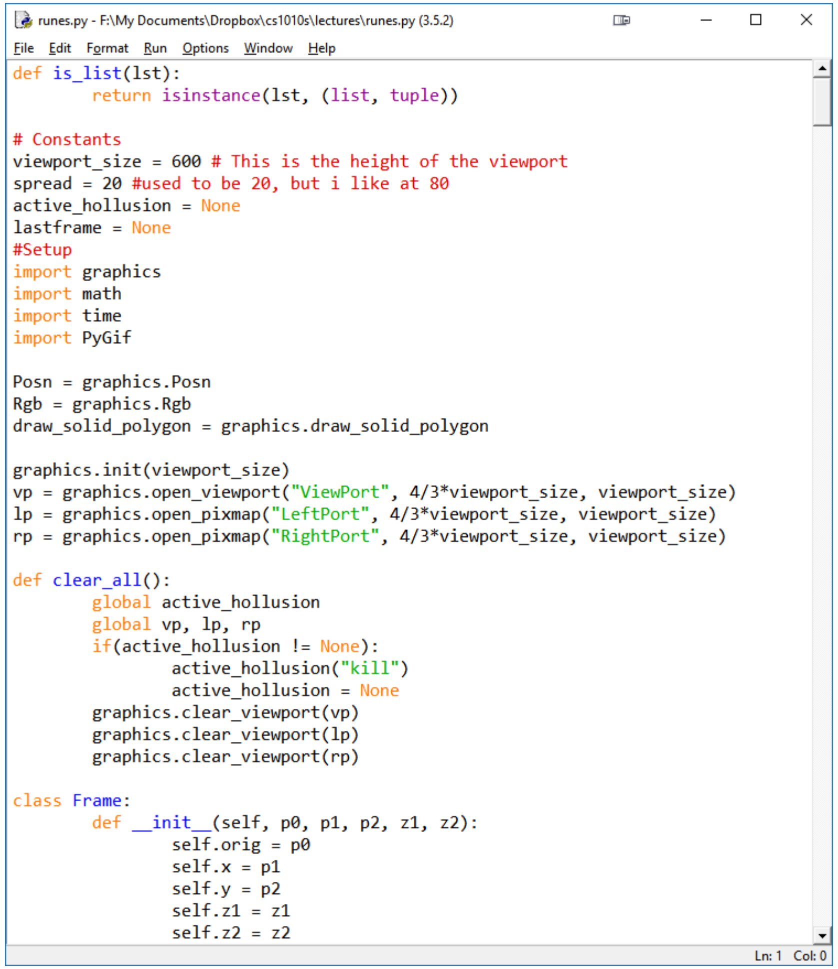
```
op(picture)
```

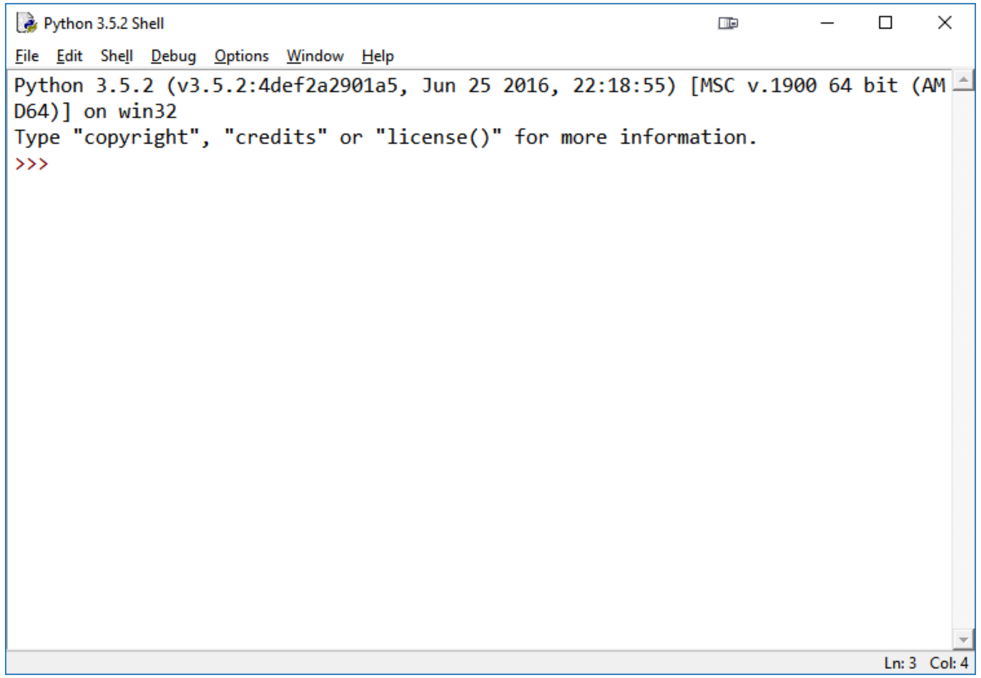
function name input(s)

Example:

```
show(heart_bb)
```

Fun with IDLE





Font matters

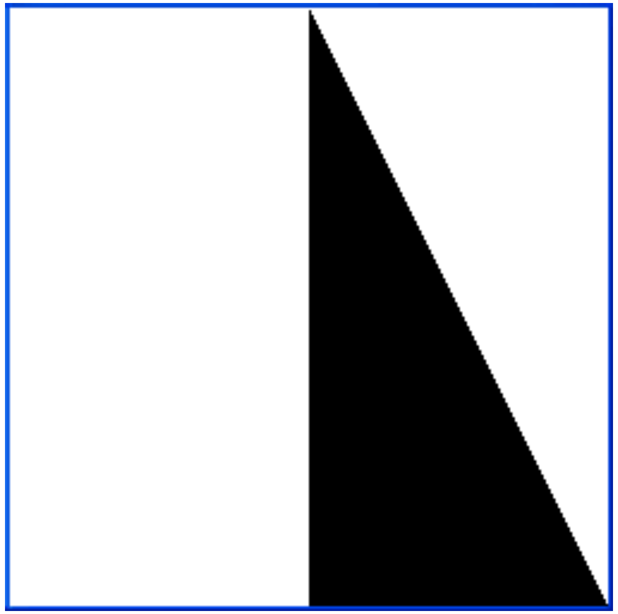
Primitive Operation

Rotating to the Right

operation picture

```
clear_all()
show(quarter_turn_right(sail_bb))
```

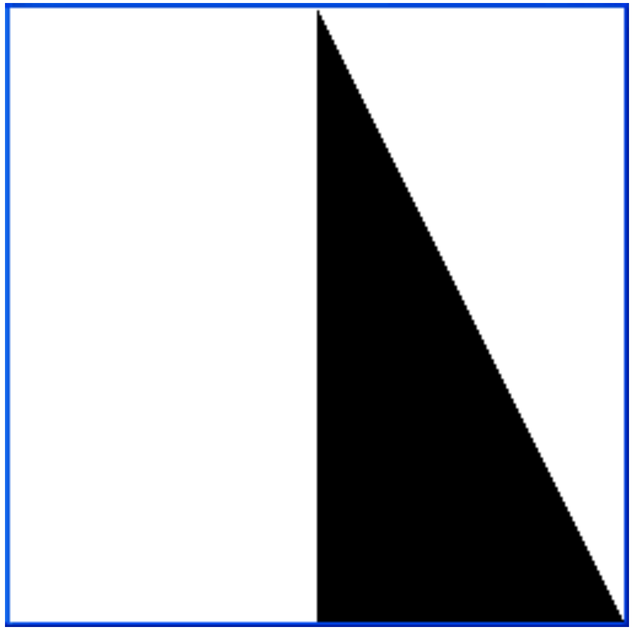
result is
another picture



Derived Operation

Rotating Upside Down

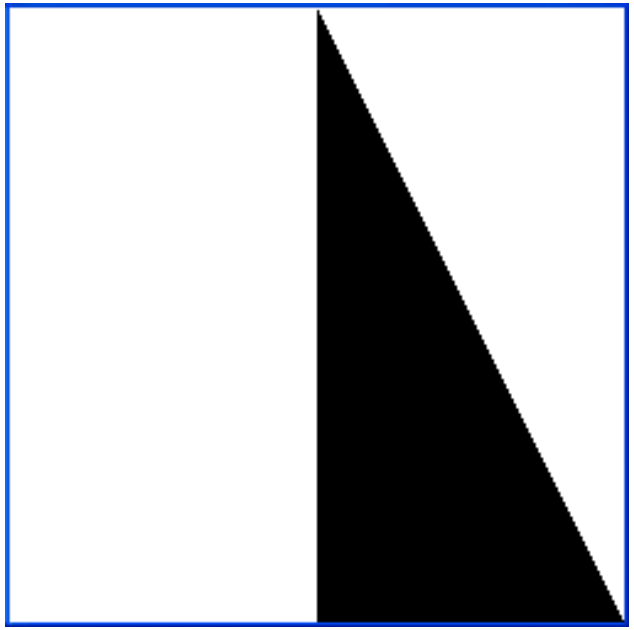
```
def turn_upside_down(pic):
    return quarter_turn_right(
           quarter_turn_right(pic))
clear_all()
show(turn_upside_down(sail_bb))
```



How about

Rotating to the Left?

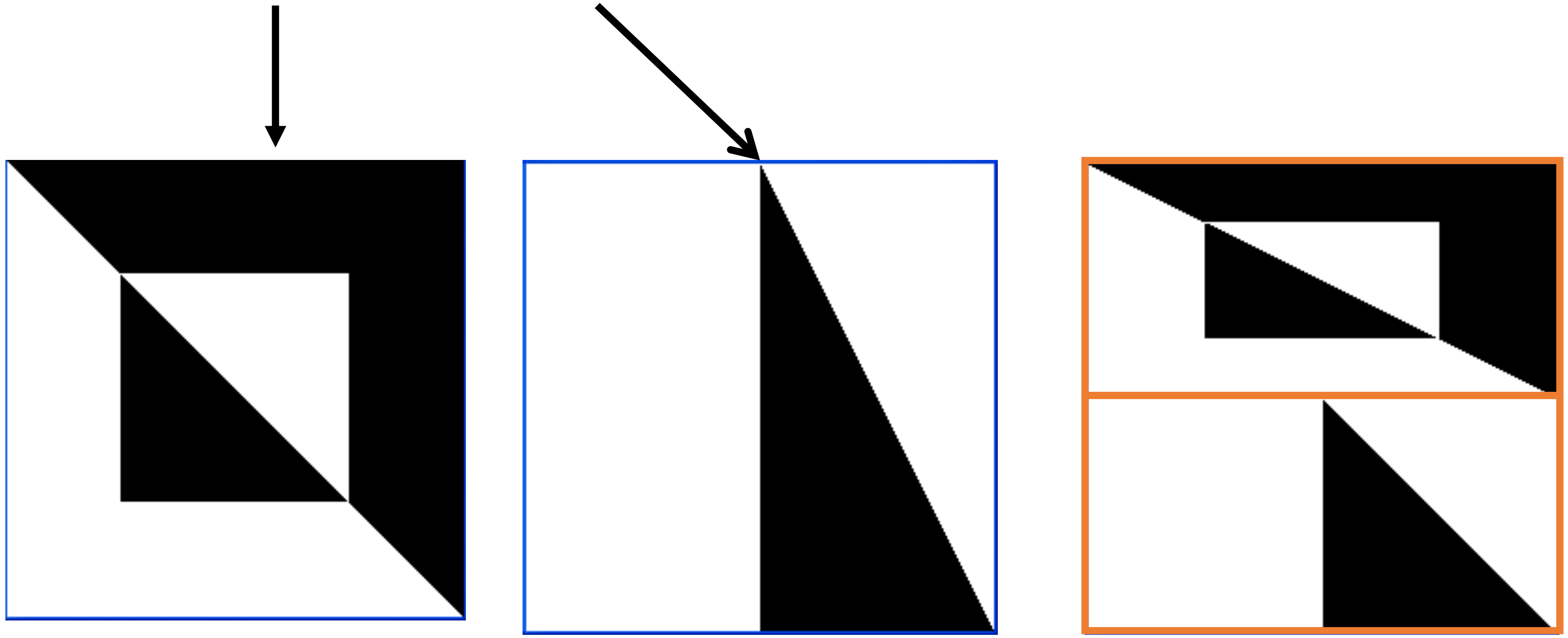
```
def quarter_turn_left(pic):
    return quarter_turn_right(
           quarter_turn_upside_down(pic))

clear_all()
show(quarter_turn_left(sail_bb))
```



Means of Combination

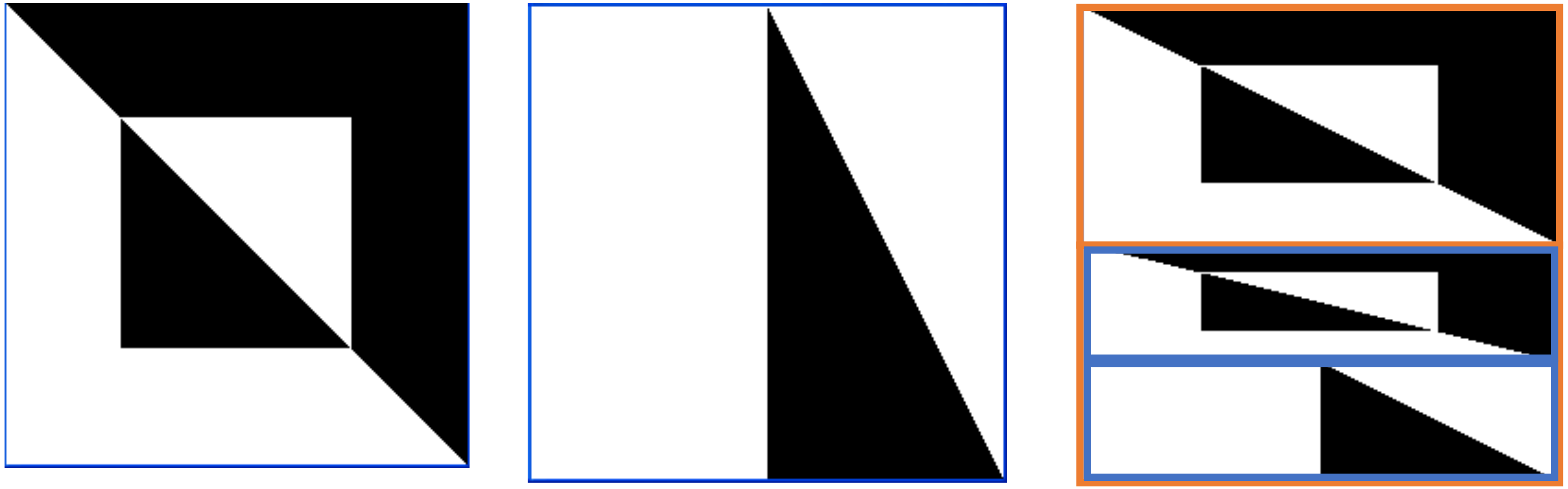
Stacking

```
clear_all()
show(stack(rcross_bb, sail_bb))
```



Multiple Stacking

```
clear_all()
show(stack(rcross_bb,
           stack(rcross_bb,
                 sail_bb)  ))
```



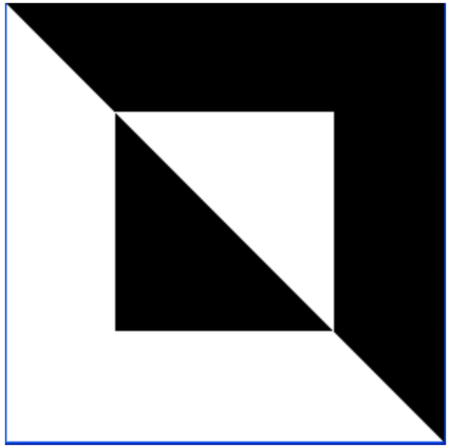
Means of Combination

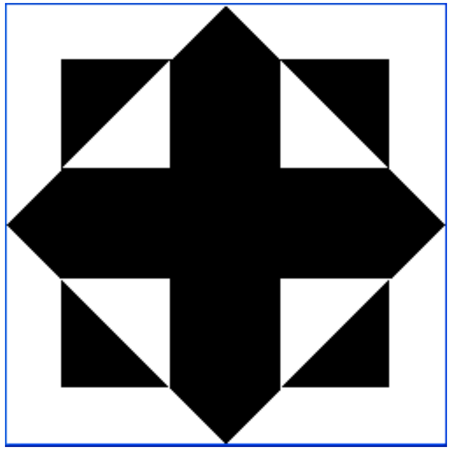
Placing Beside

```
def beside(pic1, pic2):
    return quarter_turn_right(
        stack(quarter_turn_left(pic2),
              quarter_turn_left(pic1)))
```

A complex object

```
clear_all()
show(
  stack(
    beside(
      quarter_turn_right(rcross_bb),
      turn_upside_down(rcross_bb)),
    beside(
      rcross_bb,
      quarter_turn_left(rcross_bb))))
```





Let's give it a name

`make_cross`

```
stack(
  beside(
    quarter_turn_right(rcross_bb),
    turn_upside_down(rcross_bb)),
  beside(
    rcross_bb,
    quarter_turn_left(rcross_bb))))
```

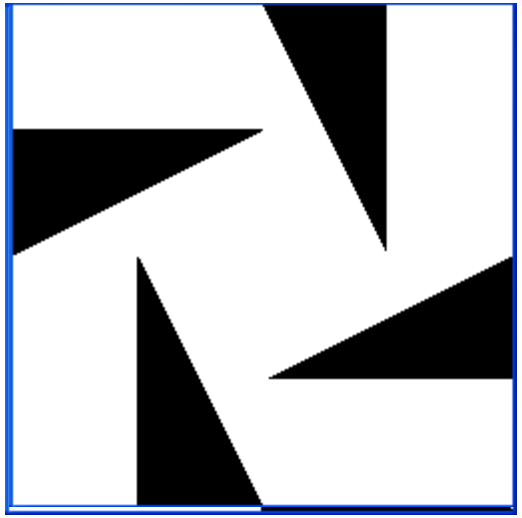
```
stack(
  beside(
    quarter_turn_right(pic),
    turn_upside_down(pic)),
  beside(
    pic,
    quarter_turn_left(pic)))
```

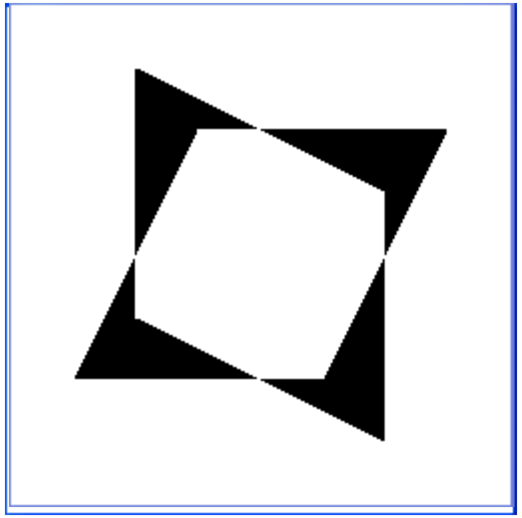
```python
def make_cross(pic):
  return stack(
    beside(
      quarter_turn_right(pic),
      turn_upside_down(pic)),
    beside(
      pic,
      quarter_turn_left(pic)))
```

return vs show

Naming your objects

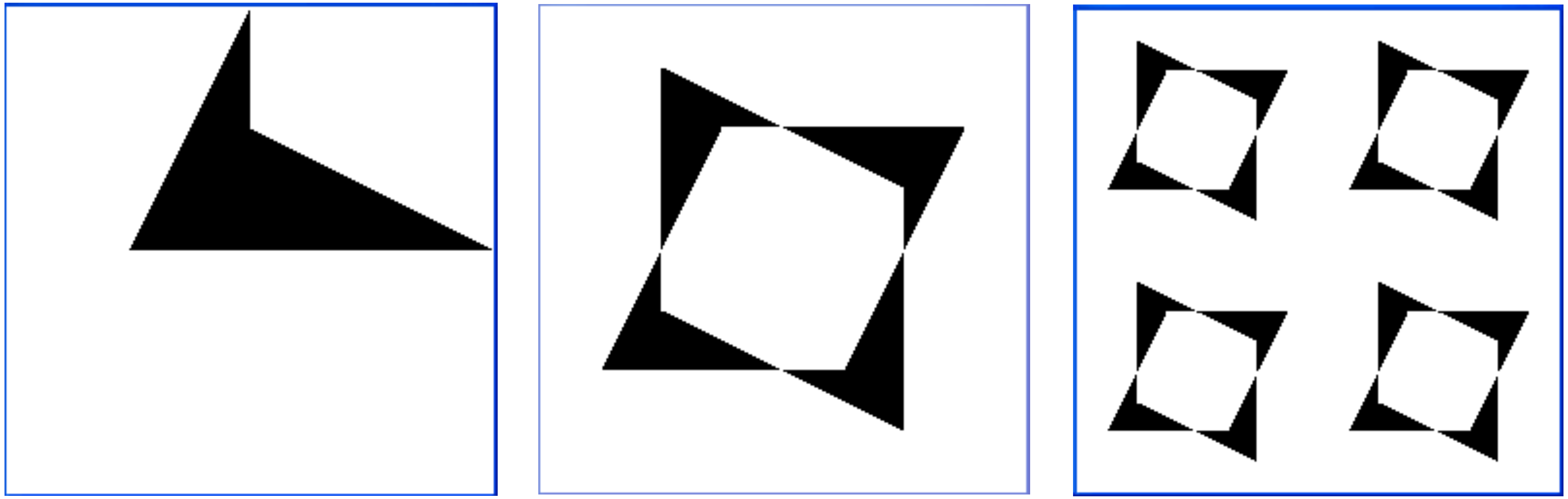
```
clear_all()
my_pic = make_cross(sail_bb)
show(my_pic)

my_pic_2 = make_cross(nova_bb)
show(my_pic_2)
```





Repeating the pattern

```
clear_all()
show(make_cross(make_cross(nova_bb)))
```

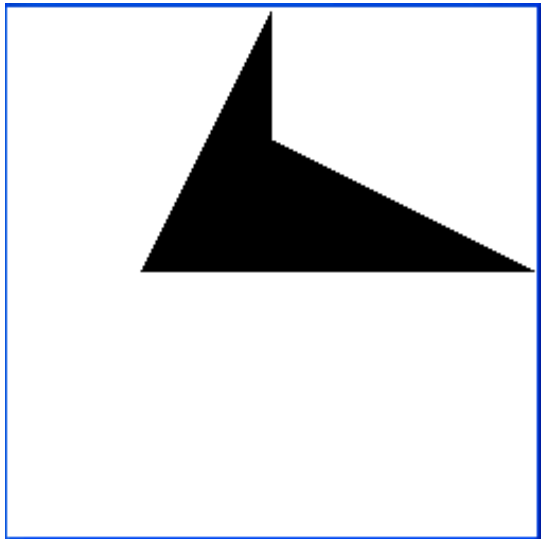


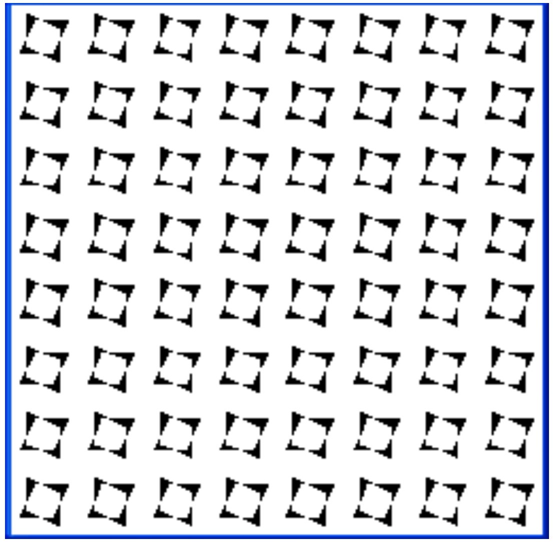
Repeating multiple times

```
clear_all()
def repeat_pattern(n, pat, pic):
    if n == 0:
        return pic
    else:
        return pat(repeat_pattern(n-1, pat, pic))
show(repeat_pattern(4, make_cross, nova_bb))
```

recursion

Qn: What does `repeat_pattern` return?





Anonymous Functions

```
def square(x):
    return x * x
```

input output

```
foo = lambda x: x * x
```

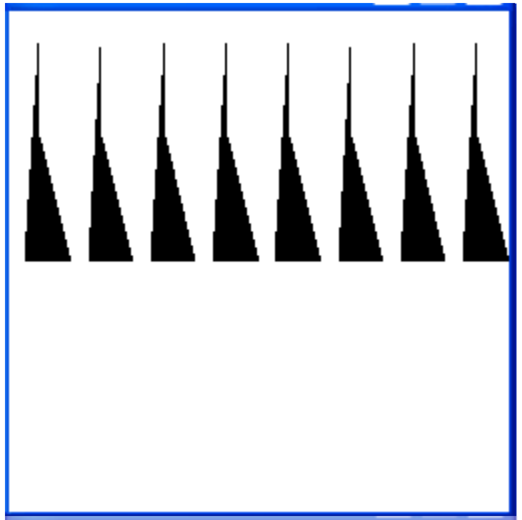
function

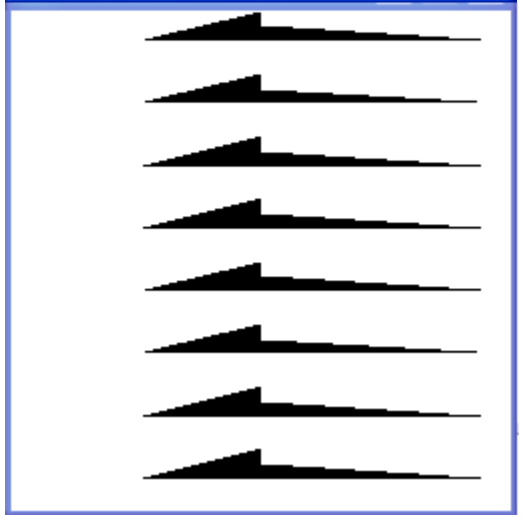
```
foo(1)     1
foo(16)    256
```

New Patterns

```
clear_all()
show(repeat_pattern(3,
     lambda pic: beside(pic, pic),
     nova_bb))

clear_all()
show(repeat_pattern(3,
      lambda pic: stack(pic, pic),
      nova_bb))
```

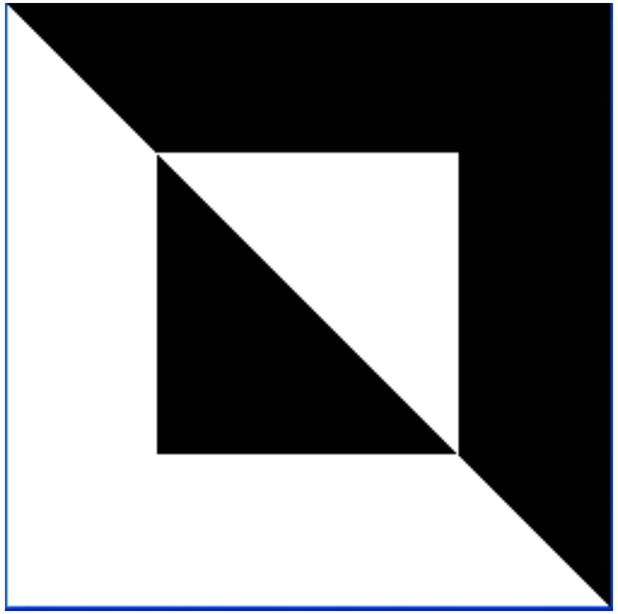
anonymous function

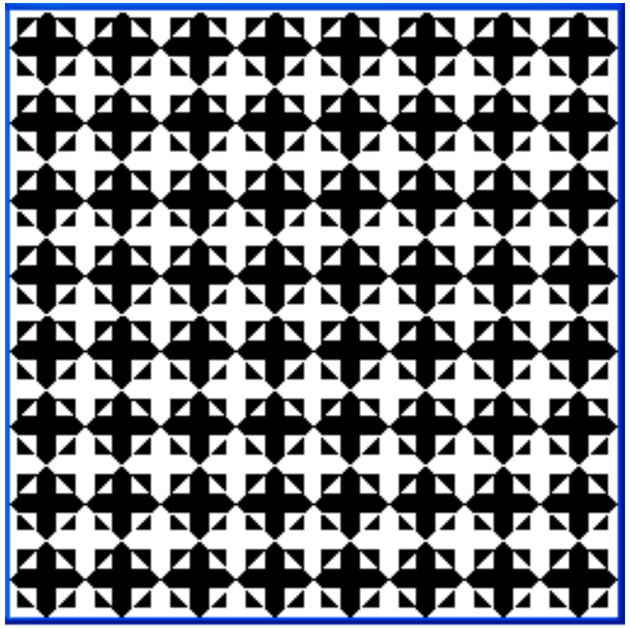




Another nice pattern

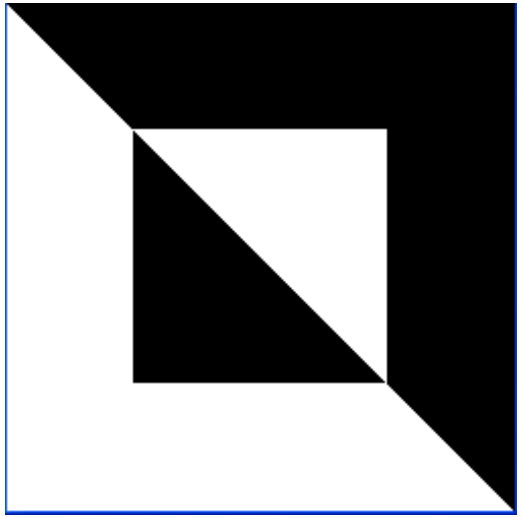
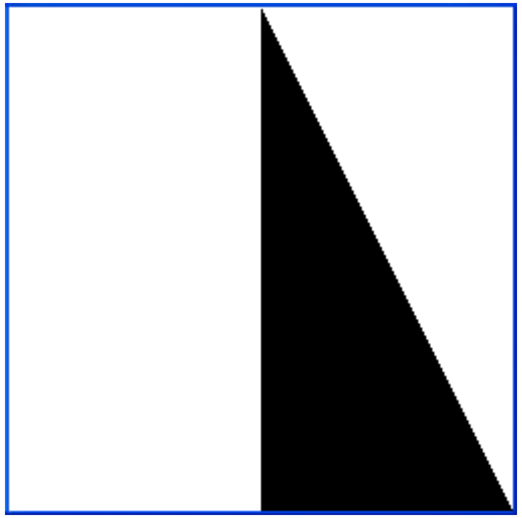
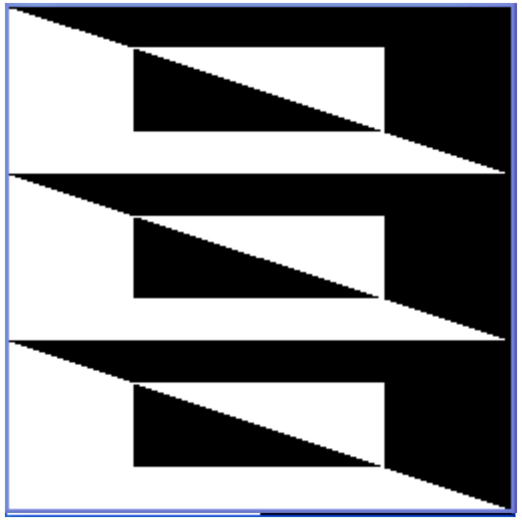
```
clear_all()
show(repeat_pattern(4, make_cross, rcross_bb))
```





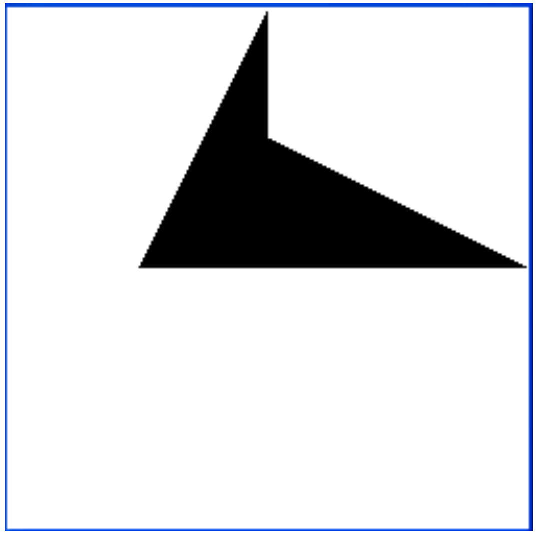
What about 3 rows?

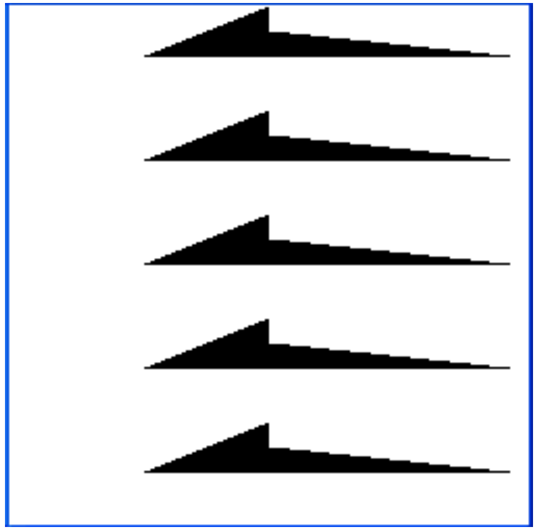
```
clear_all()
show(stack_frac(1/3, rcross_bb, sail_bb))
clear_all()
show(stack_frac(1/3, rcross_bb,
                     stack(rcross_bb, rcross_bb)))
```

Repeating n times

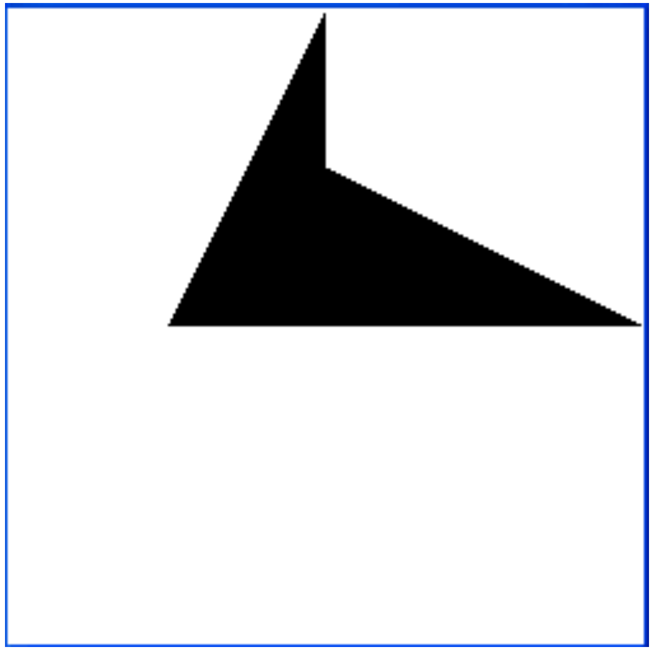
```
def stackn(n, pic):
    if n == 1:
        return pic
    else:
        return stack_frac(1/n,
                          pic,
                          stackn(n-1, pic))

clear_all()
show(stackn(3, nova_bb))

clear_all()
show(stackn(5, nova_bb))
```

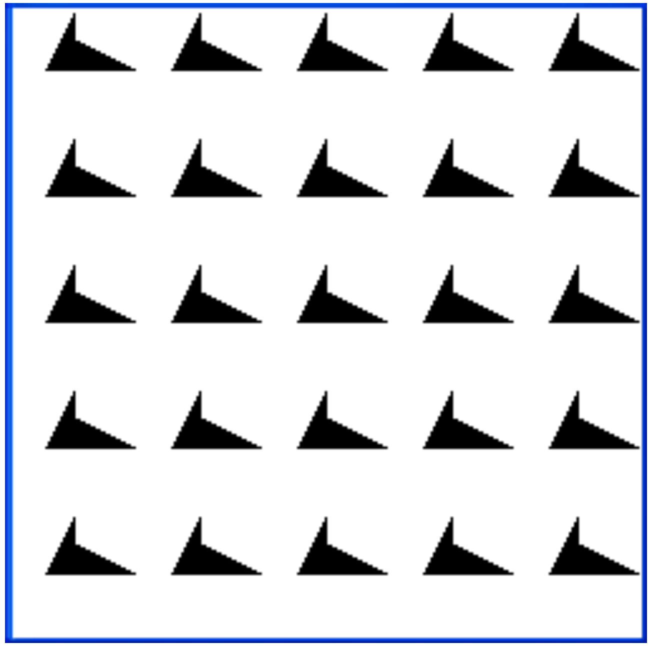




A rectangular quilting pattern

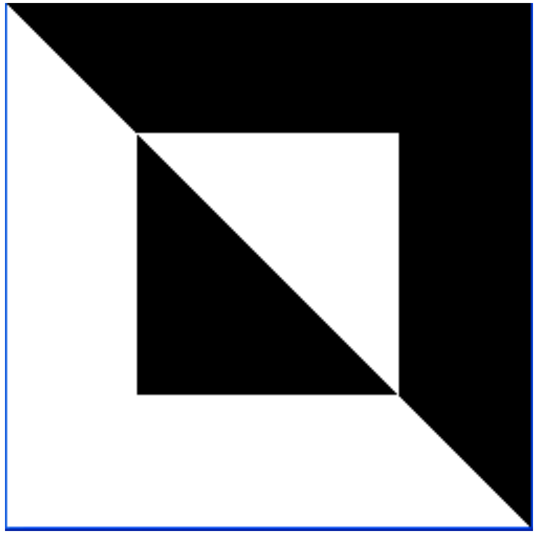
```
clear_all()
show(stackn(5, quarter_turn_right(
                  stackn(5, quarter_turn_left(nova_bb)))))
```

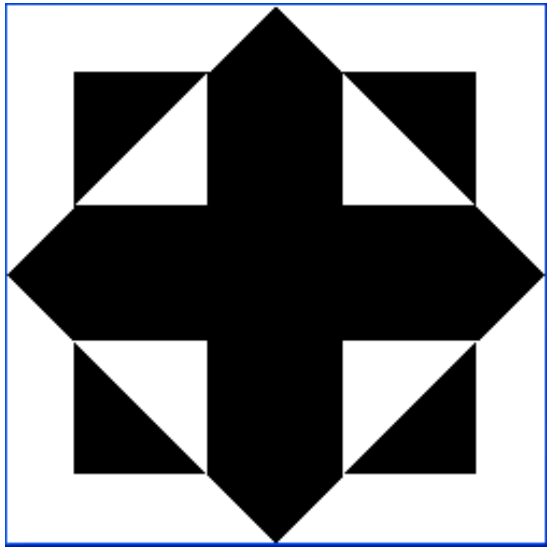


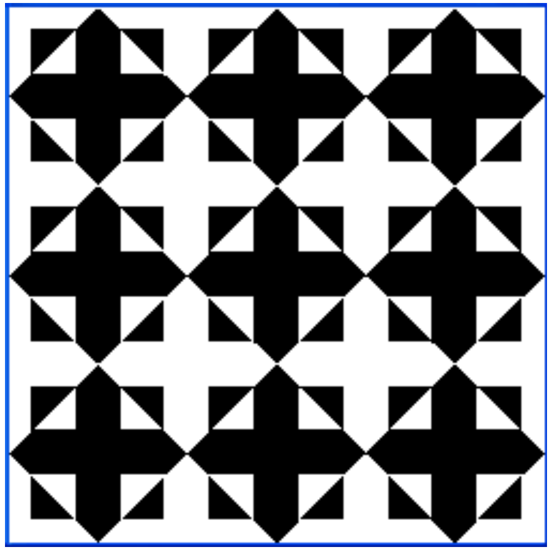


A rectangular quilting proc

```
def nxn(n, pic):
    return stackn(n, quarter_turn_right(
                 stackn(n, quarter_turn_left(pic))))
clear_all()
show(nxn(3, make_cross(rcross_bb)))
```







After all this…

No idea how a picture is represented

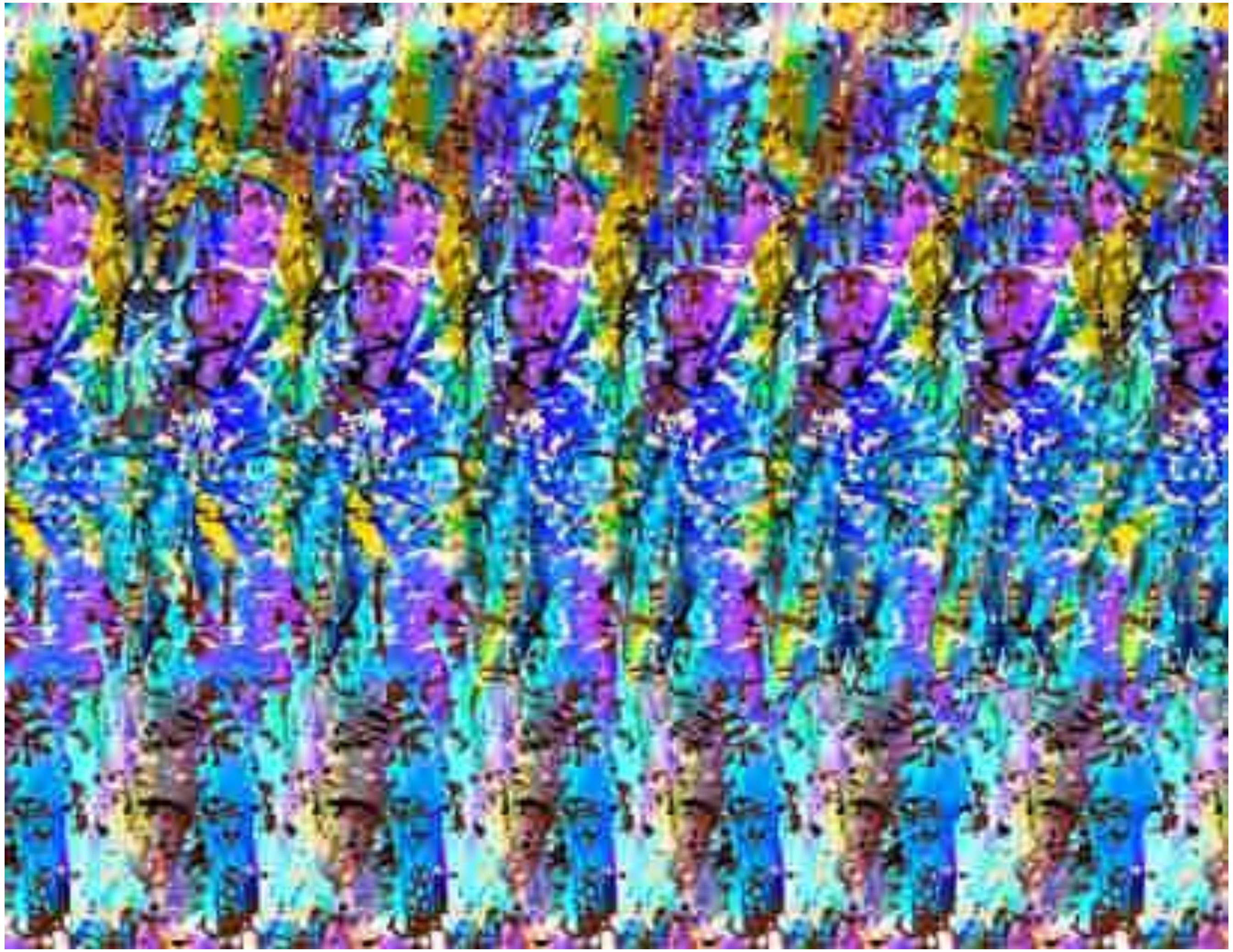
No idea how the operations do their work

Yet, we can build complex pictures

This is

Functional Abstraction

We can make Sterograms!

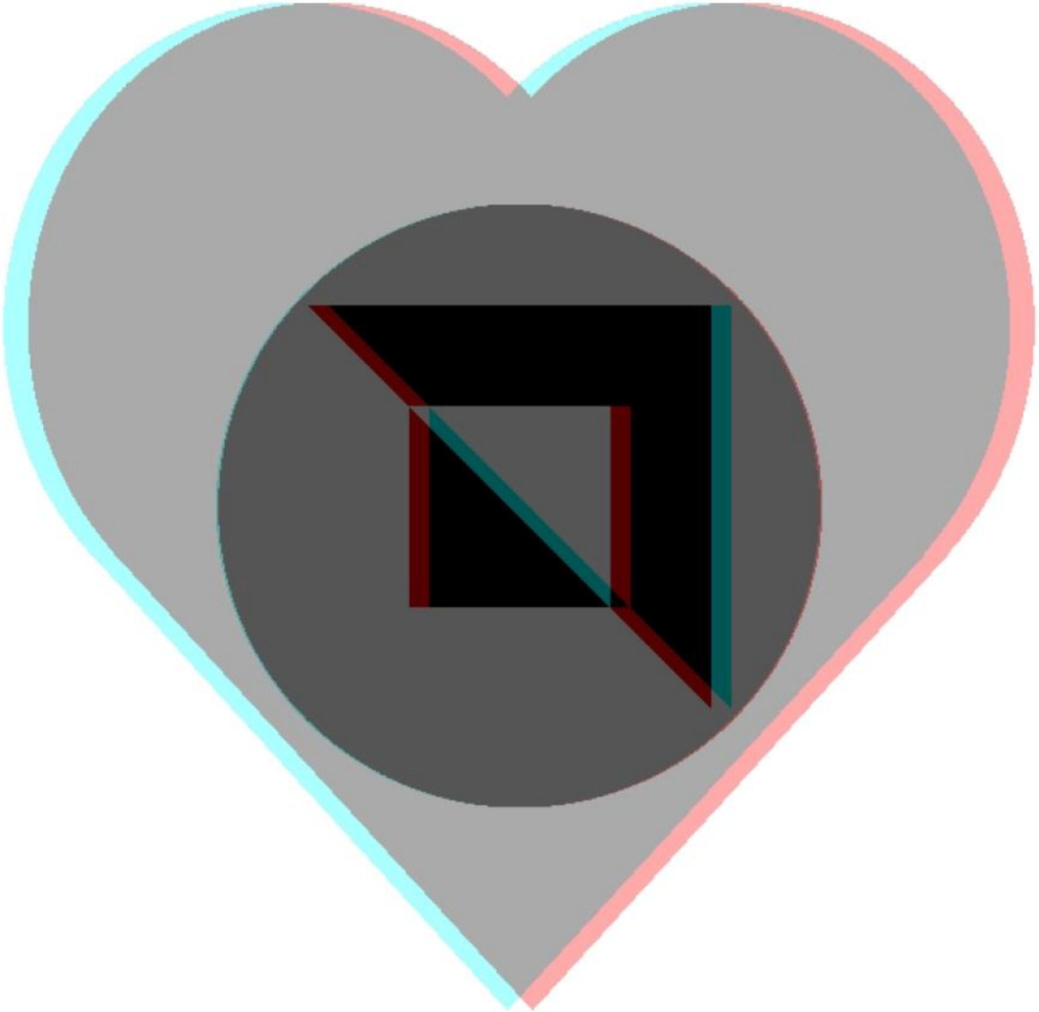


Black Box

Functional Abstraction

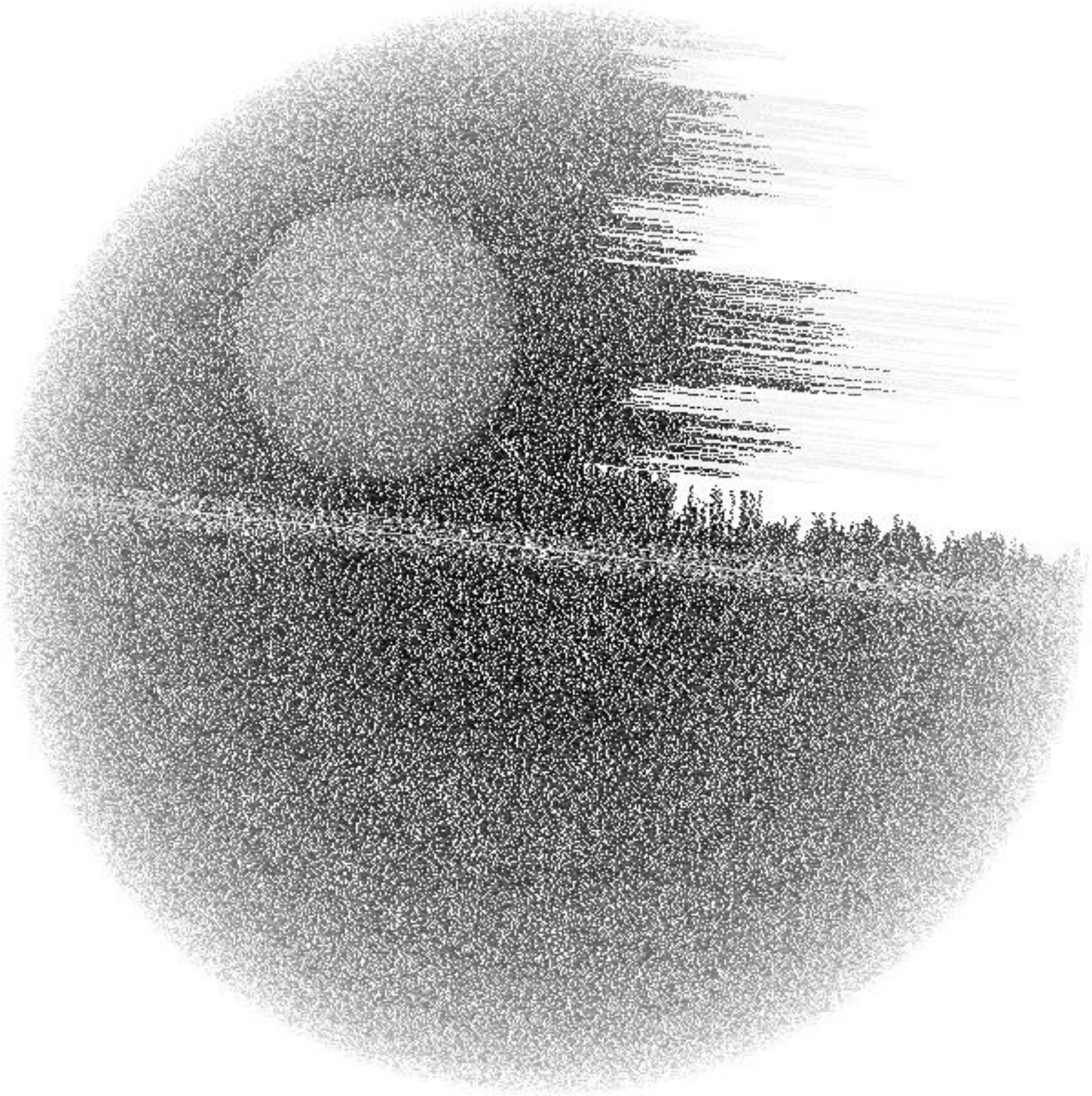
Can't see stereograms?

Anaglyphs



And if you think this is cool…

You ain't seen nothing yet!



What have we learnt?

WHAT

Functional Abstraction = Black-box

HOW

`def` and `lambda`

Functions are objects

(in Python)

WHY?

Help us manage

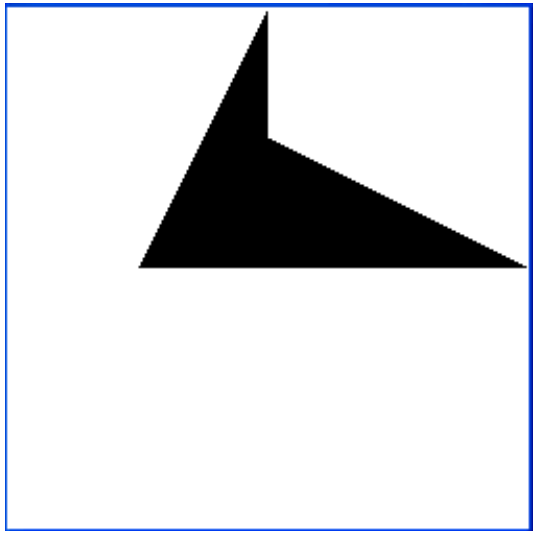
complexity

Allow us to focus on high-level problem solving

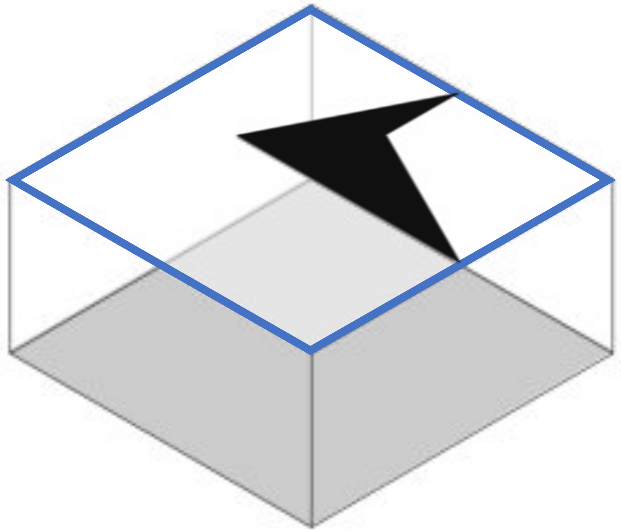
Creating 3D objects

We use greyscale to represent depth

- Black is nearest to you
- White is furthest away

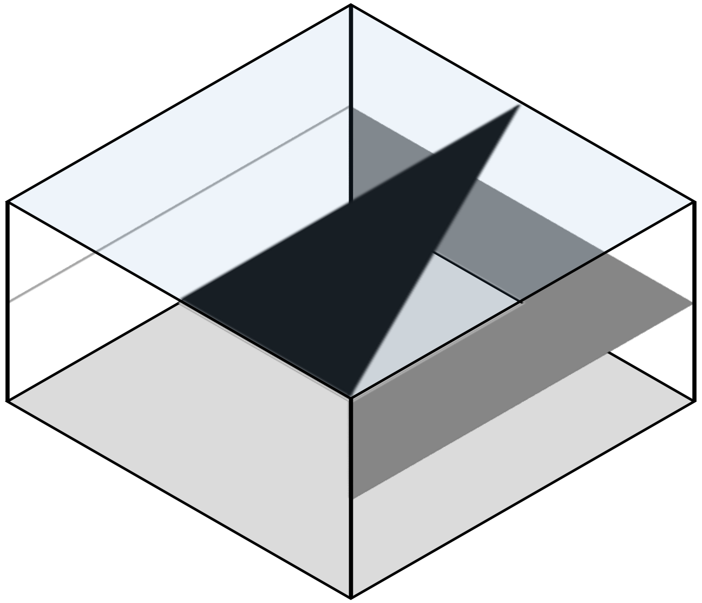


means



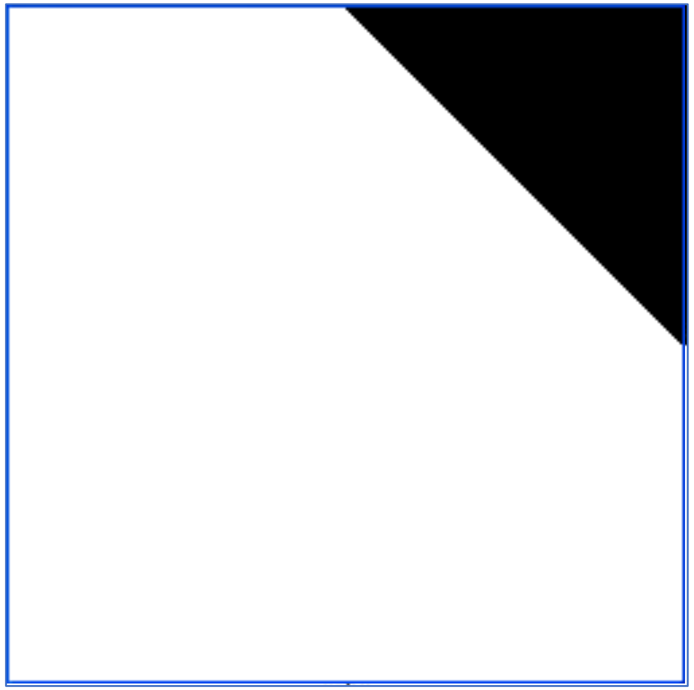
Overlay Operation

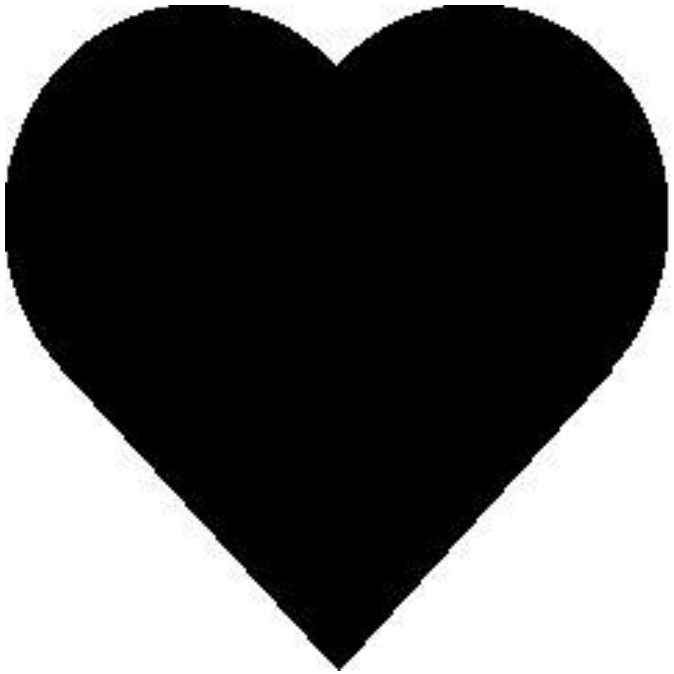
```
clear_all()
show(overlay(sail_bb, rcross_bb))
```



Advanced Overlay Operation

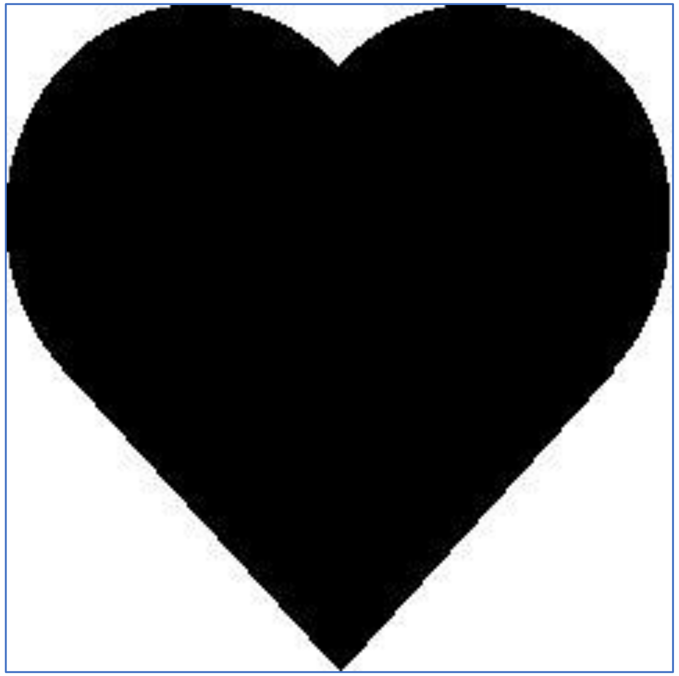
```
clear_all()
show(overlay_frac(1/4, corner_bb, heart_bb))
```

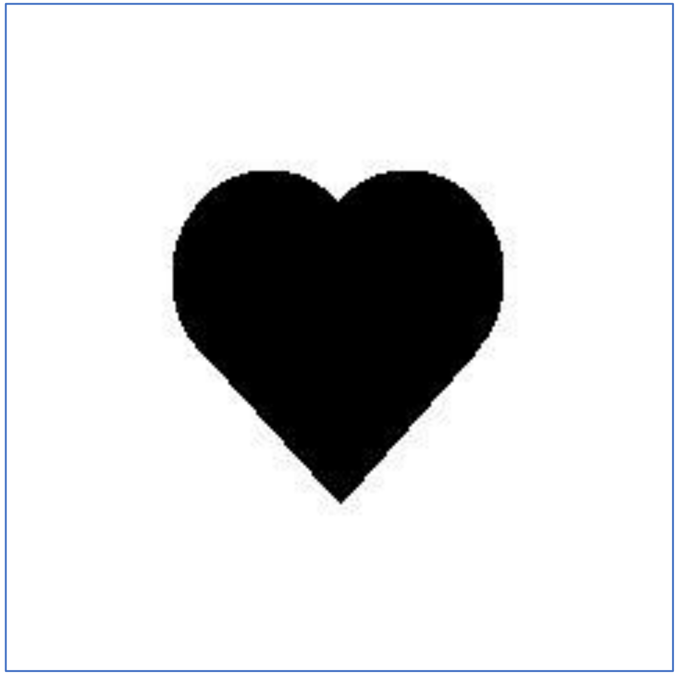




Scaling

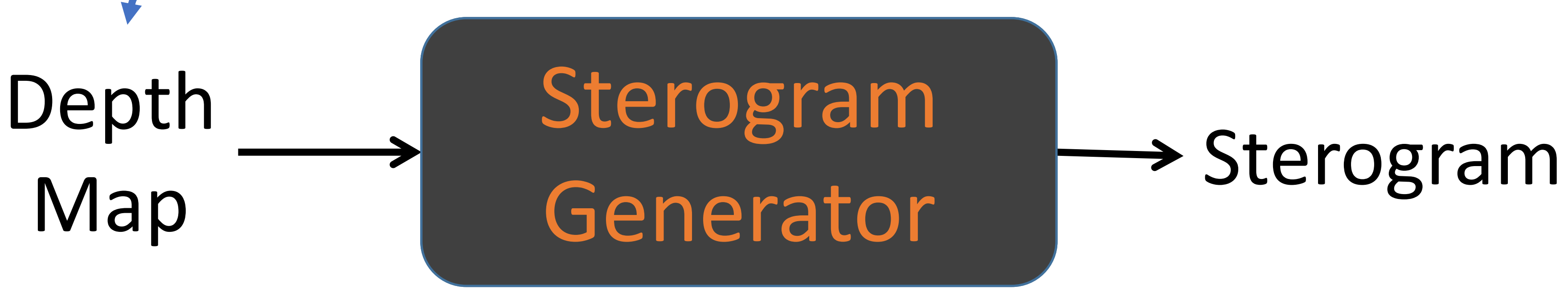
```
clear_all()
show(scale(1/2, heart_bb))
```

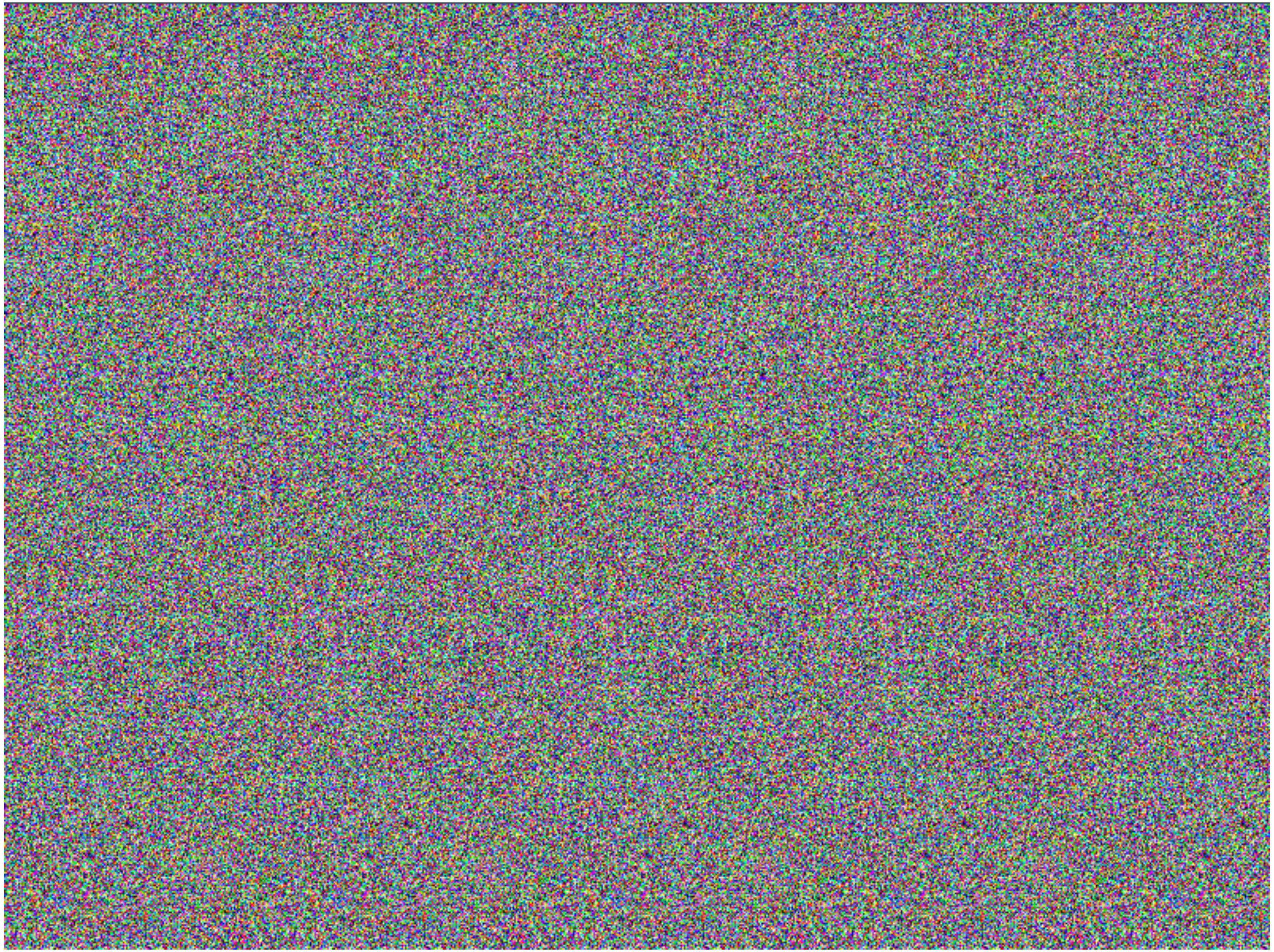




Recall

```
stereogram(scale(1/2, heart_bb))
```



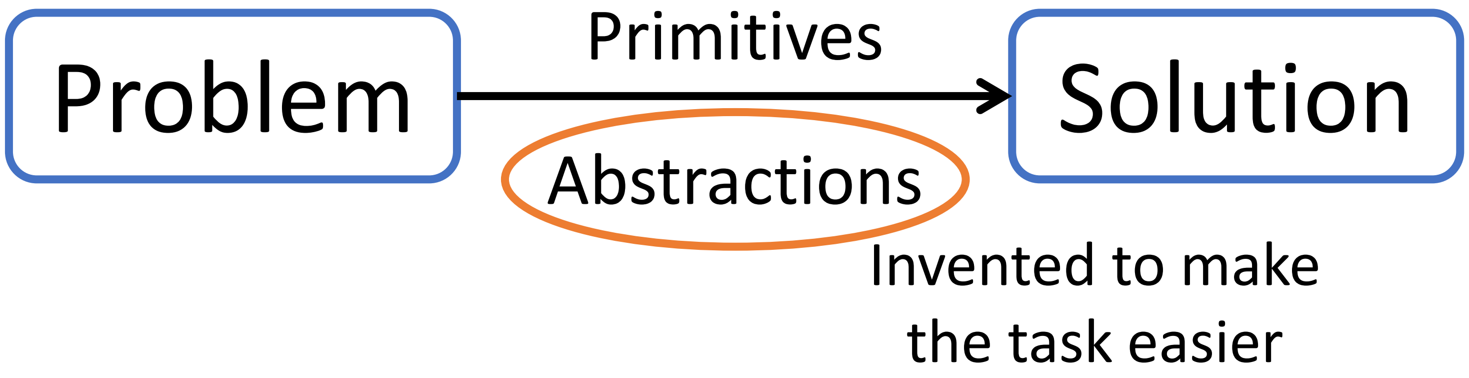


<Break>

Managing Complexity

Computers will follow orders precisely

Abstractions

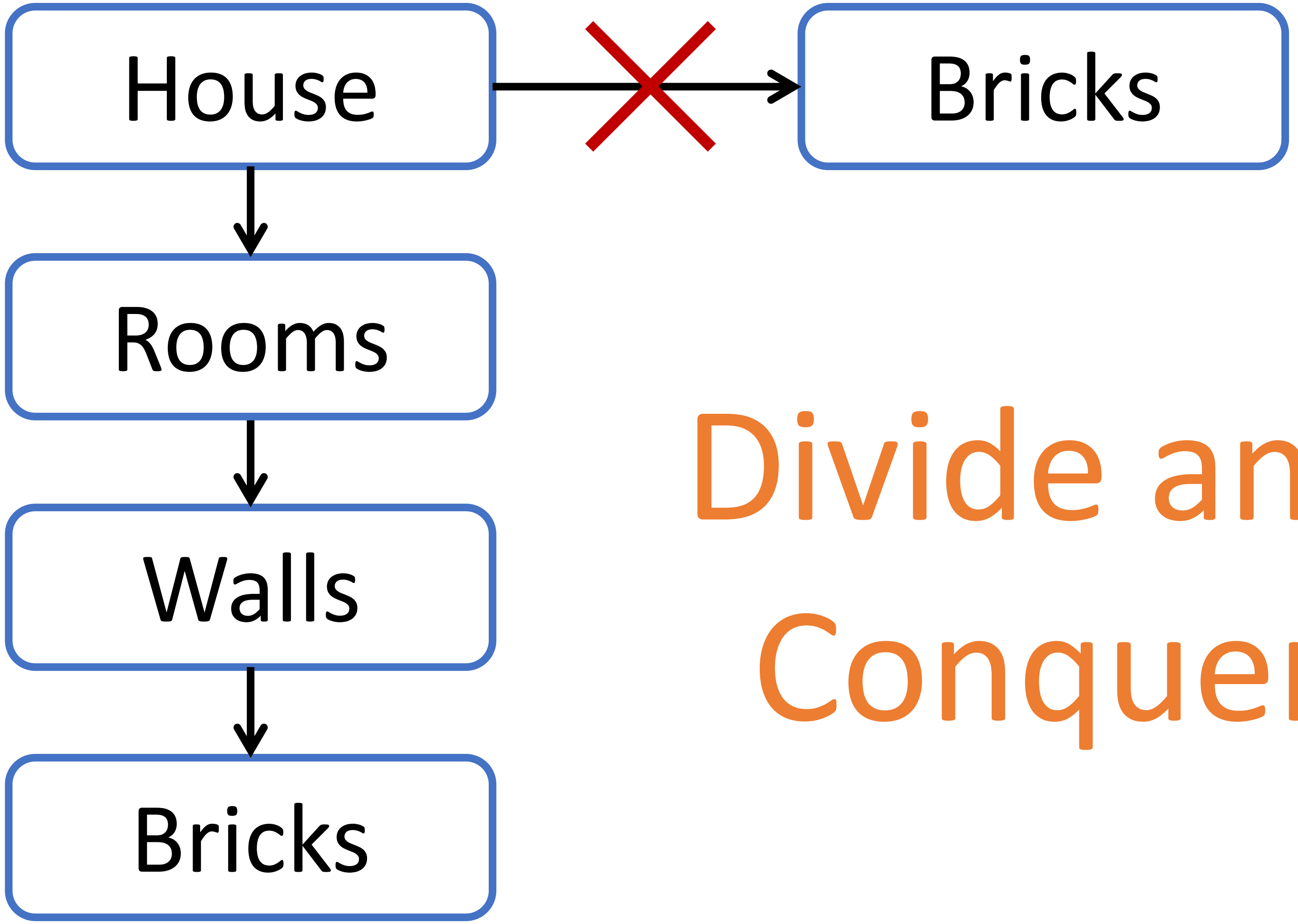


What makes a good abstraction?

Good Abstraction

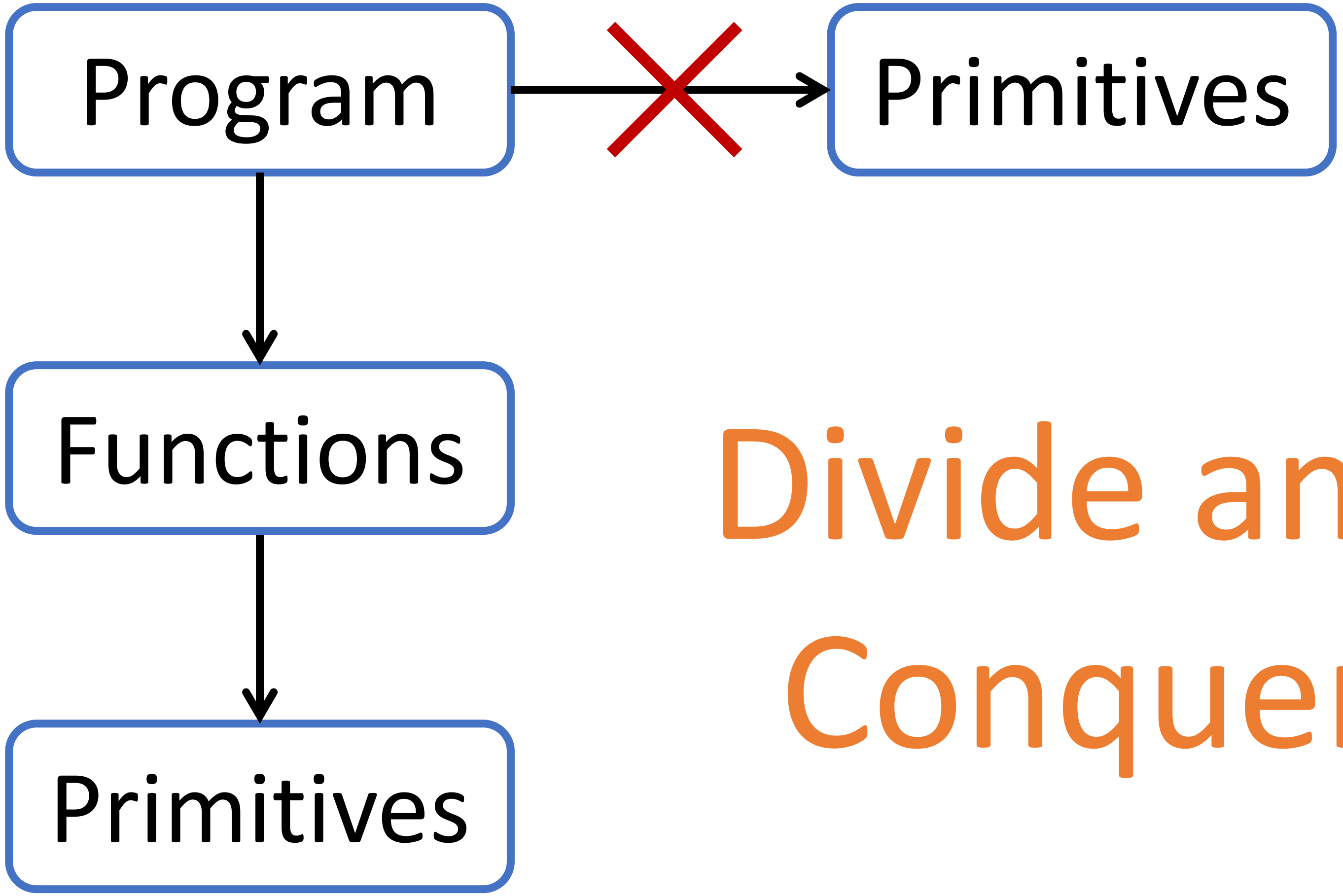
1. Makes it more natural to think about tasks and subtasks

Example



Divide and Conquer

Programming



Divide and Conquer

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand

Compare:

```
def hypotenuse(a, b):
    return sqrt((a*a) + (b*b))
```

Versus:

```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))
def sum_of_squares(x, y):
    return square(x) + square(y)
def square(x):
    return x * x
```

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns

```
stack(
  beside(
    quarter_turn_right(rcross_bb),
    turn_upside_down(rcross_bb)),
  beside(
    rcross_bb,
    quarter_turn_left(rcross_bb)))
```

```
stack(
  beside(
    quarter_turn_right(pic),
    turn_upside_down(pic)),
  beside(
    pic,
    quarter_turn_left(pic)))
```

```python
def make_cross(pic):
  return stack(
    beside(
      quarter_turn_right(pic),
      turn_upside_down(pic)),
    beside(
      pic,
      quarter_turn_left(pic)))
```

Allows Code Reuse

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse

- Function `square` used in `sum_of_squares`.
- `square` can also be used in calculating area of circle.

Another Example

Function to calculate area of circle given the radius

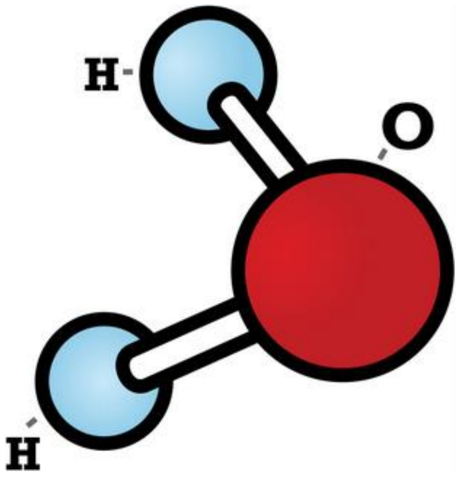
```
pi = 3.14159
def circle_area_from_radius(r):
  return pi * square(r)
```

given the diameter:

```
def circle_area_from_diameter(d):
  return circle_area_from_radius(d/2)
```

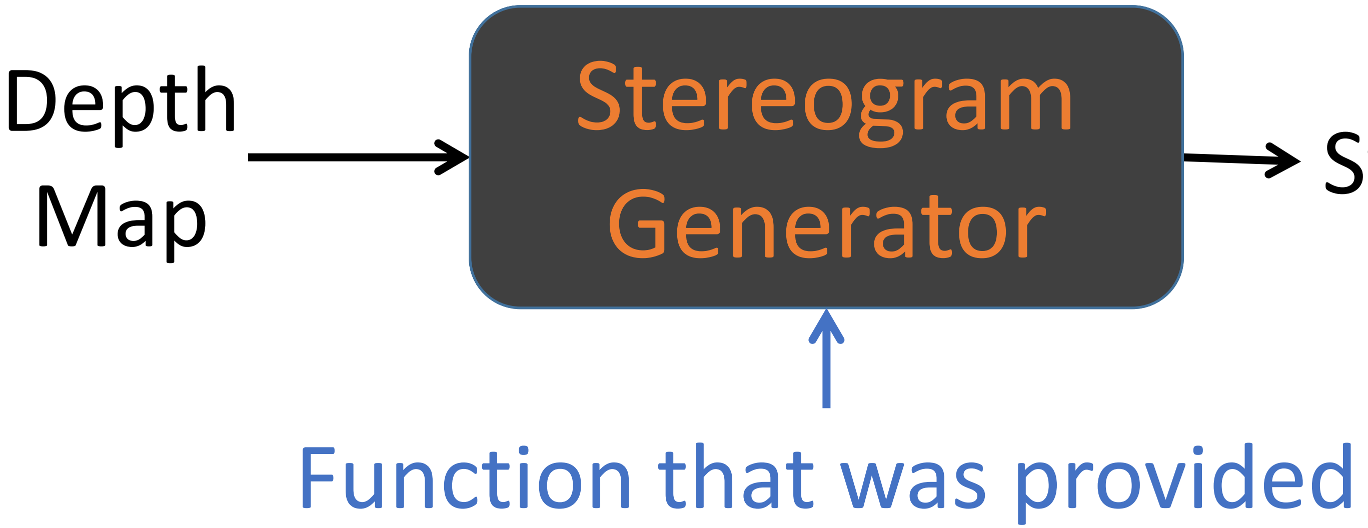
Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details



Water molecule represented as 3 balls

Ok for some chemical analyses, inadequate for others.


Depth
Map
Stereogram
Generator
Stereogram
Function that was provided

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation

Recap

Functional Abstraction

=

Black Box

No need to know how a car works to drive it!

Functional Abstraction

Separates specification from implementation

Specification: WHAT

Implementation: HOW

Example

```
def square(x):
    return x * x
```

```
def square(x):
   return exp(double(log(x)))
```

```
def double(x): return x + x
```

To think about

Why would we want to implement a function in different ways?

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation
7. Makes debugging (fixing errors) easier

Good Abstraction

Where is the bug?

```
def hypotenuse(a, b):
  return sqrt(sum_of_squares(a, b))

def sum_of_squares(x, y):
  return square(x) + square(y)

def square(x): return x + x
```

```
def hypotenuse(a, b):
  return sqrt((a + a) * (b + b))
```

Variable Scope

Variable Scope

```
x = 10
def square(x): return x * x
def double(x): return x + x
def addx(y): return y + x

square(20)
square(x)
addx(5)
```

Which x ?

Variable Scope

formal parameter

```
def square(x):
    return x * x
```

body

A function definition binds its formal parameters.

i.e. the formal parameters are visible only inside the definition (body), not outside.

Variable Scope

formal parameter

```
def square(x):
    return x * x
```

body

- Formal parameters are bound variables.
- The region where a variable is visible is called the scope of the variable.
- Any variable that is not bound is free.

Variable Scope

```
x = 10

def square(x):
    return x * x

def double(x):
    return x + x

def addx(y):
    return y + x
```

x is bound

x is bound

y is bound, x is free

Example

```
pi = 3.14169

def circle_area_from_radius(r):
    pi = 22/7    #local pi
    return pi * square(r)
```

Which pi?

Block Structure

```
def hypotenuse(a, b):

    def sum_of_squares():
        return square(a) + square(b)

    return math.sqrt(sum_of_squares())
```

The variables a and b in sum_of_squares refer to the formal parameters of hypotenuse.

<u>Hides</u> irrelevant details (sum_of_squares) from the user of hypotenuse.

Wishful Thinking

WHAT

Top-down design approach:

Pretend you have whatever you need

WHY

Easier to think with in the goal in mind

Analogy

Suppose you are to build a house.

Where do you start?

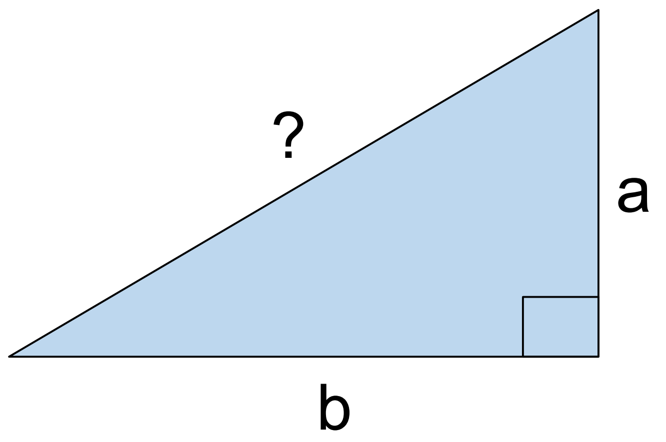
Individual bricks

Building plan

Example

Suppose you want to compute hypotenuse

```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))

def sum_of_squares(x, y):
    return square(x) + square(y)

def square(x):
    return x * x
```



Another Example

Comfort Delgro, the largest taxi operator in Singapore, determines the taxi fare based on distance traveled as follows:

• For the first kilometre or less:	$2.40
• Every 200 metres thereafter or less up to 10 km:	$0.10
• Every 150 metres thereafter or less after 10 km:	$0.10

Problem:

Write a Python function that computes the taxi fare from distance travelled.

How do we start?

Formulate the problem

Function

Needs a name

Pick an appropriate name

(not foo)

Formulate the problem

distance → Taxi Fare → fare

- What data do you need? (be thorough)
- Where would you get it? (argument/ computed?)

Results should be unambiguous

- What other abstractions may be useful?
- Ask the same questions for each abstraction.

How can the result be computed from data?

1. Try simple examples
2. Strategize step by step
3. Write it down and refine

Solution

- What to call the function? taxi_fare
- What data are required? distance
- Where to get the data? function argument
- What is the result? fare

How can the result be computed from data?

- e.g. #1: distance = 800 m, fare = $2.40
- e.g. #2: distance = 3,300 m

  $$\text{fare} = \$2.40 + \lceil 2300/200 \rceil \times \$0.10$$
  $$= \$3.60$$

- e.g. #3: distance = 14,500 m

  $$\text{fare} = \$2.40 + \lceil 9000/200 \rceil \times \$0.10 + \lceil 4500/150 \rceil \times \$0.10 = \$9.90$$

Pseudocode

Case 1: distance <= 1000

fare = \$2.40

Case 2: 1000 < distance <= 10,000

$fare = \$2.40 + \$0.10 * \lceil (distance - 1000)/200 \rceil$

What's this?

Case 3: distance > 10,000

$fare = \$6.90 + \$0.10 * \lceil (distance - 10{,}000)/150) \rceil$

Note: the Python function `ceil` rounds up its argument. `math.ceil(1.5) = 2`

Solution

```
def taxi_fare(distance):  # distance in metres
  if distance <= 1000:
    return 2.4
  elif distance <= 10000:
    return 2.4 + (0.10 * ceil((distance - 1000) / 200))
  else:
    return 6.9 + (0.10 * ceil((distance - 10000) / 150))

# check: taxi_fare(3300) = 3.6
```

Can we improve this solution?

Coping with Change

What if…

1. the starting fare increases?
2. stage distance changes?
3. increment amount changes?

CAB CONFUSION

Singapore has many different types of taxis plying the roads, all with different flag-down rates. **LIM YONG** and **BRYANDT LYN** help sort through the choices available.

Flag-down rates for first kilometre or less. Figures in brackets denote subsequent fare rates for:
- Every 400m thereafter or less up to 10km
- Every 350m thereafter or less after 10km
- Every 45 seconds of waiting or less

NOTE: Fare comparisons do not take into account surcharges, which vary by company, time and location. Fares correct as at Nov 20.

Avoid Magic Numbers

It is a terrible idea to hardcode numbers (magic numbers):

- Hard to make changes in future

Define abstractions to hide them!

Solution v2

```
def taxi_fare(distance): # distance in metres
    if distance <= stage1:
        return start_fare
    elif distance <= stage2:
        return start_fare + (increment * ceil((distance - stage1) / block1))
    else:                                    recursive call
        return taxi_fare(stage2) + (increment * ceil((distance - stage2) / block2))
stage1 = 1000
stage2 = 10000
start_fare = 2.4
increment = 0.1
block1 = 200
block2 = 150
```

in 2018

```
def taxi_fare(distance): # distance in metres
    if distance <= stage1:
        return start_fare
    elif distance <= stage2:
        return start_fare + (increment * ceil((distance - stage1) / block1))
    else:
        return taxi_fare(stage2) + (increment * ceil((distance - stage2) / block2))
stage1 = 1000
stage2 = 10000
start_fare = 3.7
increment = 0.22
block1 = 400
block2 = 350
```

Summary

- Functional Abstraction
- Good Abstractions
- Variable Scoping
- Wishful Thinking

Recitation Thursday/Friday