Programming 2: OOP (Lecture 1)
Classes, Objects, Encapsulation & Abstraction

Comp 201

Forman Christian University

Course Information

Instructor

- Fakhir Shaheen
  - fakhirshaheen@fccollege.edu.pk
  - linkedin.com/in/fakhirshaheen

- Office:
  - S-426 (E)
  - Hours: TR 9:15 am - 11:00 am

Grading Breakdown

Component	**Percentage**
Assignments/Project	15%
Labs	10%
Quizzes	15%
Mid	25%
Final	35%
Total	**100%**

Today's Agenda

1. What is OOP?
2. Classes and Objects
3. Encapsulation
4. Data Abstraction
5. Composition

What is OOP?

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**
- **OOP** but using pure C:
  - Linux Kernel
  - GTK+
  - Doom

Object Oriented Programming

- **OOP** is a method of software design and programming
  - revolve around the concept of **objects**
- **OOP** but using pure C:
  - Linux Kernel
  - GTK+
  - Doom
- Set of programming practices *(language, syntax independent)*

The Object Metaphor

Objects:

1. are like **black boxes**

The Object Metaphor

Objects:

1. are like **black boxes**
2. have **state** (`data`) and **behavior** (`methods`)
3. have unique **identities**

The Object Metaphor

Objects:

1. are like **black boxes**
2. have **state** (`data`) and **behavior** (`methods`)
3. have unique **identities**
4. can **interact** with each other by sending **messages**

OOP vs Procedural Programming

Procedural

- Focus on **functions**
- Data and functions are separate
- Top-down approach
- Hard to manage large programs
- Example: C, early Python scripts

OOP vs Procedural Programming

Procedural

- Focus on **functions**
- Data and functions are separate
- Top-down approach
- Hard to manage large programs
- Example: C, early Python scripts

Object-Oriented

- Focus on **objects**
- Data and functions bundled together
- Bottom-up approach
- Easier to manage complexity
- Example: Python classes, Java

Object-Oriented Programming (OOP)

What is OOP?

A programming paradigm focused on **objects** - combining data and behavior to model real-world systems.

Object-Oriented Programming (OOP)

What is OOP?

A programming paradigm focused on **objects** - combining data and behavior to model real-world systems.

- **Encapsulation** - Combine data and methods
- **Abstraction** - Hide internal complexity
- **Inheritance** - Reuse and extend classes
- **Polymorphism** - One interface, many behaviors

Design principles for clean, reusable code

Data Abstraction

Abstraction

Abstraction

- Hiding the details, showing only the necessary

Abstraction

- Hiding the details, showing only the necessary
- **Expressions**:

```
1 pi = 355/113
2 area = pi * (R**2)
3 circ = 2 * pi * R
```

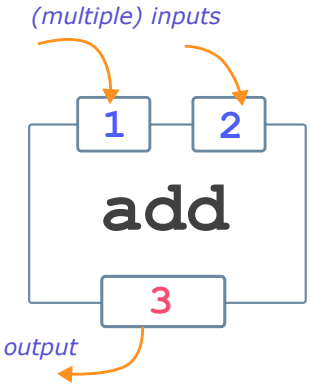
Abstraction

- Hiding the details, showing only the necessary
- **Expressions**:

```
1 pi = 355/113
2 area = pi * (R**2)
3 circ = 2 * pi * R
```

- **Functions**:

```
1 def add(x,y):
2     return x+y
3 add(1,2)
4 add(4,5)
```



Compound Data

- Many things in reality are made of compund data *(non-scalar)*
- Try to Identify components of each:
  - `Vector` →
  - `Student` →
  - `Rational` →
  - `Book` →

Compound Data

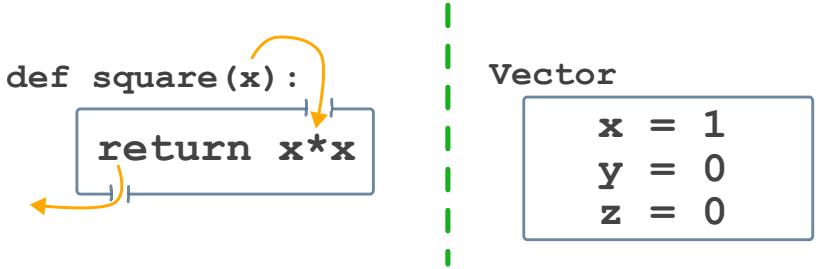
- Many things in reality are made of compund data *(non-scalar)*
- Try to Identify components of each:
  - `Vector` → `x,y,z`
  - `Student` → `name, rollno, gpa`
  - `Rational` → `num, denom`
  - `Book` → `title, author, price`

Data Abstraction

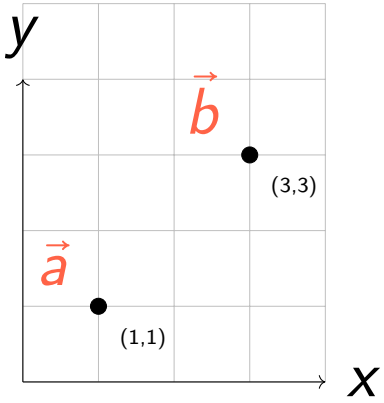
- We want to do the same with **data** that we did with **code**
- Put individual components *(data)* in a **black box**

Data Abstraction

- We want to do the same with **data** that we did with **code**
- Put individual components *(data)* in a **black box**



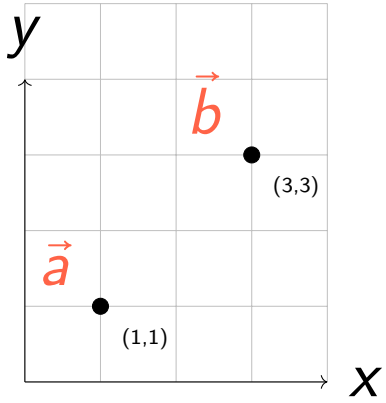
Example:



From high school math:

- $\vec{c} = \vec{a} + \vec{b}$
- $\vec{d} = \vec{a} - \vec{b}$

Example:



From high school math:

- $\vec{c} = \vec{a} + \vec{b}$
- $\vec{d} = \vec{a} - \vec{b}$

We're treating $\vec{a}$ and $\vec{b}$ as black boxes

Not concerned with their internal details

Big Idea

Data Abstraction:

Seperating how data is **represented** from how it is **manipulated**

Example: Rational Numbers

- Have the form: $\frac{numerator}{denominator}$

Example: Rational Numbers

- Have the form: $\frac{numerator}{denominator}$
- A rational 1/3 **can't be represented exactly** on a computer:

```
>>> 1/3
0.3333333333333333
>>> 1/3 == 0.33333333333333300000
True
```

Rational Numbers

- But can represent exactly as compound data *(non-scalar)*
- Imagine the following functions *(again, black boxes)*:

```
rational(n,d) # constructor, returns rational
numer(x) # accessor function
denom(x) # accessor function
```

Rational Numbers

So far we have three functions:

```
rational(n,d)      numer(x)      denom(x)
```

Rational Numbers

So far we have three functions:

```
rational(n,d)      numer(x)      denom(x)
```

- No idea how these functions are implemented
- No idea how a rational number stores numerator and denominator

Rational Numbers

So far we have three functions:

```
rational(n,d)      numer(x)      denom(x)
```

- No idea how these functions are implemented
- No idea how a rational number stores numerator and denominator
- But we can still use these as black boxes to build more complex functions

Rational Numbers

Adding two rational numbers:

$$\frac{1}{2} + \frac{2}{3} = \frac{3 \times 1 + 2 \times 2}{2 \times 3} = \frac{7}{6}$$

Rational Numbers

Adding two rational numbers:

$$\frac{1}{2} + \frac{2}{3} = \frac{3 \times 1 + 2 \times 2}{2 \times 3} = \frac{7}{6}$$

Complete the following function:

```
def add_rationals(x, y):
    ''' Adds two rational numbers x,y
    Returns: A rational number '''
```

Rational Numbers

Solution:

```
def add_rationals(x, y):
    ''' Adds two rational numbers x,y
    Returns: A rational number '''

    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx*dy)
```

Rational Numbers: You Try!

```
def mul_rationals(x, y):
    ''' Multiplies two rational numbers x,y
    Returns: A rational number '''

def print_rational(x):
    ''' Prints a rational number as n/d '''

def rationals_are_equal(x, y):
    ''' Checks if two rational numbers are
      equal '''
```

Rational Numbers

```
1  def add_rationals(x, y):
2      ...
3
4  def mul_rationals(x, y):
5      return rational(numer(x)*numer(y), denom(x)*denom(y))
6
7  def print_rational(x):
8      print(numer(x), '/', denom(x))
9
10 def rationals_are_equal(x, y):
11     return numer(x) * denom(y) == numer(y) * denom(x)
```

Rational Numbers

Data Abstraction: separation of

1. how data is manipulated ✓
   - `rational`, `numer`, `denom`
   - `add_rationals`, `mul_rationals`, `print_rational`, `rationals_are_equal`

Rational Numbers

Data Abstraction: separation of

1. how data is manipulated ✓
   - `rational`, `numer`, `denom`
   - `add_rationals`, `mul_rationals`, `print_rational`, `rationals_are_equal`
2. how data is represented ←

Representing Rational Numbers

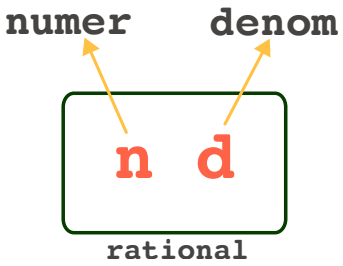
You can use a tuple or a list to store n,d:

```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```

Representing Rational Numbers

You can use a tuple or a list to store n,d:

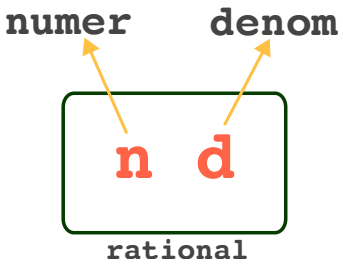
```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```



Representing Rational Numbers

You can use a tuple or a list to store n,d:

```
def rational(n, d):
    return [n, d]
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```



Only way to access the data is through accessors

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6

>>> print_rational(add_rationals(third, third))
6 / 9
```

Rational Numbers

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2

>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6

>>> print_rational(add_rationals(third, third))
6 / 9
```

Some rationals are not in **simplest form**

Rational Numbers

Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

Rational Numbers

Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

```
>>> print_rational(add_rationals(third, third))
2 / 3
```

Rational Numbers

Converting to simplest form:

```
from fractions import gcd
def rational(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

```
>>> print_rational(add_rationals(third, third))
2 / 3
```

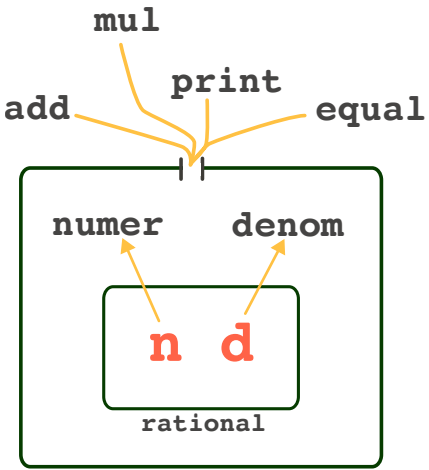
ZERO change required elsewhere

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using only...
Use rational numbers to perform computation	whole data values	`add_rational, mul_rational, rationals_are_equal, print_rational`
Create rationals or implement rational operations	numerators and denominators	`rational, numer, denom`
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Each function in last column enforce an **abstraction barrier**

Abstraction Barriers



Big Idea

Abstraction Barrier Violation happens when a higher-level function is **bypassed** to use lower-level implementation details.

Abstraction Barrier Violation:

```
def square_rational(x):
    return mul_rational(x, x)
```

Abstraction Barrier Violation:

```
def square_rational(x):
    return mul_rational(x, x)

def square_rational_violating_once(x):
    return rational(numer(x)*numer(x), denom(x)*denom(x))
```

Abstraction Barrier Violation:

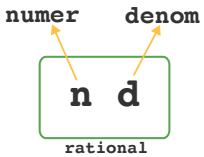
```
def square_rational(x):
    return mul_rational(x, x)

def square_rational_violating_once(x):
    return rational(numer(x)*numer(x), denom(x)*denom(x))

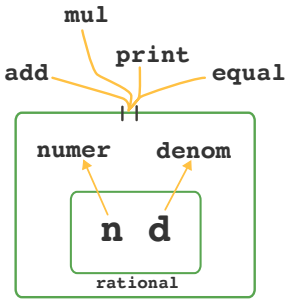
def square_rational_violating_twice(x):
    return [x[0] * x[0], x[1] * x[1]]
```

Abstraction Barrier Violation

- **Only** accessors and constructors should access **n,d** directly

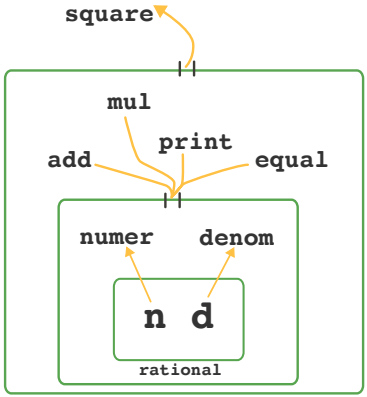


Abstraction Barrier Violation



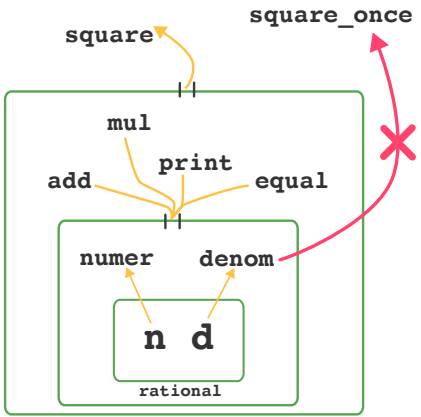
- **Only** accessors and constructors should access **n,d** directly
- These operations **should not** access **n,d** directly

Abstraction Barrier Violation



- **Only** accessors and constructors should access **n,d** directly
- These operations **should not** access **n,d** directly
- **square** should only access rational numbers through `add`, `mul`, `equal`, `print`

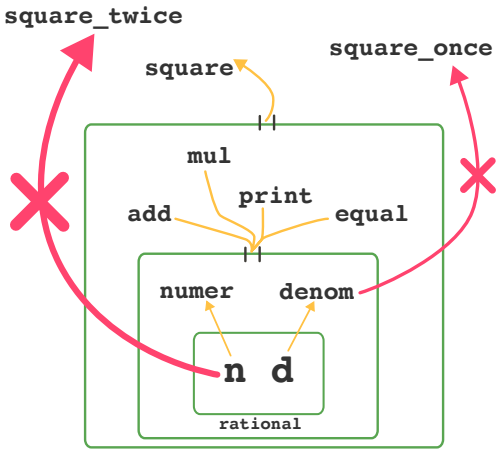
Abstraction Barrier Violation



- **Abstraction barrier** violated **Once** (Higher-level functions bypassed)

```
def sq_violate_once(x):
    return rational(
        numer(x)*numer(x),
        denom(x)*denom(x) )
```

Abstraction Barrier Violation



- **Abstraction barrier** violated **Once**
  (Higher-level functions bypassed)

```
def sq_violate_once(x):
    return rational(
        numer(x)*numer(x),
        denom(x)*denom(x) )
```

- **Abstraction barrier** violated **Twice**

```
def sq_violate_twice(x):
    return [x[0]*x[0],
            x[1]*x[1]]
```

You Try!

Right now our rational numbers are internally represented as lists.

1. Change the internal representation to a dictionary.
2. What portion of the entire program really needed to change?

The Power of Abstraction

Changing the representation of rational numbers **requires no changes** to any other parts of code:

```
def rational(n, d):
    return {"numerator":n, "denominator":d}
def numer(x):
    return x["numerator"]
def denom(x):
    return x["denominator"]
```

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]`

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` **# use a Constructor!** `make_point(x,y)`

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` **# use a Constructor!** `make_point(x,y)`

2. ```
   p = make_point(3,4)
   y = p[1]
   ```
   ```

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

1. `p = [1,2]` **# use a Constructor!** `make_point(x,y)`

2. `p = make_point(3,4)`
   `y = p[1]` **#** 1^{st} **level violated. Use accessors**
   `y = get_y(p)`

You Try!

Suppose we need to use **Point** in our game. Spot **Abstraction Barrier Violations**, **How many Barriers Violated**. Suggest how to fix them:

```
def distance(p1, p2):
    return ((p1[0]-p2[0])**2 +
            (p1[1]-p2[1])**2)**0.5
```

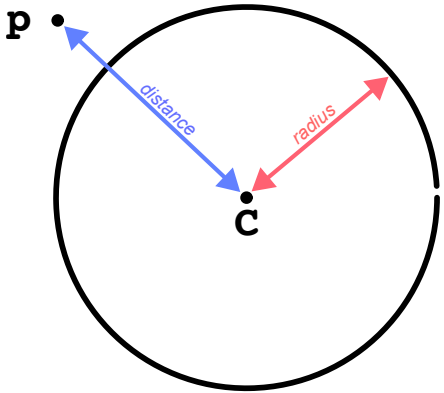
You Try! *(solution)*

1. **One level violated**
   - Bypassing the higher-level functions *(accessors)*
   - Directly accessing internal data *(very bad)*

```
def distance(p1, p2):
    return ((get_x(p1) - get_x(p2))**2 +
            (get_y(p1) - get_y(p2))**2)**0.5
```

You Try!

Check if a point is within a circle:



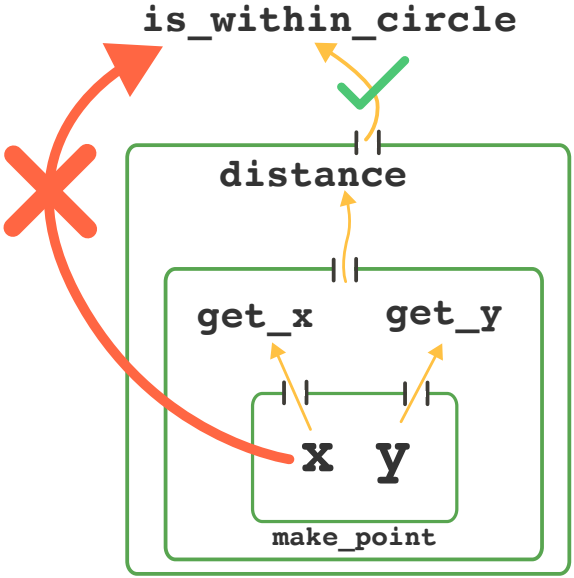
You Try!

Check if a point is within a circle. Identify **Abstraction Barrier Violations**:

```
def is_within_circle(point,center,radius):
    dist = ((center[0] - point[0])**2 +
            (center[1] - point[1])**2)**0.5

    return dist <= radius
```

You Try!

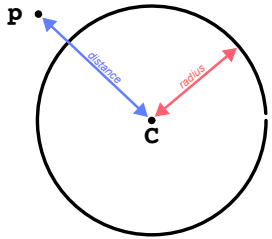


You Try!

Solution:

```
def is_within_circle(point,center,radius):
    return distance(point,center) <= radius
```

Circle class



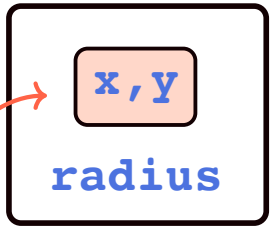

```
1 class Point:
2     ...
3 class Circle:
4     def __init__(self,x,y,radius):
5         self.x = x
6         self.y = y
7         self.radius = radius
```

Abstraction violation

could've used 'Point' class

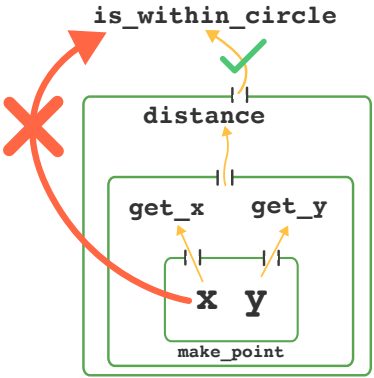
Circle class

```
1 class Point:
2     ...
3 class Circle:
4     def __init__(self,center,radius):
5         self.center = center
6         self.radius = radius
```

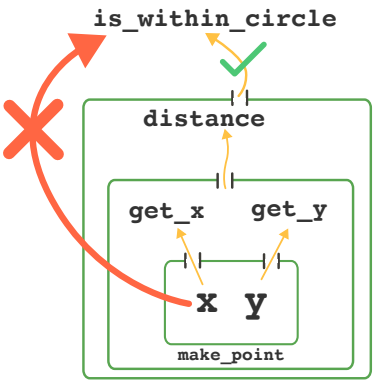


Encapsulation

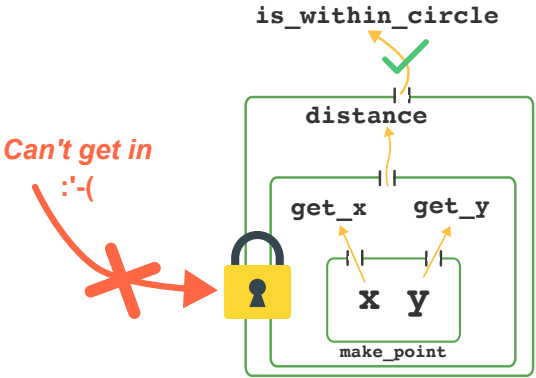
Abstraction



Abstraction



Encapsulation



Python is NOT Great at Encapsulation

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6
7
8
```

Python is NOT Great at Encapsulation

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6 print(p.name)    # access data from outside
7
8
```

Python is NOT Great at Encapsulation

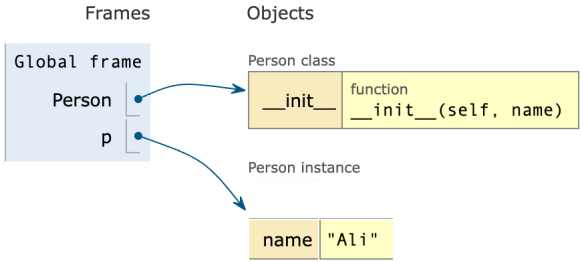
```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6 print(p.name)    # access data from outside
7 p.name = 'Ali'   # change data from outside
8
```

Python is NOT Great at Encapsulation

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6 print(p.name)    # access data from outside
7 p.name = 'Ali'  # change data from outside
8 p.age = 20      # add new data from outside
```

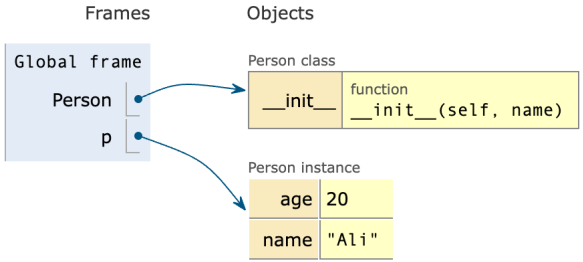
Python is NOT Great at Encapsulation

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6 print(p.name)
7 p.name = 'Ali'
8
```



Python is NOT Great at Encapsulation

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5 p = Person('Zain')
6 print(p.name)
7 p.name = 'Ali'
8 p.age = 20
```



Classes and Objects

Class: A Blueprint

- A **class** is a blueprint/template for creating objects

Class: A Blueprint

- A **class** is a blueprint/template for creating objects
- An **object** is an instance of a class

Class: A Blueprint

- A **class** is a blueprint/template for creating objects
- An **object** is an instance of a class
- Analogy:
  - **Class** = Cookie Cutter
  - **Object** = Cookie

Class: A Blueprint

- A **class** is a blueprint/template for creating objects
- An **object** is an instance of a class
- Analogy:
  - **Class** = Cookie Cutter
  - **Object** = Cookie

One class → Many objects (each with its own data)

```
1
2
3  class Point:
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Creating Objects using The 'Class' Syntax

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

keyword

class name

```
class Point:
```

Creating Objects using The 'Class' Syntax

Creating Objects using The 'Class' Syntax

```
1
2
3  class Point:
4
5
6
7
8
9
10
11
12
13
14
15 p1 = Point(1, 2)
16
17
```

```
1
2
3  class Point:
4
5
6
7
8
9
10
11
12
13
14
15 p1 = Point(1, 2)
16
17
```

Creating Objects using The 'Class' Syntax

<self>

```
1
2
3  class Point:
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7
8
9
10
11
12
13
14
15 p1 = Point(1, 2)
16
17
```

constructor

Creating Objects using The 'Class' Syntax

<self>

Creating Objects using The 'Class' Syntax

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8
9
10
11
12
13
14
15 p1 = Point(1, 2)
16
17
```

p1

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16
17
```

public interface

Creating Objects using The 'Class' Syntax

<self> p1

x → 1

y → 2

Creating Objects using The 'Class' Syntax

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16
17
```

The first parameter is always **'self'**

<self> p1

x → 1

y → 2

Creating Objects using The 'Class' Syntax

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16
17
```

'self' is passed automatically by Python

<self> p1

x → 1

y → 2

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16 p1.print()
17
```

Accessing attributes using the **dot notation**

Creating Objects using The 'Class' Syntax

<self> p1

x → 1

y → 2

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16 p1.print()
17
```

Creating Objects using The 'Class' Syntax

<self> p1

x → 1

y → 2

<obj>.<attribute>

note: *no parameters passes*

```
1
2
3  class Point:
4    def __init__(self, x, y):
5      self.x = x
6      self.y = y
7
8    def get_x(self):
9      return self.x
10   def get_y(self):
11     return self.y
12   def print(self):
13     print(f'({self.x}, {self.y})')
14
15 p1 = Point(1, 2)
16 p1.print()
17
```

p1

Creating Objects using The 'Class' Syntax

<self> p1

x → 1

y → 2

Big Idea

The `__init__()` method is invoked **automatically** when an object is created.

Big Idea

The `self` parameter is the object which invokes the method using the dot operator.

Creating Multiple Objects

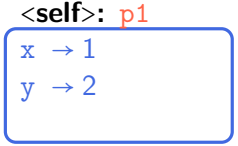
```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9
10
11
12
13
14
```

Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10
11
12
13
14
```

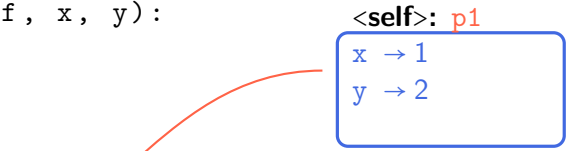
Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10
11
12
13
14
```



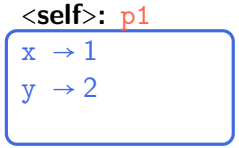
Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10
11
12
13
14
```



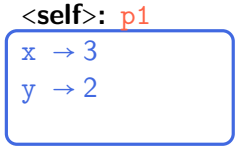
Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11
12
13
14
```



Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12
13
14
```



Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13
14
```

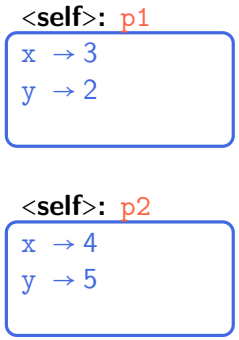
<self>: p1

x → 3

y → 2

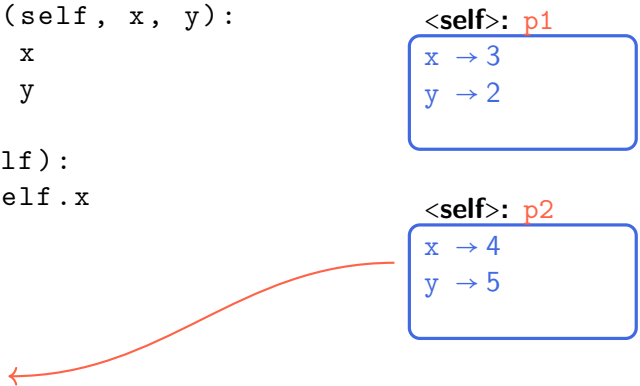
Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13
14
```



Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13
14
```



Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14
```

<self>: p1

x → 3
y → 2

<self>: p2

x → 4
y → 5

Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14
```

<self>: p1

x → 3
y → 2

<self>: p2

x → 4
y → 5

Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14
```

p2

<self>: p1

x → 3
y → 2

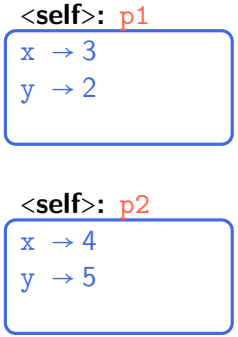
<self>: p2

x → 4
y → 5

Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14
```

p2



Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14 print(p1.get_x())
```

<self>: p1

x → 3
y → 2

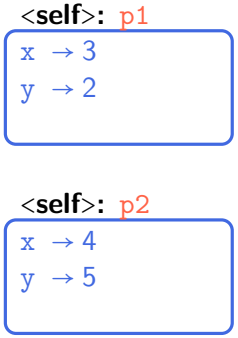
<self>: p2

x → 4
y → 5

Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14 print(p1.get_x())
```

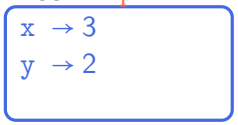
p1

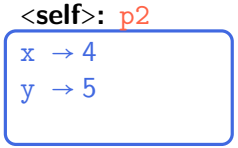


Creating Multiple Objects

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def get_x(self):
7          return self.x
8
9  p1 = Point(1, 2)
10 print(p1.x)
11 p1.x = 3
12 p2 = Point(4, 5)
13 print(p2.get_x())
14 print(p1.get_x())
```

Observe:
Methods have **one** copy. They are shared between different objects. Hence the '**self**' parameter





Big Idea

The methods are created only **once**.
Different objects **share** the same methods by passing themselves as '**self**'.

You Try!

- Create a class `Rectangle` with attributes `length` and `width`.
- Add methods to calculate the area and perimeter of the rectangle.
- Create two rectangle objects and test the methods.

Questions?