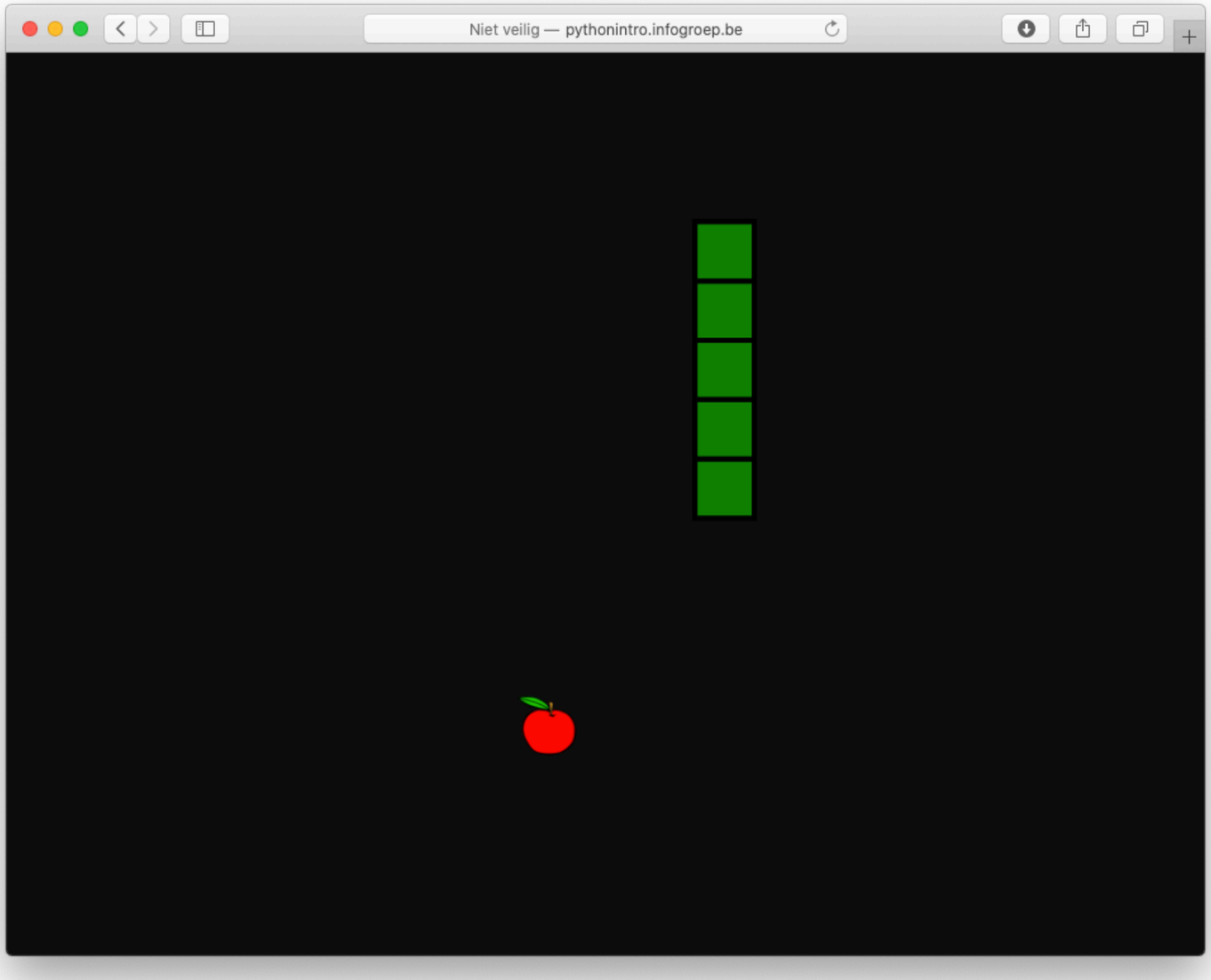
Python Advanced Concepts

bram@infogroep.be
@Infogroep
2020-11-19

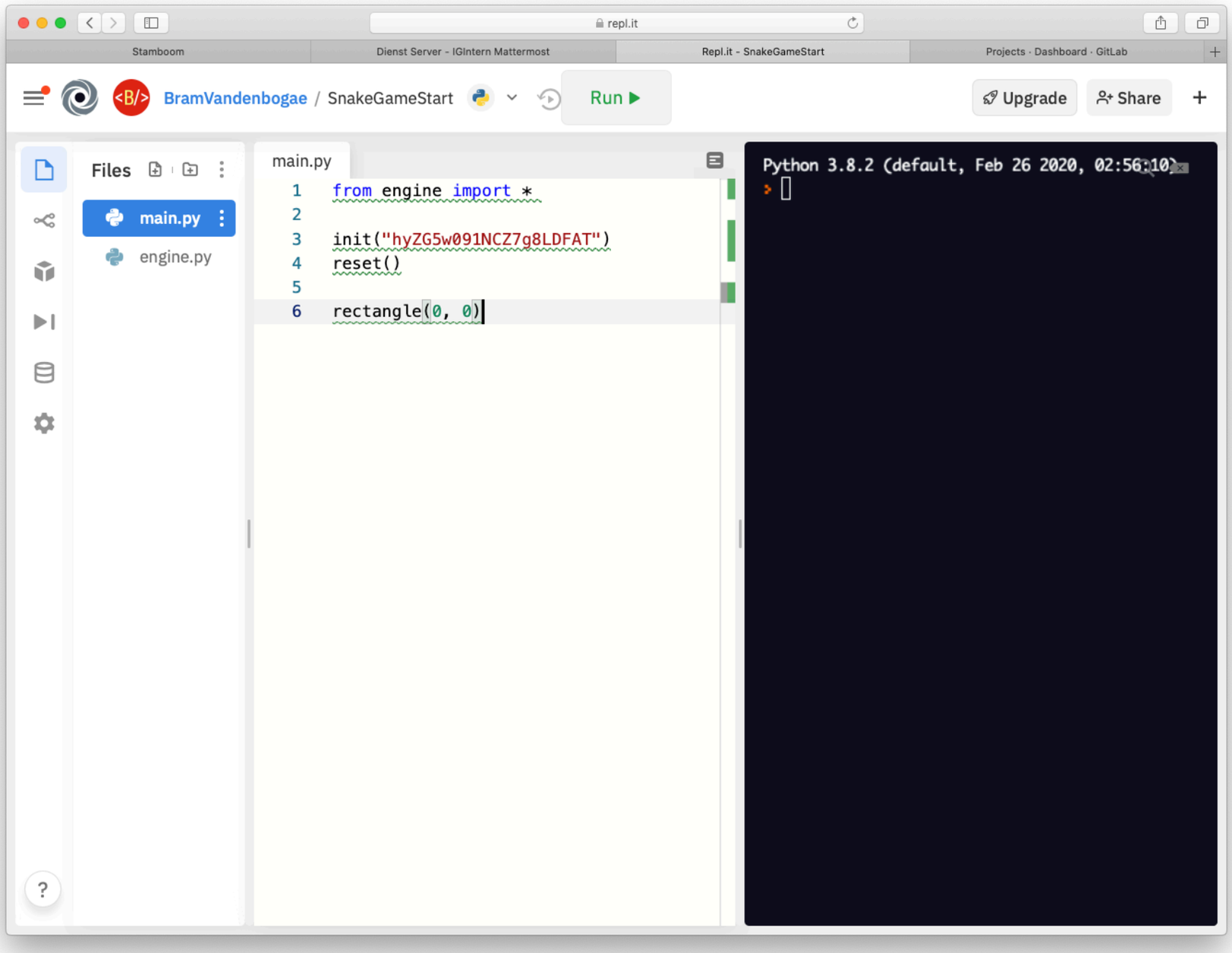
Goal

See last week: Introduction to Python

Goal: re-implement using classes



Participate



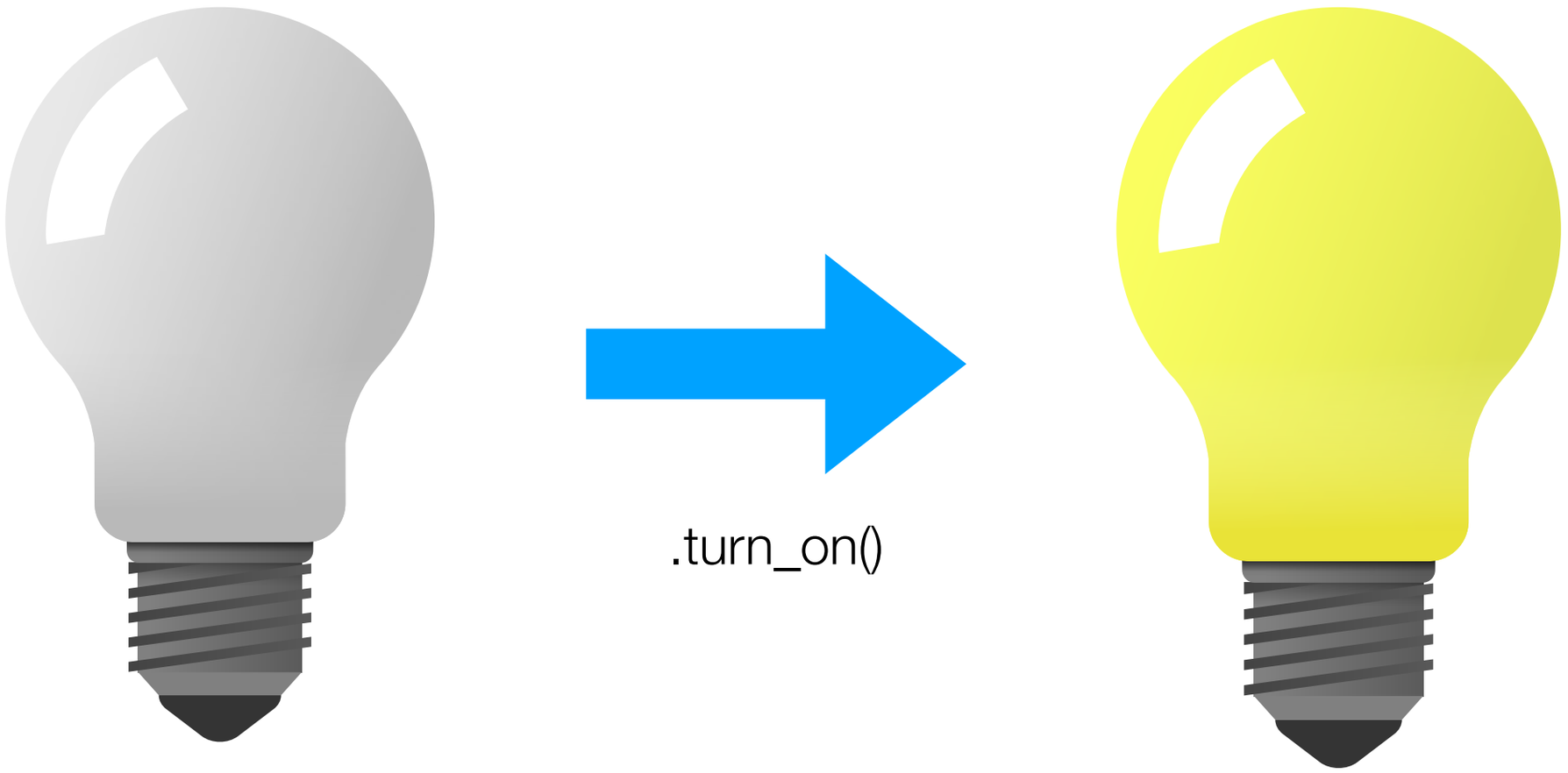
https://repl.it/@BramVandenbogae/SnakeGameStart

Topics

- Focus: Object Oriented Programming
- What are objects and classes?
- Defining methods?
- What is inheritance?
- Overriding methods
- Multiple inheritance

What are Objects? (1)

Modelling real-world things



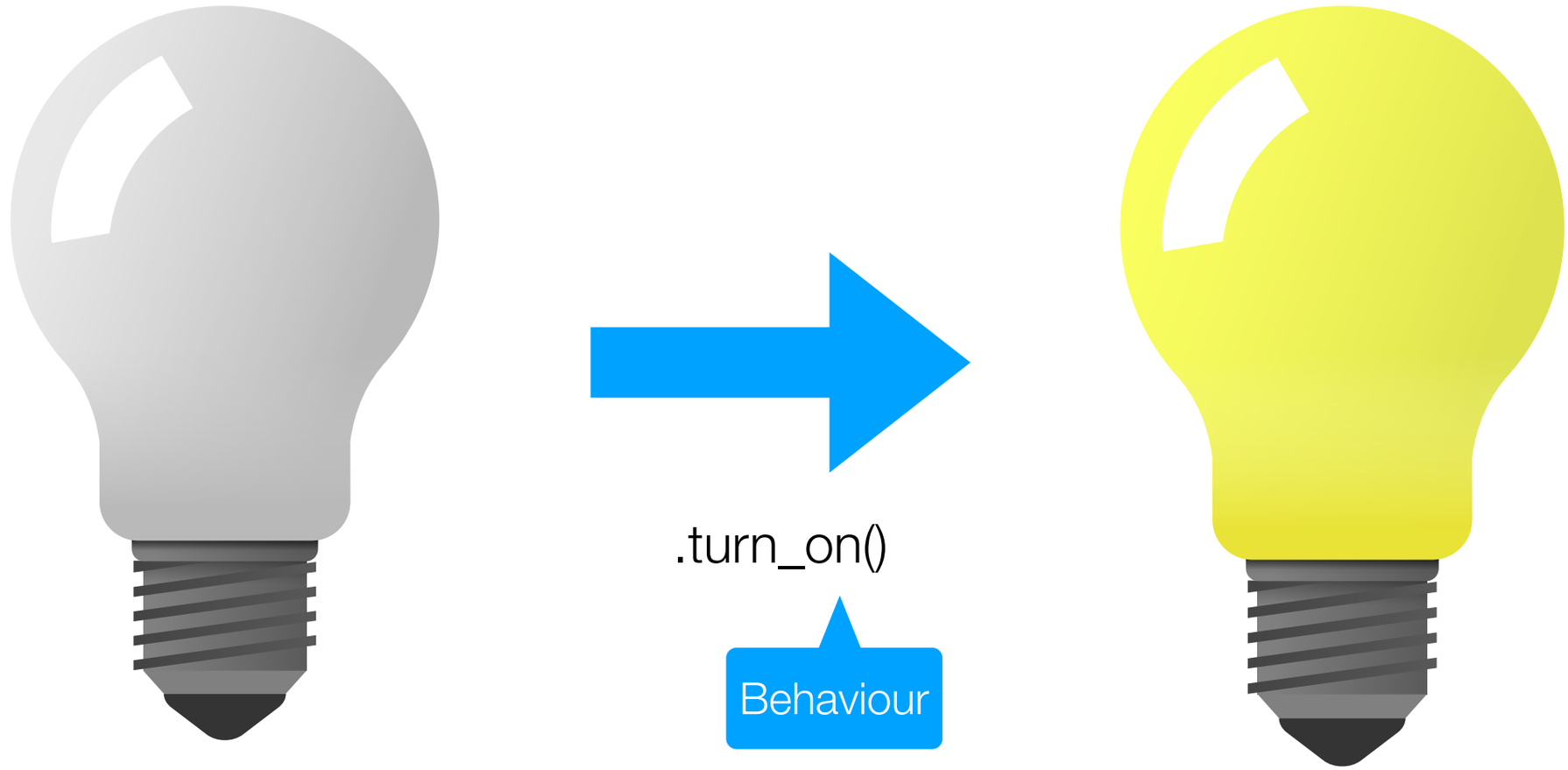
state: off

state: on

The light bulb is an object

What are Objects? (2)

Objects have **state** and **behaviour**



State

state: off

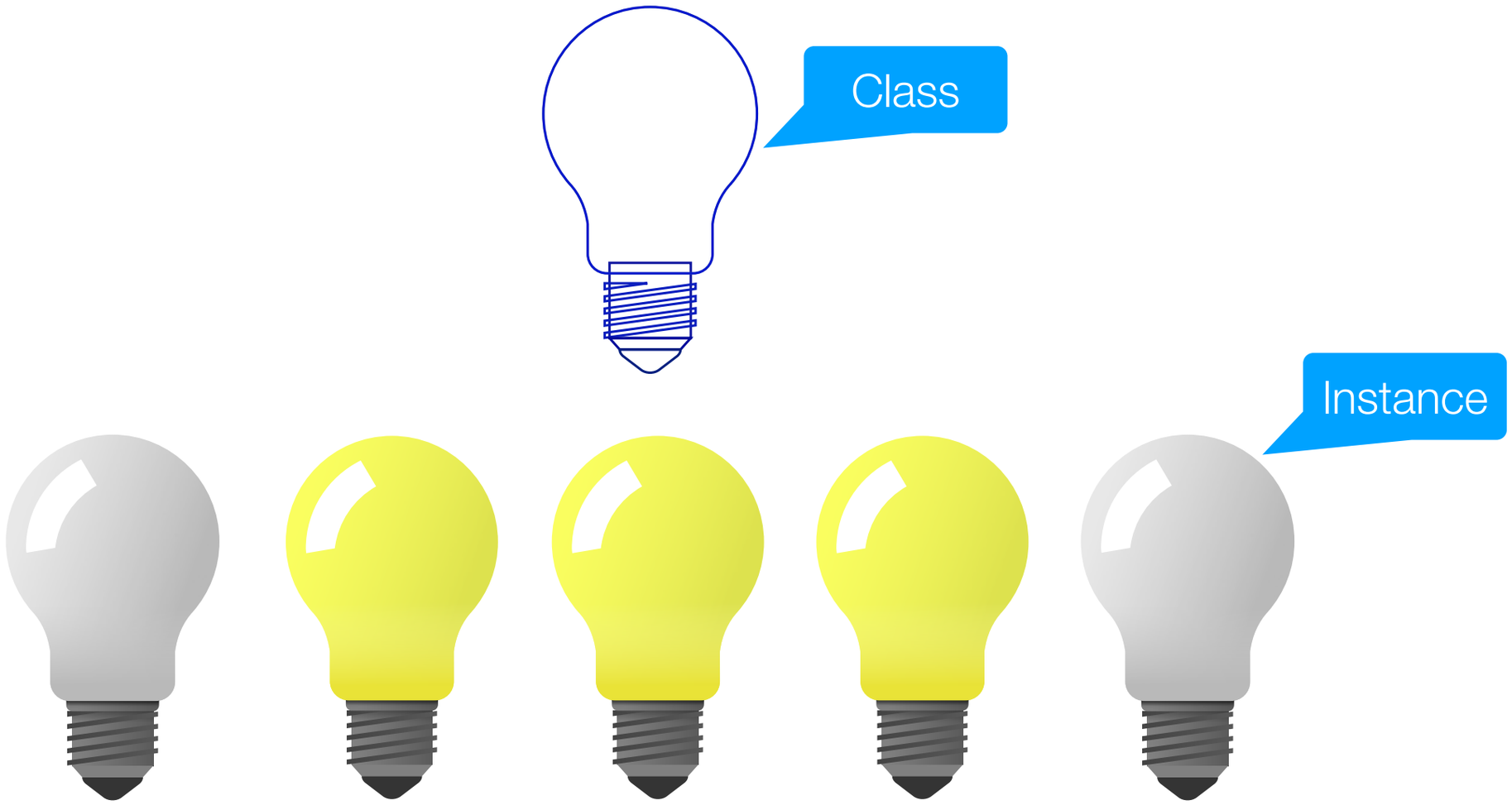
state: on

State is described by **variables**

Behaviour by **code**

What are Classes? (1)

There is more than one light bulb in the world



A class describes how objects should *look* and *behave*

What are Classes? (2)

A class is a **blueprint** for objects

They describe how instances (objects) should look

Fields (properties) describe the **state** of objects

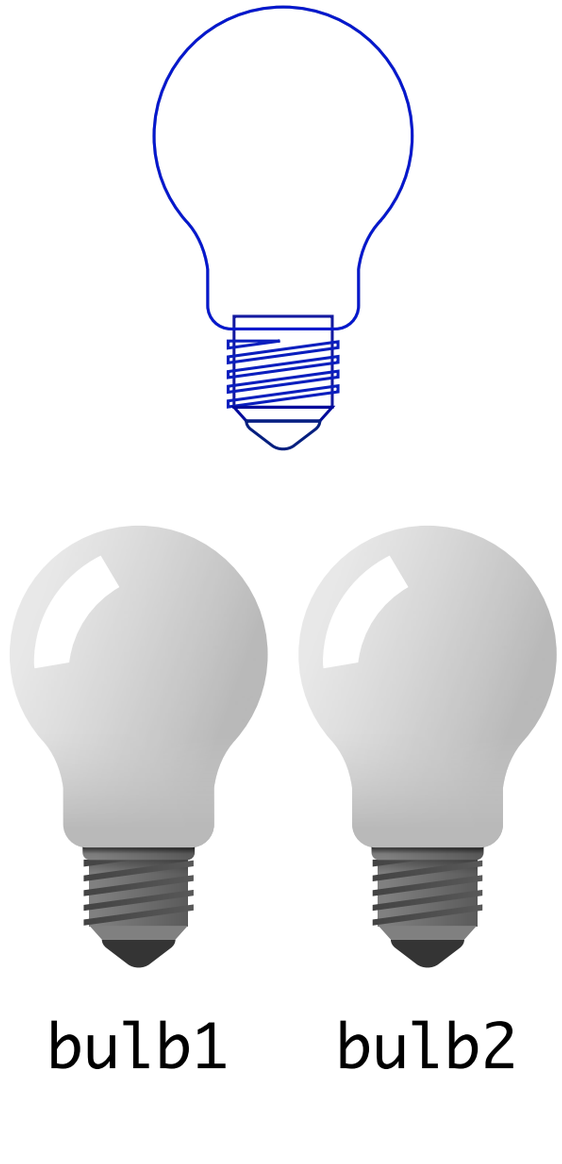
Methods describe the **behaviour** of objects

Defining a Class

```
class Bulb:
    pass
```

```
bulb1 = Bulb()
bulb2 = Bulb()
```

Creates a new instance of the class



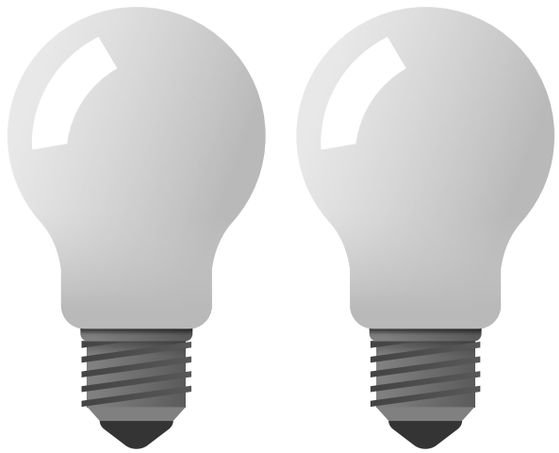
Describing State with Fields

```
class Bulb:
    def __init__(self):
        self.powered = False

bulb1 = Bulb()
print(bulb1.powered)
# => False

bulb2 = Bulb()
print(bulb2.powered)
# => False
```

Initialisation **method**

State described by **field**

Read the field

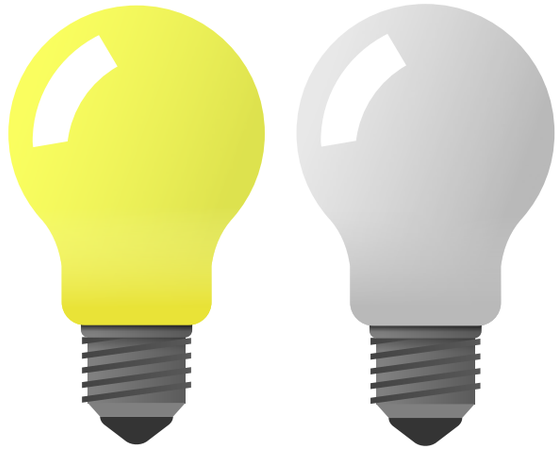


bulb1 bulb2

Changing State

```
class Bulb:
    def __init__(self):
        self.powered = False

    def turn_on(self):
        self.powered = True

    def turn_off(self):
        self.powered = False

    def toggle(self):
        self.powered = not self.powered

bulb1 = Bulb()
bulb1.turn_on()
print(bulb1.powered)
# => True

bulb2 = Bulb()
print(bulb2.powered)
# => False
```

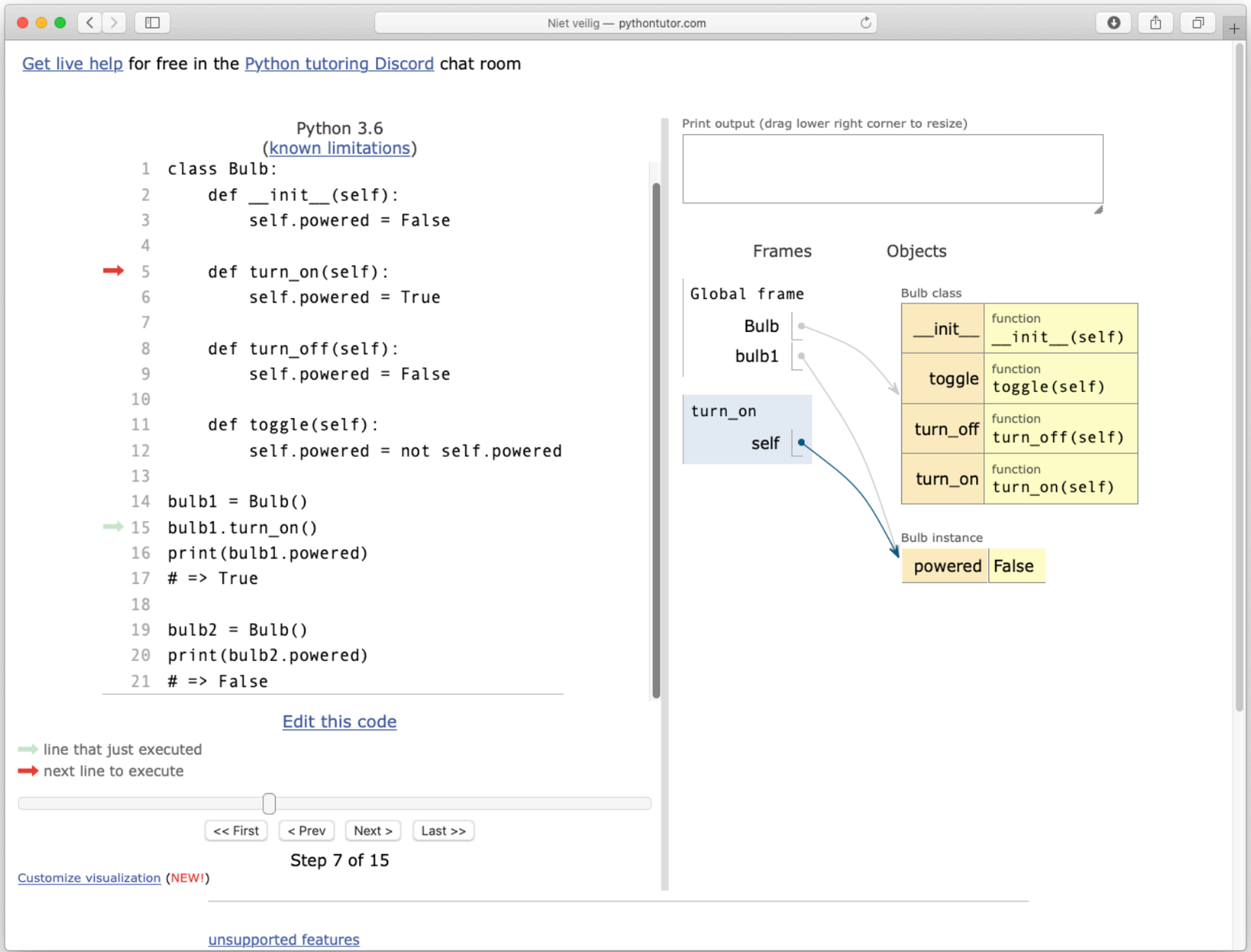
Run the behaviour (method)



bulb1 bulb2

What is `self`? (1)

`self` always points to a specific instance



Bank Account

A **bank account** should have a **balance**

A user should be able to **deposit** money on a bank account, and **withdraw** money from it.

Finally, money should be able to be **transferred** between bank accounts

Bank Account

```
class BankAccount:
    def __init__(self, init_amount):
        self.balance = init_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def transfer(self, other, amount):
        self.withdraw(amount)
        other.deposit(amount)

b1 = BankAccount(100)
b2 = BankAccount(200)
b2.transfer(b1, 100)
print(b1.balance)
# => 200
```

+= is short for self.balance = self.balance + amount

Redesigning Game: Apple

An **apple** should be able to **spawn**
and **move** on a random **row** and **column**

```
class Apple:
    def __init__(self, row, column):
        self.row = row
        self.column = column
        self.shape = apple(row, column)

the_apple = Apple(5, 5)
```

Moving the Apple (1)

```
class Apple:
    # ... see previous slide ...

    def move(self, row, column):
        self.row = row
        self.column = column
        move(self.shape, self.row, self.column)

def move_random(self):
    """
    Move randomly
    """

    self.move(random(0, 14), random(0, 14))
```

Documentation string

Moving the Apple (2)

Move the apple 5 times randomly on the screen

```
the_apple = Apple(5, 5)
for i in range(0, 10):
    the_apple.move_random()
    sleep(0.5)
```

Checking Collisions

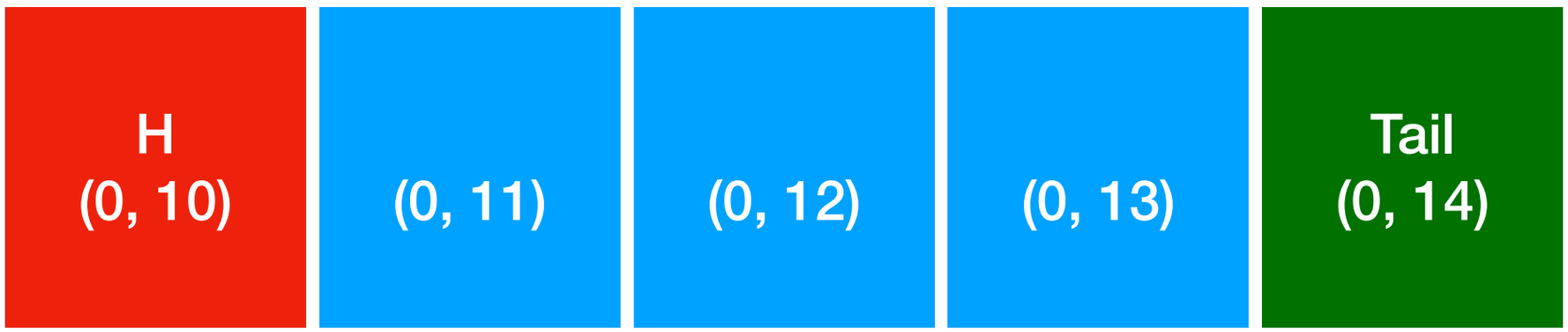
```
class Apple:
  # ... see previous slide ...

  def hasSamePosition(self, other):
    return hasSamePosition(self.shape, other.shape)
```

Redesigning Game: Snake

A snake consists of **snake elements**

A snake can **move** into a certain **direction**

We should be able to **check** whether the snake **collides with itself**



Snake Class Outline

```
class Snake:
    def __init__(self, row, column, length):
        pass

    def head(self):
        pass

    def has_collision(self):
        pass

    def move(self):
        pass

    def extend(self):
        pass
```

Snake (1)

```
class Snake:
  def __init__(self, row, column, length):
    self.row = row
    self.column = column
    self.snake = [ SnakeElement(row, column+i) for i in range(column, column+length)  ]

  def move(self):
    key = lastkey()
    if isKeyUp(key):
      self.__move_increment(-1, 0)
    elif isKeyDown(key):
      self.__move_increment(1, 0)
    elif isKeyLeft(key):
      self.__move_increment(0, -1)
    elif isKeyRight(key):
      self.__move_increment(0, 1)
    else:
      self.__move_increment(0, -1)
```

List comprehension

List Comprehensions

```
mylist = [ x*2 for x in range(0, 4) ]
print(mylist)
# => [0, 2, 4, 6]
```

Snake (2)

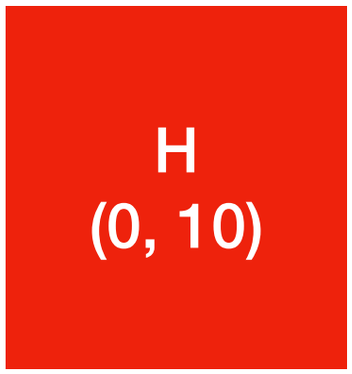
```
# ... see previous slide ...
def __move_increment(self, rowIncrement, columnIncrement):
  # first remove the tail
  tail = self.snake.pop()

  # get the head
  head = self.snake[0]

  new_row = (head.row+row_increment) % 15
  new_column = (head.column+column_increment) % 15

  # now move the tail...
  tail.move(new_row, new_column)

  # ...and add it to the front of the list
  self.snake.insert(0, tail)

# ... see previous slide ...
```

A “private” method

Snake (3)

```
# ... see previous slide ...

def head(self):
  return self.snake[0]

def has_collision(self):
  head = self.head()
  collision = False
  for square in self.snake[1:]:
    if hasSamePosition(head, square):
      collision = True

  return collision
```

Snake Element




```
class SnakeElement:
  def __init__(self, row, column):
    self.row = row
    self.column = column
    self.shape = rectangle(row, column)

  def move(self, row, column):
    self.row = row
    self.column = column
    move(self.shape, row, column)

  def hasSamePosition(self, other):
    return hasSamePosition(self.shape, other.shape)
```

```
head = SnakeElement(0, 10)
```

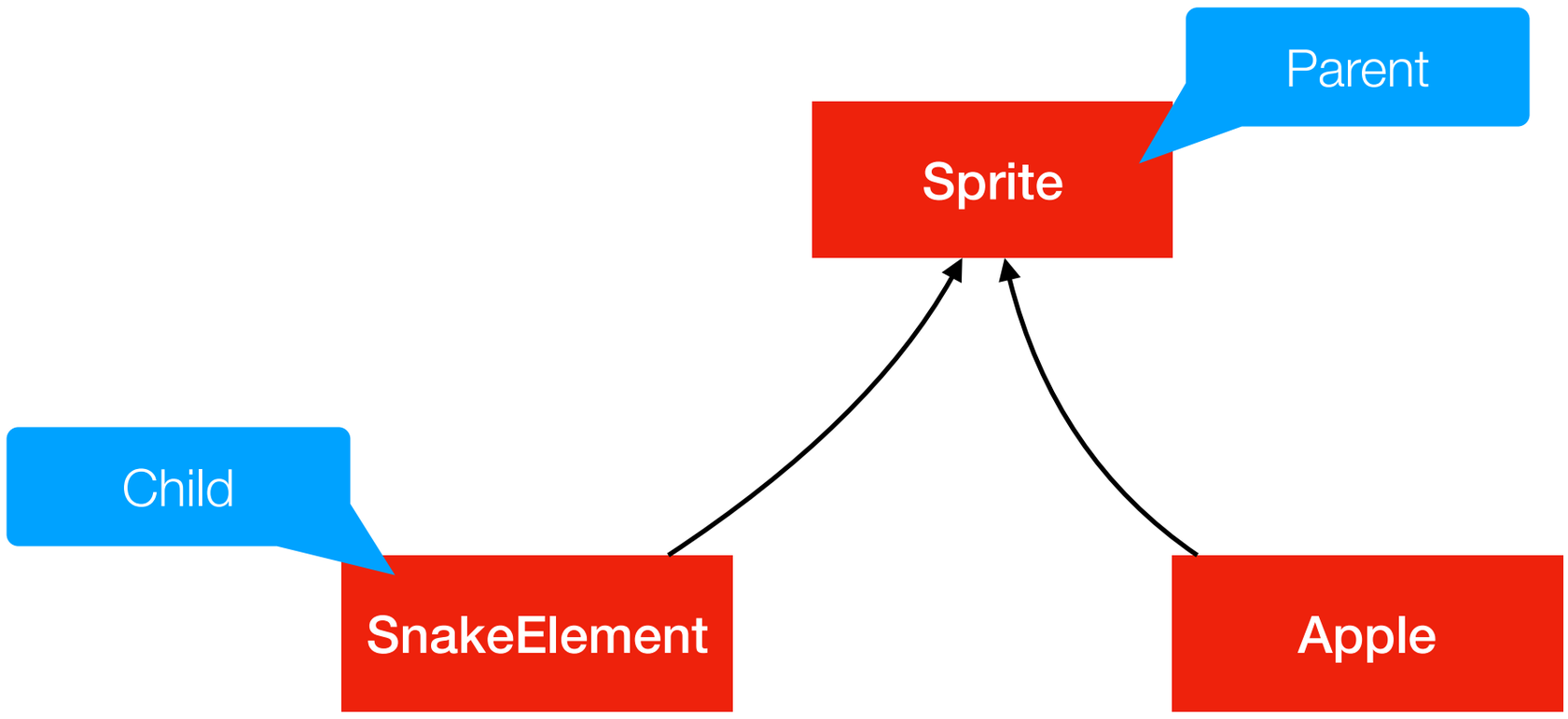
Inheritance (1)

```
class SnakeElement:
  def __init__(self, row, column):
    self.row = row
    self.column = column
    self.shape = rectangle(row, column)

def move(self, row, column):
  self.row = row
  self.column = column
  move(self.shape, row, column)
```

```
class Apple:
  def __init__(self, row, column):
    self.row = row
    self.column = column
    self.shape = apple(row, column)

def move(self, row, column):
  self.row = row
  self.column = column
  move(self.shape, row, column)
```

SnakeElement and Apple have a few things in common:
row, column and shape

Inheritance (2)



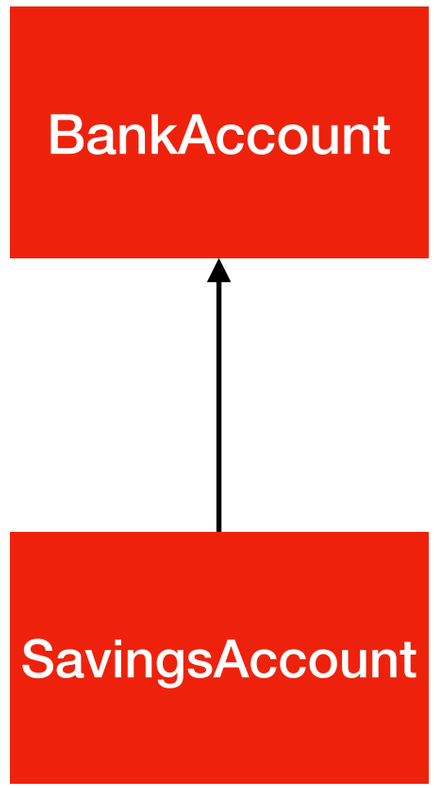
Classes can inherit properties and methods from their parents

Inheritance (3)

```
class BankAccount:
    def __init__(self, init_amount):
        self.balance = init_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def transfer(self, other, amount):
        self.withdraw(amount)
        other.deposit(amount)

class SavingsAccount(BankAccount):
    def withdraw(self, amount):
        if self.balance - amount >= 0:
            super().withdraw(amount)
        else:
            print("Insufficient money")

b1 = SavingsAccount(0)
b1.deposit(50)
b1.withdraw(100)
# => Insufficient Money
```

Override (overwrite) method from parent

Use withdraw of the parent



Generic Sprite

```
class Sprite:
  def __init__(self, row, column, shape):
    self.row = row
    self.column = column
    self.shape = shape

  def move(self, row, column):
    self.row = row
    self.column = column
    move(self.shape, self.row, self.column)

  def hasSamePosition(self, other):
    return hasSamePosition(self.shape, other.shape)
```

Apple as Sprite

```
class Apple(Sprite):
    def __init__(self, row, column):
        super().__init__(row, column, apple(row, column))

def move_random(self):
 """
 Move randomly
 """

 self.move(random(0, 14), random(0, 14))
```

Override (overwrite)
method from parent

Inherited methods are gone

SnakeElement as Sprite

```
class SnakeElement(Sprite):
    def __init__(self, row, column):
        super().__init__(row, column, rectangle(row, column))
```

Override (overwrite)
method from parent

Combining Everything (1)

A **Game** should have some code that **executes one iteration of the game loop.**

It should keep track of a **score**.

It should be able to display a **game over** message,
and it should be able to check when the snake **hits the apple.**

```
class Game:
    def __init__(self):
        pass

    def gameover(self):
        pass

    def check_apple(self):
        pass

    def gametick(self):
        pass
```

Combining Everything (2)

```
class Game:
  def __init__(self):
    self.apple = Apple(random(0, 15), random(0, 15))
    self.snake = Snake(0, 10, 4)
    self.score = 0

  def gameover(self):
    print("Game over, your score was ", self.score)
```

```
def check_apple(self):
  head = self.snake.head()
  if head.hasSamePosition(self.apple):
    self.score = self.score + 1
    self.snake.extend()
    self.apple.move_random()

def gametick(self):
   self.snake.move()
   self.check_apple()
   return self.snake.has_collision()
```

Combining Everything (3)

```
game = Game()
while True:
    if game.gametick():
        break

game.gameover()
```

Summary

- What are objects and classes?
- Defining methods?
- What is inheritance?
- Overriding methods
- Multiple inheritance