CS1010S Programming Methodology

Lecture 2

Functional Abstraction

18 Jan 2023

Admin Matters

- Recitation classes start tomorrow.
- Late Policy
  - < 10 min: OK
  - < 24 hours: -10%
  - > 24 hours: -20%
- Don’t forget to click on finalise to submit the missions!
- Ask early for extensions if needed.

Don’t Stress

But please do your work

Try not to submit at 23:58

DO NOT plagiarize!

Quick Revision!

Variables

- Each variable has:
  - `Name`
  - `Value`
  - `ID`
- Every variable can hold data of a *single kind* at a given time.

Operators

- Assignment

```
a = 5
```

- Equality testing

```
a == 5
```

- Not equal

```
a != 5
```

#Comments

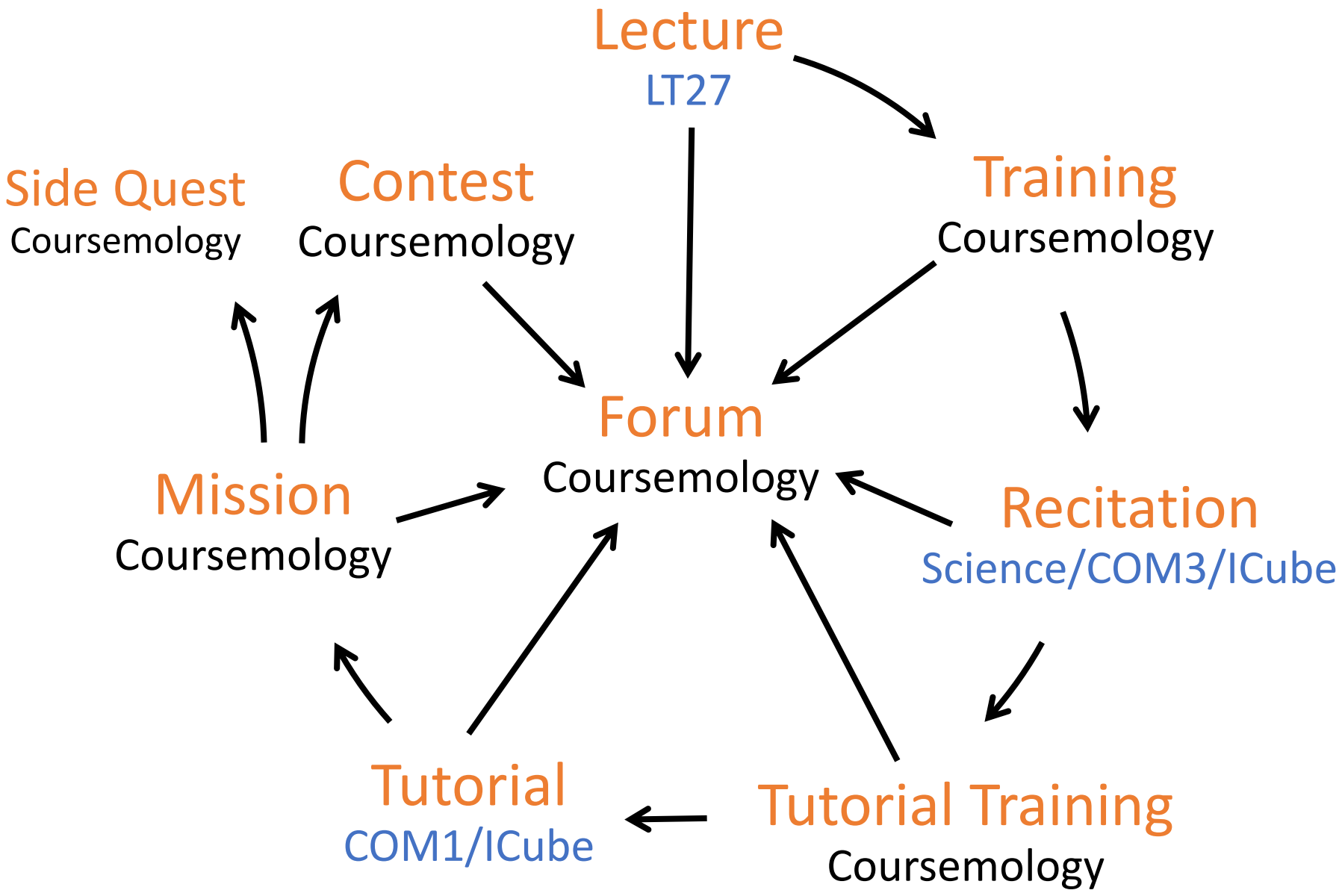
```
# this is not a hashtag!

>>> print("Good to go")
"Good to go"

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

Python Imaging Library

What's this?

```
from PIL import *
```

(Mission 0)

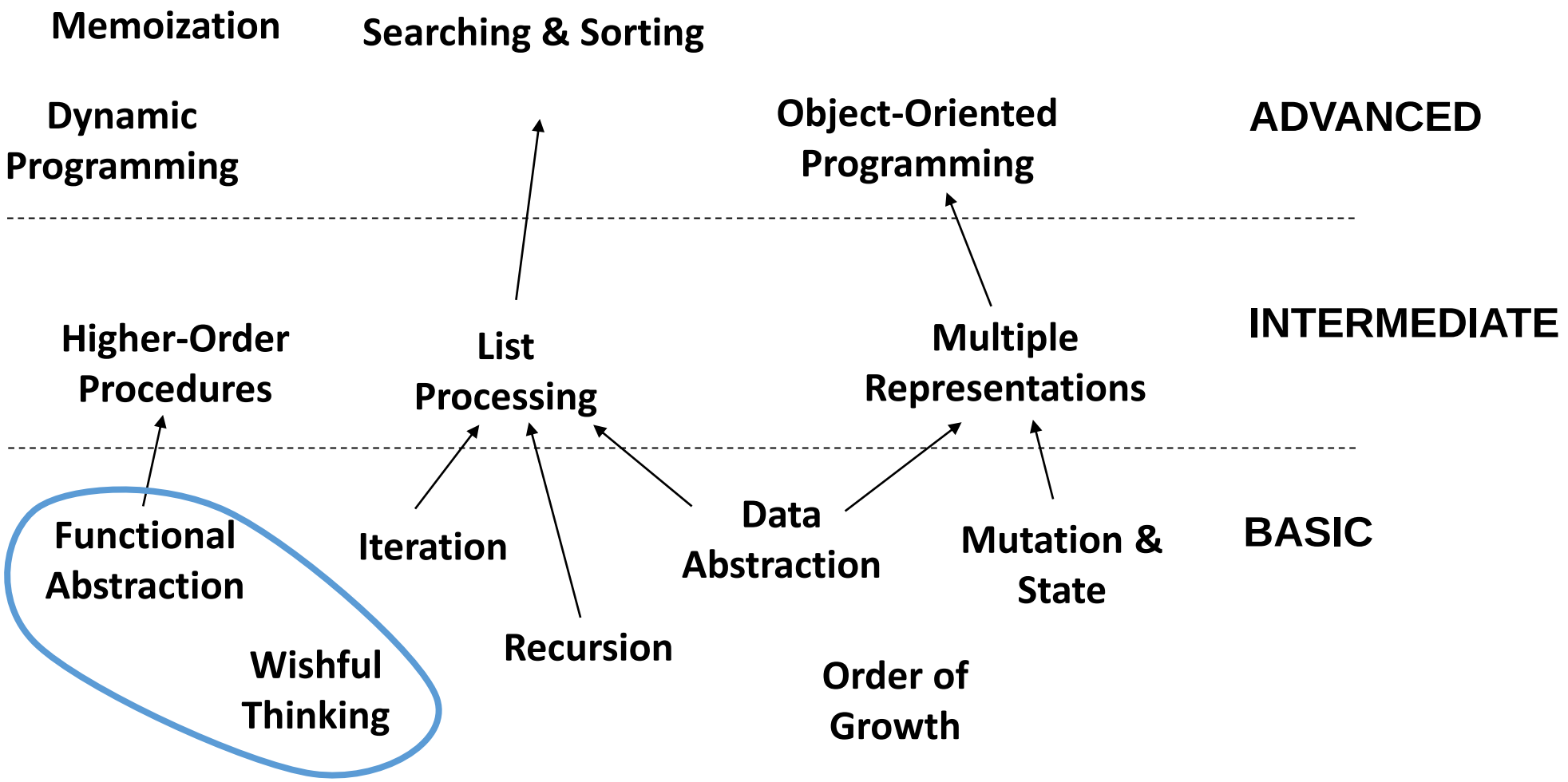

Lecture
LT27
Side Quest
Coursemology
Contest
Coursemology
Training
Coursemology
Forum
Coursemology
Mission
Coursemology
Recitation
Science/COM3/ICube
Tutorial
COM1/ICube
Tutorial Training
Coursemology

Trainings

Please don't anyhow hantam

Computational Thinking

CS1010S Road Map

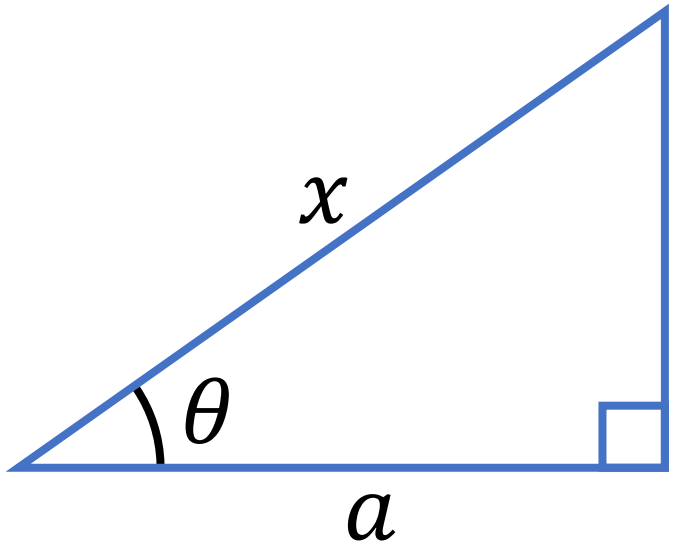


Fundamental **concepts** of computer programming

Functional Abstraction

What is a function?

Functions aren’t new to us!



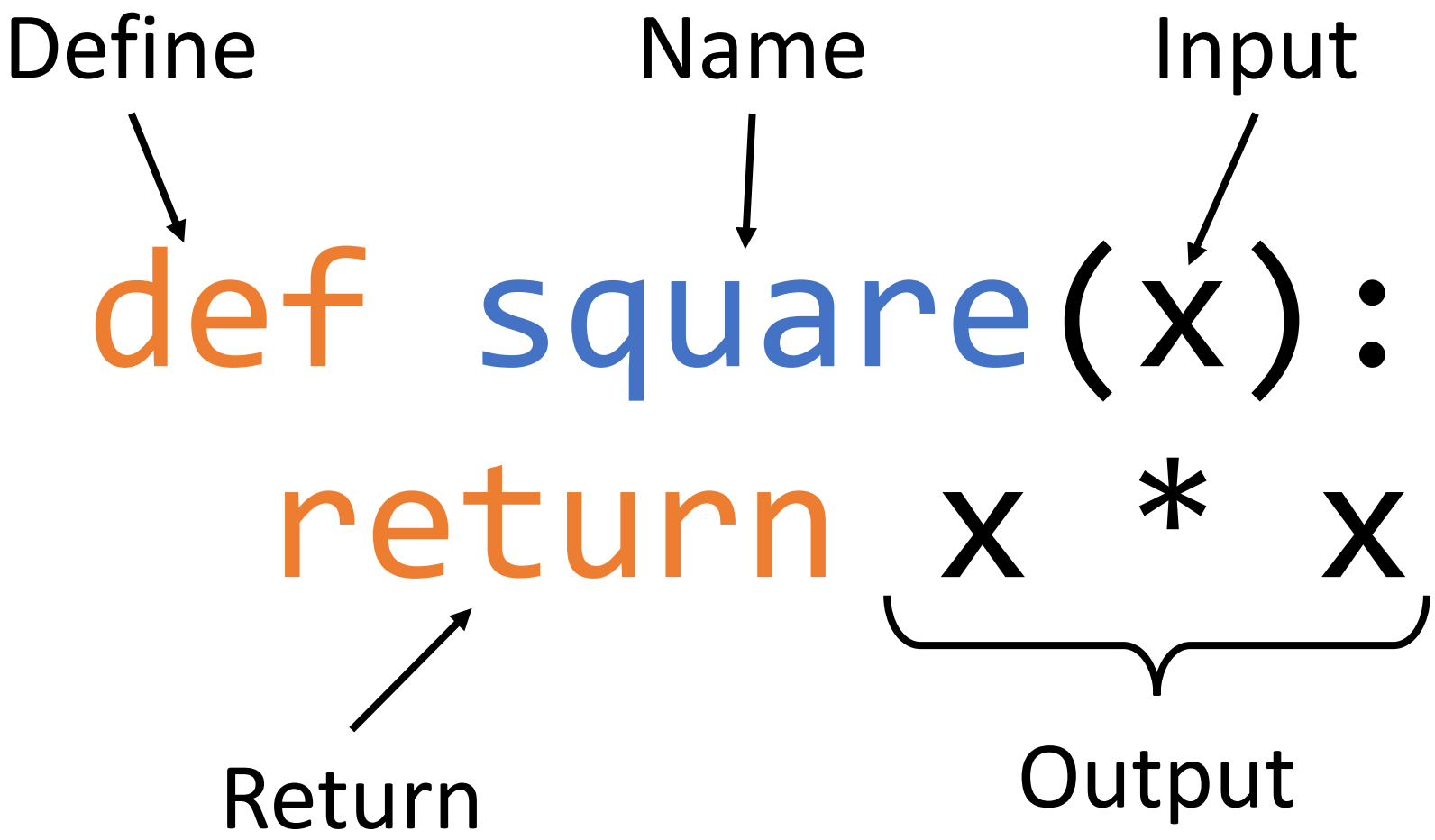
Find x?

$$a = x\cos(\theta)$$

function input

Question:

How do we square a number?


Define
Name
Input
def square(x):
return x * x
Return
Output

```
square(21)
```
441

```
square(2 + 5)
```
49

```
square(square(3))
```
81

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

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

sum_of_squares(3, 4)
```

25

```
from math import sqrt

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

```
hypotenuse(5, 12)
```

13

Syntax of writing a function!

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

- Name
  - Symbol associated with the function
- Parameters (inputs)
  - 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

Definition versus Invocation

Definition

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

def <name> (<parameters>):

*   <body>*

Invocation

```
square(5)
square(6)
```

<name> (<parameters>)

Definition versus Invocation

```
def fun(x): return x / 0

fun(2)
```

Will this raise an error?

The body of the function is NOT executed until it is called!

Function – a black box

Don't need to know how it works

Just know what it does

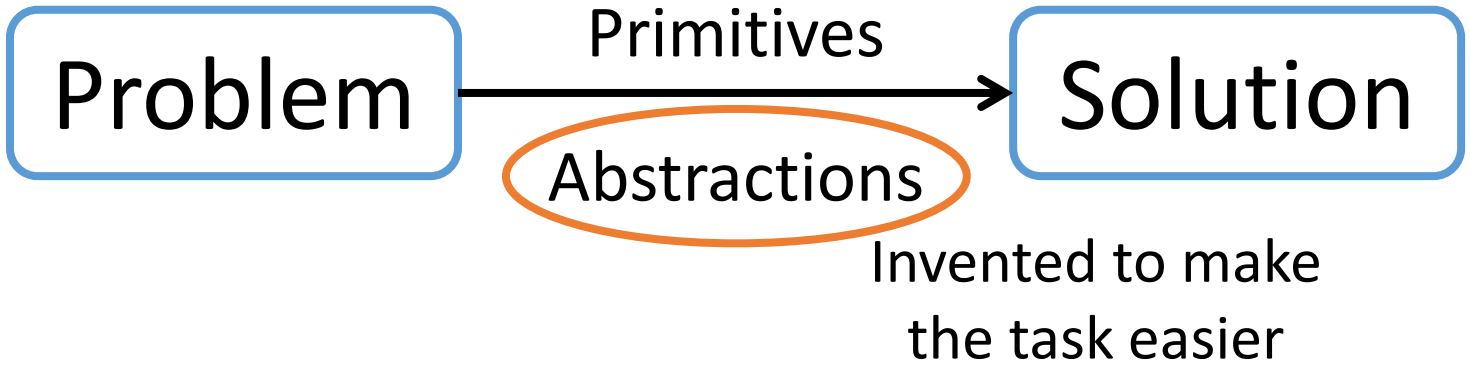
Output is returned with return

Return type can be None

Functional Abstraction

Managing Complexity

Functional Abstraction

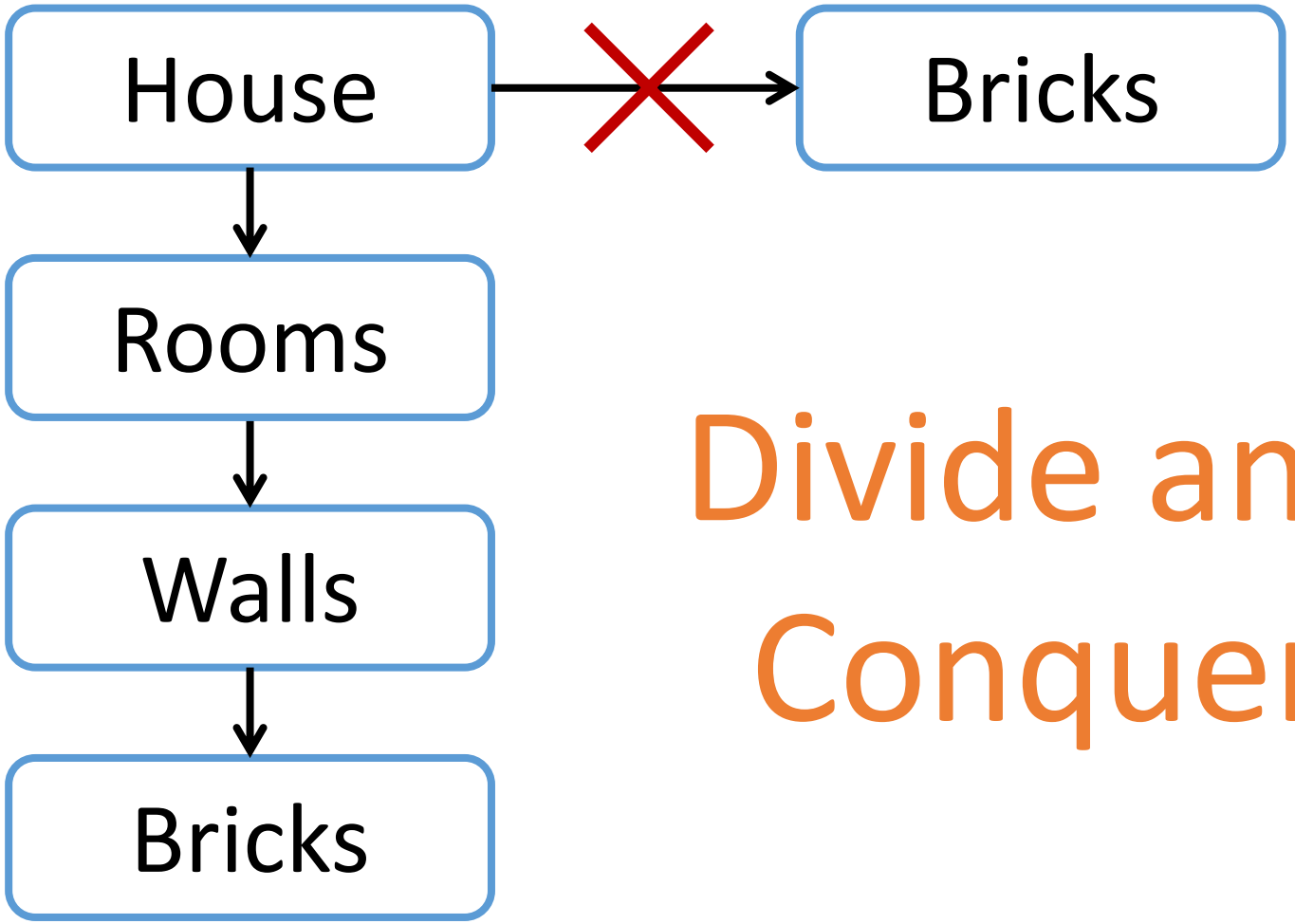


What makes a good abstraction?

Why functions?

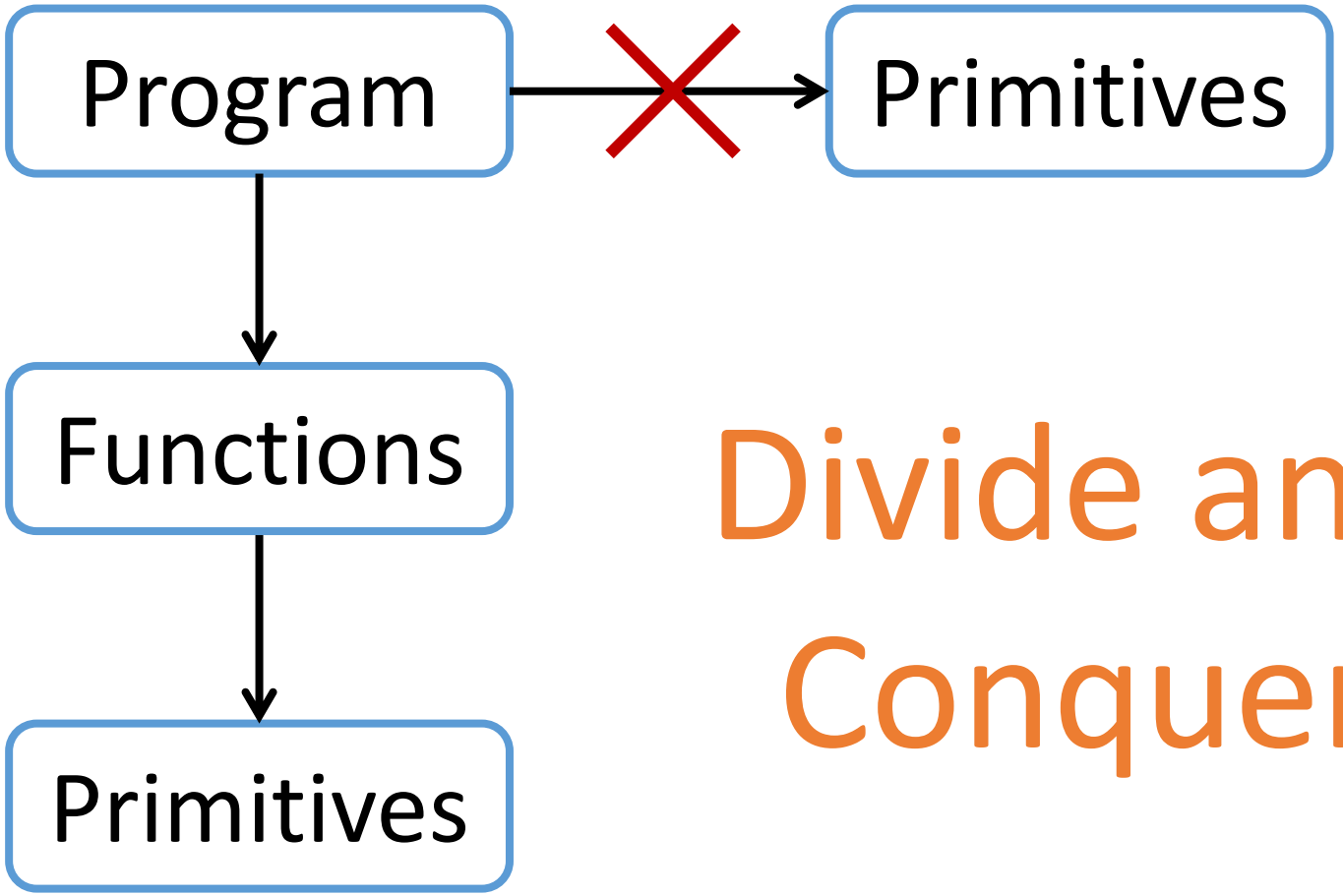
Makes it more natural to think about tasks and subtasks

Example



Divide and Conquer

Programming



Divide and Conquer

Why functions?

Makes program easier to understand

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

```
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
```

Why functions?

Captures common patterns

Why functions?

Allows for code reuse

Another Example

Function to calculate area of circle given the radius

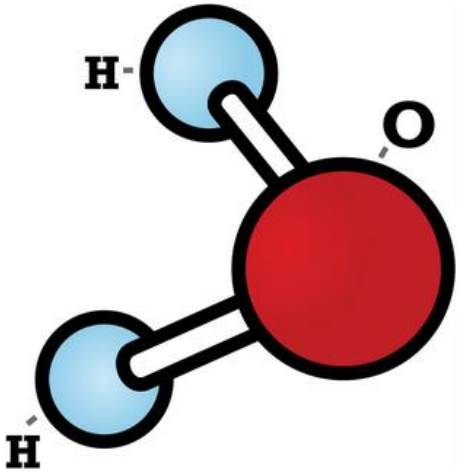
```
pi = 3.14159
def circle_area_from_radius(r):
  return pi * square(r)
```

given the diameter:

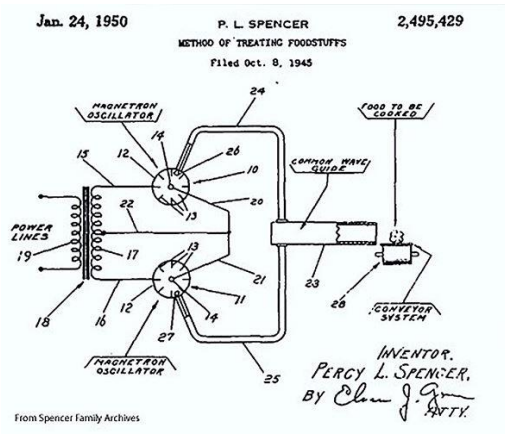
```
def circle_area_from_diameter(d):
  return circle_area_from_radius(d/2)
```

Why functions?

Hides irrelevant details

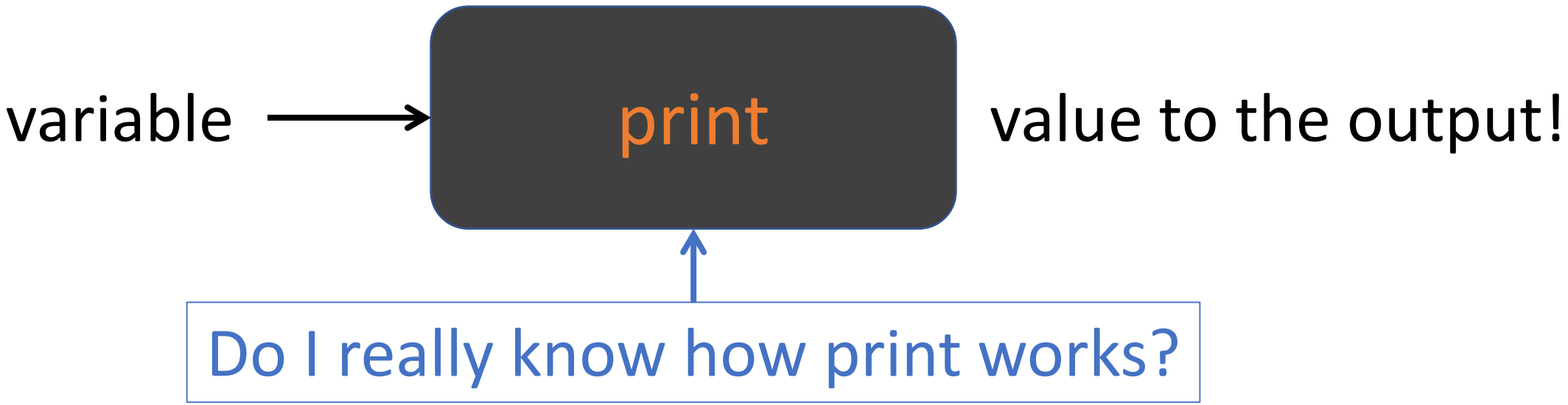


Ok for some chemical analyses, inadequate for others.

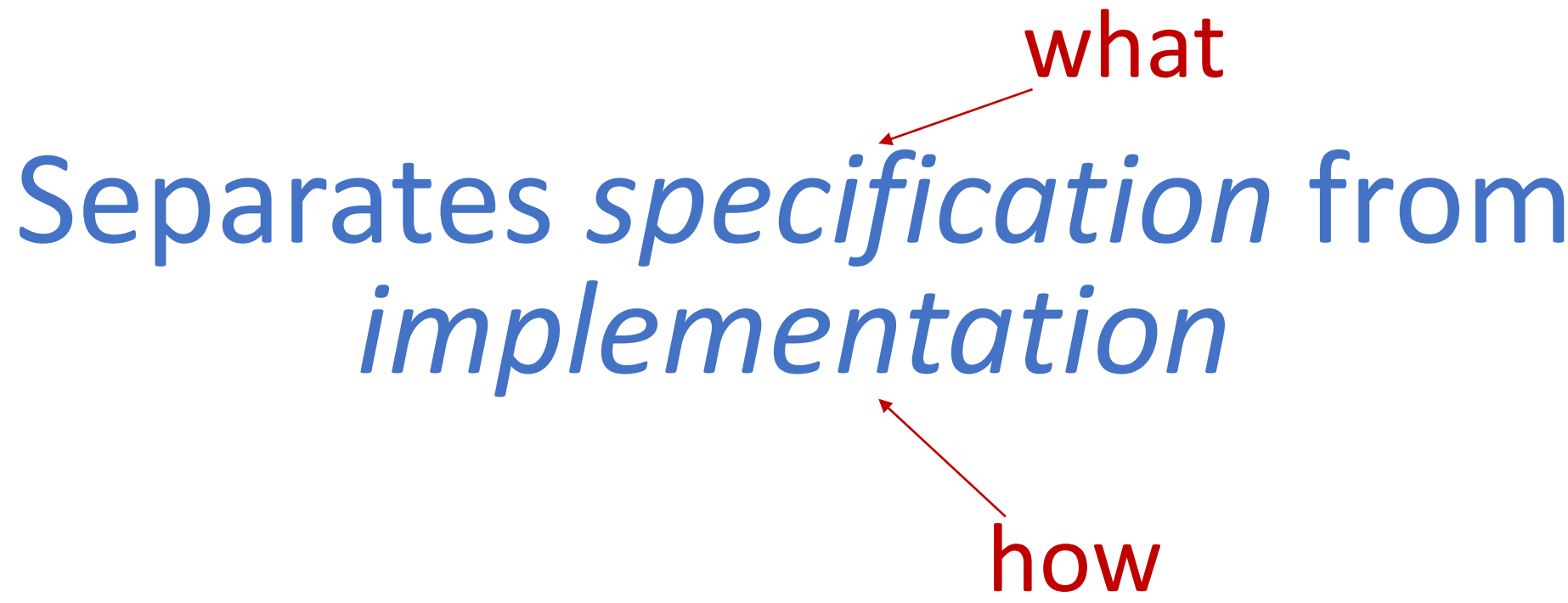


Water molecule represented as 3 balls

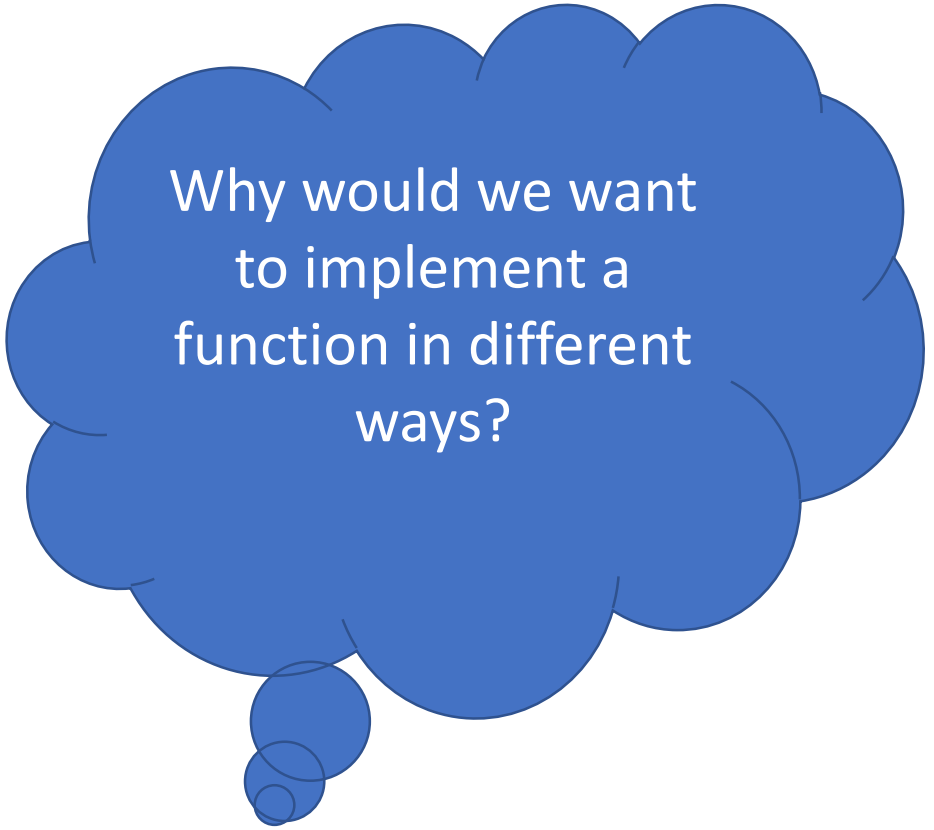
No need to know how a car works to drive it!


variable
print
value to the output!
Do I really know how print works?

Why functions?



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

def square(x):
    return exp(double(log(x)))
def square(x):
    return sqrt(x**4)
def square(x):
    return x**2
```



Why functions?

Makes debugging easier

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

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
```

Functional abstraction (Summary)

1. Divides the messy problem into natural subtasks
2. Makes program easier to understand
3. Captures common patterns
4. Allows code reuse
5. Hide irrelevant details
6. Separates specification from implementation
7. Makes debugging easier

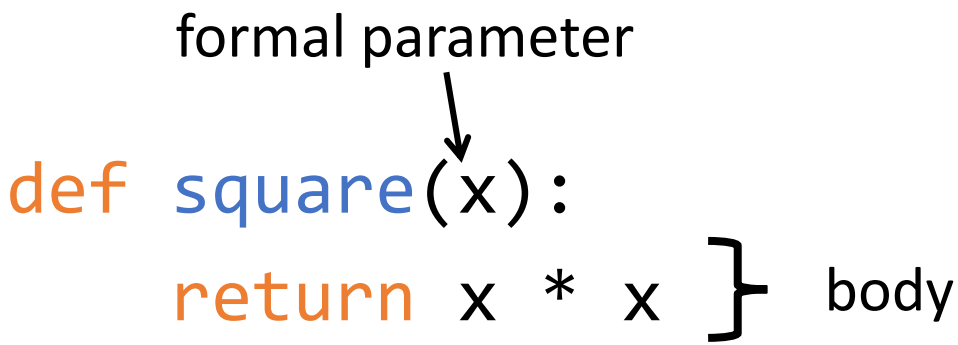
Scope of variables

```
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 ?

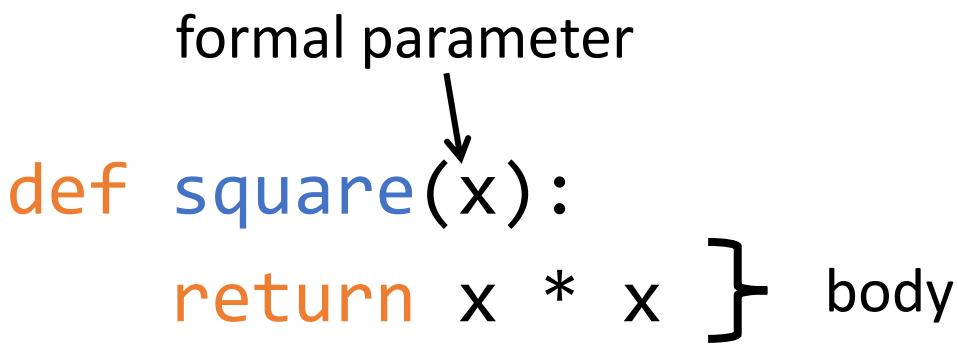
Scope of variables



A function definition binds its formal parameters.

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

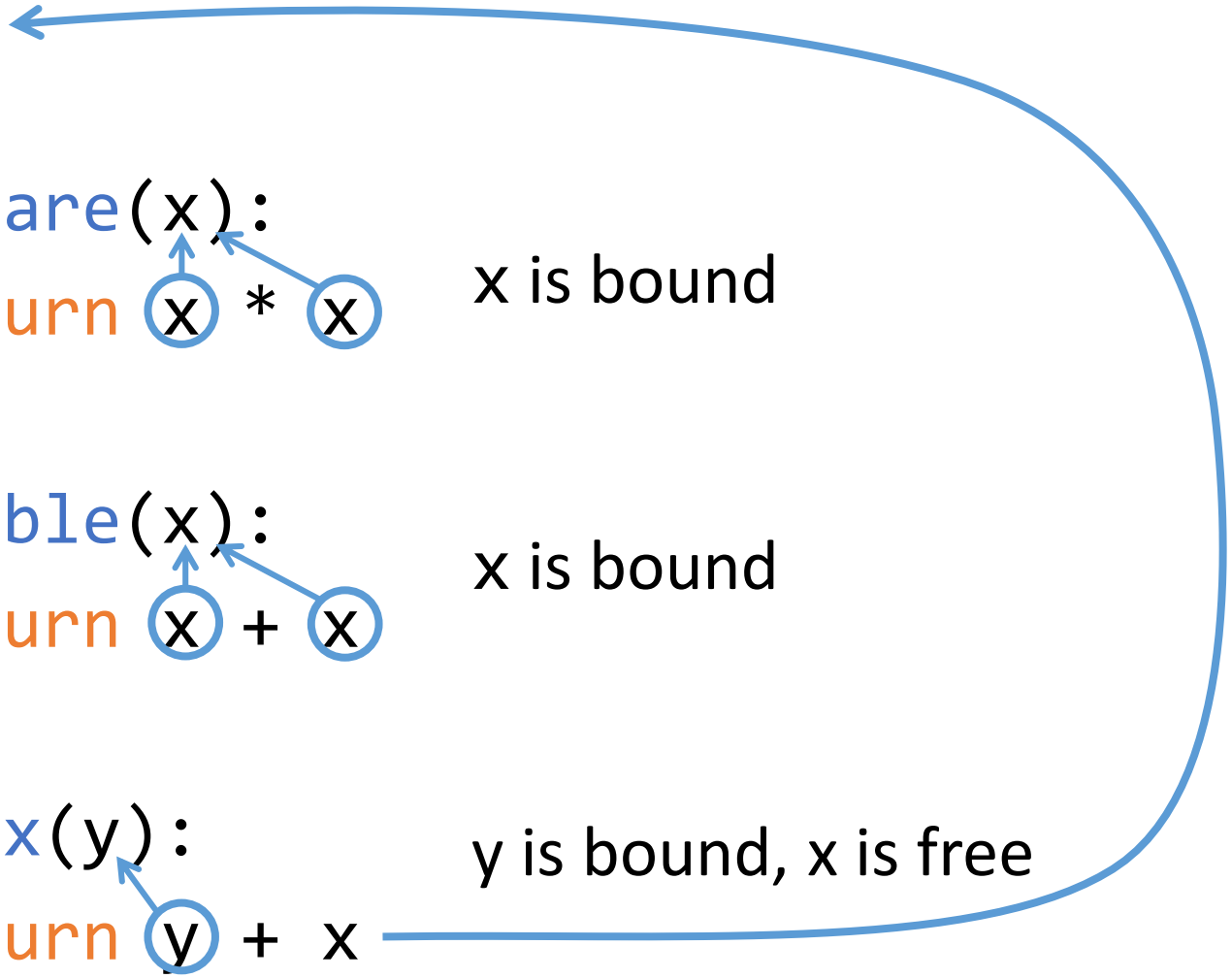
Scope of variables



- 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.

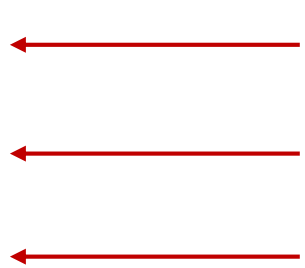
Scope of variables

```
x = 10

def square(x):
    return x * x

def double(x):
    return x + x

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



Example

```
x, y = 10, 20
def square(x): return x * x
def double(x): return x + x
def addx(y): return y + x

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



Global
x = 10
y = 20

square
x = 20
return = 400

square
x = 10
return = 100

addx
y = 5
return = 15

Example

```
a, b = 3, 4
def hypotenuse(a, b):
    def sum_of_squares():
        return square(a) + square(b)

    return math.sqrt(sum_of_squares())

c = hypotenuse(4, a)
```

Global
a = 3
b = 4
c = 5

hypotenuse
a = 4
b = 3
return=5

math.sqrt
return = 5

Abstract Environment

Picture Language

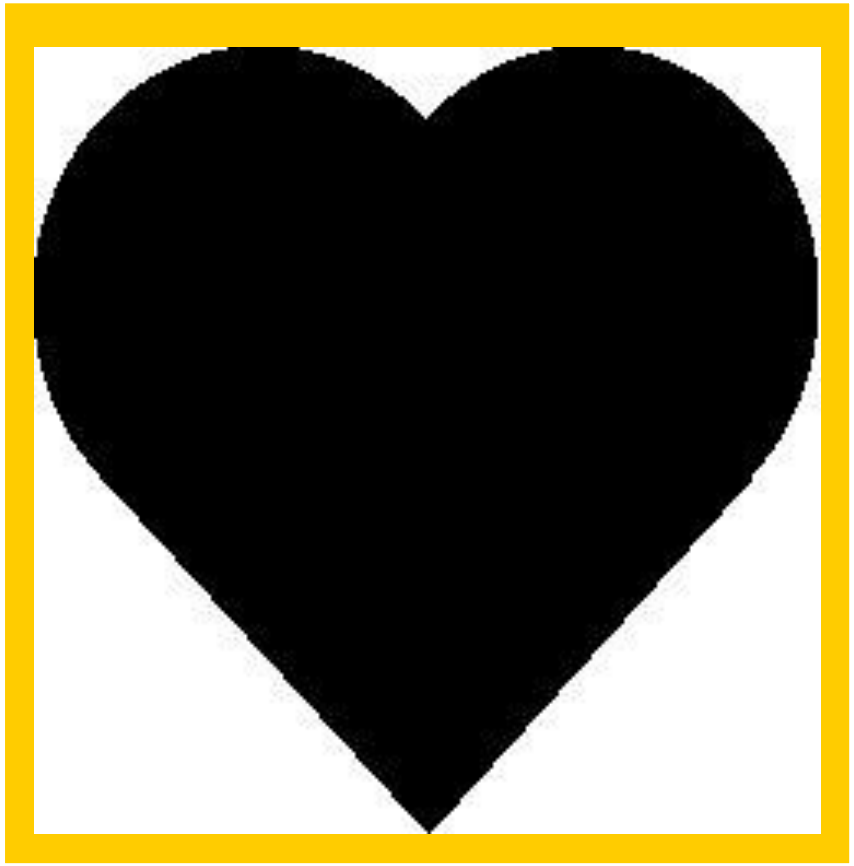
(runes.py)

Also graphics.py + image2gif.py

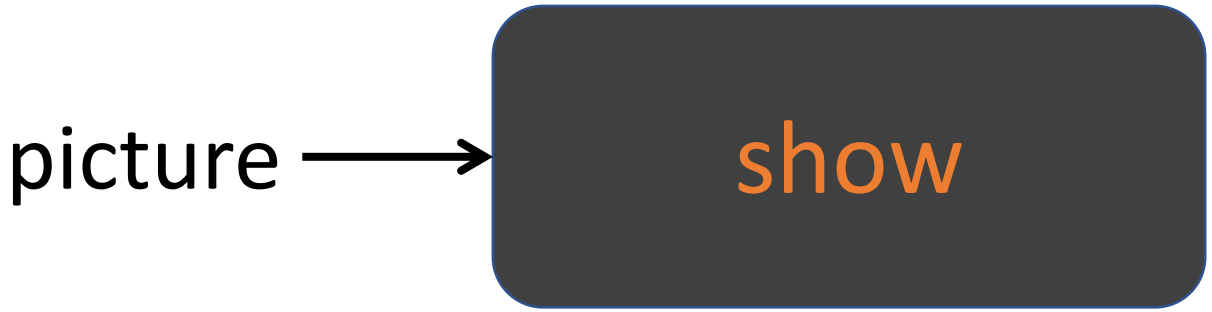
Primitives: show

```
show(rcross_bb)
show(corner_bb)
show(sail_bb)
show(nova_bb)
show(heart_bb)
```

picture object



Primitives are functions!



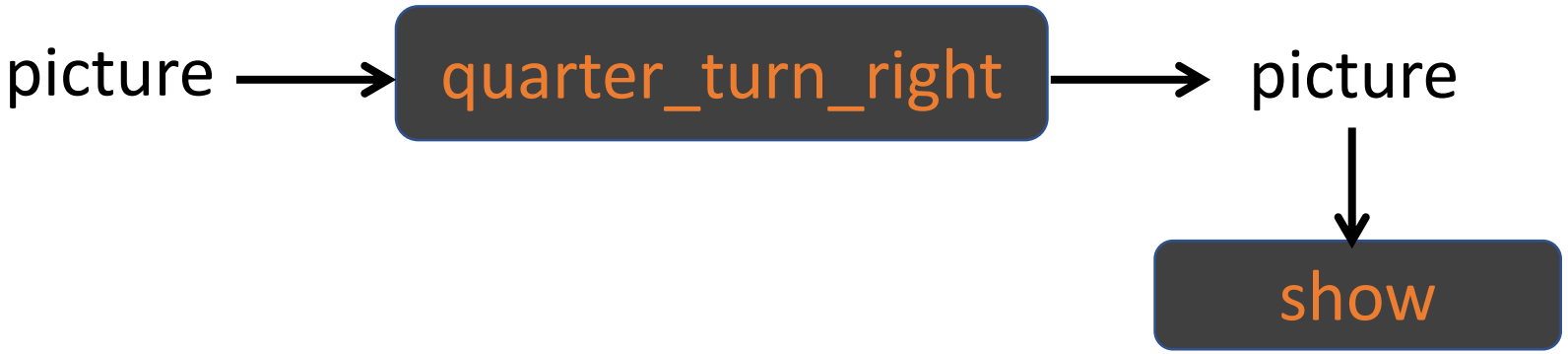
Primitives: `quarter_turn_right`

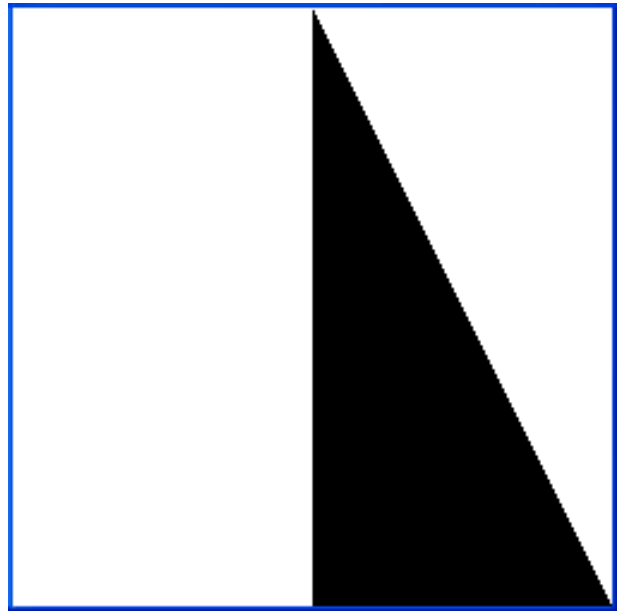
```
clear_all()
show(quarter_turn_right(sail_bb))
```

operation

picture

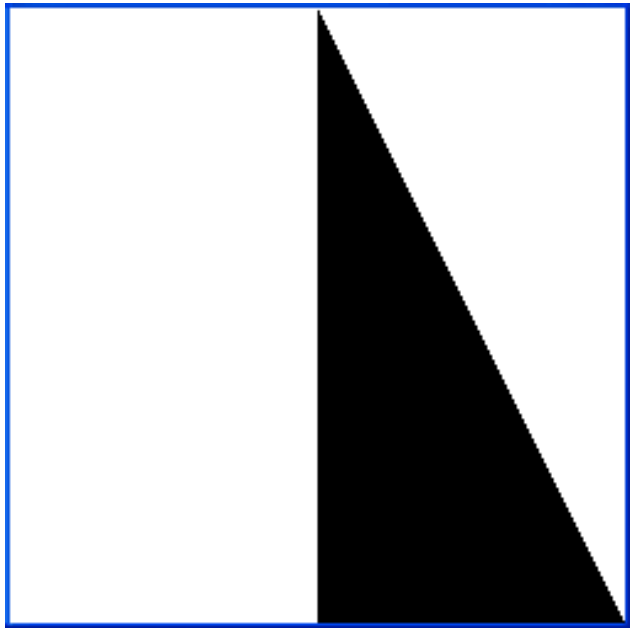
result is another picture





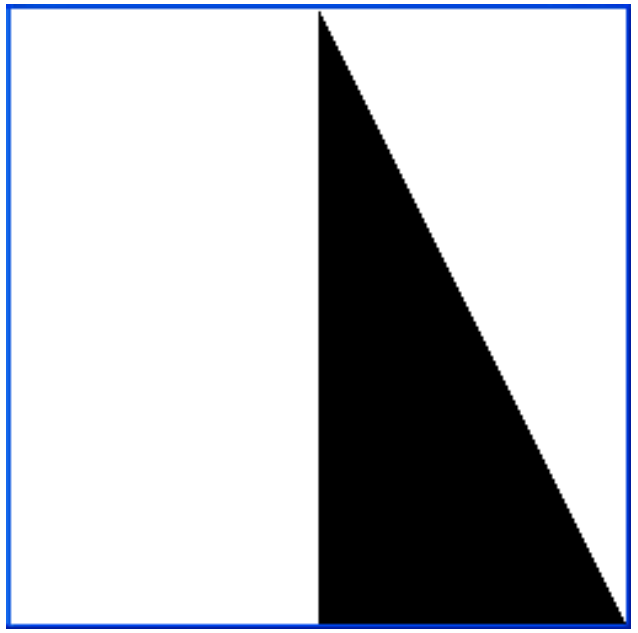
Derived: turn_upside_down

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



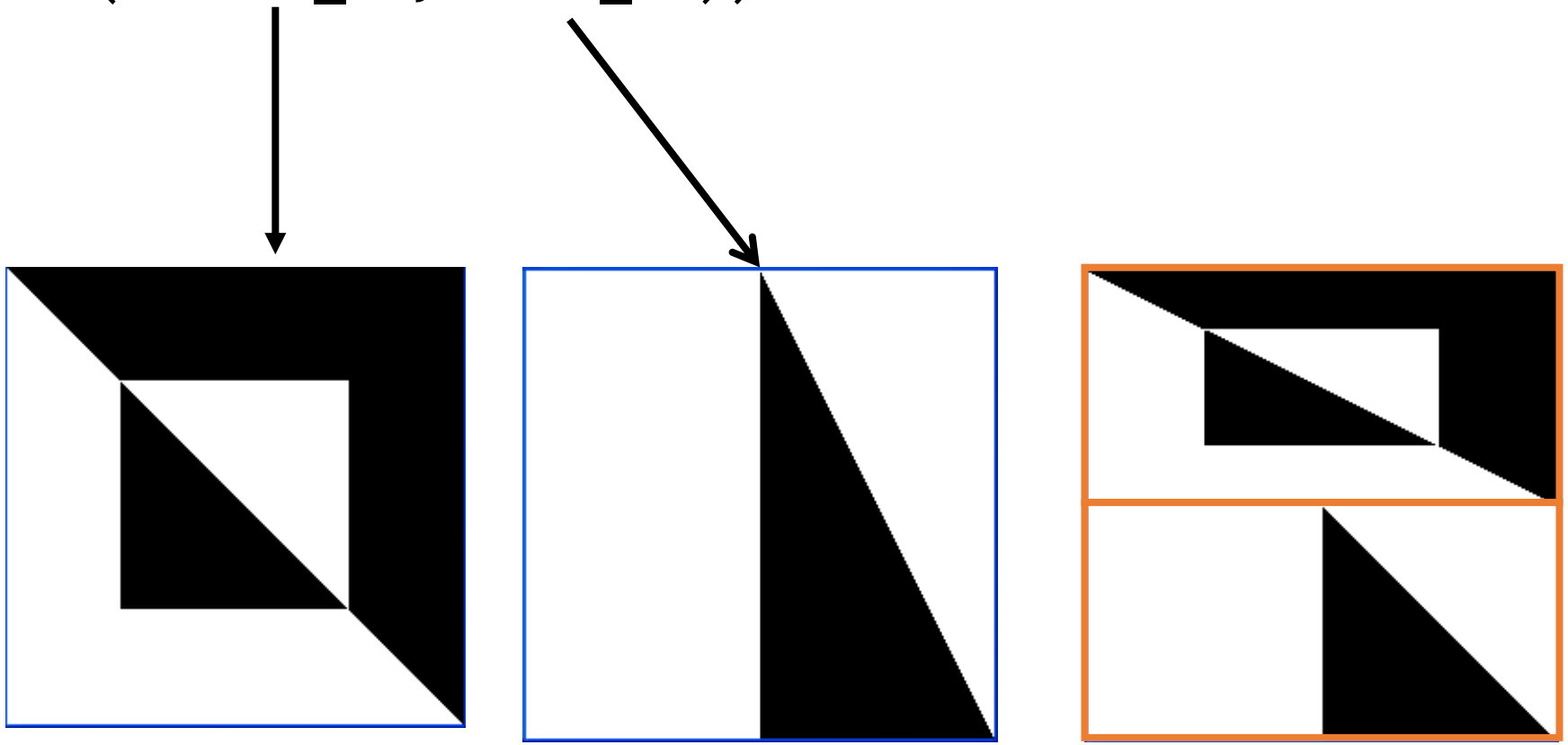
Derived: quarter_turn_left

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

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



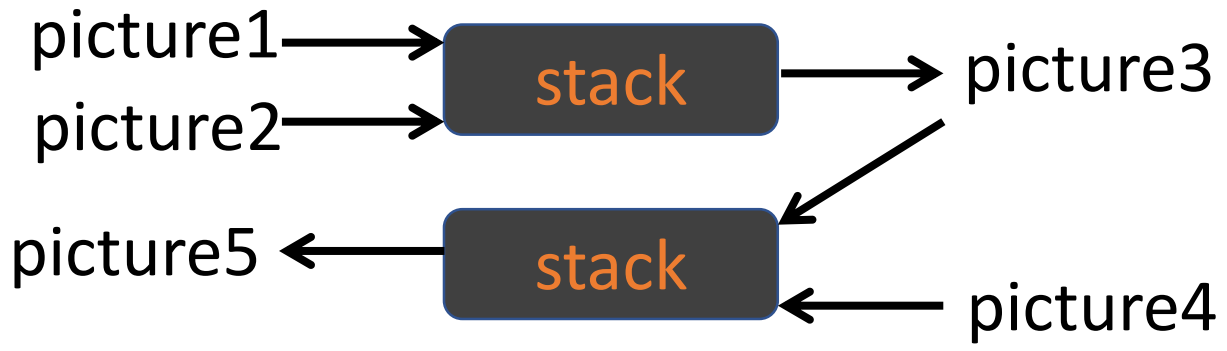
Primitive: `stack`

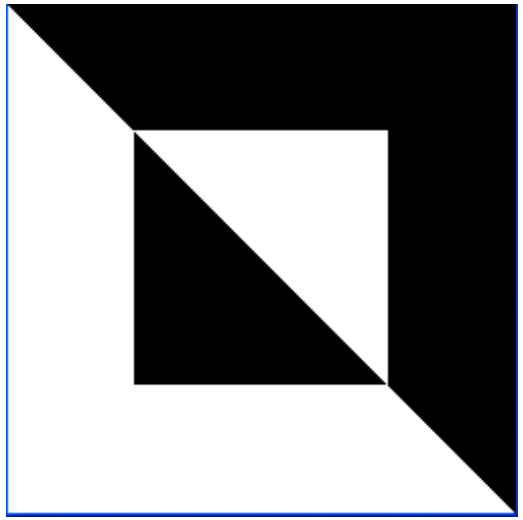

```
clear_all()
show(stack(rcross_bb, sail_bb))
```

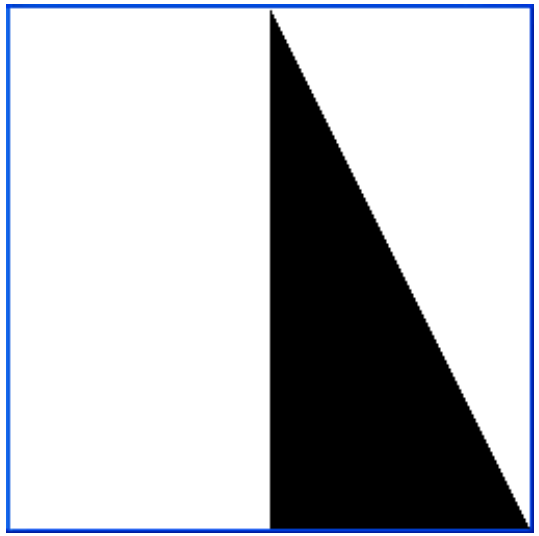


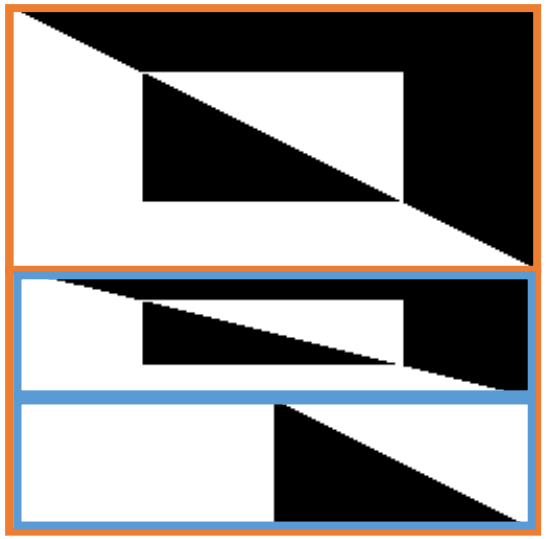
Derived *multiple stacking*

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



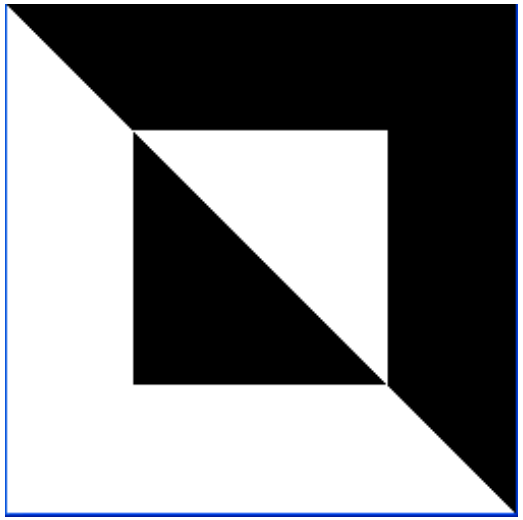


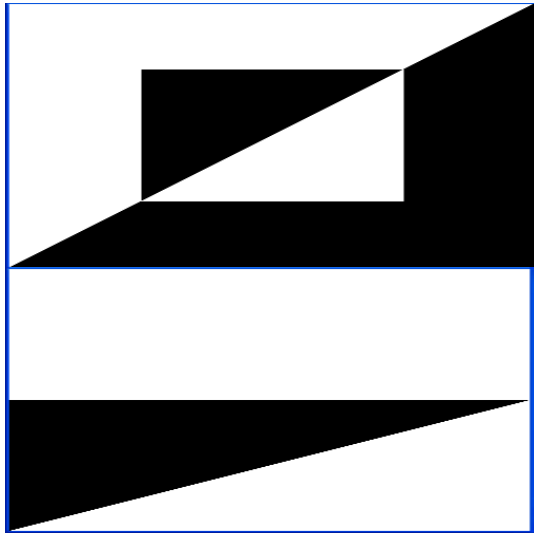




Derived: `beside`

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

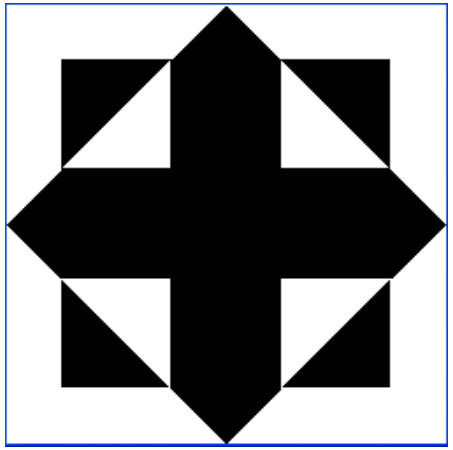




Derived – a more complex function!

```
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



How to write `make_cross` function?

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

How to write `make_cross` function?

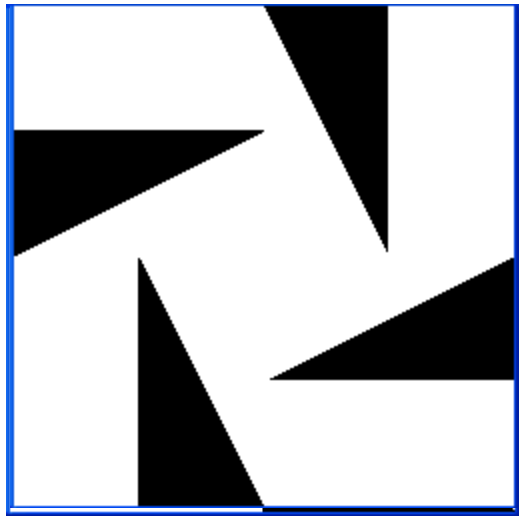
```
stack(
  beside(
    quarter_turn_right(pic),
    turn_upside_down(pic)),
  beside(
    pic,
    quarter_turn_left(pic))))
```

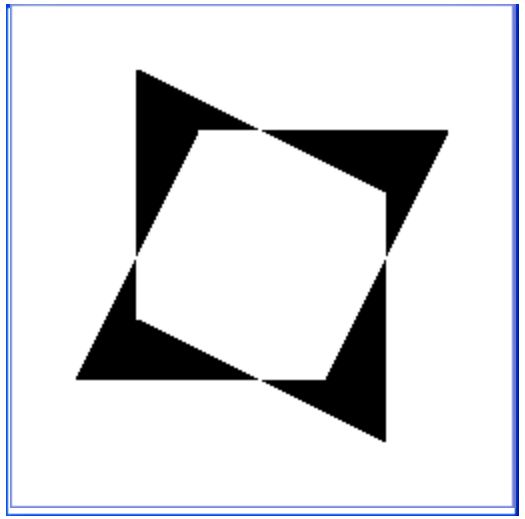
How to write `make_cross` function?

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

Storing the return values!

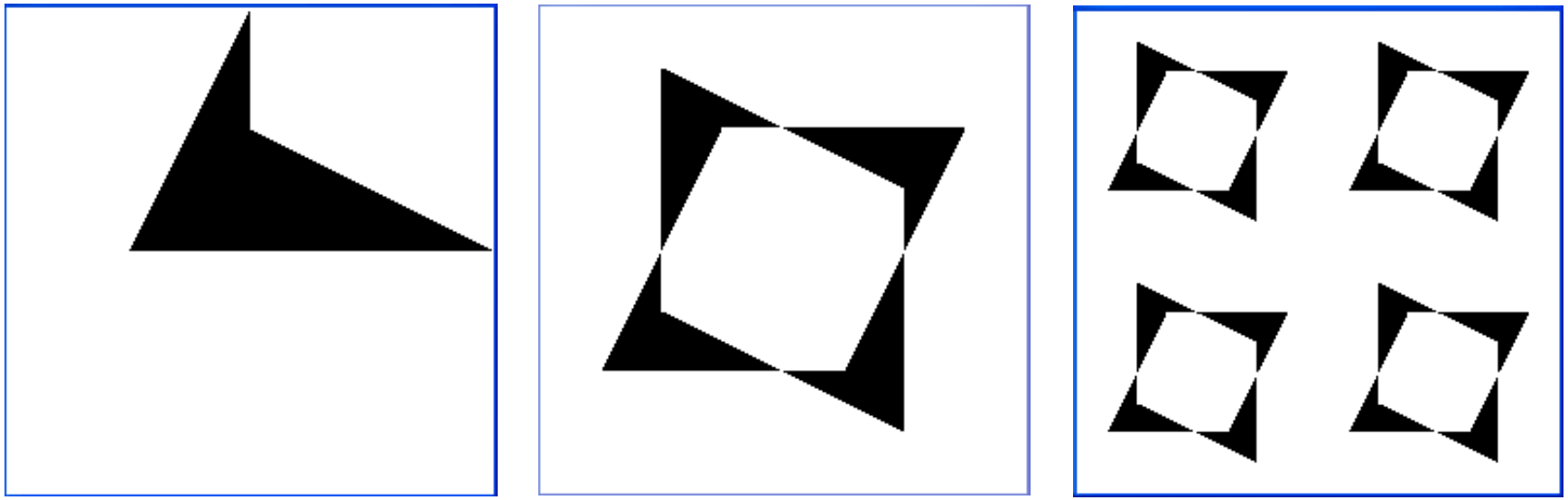
```
my_pic = make_cross(sail_bb)
show(my_pic)

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





Repeating the pattern

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

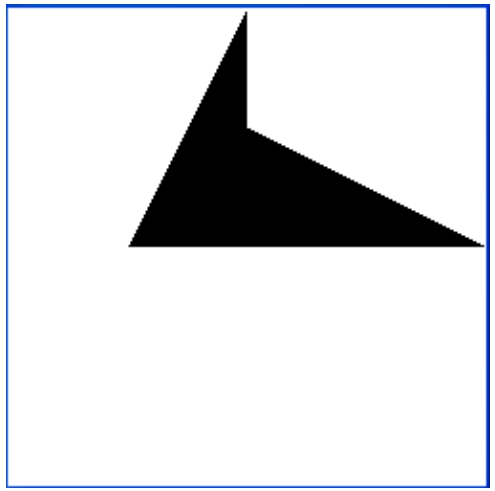


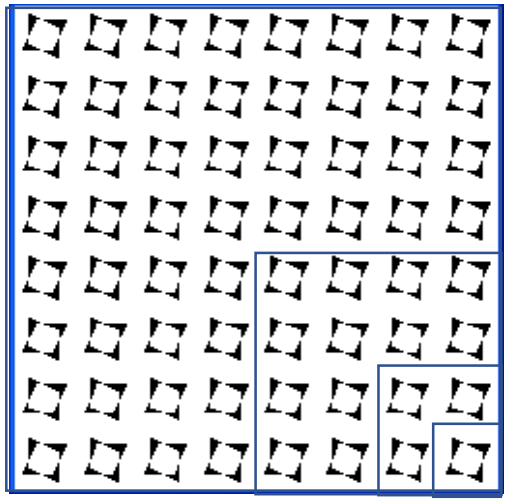
Repeating the pattern

```
def repeat_pattern(n, pat, pic):
    if n == 0:
        return pic
    else:
        return pat(repeat_pattern(n-1, pat, pic))
```

recursion

```
show(repeat_pattern(4, make_cross, nova_bb))
```





pat(pat(pat(pat(pic))))

pat(pat(pat(pic)))

pat(pat(pic))

pat(pic)

pic

repeat_pattern(4)

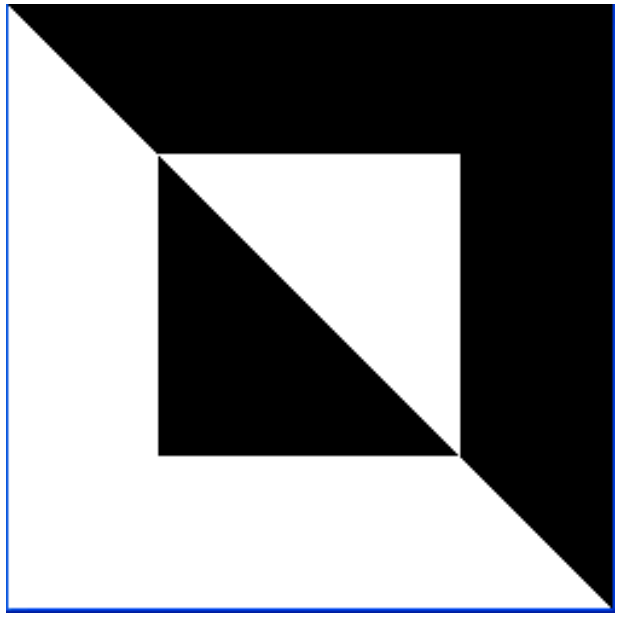
repeat_pattern(3)

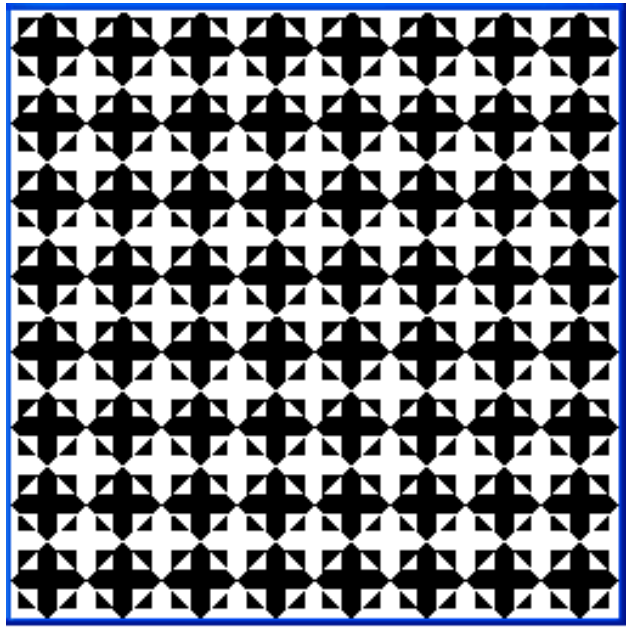
repeat_pattern(2)

repeat_pattern(1)

repeat_pattern(0)

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





Anonymous(aka lambda) functions

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

input output

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

function

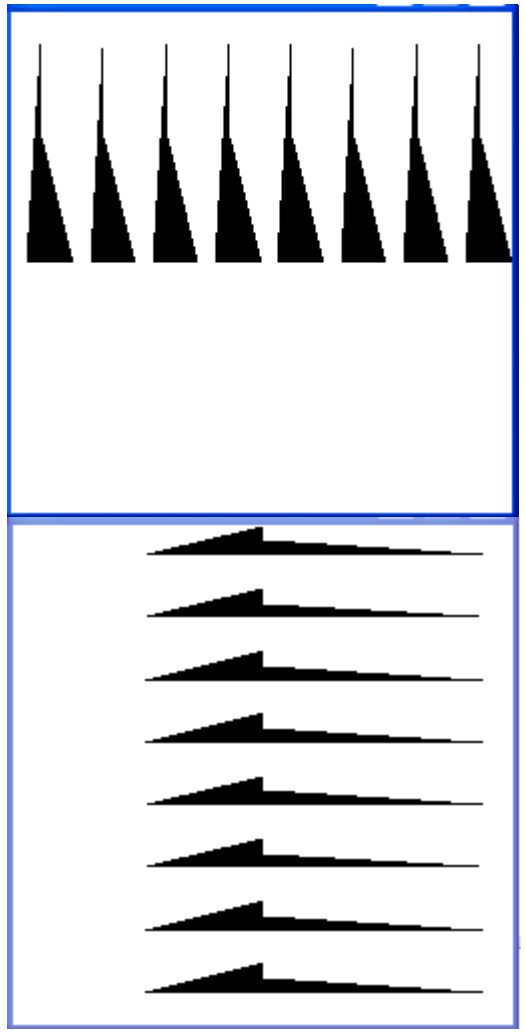
```
foo(1)
```
1

```
foo(4)
```
16

New patterns!

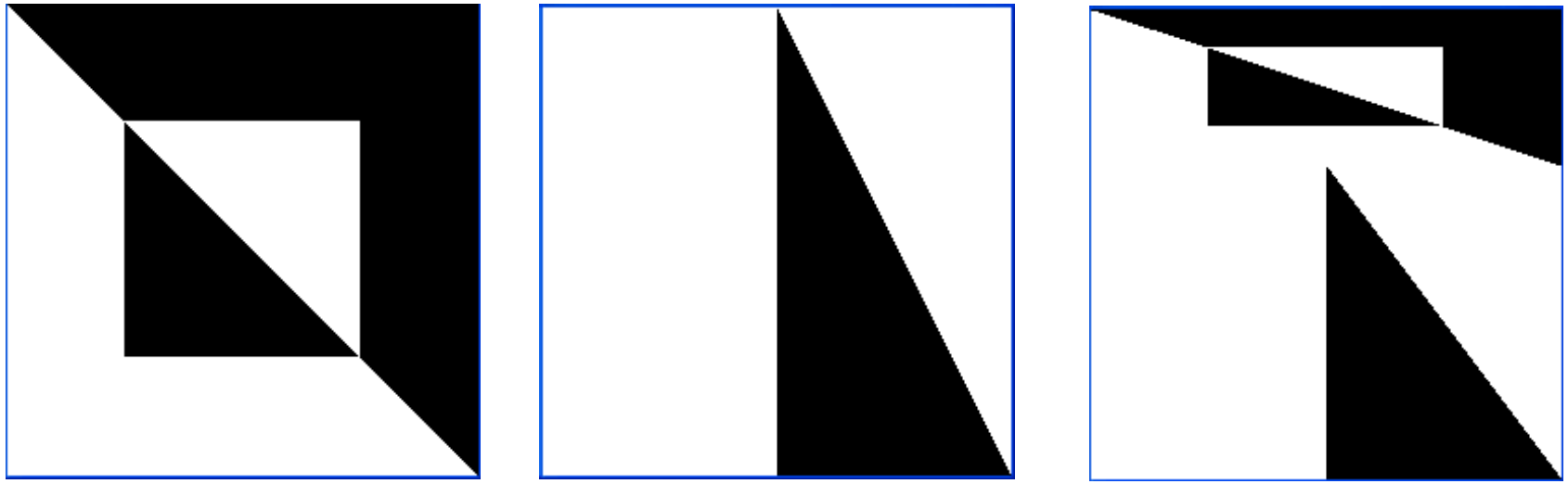
anonymous function

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

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



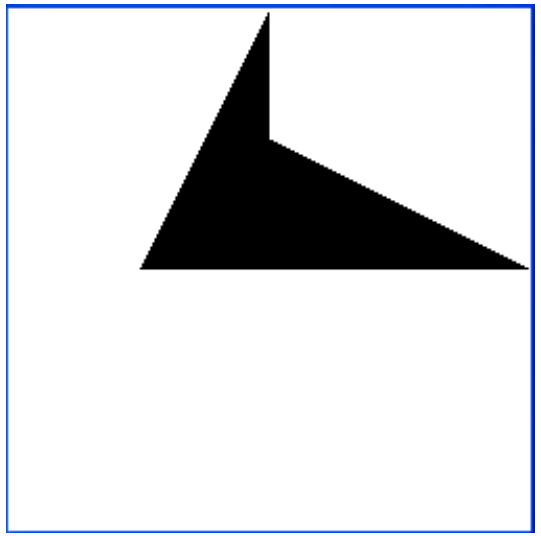
Primitive: stack_frac

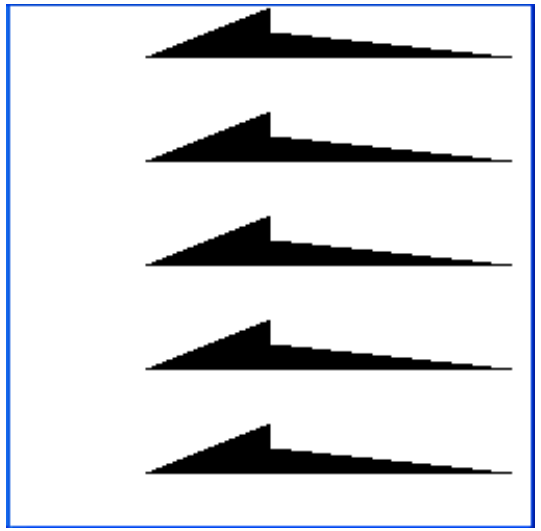
```
clear_all()
show(stack_frac(1/3, rcross_bb, sail_bb))
```

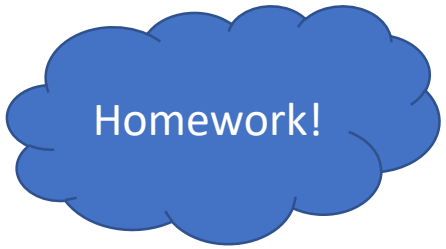


Repeating the pattern

```
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))
```

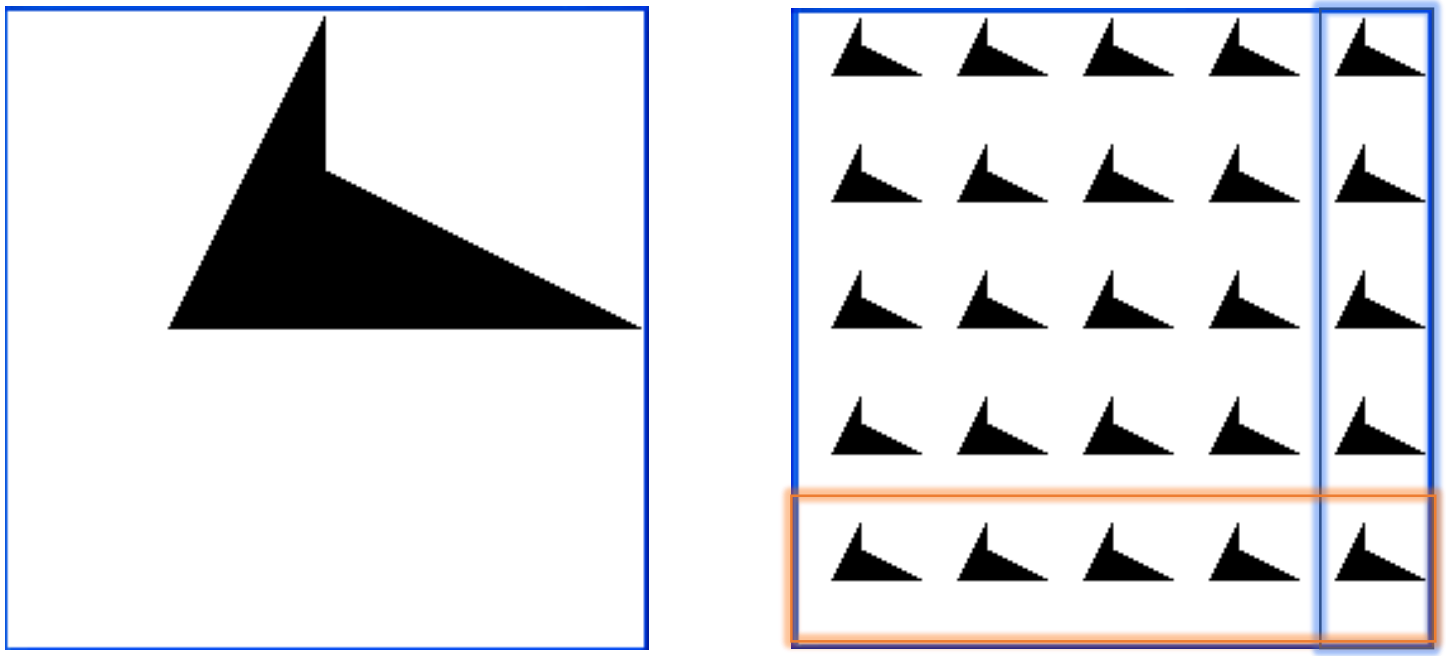






One final pattern!

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



No idea how a picture is represented

No idea how the operations do their work

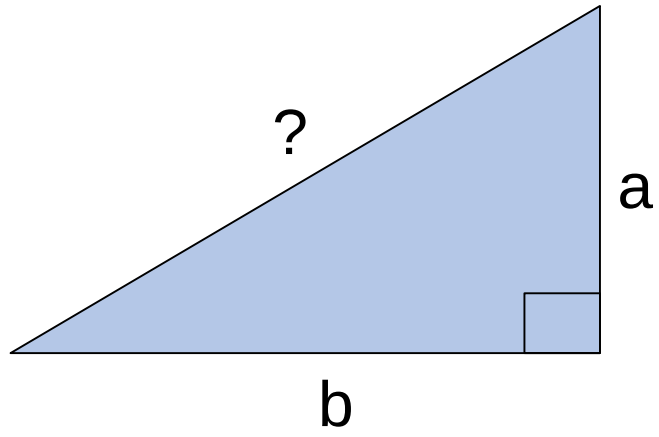
Yet, we can build complex pictures

Functional Abstraction

Wishful Thinking

Pretend you have whatever you need

```
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: Taxi Fare Calculation

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

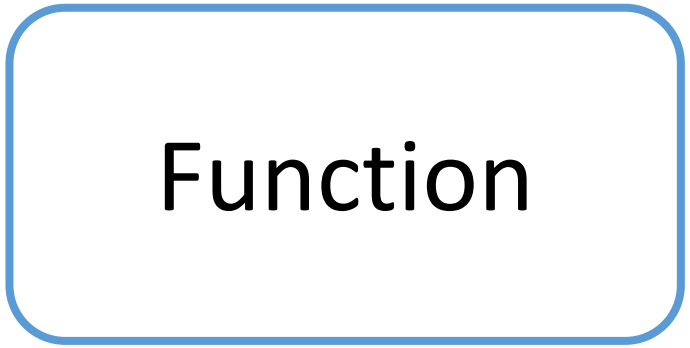
- For the first kilometre or less: $2.40
- Every 200 metres thereafter or less up to 10 km: $0.10
- Every 150 metres thereafter or less after 10 km: $0.10

Problem:

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

How do we start?

Formulate the problem



Needs a name

Pick an appropriate name

(not ~~foo~~)

Formulate the problem

- What data do you need?
- Where would you get it?

Results should be unambiguous

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

How to compute the result?

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

How to compute the result?

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

  fare = $2.40 + $\lceil 2300/200 \rceil \times \$0.10$

  = $3.60
- e.g. #3: distance = 14,500 m

  fare = $2.40 + $\lceil 9000/200 \rceil \times \$0.10 + \lceil 4500/150 \rceil \times \$0.10$

  = $9.90

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

Pseudocode

Case 1: distance <= 1000

fare = $2.40

Case 2: 1000 < distance <= 10,000

fare = $2.40 + $0.10 * ⌈(distance – 1000)/200⌉

Case 3: distance > 10,000

fare = $6.90 + $0.10 * ⌈(distance – 10,000)/150)⌉

Note: the Python function `ceil` rounds up its argument.

```
math.ceil(1.5) = 2
```

Program

```
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?

Generalisability?

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.

$3

Comfort and CityCab
Toyota Crown
(22¢)

Trans-Cab
Toyota Crown
(22¢)

SMRT
Toyota Crown
(22¢)

Premier
Toyota Crown
(22¢)

Nissan Cedric
(22¢)

$3.20

Comfort and CityCab
Hyundai Sonata
(22¢)

Premier
Kia Magentis
(22¢)

Toyota Wish (CNG)
(22¢)

Hyundai i30 Wagon
(22¢)

Prime
Toyota Axio
(22¢)

Honda Fit
(22¢)

Honda Airwave
(22¢)

Honda Partner
(22¢)

$3.40

Comfort and CityCab
Toyota Camry Hybrid
(22¢)

Prime
Toyota Allion
(22¢)

Toyota Premio
(22¢)

Toyota Wish
(22¢)

Toyota Fielder
(22¢)

Trans-Cab
Toyota Wish
(22¢)

SMRT
Chevrolet Epica
(22¢)

Hyundai Avante
(22¢)

Honda Stream
(22¢)

$3.50

Prime
Toyota Prius 1.5
(22¢)

Premier
Toyota Prius
(22¢)

Skoda Superb
(22¢)

$3.60

Trans-Cab
Chevrolet Epica II
(22¢)

$3.70

Comfort and CityCab
Hyundai i40
(22¢)

Premier
Kia Optima
(22¢)

Prime
Toyota Camry/
Camry Hybrid
(22¢)

Toyota Estima
(22¢)

Honda Stepwagon
(22¢)

Toyota Prius 1.8
(22¢)

$3.80

SMRT
Toyota Prius
(22¢)

Hyundai Azera (CNG)
(22¢)

$3.90

Comfort and CityCab
Limousine
(30¢)

Premier
Kia Carnival
(30¢)

SMRT
Mercedes-Benz
(22¢)

London cab
(22¢)

Ssangyong Space
(22¢)

Hyundai Starex
(22¢)

Trans-Cab
Mercedes-Benz
(30¢)

Renault Latitude
(22¢)

$4.50

Prime Toyota Vellfire (33¢)

Premier Mercedes-Benz E-220 (30¢)

$5

SMRT Chrysler 300C (33¢)

Sources: COMFORT TRANSPORTATION AND CITYCAB, PREMIER TAXIS, PRIME CAR RENTAL & TAXI SERVICES, SMRT, TRANS-CAB SERVICES

PHOTOS: ST FILE, COMFORT, PREMIER TAXIS, PRIME TAXI, SMRT, TRANS-CAB TAXI

Avoid magic numbers!

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

- Hard to make changes in future

Define abstractions to hide them!

```
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 = 3.2
increment = 0.22
block1 = 400
block2 = 350
```

Summary

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

Why functions are good!

1. Divides the messy problem into natural subtasks
2. Makes program easier to understand
3. Captures common patterns
4. Allows code reuse
5. Hide irrelevant details
6. Separates specification from implementation
7. Makes debugging easier