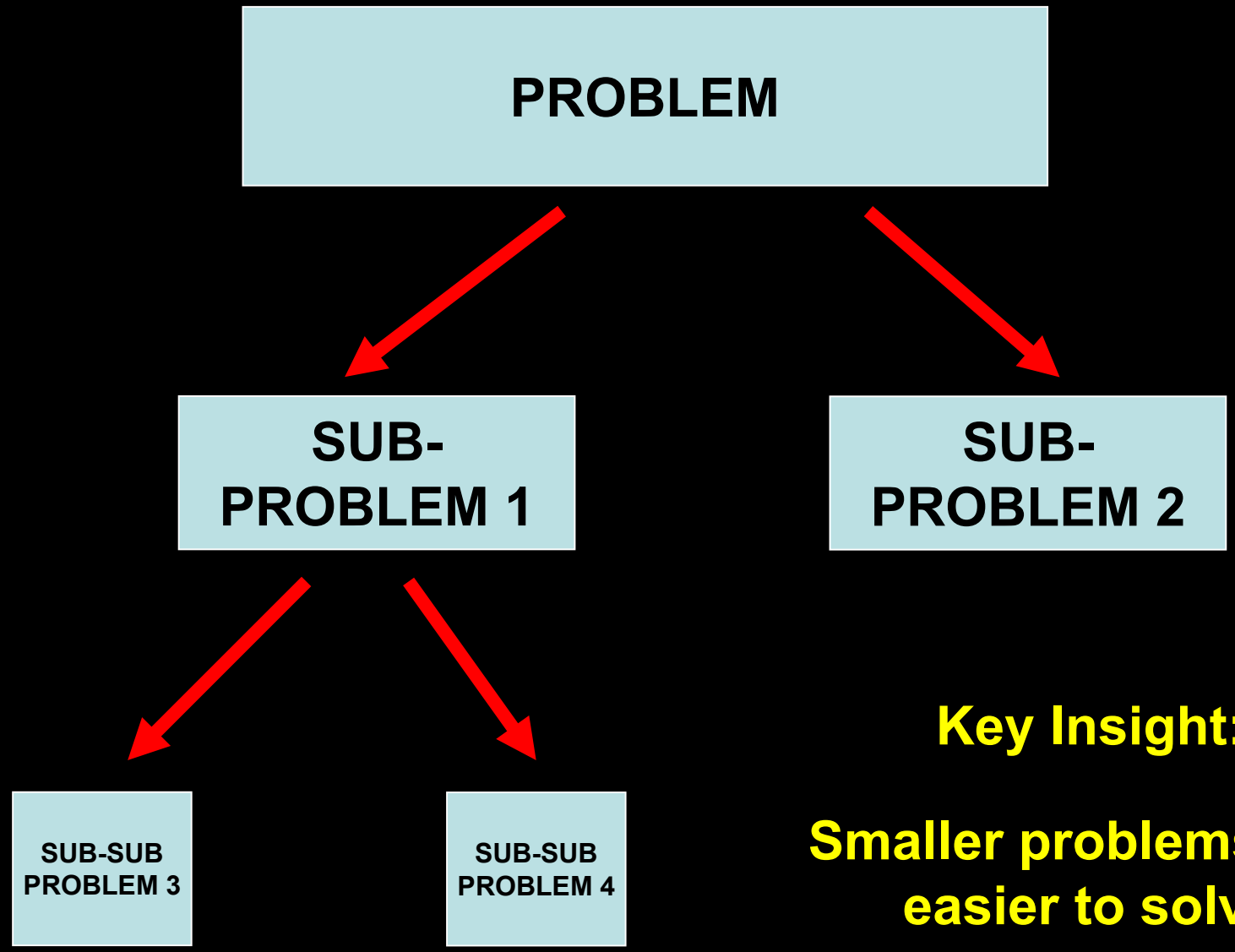
Lecture 3b - Good Abstraction

LT3b Objectives

- Know what abstraction means and understand its advantages
- Understand key approaches to problem solving

Managing Complexity

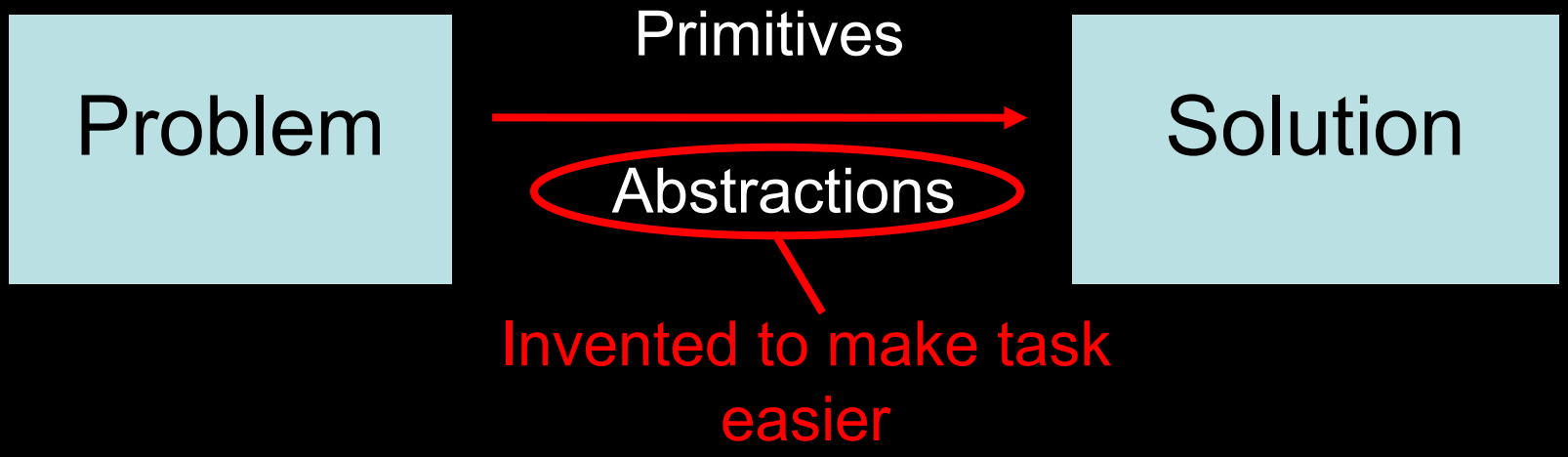
Divide-and-conquer



Key Insight:

Smaller problems are easier to solve

Abstractions



What makes a good abstraction?

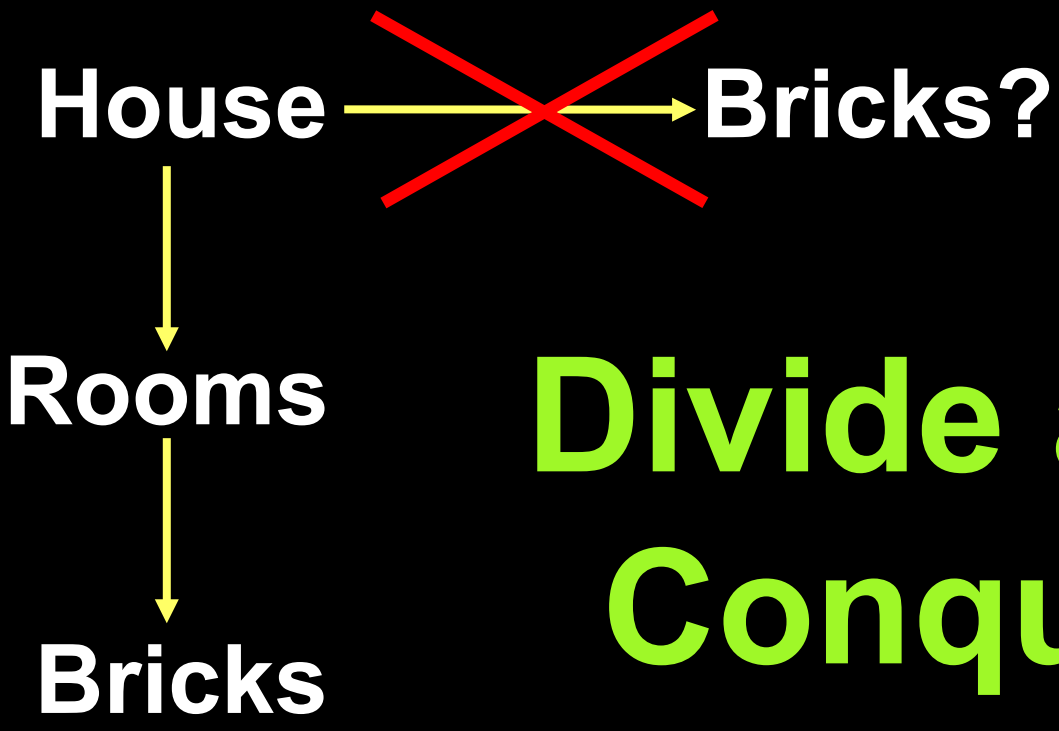
Good Abstraction

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

Good Abstraction

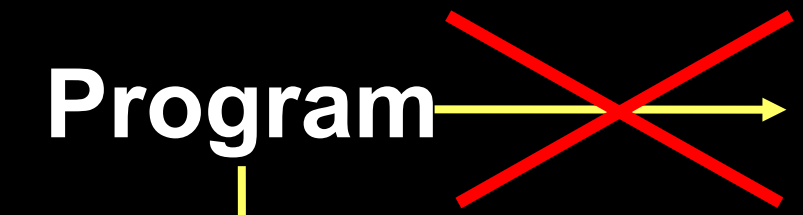
1. Makes it more natural to think about tasks and subtasks

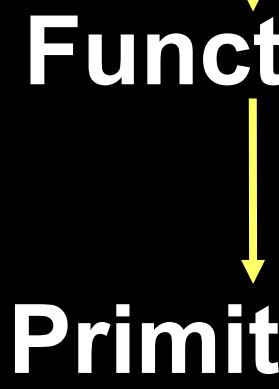
Example



Divide and Conquer

Programming





Divide and Conquer

Good Abstraction

2. Makes programs easier to understand

Compare:

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

  def square(x):
      return x * x

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

Versus:

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

Good Abstraction

3. Captures common patterns

Item	Amount ($)
Food	300
Entertainment	50
Saving	400
Others	250

```
my_finances =
     'food = $' + str(1000*0.3)\
     + '; entertainment = $' + str(1000*0.05)\
     + '; savings = $' + str(1000*0.4)\
     + '; others = $' + str(1000*0.25)
```

```
my_finances =
    'food = $' + str(1000*0.3)\
    + '; entertainment = $' + str(1000*0.05)\
    + '; savings = $' + str(1000*0.4)\
    + '; others = $' + str(1000*0.25)
```

Using Function to capture the common patterns

```
def my_finances(salary):
    return 'food = $' + str(salary*0.3)\
    + '; entertainment = $' + str(salary*0.05)\
    + '; savings = $' + str(salary*0.4)\
    + '; others = $' + str(salary*0.25))
```

```
>>> my_finances(1000)
```

Good Abstraction

4. Allows for the code to be reused

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

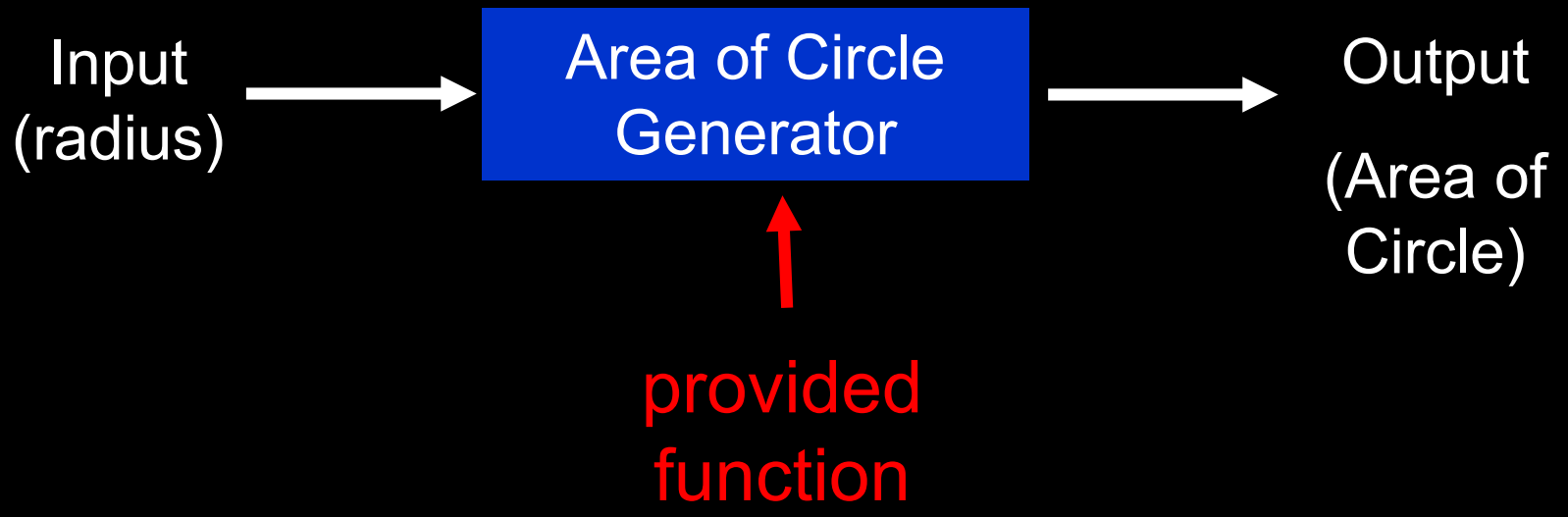
Function `square` used to find the hypothenus can also be used to calculate the area of a circle.

Function to calculate area of circle, given its radius:

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

Good Abstraction

5. Hides the implementation details


Input
(radius)
Area of Circle
Generator
Output
(Area of
Circle)
provided
function

Good Abstraction

6. Separates specification from implementation

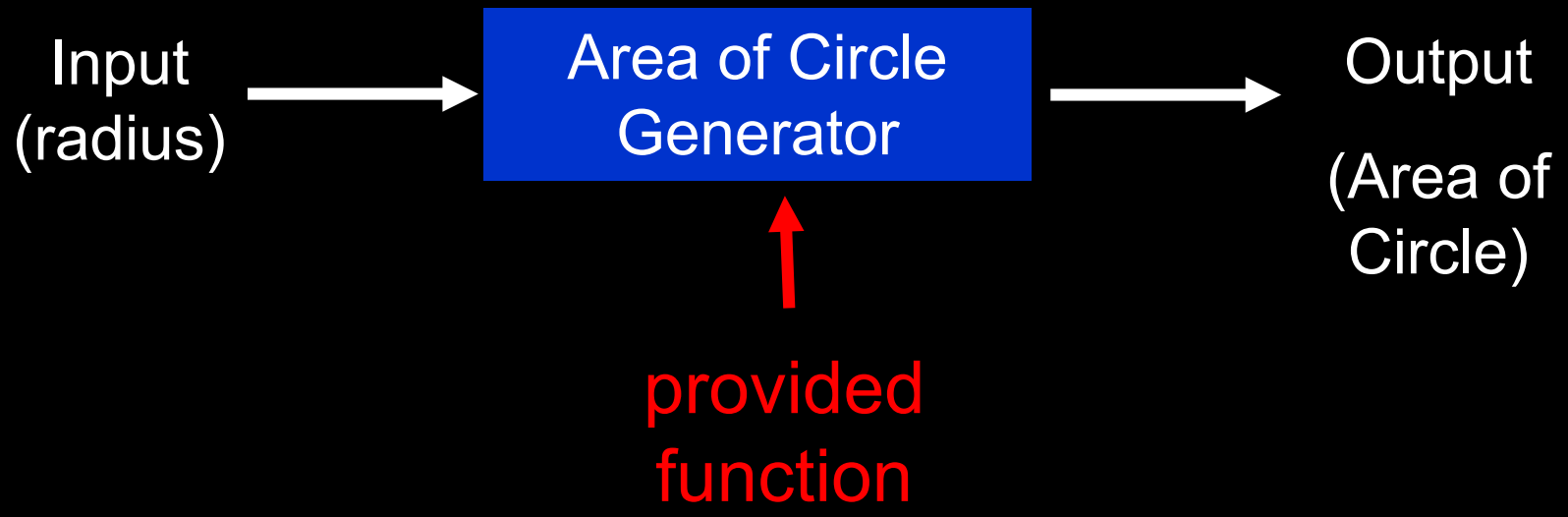
Recap: Functional Abstraction = Black Box

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

Functional Abstraction: Separates specification from implementation

Specification: WHAT it should do

Implementation: HOW it does it



Specification: WHAT

(Calculate the area of a circle given its radius

Implementation: HOW (inside the box)

Good Abstraction

7. Makes debugging (fixing errors) easier

Where is the bug?
Which is easier to check?

Code #1:

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

Where is the bug?
Which is easier to check?

Code #2:

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

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

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

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 the code to be reused
5. Hides the implementation details
6. Separates specification from implementation
7. Makes debugging (fixing errors) easier

Wishful Thinking

Top-down approach:
Pretend you have whatever you need (!)

An Example

Singapore Taxi determines the taxi fare based on distance traveled as follows:

For the first kilometre or less: $3.00

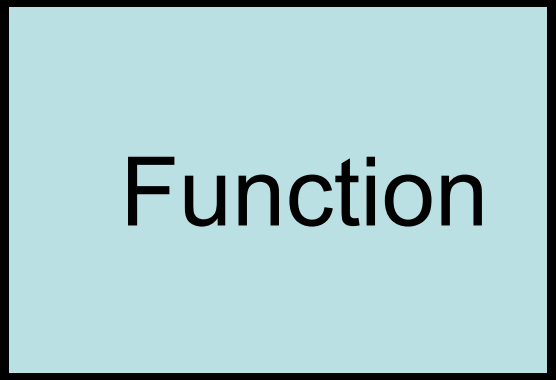
Every 400 metres thereafter or less up to 10 km: $0.22

Every 400 metres thereafter or less after 10 km: $0.30

Problem: Write a Python function that computes the taxi fare from the distance traveled.

How do we start?

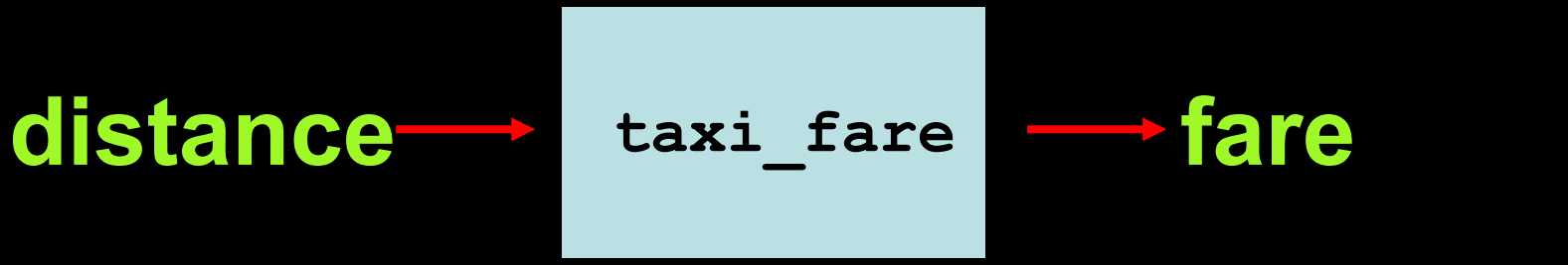
Formulate the problem



Needs a name!

Pick an appropriate name

Formulate the problem



- 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`
- How to get data? `use a argument`
- What is the result? `fare`

How can the result be computed from data?

- e.g.#1: distance = 800 m, fare = $3.00
- e.g.#2: distance = 3300 m,

  fare = $3.00 +roundup(2300/400) × $0.22 = $4.32
- e.g.#3: distance = 14500 m,

  fare = $3.00 +roundup(9000/400) × $0.22 + roundup(4500/400)× $0.25 = $11.06

Pseudocode

Case 1: distance <= 1000

fare = $3.00

Case 2: 1000 < distance <= 10,000

fare = $3.00 + $0.22 *

roundup((distance – 1000)/ 400)

Case 3: distance > 10,000

What's this? fare for 10 km

fare = $8.06 + $0.25 *

roundup((distance – 10,000)/ 400)

Solution

```
def taxi_fare(distance): # distance in metres
  if distance <= 1000:
    return 3.0
  elif distance <= 10000:
    return 3.0 + (0.22 *
                  ceil((distance - 1000) / 400))
  else:
    return 8.06 + (0.25 *
                  ceil((distance - 10000) / 400))

# check: taxi_fare(3300) = 4.32
```

Can we improve this solution?

Avoid Magic Numbers

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

- Hard to make changes in future.

Coping with Change

1. What if starting fare is increased to $3.20?

2. What if 400 m is decreased to 300 m?

Which are the magic numbers?

```
def taxi_fare(distance): # distance in metres
  if distance <= 1000:
    return 3.0
  elif distance <= 10000:
    return 3.0 + (0.22 *
                  ceil((distance - 1000) / 400))
  else:
    return 8.06 + (0.25 *
                  ceil((distance - 10000) / 400))
```

How to replace the magic numbers?

```
def taxi_fare(distance): # distance in metres
  start_fare = 3.0
  stage = 400

  if distance <= 1000:
    return start_fare
  elif distance <= 10000:
    return start_fare + (0.22 *
                  ceil((distance - 1000) / stage))
  else:
    return 8.06 + (0.25 *
                  ceil((distance - 10000) / stage))
```

How to remove this one???

```
taxi_fare(1000)
```

```
def taxi_fare(distance): # distance in metres
  start_fare = 3.0
  block = 400

  if distance <= 1000:
    return start_fare
  elif distance <= 10000:
    return start_fare + (0.22 *
                ceil((distance - 1000) / block))
  else:
    return taxi_fare(1000) + (0.25 *
                ceil((distance - 10000) / block))
```

Easily cope with changes

1. Increase the starting fare to $3.20

2. Decrease the 400 m block to 300 m

Can easily update these magic numbers

```
def taxi_fare(distance): # distance in metres
  start_fare = 3.2
  block = 300

  if distance <= 1000:
    return start_fare
  elif distance <= 10000:
    return start_fare + (0.22 *
                ceil((distance - 1000) / block))
  else:
    return taxi_fare(1000) + (0.25 *
                ceil((distance - 10000) / block))
```

Remove all the magic numbers !

```
def taxi_fare(distance): # distance in metres
    stage1 = 1000
    stage2 = 10000
    start_fare = 3.0
    increment = 0.22
    block = 400

    if distance <= stage1:
        return start_fare
    elif distance <= stage2:
        return start_fare + (increment *
            ceil((distance - stage1) / block))
    else:
        return taxi_fare(stage1) +
            (increment * ceil((distance - stage2) / block))
```