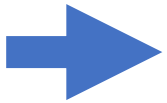
Object Oriented Programming

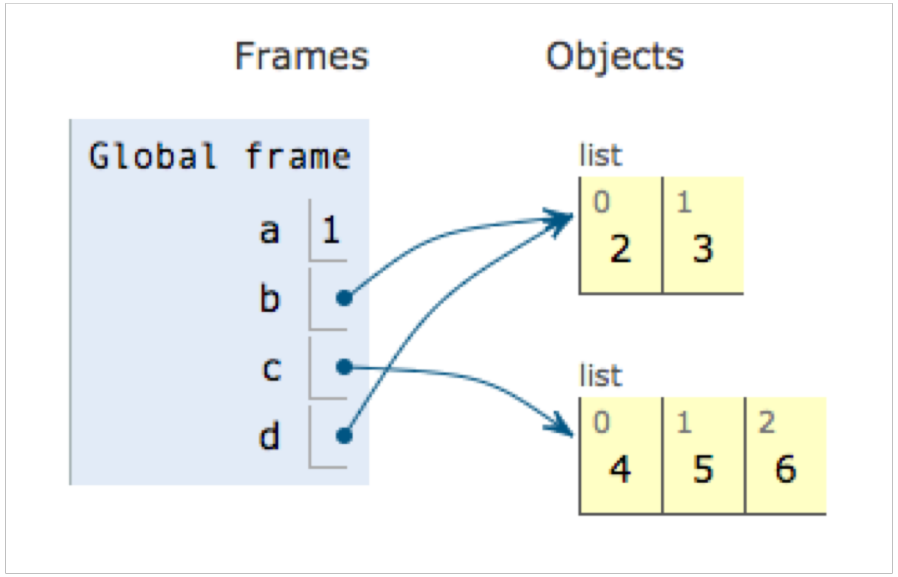
Prof. Dr. Harald Gall

University of Zurich, Department of Informatics

What about these Objects all the time?

```
a = 1
b = [2,3]
c = [4,5,6]
d = b
```





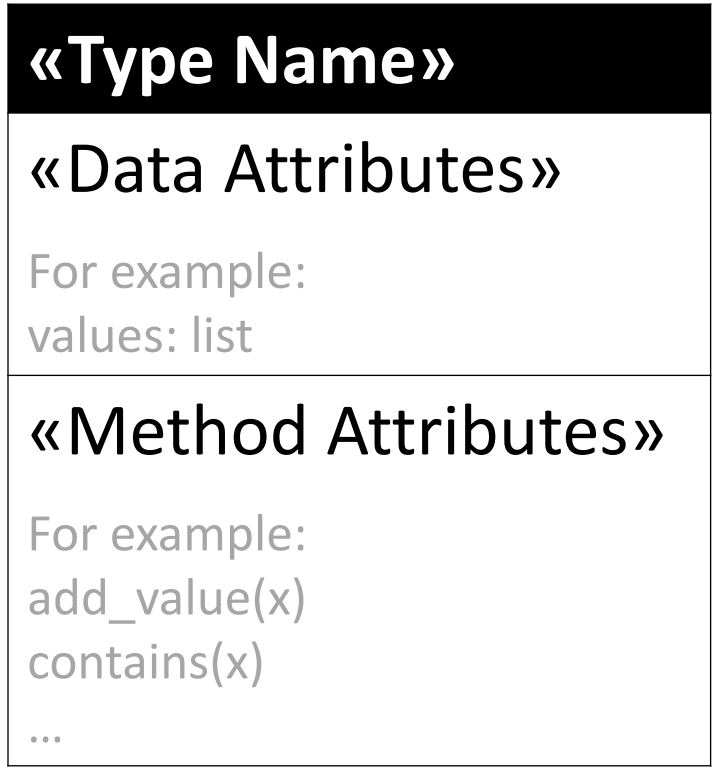
Everything in Python is an Object

- Integers 3
- Strings 'Hello, world!'
- Lists [“dog”, “bird” , “cat”]
- Dictionaries {’a': 1, ‘b’: 2, ‘c’: 3}
- Tuples (1, 2)
- ...

Definition: Classes

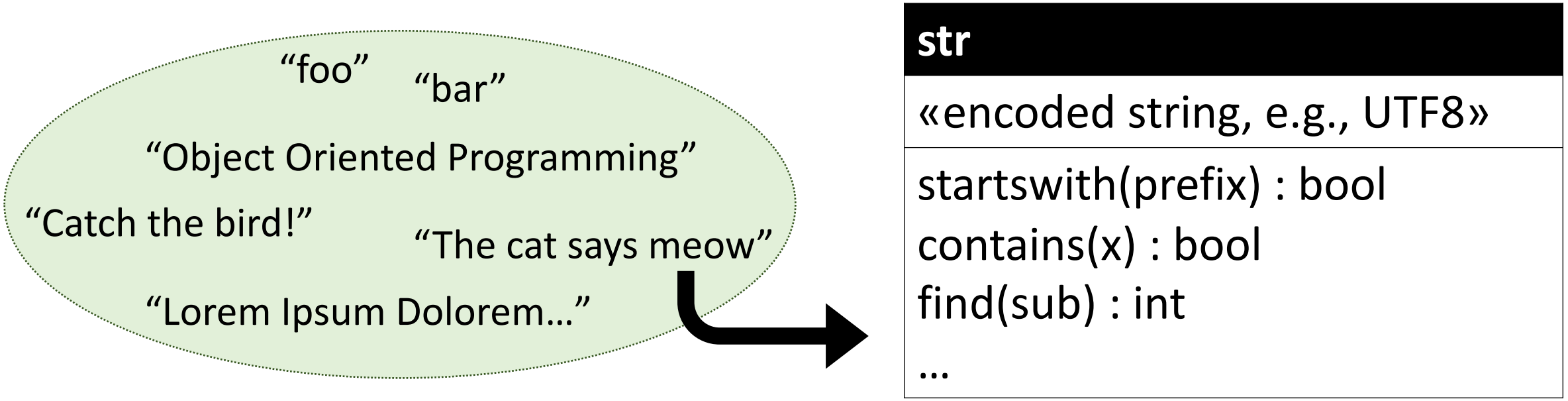
A class defines a new object type by specifying its name, its internal representation (data attributes), and the corresponding operations (method attributes) used to interact with the internal data.

Abstract Representation:



Terminology: Classes, Object, Instance

An *object* is the *instance* of a *class* definition.



str Objects **Each One Is an Instance of** **Class Definition**

Another Concrete Example

Instance: `[12, 3, 45, 678]`

Type Name: list

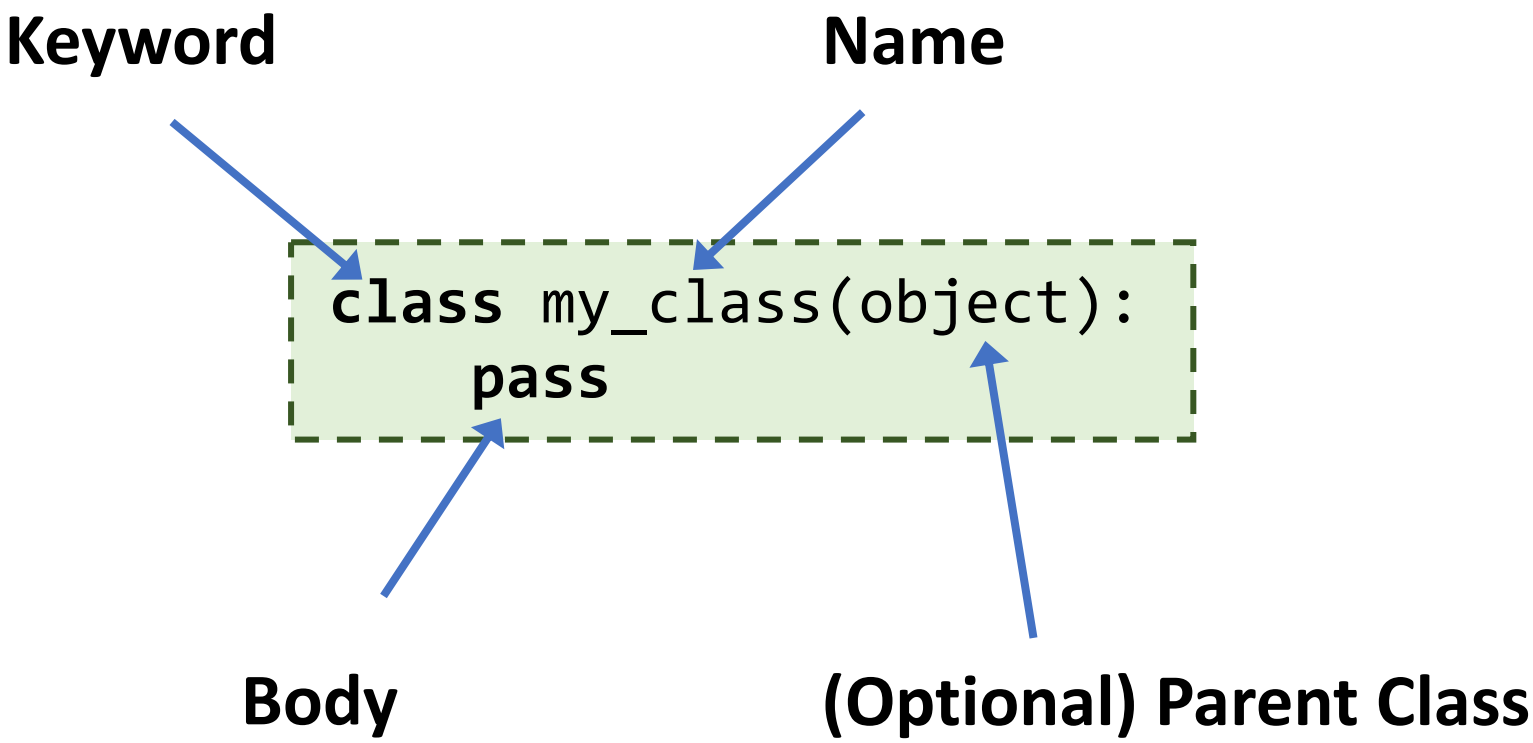
Internal Representation:

Operations:

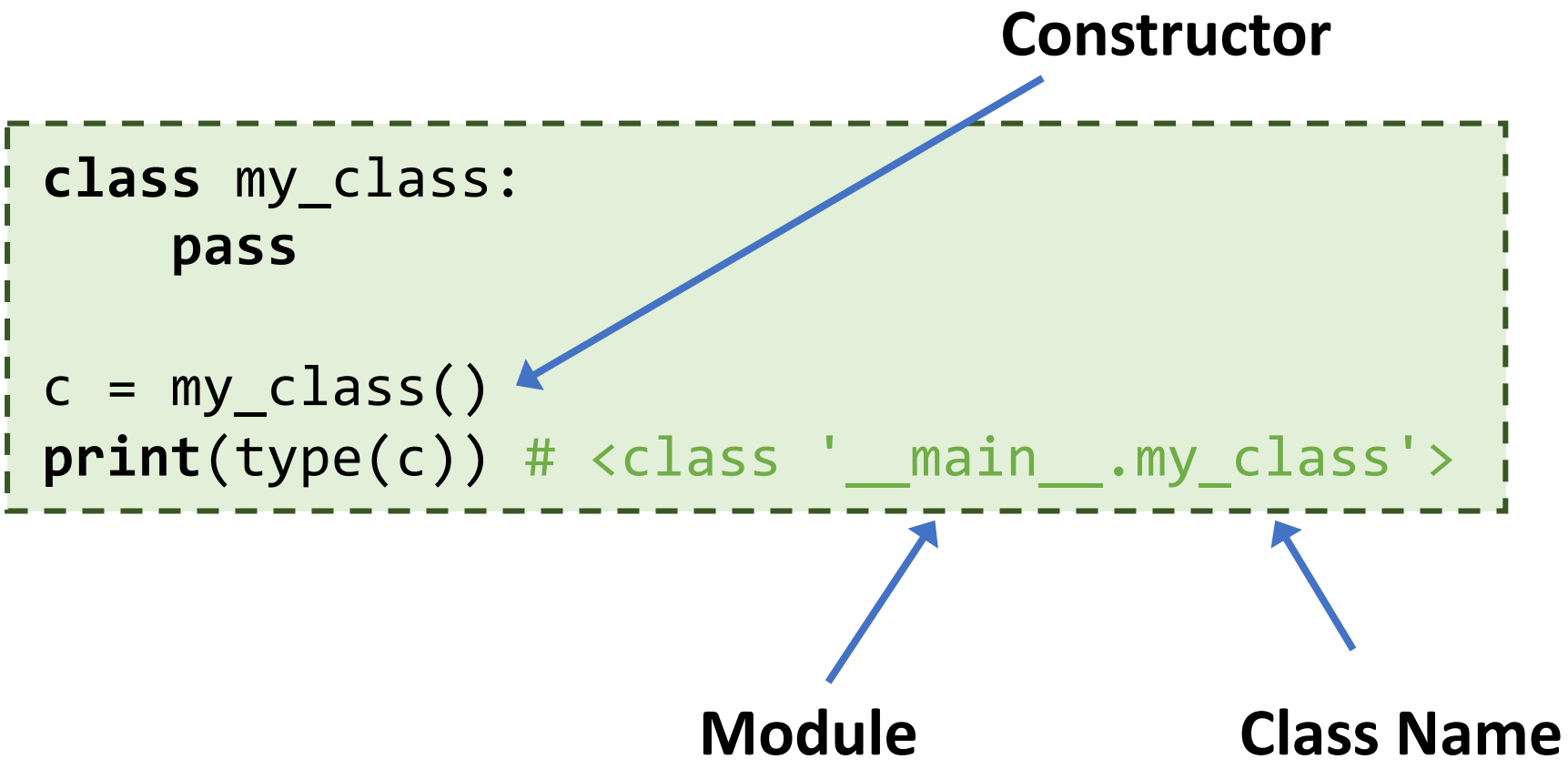
- append(x)
- clear()
- reverse()
- …

Working with Classes in Python

Empty Class Definition in Python



Creating an Instance of The Empty Class



Deleting Instances of a Class

We can either use the `del` operator or just leave it to Python to decide when the object instance is not *reachable* anymore (garbage collection).

```
c = my_class()
print(c)
del c
print(c) # NameError: name ’c' is not defined
```

Using a Constructor to Define Data Attributes

- The special method `__init__` can be used to define data attributes
- The first parameter of any method is a reference to the instance
- *By convention*, the first parameter, `self`, points to the instance
- Data attributes are also called instance variables

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
```

Instantiation and Access to Data Attributes

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

c1 = ComplexNumber(1, 2)
print(c1.real) # prints ‘1’
```

The self reference is provided implicitly by Python

Use the dot notation to access data attributes

Instantiate Multiple Object of the Same Type

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

c1 = ComplexNumber(1, 2)
c2 = ComplexNumber(3, 4)

print(c1.real) # prints ‘1’
print(c2.real) # prints ‘3’
```

Instance Variables Can Be Made Private

- By default, instance variables in Python are publicly accessible
- If instance variables should only be used internally, they can be marked by starting their names with __ (double underscore)
- Python mangles such variable names (they are not easily accessible afterwards)

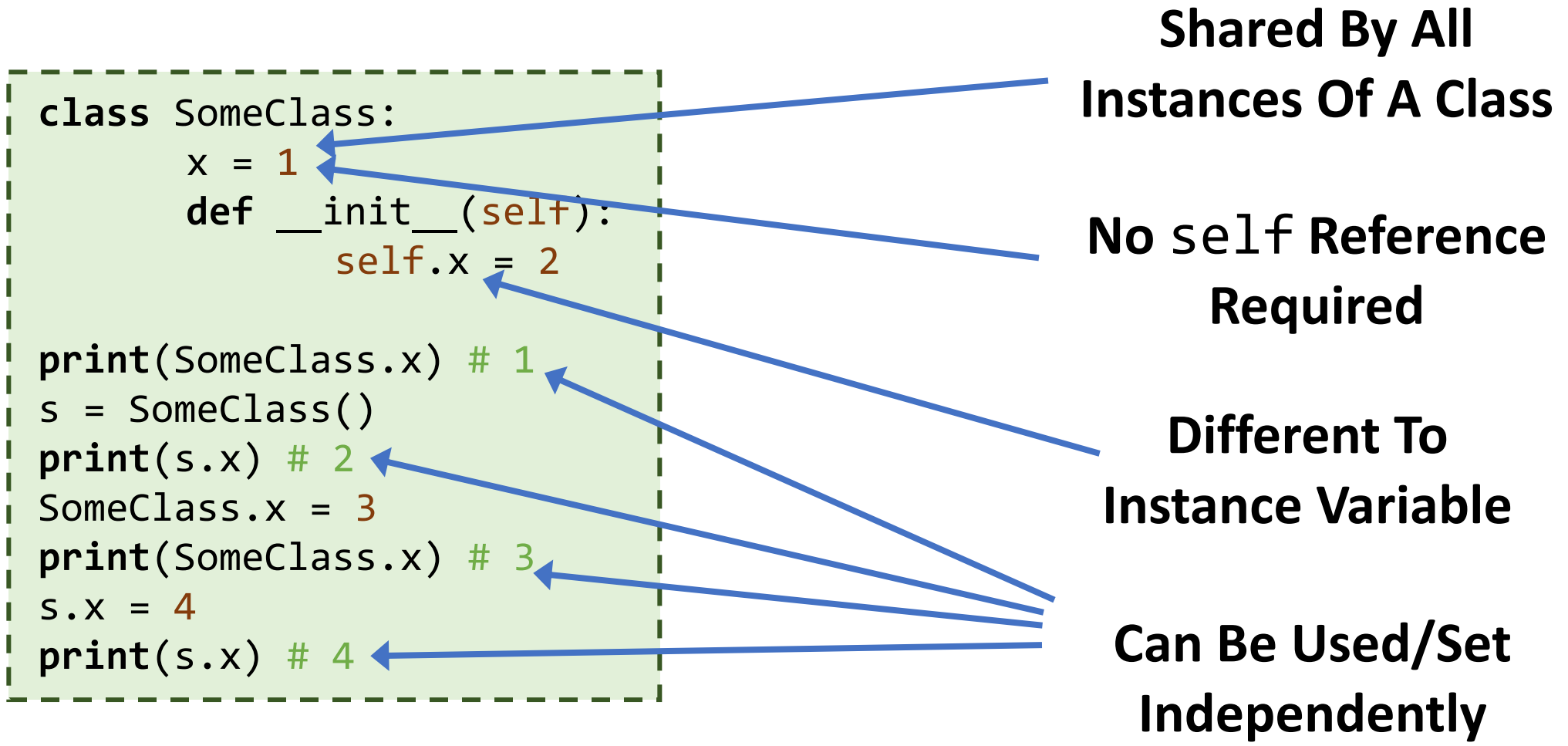
```
class Example:
    def __init__(self, a, b):
        self.public = a
        self.__hidden = b

o = Example(1, 2)
print(o.public) # prints ‘a’
print(o.__hidden)
# AttributeError: ’Example' object has no attribute '__hidden'
```

Class Variables

```
class SomeClass:
        x = 1
        def __init__(self):
                self.x = 2

print(SomeClass.x) # 1
s = SomeClass()
print(s.x) # 2
SomeClass.x = 3
print(SomeClass.x) # 3
s.x = 4
print(s.x) # 4
```



Declaring Method Attributes

Methods Are Functions Defined In A Class

```
class ComplexNumber:
    …
    def add_to_real(self, delta):
        self.real += delta

c = ComplexNumber(1, 2)
print(c.real) # prints ‘1’
ComplexNumber.add_to_real(c, 2)
print(c.real) # prints ‘3’
```

Also Methods Make Use of the `self` Reference

Methods Can Access Data Attributes

Use the dot notation to access method attributes

Three Equivalent Ways of Invoking a Method

```
c = ComplexNumber(1, 2) # create

ComplexNumber.add_to_real(c, 2)

m = ComplexNumber.add_to_real
m(c, 2)

c.add_to_real(2)
```

Example From Last Slide

Methods Are First Class Objects, Like Functions

Python Provides Convenient Short-Hand Form Through Dot Notation

Static Methods

- Functions that are defined inside the class but don't access any of its instance variables (therefore do not need `self`)
- they are defined using the **@staticmethod** annotation

```
class ComplexNumber:
    …
    @staticmethod
    def add(c1, c2):
        return ComplexNumber(c1.real+c2.real, \
                             c1.imag+c2.imag)

n1 = ComplexNumber(1, 2)
n2 = ComplexNumber(2, 3)
n3 = ComplexNumber.add(n1, n2)
print(n3.real) # 3
```

Concrete Example 2: Safe

https://www.allaboutsafes.co.uk/safes-c128/chubbsafes-elements-air-hotel-safe-p112

Concrete Example 2: Safe

Type name	**Safe**
Internal Representation	???
Operations	set_content(pin: int, content: string) get_content(pin: int) : string

```
s = Safe()
s.set_content(123, “My Secret”)
print(s.get_content(123)) # “My Secret”
print(s.get_content(456)) # None
```

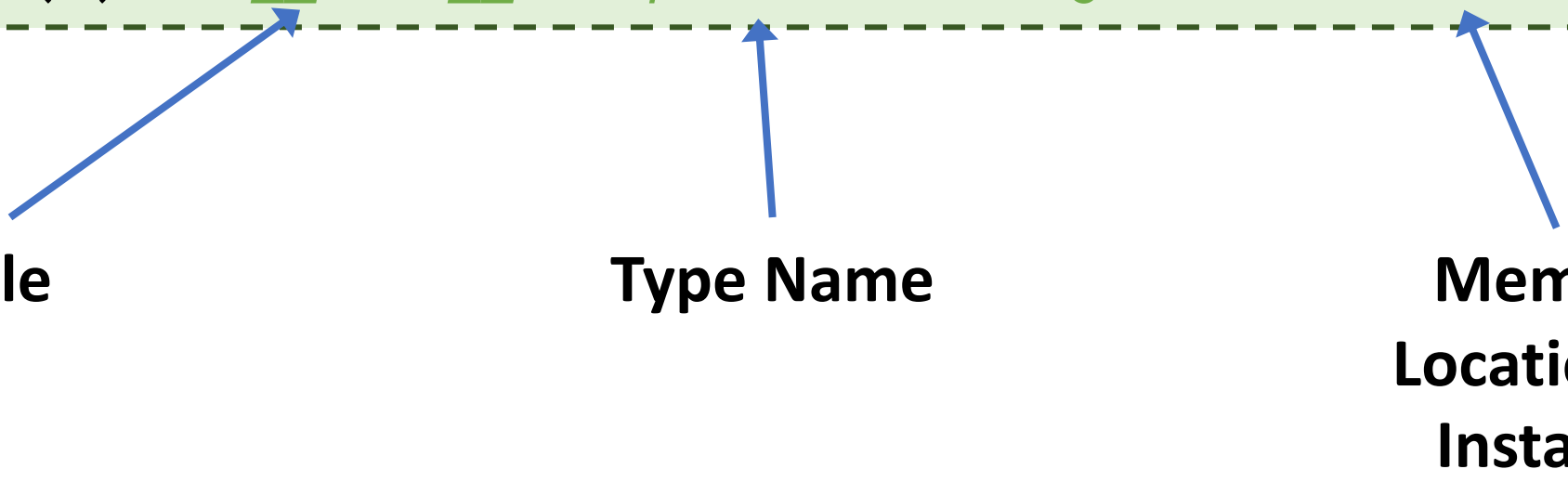
Advantages of Defining Classes

- Reduces complexity
  (coherent bundle of data and its methods)
- Facilitates testing (we can test the classes separately)
- Encourages reuse (we can reuse classes defined by other people)
- Users don’t need to know the inner workings of a class, they just rely on the public interface (= operators)
- It is possible to hide information that users don’t need to understand
- Class can Inherit Data and Behavior From Parent Class (Next Week)

Special Methods

By Default, Printing An Object Is Not Helpful

```
n = ComplexNumber(1, 2)
print(n) # <__main__.ComplexNumber object at 0x108ed82b0>
```



Redefine __str__ to Improve Output

```
class ComplexNumber(object):
    …
    def __str__(self):
        return "%.2f + %.2fi" % (self.real, self.imag)

n = ComplexNumber(1, 2)
print(n) # 1.00 + 2.00i
```

Goal of `__str__` is to create readable representation of an object.

Why doesn't it work in collections?

```
n = ComplexNumber(1, 2)
print(n) # 1.00 + 2.00i
print([n]) # [ <__main__.ComplexNumber object \
                    at 0x1140dd1d0> ]
```

Collections use `__repr__` to print the contained objects.

Define __repr__ Instead!

```
class ComplexNumber(object):
    …
    def __repr__(self):
        return ”ComplexNumber(%f,%f)" % (self.real, self.imag)

n = ComplexNumber(1, 2)
n_repr = n.__repr__() # “ComplexNumber(1,2)”

print([n]) # [ ComplexNumber(1,2) ]
print(eval(n_repr) == n) # True (see next slide)
```

Goal of `__repr__` is to create unambiguous representation of an object.

By Default, Different Objects Are Not Equal

```
n1 = ComplexNumber(1, 2)
n2 = ComplexNumber(1, 2)

print(n1 == n2) # False
```

Redefine __eq__ to Allow Equality Check

```
class ComplexNumber(object):
  …
  def __eq__(self, other):
    return isinstance(other, ComplexNumber) \
           and self.real == other.real \
           and self.imag == other.imag

n1 = ComplexNumber(1, 2)
n2 = ComplexNumber(1, 2)
print(n1 == n2) # True
```

Make Sure Types Match

Only then Compare Data Attributes

By Default, Classes Cannot Be Hashed

```
n = ComplexNumber(1, 2)

d = {}
d[n] = “…” # TypeError: unhashable type: 'ComplexNumber'
```

Define __hash__ to Use Instances as Keys

```
class ComplexNumber(object):
    …
    def __hash__(self):
        return (101 * self.x) + self.y

d = { ComplexNumber(1, 2): “…” }

print(d) # { ComplexNumber(1,2): ’…’ }
```

__hash__ must return the same value for objects that are __eq__.

Other Special Methods…

- __len__ (e.g., len(o))
- __add__ (e.g., a + b)
- __sub__ (e.g., a - b)
- …

https://docs.python.org/3/reference/datamodel.html

Objects and Data Structures

An Example of Good Coding Practice

Example of an Object

```
class Car:
    def __init__(self):
        self.__speed = 0

    def accelerate(self):
        self.__speed += 1

    def break(self):
        if self.__speed > 0:
            self.__speed -= 1

    def get_state(self):
        return "running" if \
               self.__speed > 0 \
               else "standing"
```

Objects hide their data behind abstractions and only expose functions that operate on that data.

Programmers just need to know how to use (and what to expect from) the public interface, but no implementation details.

Example of a Data Structure

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def get_age(self):
        return self.__age

p = Person(“Bob”, 27)
print(p.name)
print(p.get_age())
```

Data structures expose their data and have no meaningful functions.

“Getters”/”Setters” introduce indirection, but they also expose Implementation details

Differences Between Objects and Data Structures

- It is easy to add new kinds of **objects** without changing existing behaviors. It is hard though to add new behaviors to existing objects.
- It is easy to extend existing **data structures** with new data, but it is hard to adapt existing functions, afterwards.

Robert C. Martin: ”Clean Code: A Handbook of Agile Software Craftsmanship”

Summary

You should be able to answer these questions

- What is a class? How to define one?
- What is the difference between a class and an instance?
- How do we declare and access data attributes?
- How do we define and invoke methods?
- What is the difference between…
  - a function and a method?
  - a data structure and an object?
  - a class variable, an instance variable and a local variable?
  - a static and a non-static method?
- What is information hiding and why is it important?
- What is the purpose of __str__/__repr__, __eq__, and __hash__?

Exercises

Exercise 1: Pizza Shop

Pizza Order Print-Out

```
Your Order:
* Pizza “Vegi” (Tomato, Cheese, Mushrooms)
* Pizza “Bacon” (Tomato, Cheese, Bacon, Peperoni)

Total: 30CHF
```

Design a Cashier System for a Pizza Shop

- What are the important concepts in the domain?
  - Data Structures? Objects?
- What are the interactions?
- What information do we want to hide?
  (Not for security! Only because we might want to change it later...)

Exercise 2

Soccer Championship

Exercise

For this exercise you should implement a soccer championship simulator by following the next steps:

(a) implement a Player class, each player will have a name and a skill variable represented by a number from 1 – 10, the latter is initialized randomly when the object is created

Exercise

(b) implement a Team class, this has the following attributes: the country string, a trainer (for simplicity just a regular player object), and a list of 11 players. The players should just be generated with names “player 1” to “player “11” (and “trainer”) when creating a team.

Exercise

(c) implement the Game class, which consists of two teams (home and away). Teams are instance variables. The game has a play method that will return the team that wins (it's the team with the most skilled players and trainer – the trainer counts double). If the skills are the same, the home team wins.

Exercise

(d) implement the soccer championship simulator with the following 8 teams. In every round, the winner advances. You should print in each round what teams are currently playing and who won by how much.

Teams: ['BR', 'AR', 'IT', 'DE', 'NL', ’EN', 'AT', 'CH']

(shuffle the list of teams so that it’s not the same teams playing all the time)

Exercise

(a) implement a Player class, each player will have a name and a skill variable represented by a number from 1 – 10, the latter is initialized randomly when the object is created

Defining the Player class

```
class Player(object):
    pass
```

Add the __init__ method

```
class Player(object):
    def __init__(self, name):
        self.__name = name
```

Initialize the skill variable randomly

```
import random

class Player(object):
    def __init__(self, name):
        self.__name = name
        self.__skill = random.randint(1, 10)
```

Define a getter for skill

```
import random

class Player(object):
    def __init__(self, name):
        self.__name = name
        self.__skill = random.randint(1, 10)

    def get_skill(self):
        return self.__skill
```

Implement the __str__ method

```
import random

class Player(object):
    def __init__(self, name):
        self.__name = name
        self.__skill = random.randint(1, 10)

    def get_skill(self):
        return self.__skill

    def __str__(self):
        return "Player " + self.__name
```

Exercise

(b) implement a Team class, this has the following attributes: the country string, a trainer (for simplicity just a regular player object), and a list of 11 players. The players should just be generated with names “player 1” to “player “11” (and “trainer”) when creating a team.

Defining the Team class

```
class Team(object):
    pass
```

Add the __init__ method with the country instance variable

```
class Team(object):
    def __init__(self, country):
        self.__country = country
```

Add the trainer

```
class Team(object):
    def __init__(self, country):
        self.__country = country
        self.__trainer = Player("Trainer")
```

And the players

```
class Team(object):
    def __init__(self, country):
        self.__country = country
        self.__trainer = Player("Trainer")
        self.__players = []
        for i in range(1, 12):
            self.__players.append(Player("Player_" + str(i)))
```

Implement the __str__ method

```
class Team(object):
    def __init__(self, country):
        self.__country = country
        self.__trainer = Player("Trainer")
        self.__players = []
        for i in range(1, 12):
            self.__players.append(Player("Player_" + str(i)))

    def __str__(self):
        return self.__country
```

Exercise

(c) implement the Game class, which consists of two teams (home and away). Teams are instance variables. The game has a play method that will return the team that wins (it's the team with the most skilled players and trainer – the trainer counts double). If the skills are the same, the home team wins.

Defining the Game class

```
class Game(object):
    pass
```

Implement the __init__ method

```
class Game(object):
    def __init__(self, team_1, team_2):
        self.__team_1 = team_1
        self.__team_2 = team_2
```

Implement the play method

```
class Game(object):
    def __init__(self, team_1, team_2):
        self.__team_1 = team_1
        self.__team_2 = team_2

    def play(self):
        pass
```

We need the total skills for a team, we have to go back to the Team's class

```
class Team(object):
    def __init__(self, country):
        self.__country = country
        self.__trainer = Player("Trainer")
        self.__players = []
        for i in range(1, 12):
            self.__players.append(Player("Player_" + str(i)))

    def __str__(self):
        return self.__country
```

Define a get_total_skills method

```
class Team(object):
    def __init__(self, country):
        self.__country = country
        self.__trainer = Player("Trainer")
        self.__players = []
        for i in range(1, 12):
            self.__players.append(Player("Player_" + str(i)))

    def __str__(self):
        return self.__country

    def get_total_skills(self):
        pass
```

Implement the get_total_skills method

```
class Team(object):
    ...

    def get_total_skills(self):
        total_skills = 0
        for player in self.__players:
            total_skills += player.get_skill()
        return total_skills
```

Add the trainer skills

```
class Team(object):
    ...

    def get_total_skills(self):
        total_skills = 0
        for player in self.__players:
            total_skills += player.get_skill()
        total_skills += 2 * self.__trainer.get_skill()
        return total_skills
```

Now we can finish implementing the play method

```
class Game(object):
    def __init__(self, team_1, team_2):
        self.__team_1 = team_1
        self.__team_2 = team_2

    def play(self):
        pass
```

We compute the skills value for each team

```
class Game(object):
    def __init__(self, team_1, team_2):
        self.__team_1 = team_1
        self.__team_2 = team_2

    def play(self):
        skills_1 = self.__team_1.get_total_skills()
        skills_2 = self.__team_2.get_total_skills()
```

And compare them

```
class Game(object):
    def __init__(self, team_1, team_2):
        self.__team_1 = team_1
        self.__team_2 = team_2

    def play(self):
        skills_1 = self.__team_1.get_total_skills()
        skills_2 = self.__team_2.get_total_skills()
        if skills_1 >= skills_2:
            return self.__team_1
        return self.__team_2
```

Exercise

(d) implement the soccer championship simulator with the following 8 teams. In every round, the winner advances. You should print in each round what teams are currently playing and who won by how much.

Teams: ['BR', 'AR', 'IT', 'DE', 'NL', ’EN', 'AT', 'CH']

(shuffle the list of teams so that it’s not the same teams playing all the time)

Defining the SoccerGameChampionship class

```
class SoccerGameChampionship(object):
    pass
```

Implement the __init__ method

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams
```

Define the play method

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        pass
```

Make a copy of the __teams instance variable

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
```

Implement playing one round

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
```

First create the groups

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = self._create_groups(teams)
```

Implement the _create_groups method

```
class SoccerGameChampionship(object):
    def _create_groups(self, teams):
        pass
```

We do not need to reference to self, we could make it a staticmethod

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        pass
```

We need to create and return the groups list

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []

        return groups
```

We need to create len(teams)/2 groups

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            pass
        return groups
```

Randomly select 2 teams at each iteration

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            team_1 = teams[random.randint(0, len(teams) - 1)]
            team_2 = teams[random.randint(0, len(teams) - 1)]
        return groups
```

Remove team_1 before selecting the other (Why?)

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            team_1 = teams[random.randint(0, len(teams) - 1)]
            team_2 = teams[random.randint(0, len(teams) - 1)]
        return groups
```

Otherwise team_1 might play against itself

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            team_1 = teams[random.randint(0, len(teams) - 1)]
            teams.remove(team_1)
            team_2 = teams[random.randint(0, len(teams) - 1)]
        return groups
```

Remove team_2

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            team_1 = teams[random.randint(0, len(teams) - 1)]
            teams.remove(team_1)
            team_2 = teams[random.randint(0, len(teams) - 1)]
            teams.remove(team_2)
        return groups
```

Add both of them to groups as a tuple

```
class SoccerGameChampionship(object):
    @staticmethod
    def _create_groups(teams):
        groups = []
        for i in range(0, int(len(teams) / 2)):
            team_1 = teams[random.randint(0, len(teams) - 1)]
            teams.remove(team_1)
            team_2 = teams[random.randint(0, len(teams) - 1)]
            teams.remove(team_2)
            groups.append((team_1, team_2))
        return groups
```

Go back to the play method

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = self._create_groups(teams)
```

We can modify how we call _create_groups

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = SoccerGameChampionship._create_groups(teams)
```

Each group gets a chance to play

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 -----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        for group in groups:
            print("Group: %s, %s" % group)
```

We create a game instance for each group

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 -----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        for group in groups:
            print("Group: %s, %s" % group)
            game = Game(group[0], group[1])
```

Call the play method

```
class SoccerGameChampionship(object):
    def __init__(self, teams):
        self.__teams = teams

    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        for group in groups:
            print("Group: %s, %s" % group)
            game = Game(group[0], group[1])
            winner = game.play()
```

Print the winner

```
class SoccerGameChampionship(object):

    ...
    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        for group in groups:
            print("Group: %s, %s" % group)
            game = Game(group[0], group[1])
            winner = game.play()
            print("Winner: %s" % winner)
```

Store the winners for a new round

```
class SoccerGameChampionship(object):

    ...
    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        teams = []
        for group in groups:
            print("Group: %s, %s" % group)
            game = Game(group[0], group[1])
            winner = game.play()
            print("Winner: %s" % winner)
            teams.append(winner)
```

We need to repeat for multiple rounds, but how many?

```
class SoccerGameChampionship(object):

    ...
    def play(self):
        teams = self.__teams[:]
        print("----------------- Round 1 ----------------")
        groups = SoccerGameChampionship._create_groups(teams)
        teams = []
        for group in groups:
            print("Group: %s, %s" % group)
            game = Game(group[0], group[1])
            winner = game.play()
            print("Winner: %s" % winner)
            teams.append(winner)
```

At each round the number of teams is halved => $\log_2(nr\_of\_teams)$

```
class SoccerGameChampionship(object):
    def play(self):
        teams = self.__teams[:]
        for i in range(int(math.log(len(teams), 2))):
            print("---------- Round %d ----------" % (i+1))
            groups = SoccerGameChampionship._create_groups(teams)
            teams = []
            for group in groups:
                print("Group: %s, %s" % group)
                game = Game(group[0], group[1])
                winner = game.play()
                print("Winner: %s" % winner)
                teams.append(winner)
```

https://docs.python.org/3/library/math.html#math.log

Let's write the code that actually runs the simulator – First we need the list of countries

```
countries = ['BR', 'AR', 'IT', 'DE', 'NL', 'NG', 'AT', 'CH']
```

Create the list of teams

```
countries = ['BR', 'AR', 'IT', 'DE', 'NL', 'NG', 'AT', 'CH']

teams = []
for country in countries:
    teams.append(Team(country))
```

Create a SoccerGameChampionship instance

```
countries = ['BR', 'AR', 'IT', 'DE', 'NL', 'NG', 'AT', 'CH']

teams = []
for country in countries:
    teams.append(Team(country))

championship = SoccerGameChampionship(teams)
```

And call the play method

```
countries = ['BR', 'AR', 'IT', 'DE', 'NL', 'NG', 'AT', 'CH']

teams = []
for country in countries:
    teams.append(Team(country))

championship = SoccerGameChampionship(teams)
championship.play()
```

Example Run

```
---------- Round 1 ----------
Group: NG, BR
Winner: BR
Group: CH, NL
Winner: CH
Group: AT, IT
Winner: IT
Group: AR, DE
Winner: DE
---------- Round 2 ----------
Group: BR, CH
Winner: BR
Group: IT, DE
Winner: DE
---------- Round 3 ----------
Group: BR, DE
Winner: BR
```