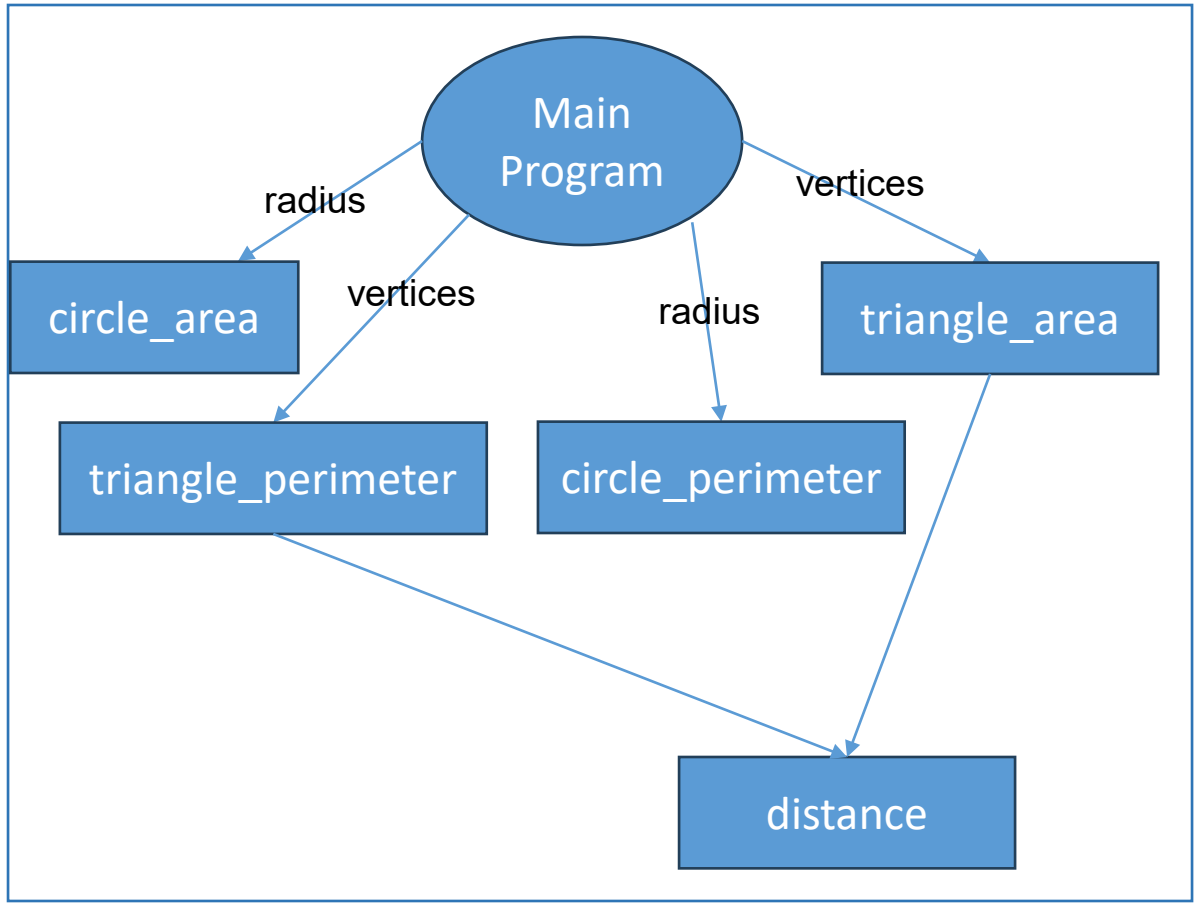
Fundamentals of Programming & Computer Science
CS 15-112

OOP – Part 1

Hend Gedawy

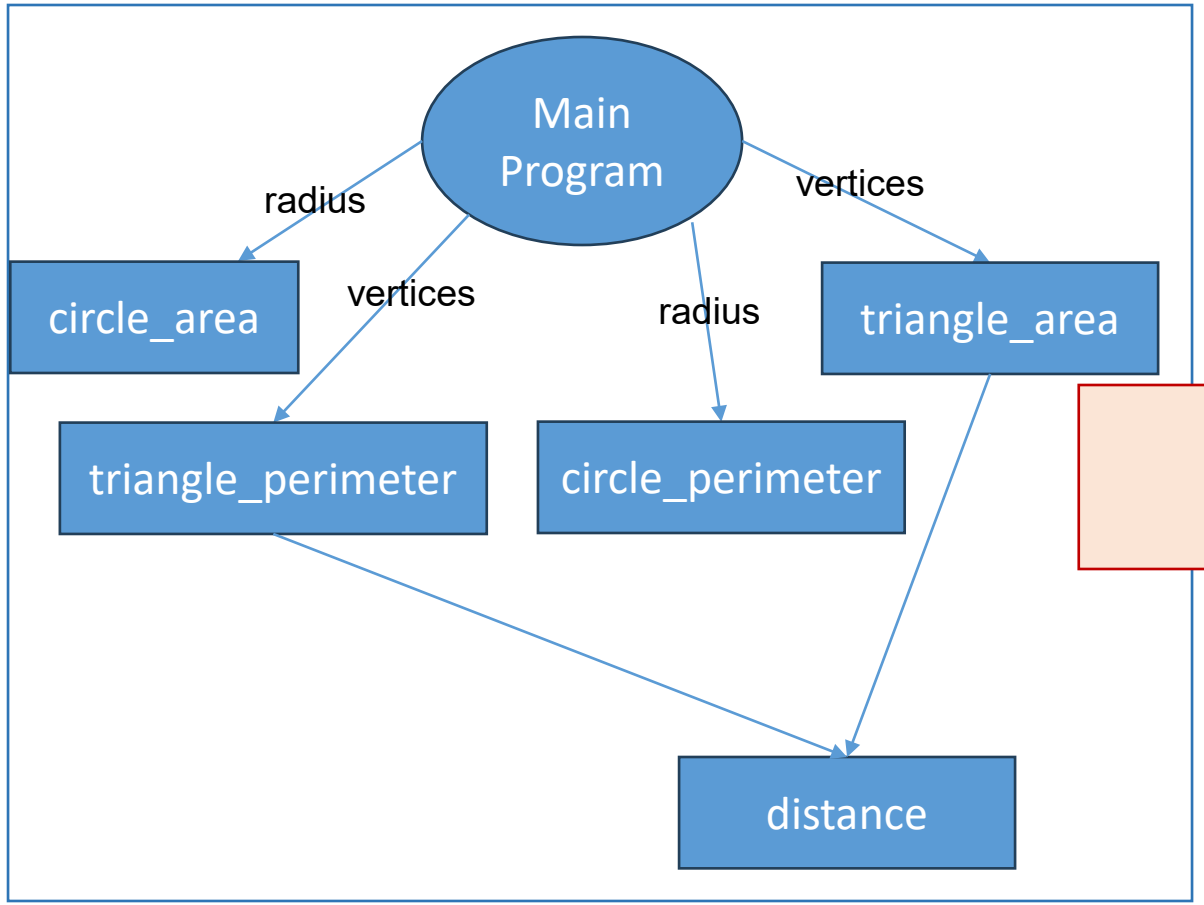
جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

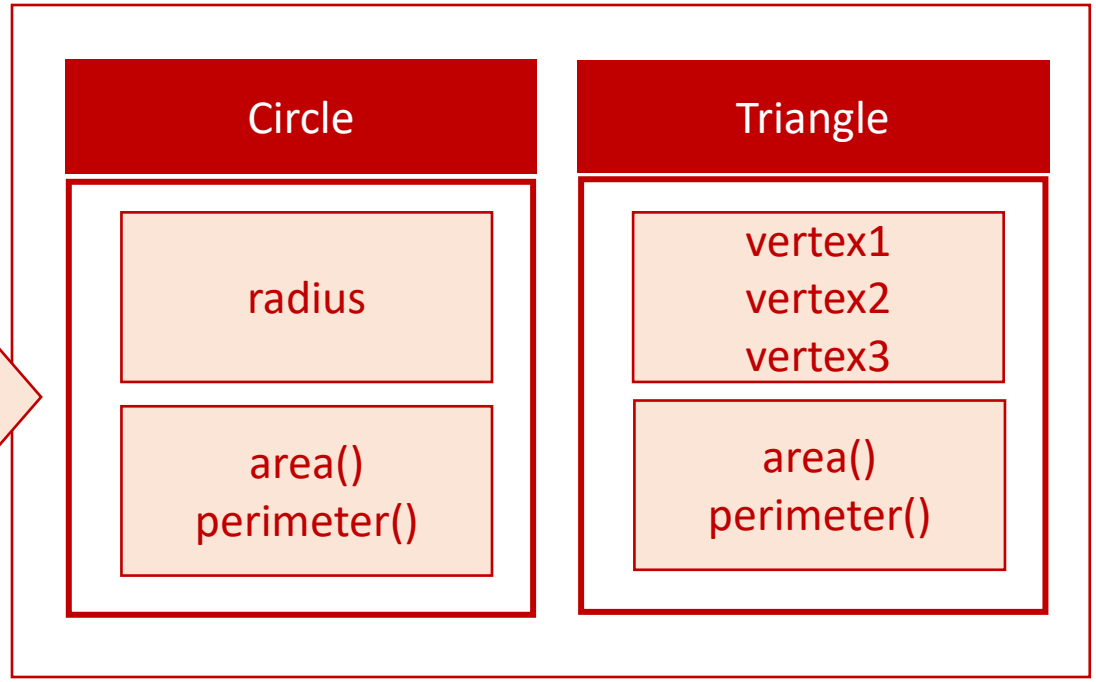
What we have been doing so far




```python
# Function to calculate area of circle
def calculate_circle_area(radius):
    return 3.14 * radius * radius

# Function to calculate perimeter of circle
def calculate_circle_perimeter(radius):
    return 2 * 3.14 * radius

# Function to calculate the distance between two points in a 2D plane.
def distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Function to calculate area of triangle
def calculate_triangle_area(side1, side2, side3):
    side1 = distance(vertices[0], vertices[1])
    side2 = distance(vertices[1], vertices[2])
    side3 = distance(vertices[2], vertices[0])
    s = (side1 + side2 + side3) / 2  # Semi-perimeter
    area = math.sqrt(s * (s - side1) * (s - side2) * (s - side3))
    return area

# Function to calculate perimeter of tirangle
def calculate_triangle_perimeter(vertices):

    side1 = distance(vertices[0], vertices[1])
    side2 = distance(vertices[1], vertices[2])
    side3 = distance(vertices[2], vertices[0])
    perimeter = side1 + side2 + side3
    return perimeter

# Main Program
circle_perimeter = calculate_circle_perimeter(3)
circle_area = calculate_circle_area(3)

vertices = [(0, 0), (3, 4), (6, 0)]
triangle_area = calculate_triangle_area(vertices)
triangle_perimeter = calculate_triangle_perimeter(vertices)
```

An Alternative Approach

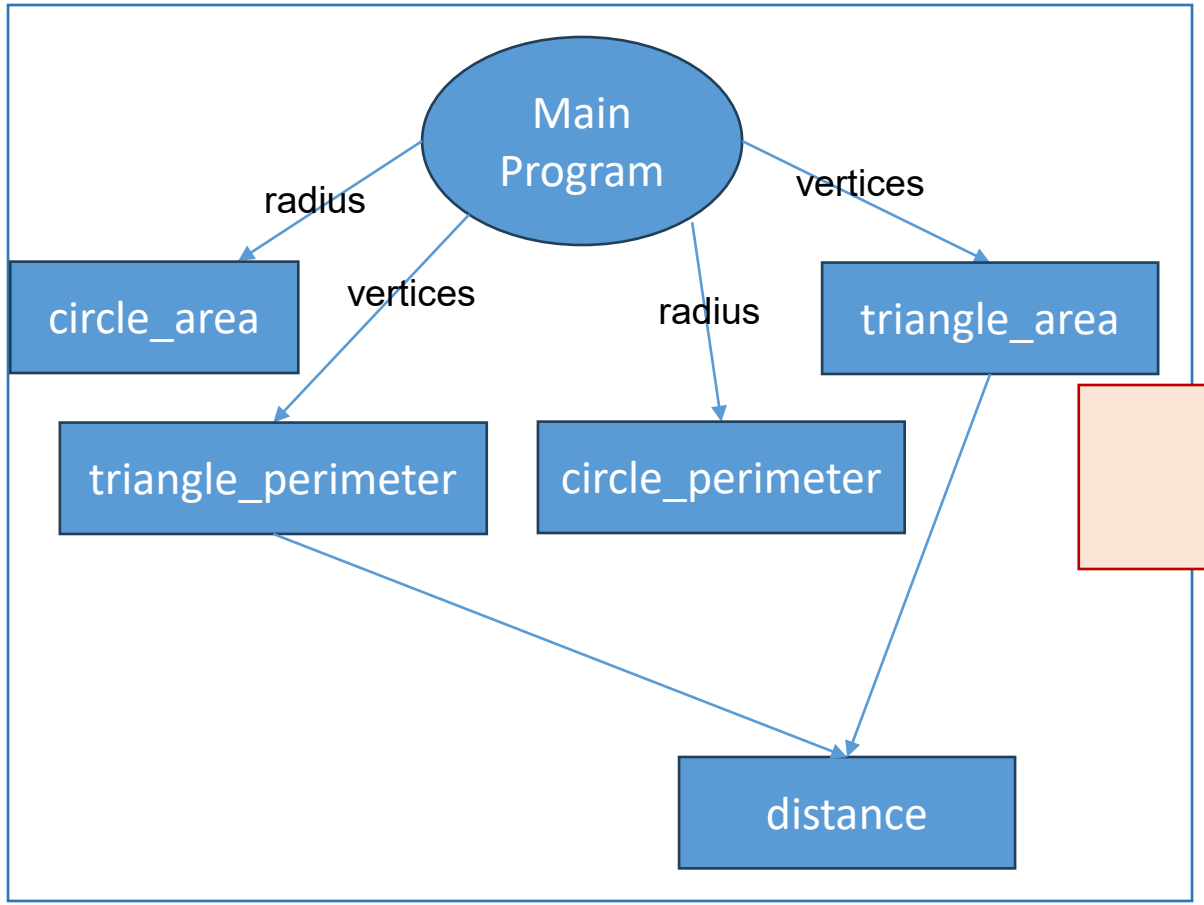




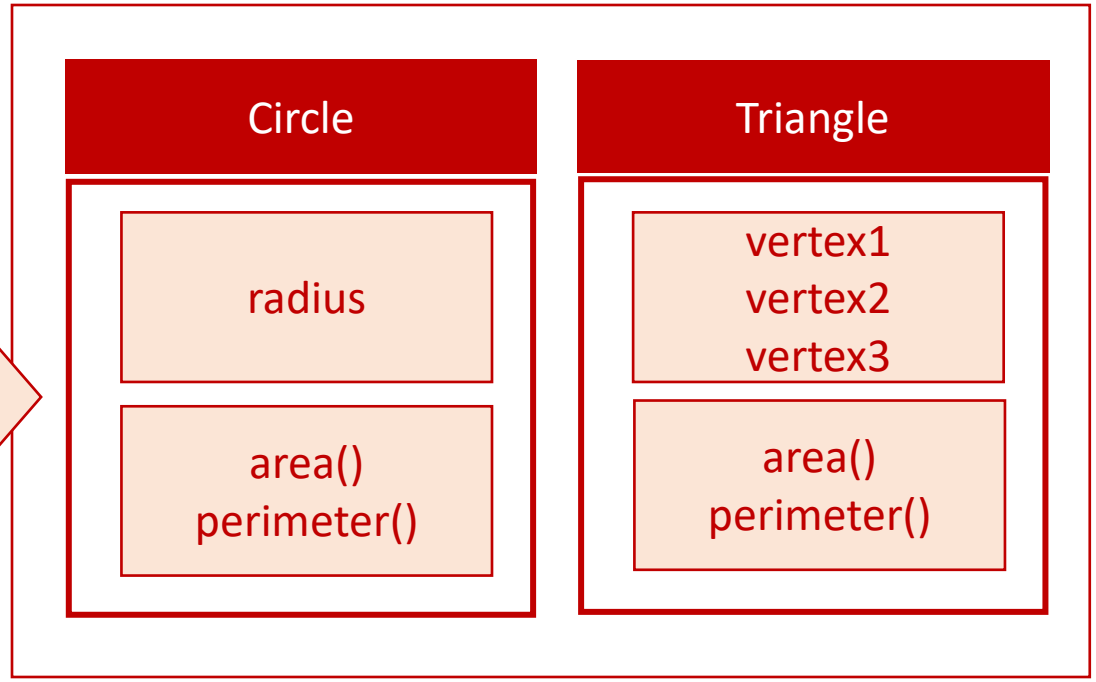
An Alternative Approach

Your program is divided into parts

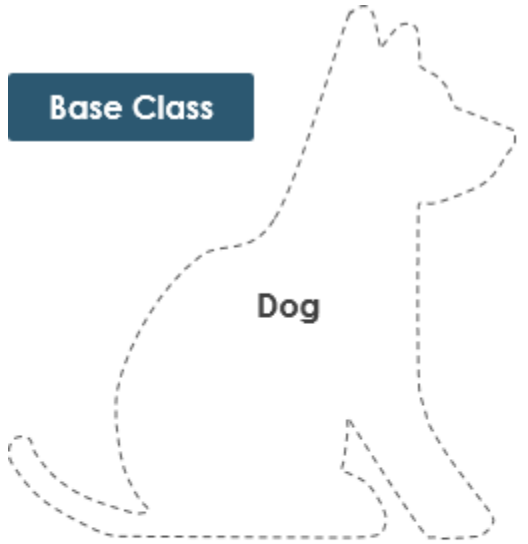
Functions



Classes



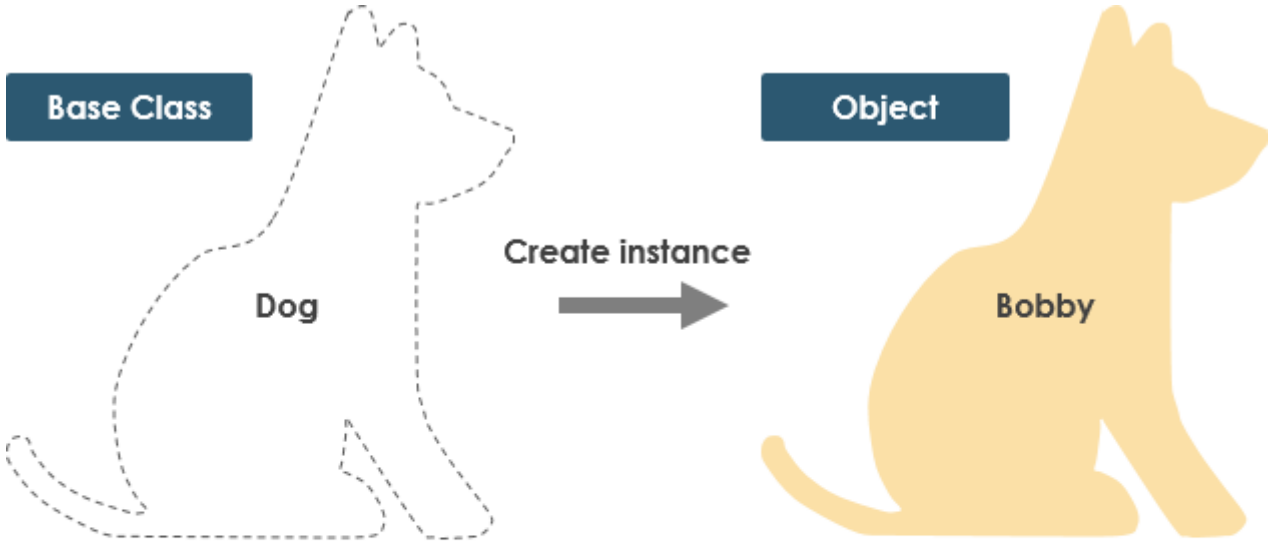
Object Oriented Programming (OOP) Approach



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Photo Credit

Classes VS Objects/Instances

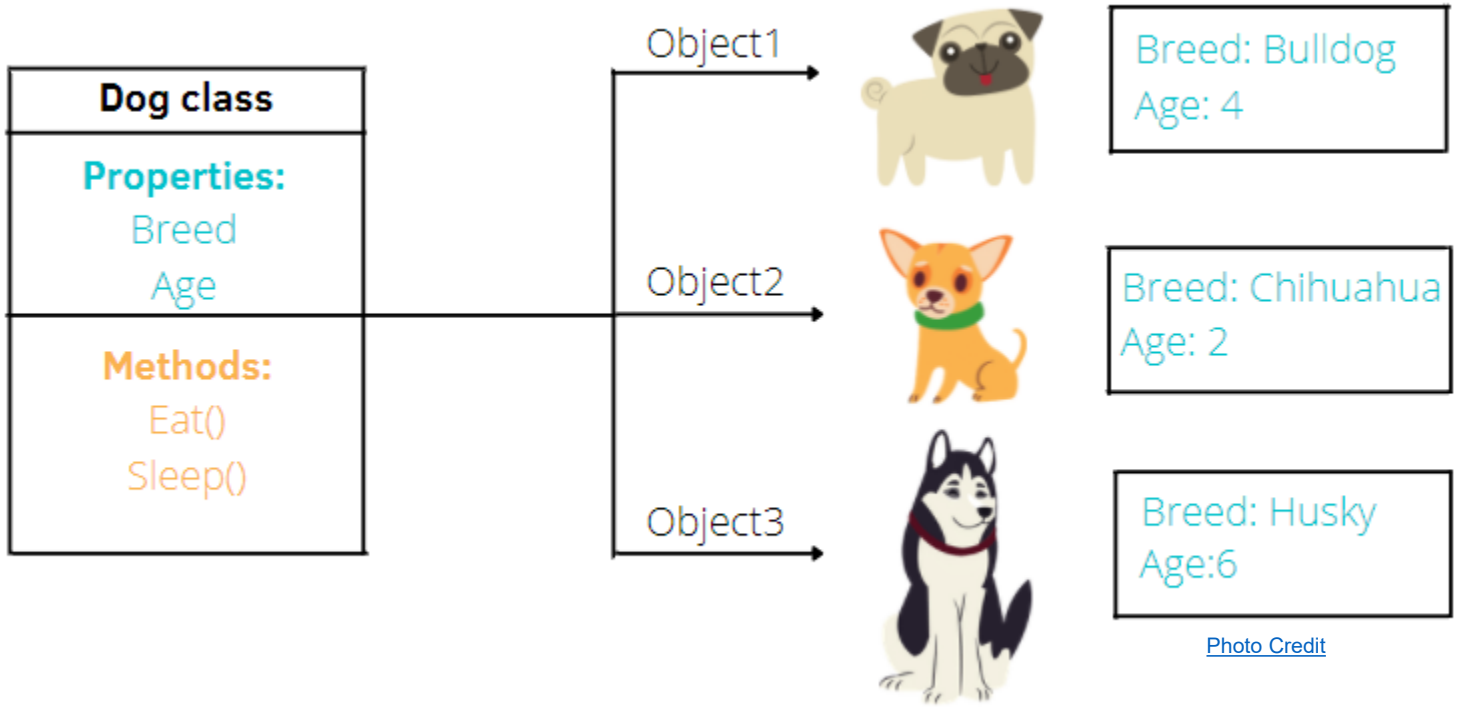


Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

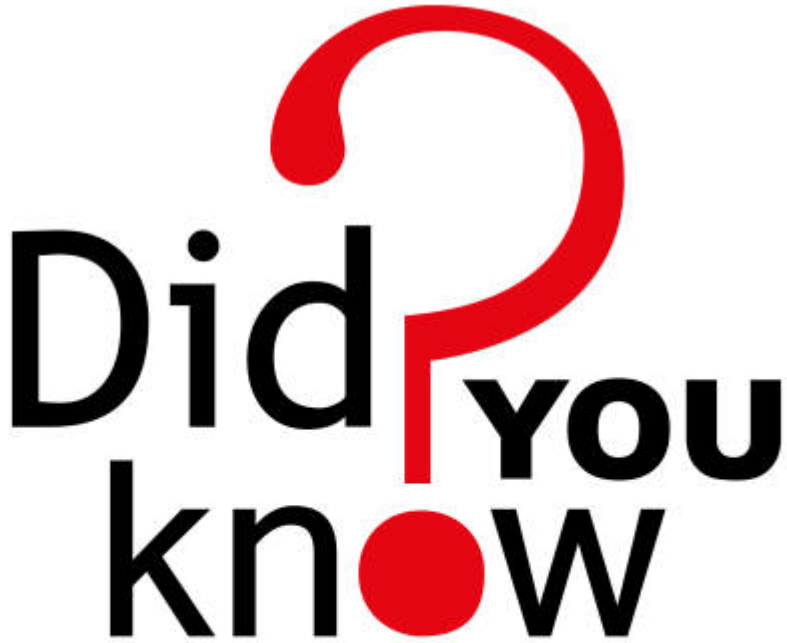
Photo Credit

Classes _ Objects/Instances



Acts as a template for a generic object.

Instances



you have already been using OOP

You have been using Built-in Objects in Python

Class	Object/Instance
str	“Hello”
int	5
list	[1,2,3,4]

You have been using Built-in Objects in Python

Class (are also Types)	Object/Instance (values of a given class or type)	
str	"Hello"	An instance of type **str**
int	5	An instance of type **int**
list	[1,2,3,4]	An instance of type **list**

```
>>> type("Hello") == str
True
>>> isinstance("Hello", str)
True
```

```
>>> type(5) == int
True
>>> isinstance(5, int)
True
```

```
>>> type([1,2,3,4])== list
True
>>> isinstance([1,2,3,4], list)
True
```

You have been calling methods on these objects

```
s = 'This could be any string!’
```

We call methods using s.method()…

```
print(s.upper()) # upper is a string method, called using the . Notation
# we say that we "call the method upper on the string s"

print(s.replace('could', 'may')) # some methods take additional
arguments
```

… rather than function(s):

```
print(len(s)) # len is a function
```

Functions Versus Methods

f(_) is a Function

len(L)

sorted(L)

max(n1, n2, n3..)

abs(n)

range(n)

sum(L)

obj.m(_) is a Method

s.upper()

l.append(x)

d.clear()

l.count(x)

s1.update(s2)

l.sort()

We use the dot notion to call a method on an object

Objects

Creating Your Own Objects

- Properties?
  - What data defines your Object?
- Methods?
  - How do you plan to use the object in your code?

Properties

Creating Empty Class & Instances

```
# Create your own class:
class Dog(object):
    # define properties and methods of a generic dog here

    # a class must have a body, even if it does nothing, so we will
    # use 'pass' for now...
    pass


# Create instances of our class:
d1 = Dog()
d2 = Dog()

# Verify the type of these instances:
print(type(d1))   # Dog (actually, class '__main__.Dog')
print(isinstance(d2, Dog)) # True
```

Properties
Setting and Getting Instances Properties

```
# Create your own class:
class Dog(object):
    # define properties and methods of a
generic dog here

    # a class must have a body, even if it does nothing, so we
    will
    # use 'pass' for now...
    pass

# Create instances of our class:
d1 = Dog()
d2 = Dog()

# Set and get properties (aka 'fields' or 'attributes') of these instances:
d1.name = 'Dot’
d1.age = 4
d2.name = 'Elf’
d2.age = 3
print(d1.name, d1.age) # Dot 4
print(d2.name, d2.age) # Elf 3
```

```
def constructor(dog, name, age):
    # pre-load the dog instance with the given name and age:
        dog.name = name
        dog.age = age
```

```
def __init__(dog, name, age):
    # pre-load the dog instance with the given name and age:
        dog.name = name
        dog.age = age
```

```
class Dog(object):
    def __init__(self, name, age):
        # pre-load the dog instance with the given name and age:
            self.name = name
            self.age = age
```

```
d1 = Dog('fred', 4) # now d1 is a Dog instance with name 'fred' and age 4
d2 = Dog('Elf', 3)
print(d1.name, d1.age) # Dot 4
print(d2.name, d2.age) # Elf 3
```

Properties

Preloading Instances w/ Properties (using **Constructors**)

Methods

```
class Dog(object):

        def __init__(self, name, age):
                self.name = name
                self.age = age


# Here is a function we will turn into a method:
def sayHi(dog):
        print(f'Hi, my name is {dog.name} and I
am {dog.age} years old!’)

d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)
sayHi(d1) # Hi, my name is Dot and I am 4 years old!
sayHi(d2) # Hi, my name is Elf and I am 3 years old!
```

Methods
1) Start with a Function

Methods
2) Turn it into Method

```
class Dog(object):

        def __init__(self, name, age):
                self.name = name
                self.age = age

        # Now it is a method (simply by indenting it inside the class!)
        def sayHi(dog):
                print(f'Hi, my name is {dog.name} and I am {dog.age} years old!’)


d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)

# Notice how we change the function calls into method calls:
d1.sayHi()  # Hi, my name is Dot and I am 4 years old!
d2.sayHi()  # Hi, my name is Elf and I am 3 years old!
```

Methods
2) Turn it into Method

```
class Dog(object):

        def __init__(self, name, age):
                self.name = name
                self.age = age

        # Now it is a method (simply by indenting it inside the class!)
        def sayHi(dog):
                print(f'Hi, my name is {dog.name} and I am {dog.age} years old!')


d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)

# Notice how we change the function calls into method calls:
d1.sayHi()  # Hi, my name is Dot and I am 4 years old!
d2.sayHi()  # Hi, my name is Elf and I am 3 years old!
```

Methods
2) Turn it into Method

```
class Dog(object):

        def __init__(self, name, age):
                self.name = name
                self.age = age

        # Use self keyword to represent the current instance
        def sayHi(self):
                print(f'Hi, my name is {dog.name} and I am {dog.age} years old!’)


d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)

d1.sayHi()  # Hi, my name is Dot and I am 4 years old!
d2.sayHi()  # Hi, my name is Elf and I am 3 years old!
```

More Methods

```
class Dog(object):

        def __init__(self, name, age):
                self.name = name
                self.age = age
                self.woofCount = 0      # We initialize the property in the constructor!

        def sayHi(self):
                print(f'Hi, my name is {dog.name} and I am {dog.age} years old!’)

        # This method takes a second parameter -- times
        def bark(self, times):
                print(f'{self.name} says: {"woof!" * times}')
                self.woofCount += times # Then we can set and get the property in this method

d = Dog('Dot', 4)

d.sayHi() # Hi, my name is Dot and I am 4 years old!
d.bark(1) # Dot says: woof!
d.bark(4) # Dot says: woof!woof!woof!woof!
```

Practice

Write the class SpiderMan to pass the test cases shown below.

Do not hardcode against the testcase values, though you can assume the testcases and comments cover the needed functionality.

```
s = SpiderMan("Peter")
# SpiderMan can shoot a web
assert(s.shootWeb() == "Peter shoots a web")
# SpiderMan can get hurt
assert(s.takeDamage(25) == "Peter gets hit for 25 hp")
```

What does the constructor take?

- What properties?

What methods is the code expecting you to implement ?

- What properties are these methods getting and setting/modifying?

Advantages of Classes and Methods

Encapsulation

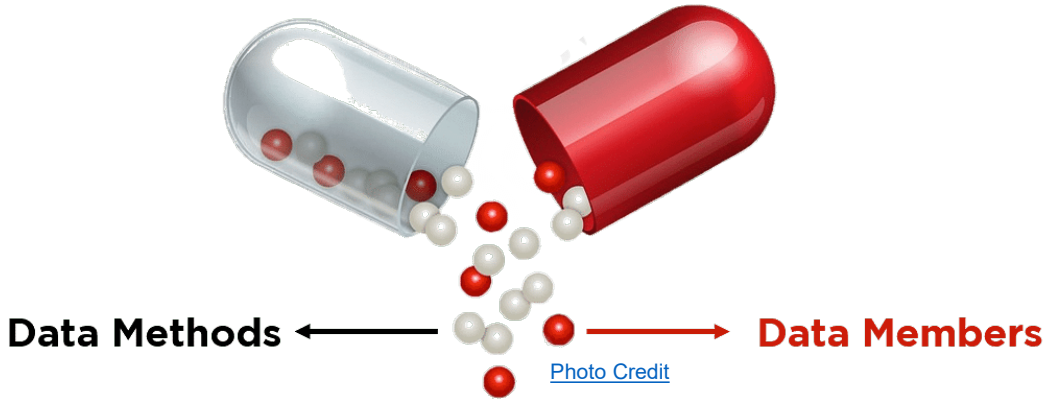


Photo Credit

- **Organizes code**
  A class includes the data and methods for that class.
- **Promotes intuitive design**
  Well-designed classes should be *intuitive*, so the data and methods in the class match commonsense expectations.
- **Restricts access**
  - `len` is a function, so we can call `len(True)` (which crashes)
  - `upper` is a method on strings but not booleans, so we cannot even call `True.upper()`

Polymorphism

The same method name can run different code based on type.

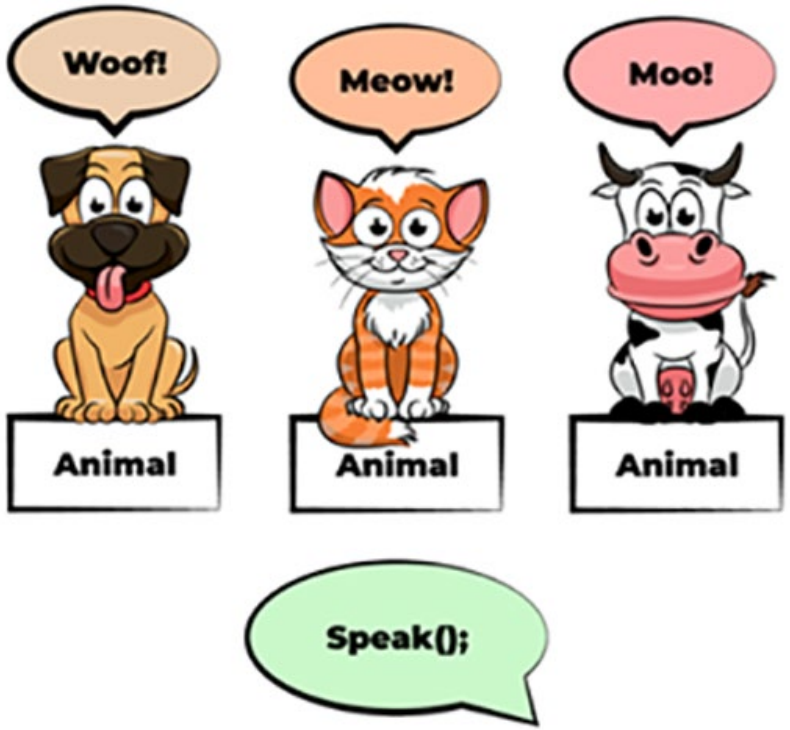


Photo Credit

```
class Dog(object):
        def speak(self):
                print('woof!’)
class Cat(object):
        def speak(self):
                print('meow!’)

for animal in [ Dog(), Cat() ]:
        animal.speak() # same method name, but one
                       # woofs and one meows!
```

More

Methods

Equality Testing

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4 a1 = A(5)
5 a2 = A(5)
6 print(a1 == a2)  # False!
```

Shouldn’t a1= a2??

__eq__
Equality Testing

The `__eq__` Method:

- Returns True if the object is equal to another object (other)
- Python uses it for testing equality of two objects

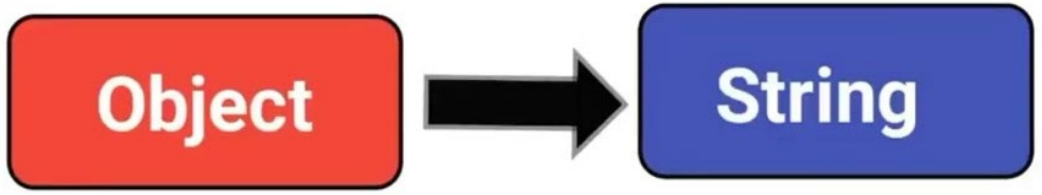
```
1  class A(object):
2      def __init__(self, x):
3          self.x = x
4      def __eq__(self, other):
5          return (self.x == other.x)
6  a1 = A(5)
7  a2 = A(5)
8  print(a1 == a2)  # True
9  print(a1 == 99)  # crash (darn!)
```

__eq__
Equality Testing

```
1  class A(object):
2      def __init__(self, x):
3          self.x = x
4      def __eq__(self, other):
5          return (isinstance(other, A) and (self.x == other.x))
6  a1 = A(5)
7  a2 = A(5)
8  print(a1 == a2)  # True
9  print(a1 == 99)  # False (huzzah!)
```

Here we don't crash on unexpected types of `other`

Converting to String




```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4 a = A(5)
5 print(a) # prints <__main__.A object at 0x102916128> (yuck!)
```

Converting to String
__ str__

```
1  class A(object):
2      def __init__(self, x):
3          self.x = x
4      def __str__(self):
5          return f'A(x={self.x})'
6  a = A(5)
7  print(a) # prints A(x=5) (better)
8  print([a]) # prints [<__main__.A object at 0x102136278>] (yuck!)
```

The `__str__` method tells Python how to convert the object to a string, but it is not used in some cases (such as when the object is in a list):

Converting to String
__ repr __

The __repr__ method is used inside lists (and other places)

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4     def __repr__(self):
5         return f'A(x={self.x})'
6 a = A(5)
7 print(a) # prints A(x=5) (better)
8 print([a]) # [A(x=5)]
```

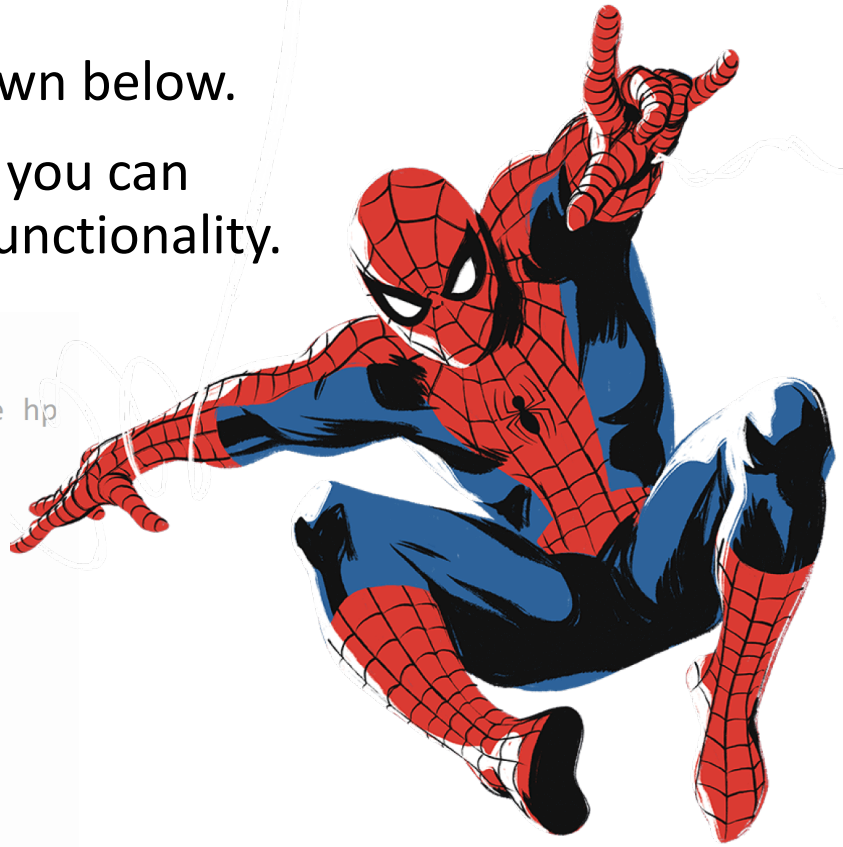
Recap: Some Special Methods

- `__init__`: constructor, initializer
  - Define initial values for the properties when an object is created
  - It does NOT return any meaningful value
- • `__repr__`:
  - Returns a string
  - Python uses it to convert an object to a string
  - E.g., `print`
- • `__eq__`: comparator
  - Returns True if the object is equal to another object
  - Python uses it for testing equality

Practice

Write the classes SpiderMan to pass the test cases shown below.

Do not hardcode against the testcase values, though you can assume the testcases and comments cover the needed functionality.



```
s = SpiderMan("Peter")
assert(str(s) == "SpiderMan(Peter) with 100 hp")
# SpiderMen can be equal to each other. This considers the name, but not the hp
assert(s == SpiderMan("Peter"))
assert(s != SpiderMan("Bob"))
assert(s != 42)
assert(s != "SpiderMan(Peter) with 100 hp")
# SpiderMan can shoot a web
assert(s.shootWeb() == "Peter shoots a web")
# SpiderMan can get hurt
assert(s.takeDamage(25) == "Peter gets hit for 25 hp")
assert(str(s) == "SpiderMan(Peter) with 75 hp")
# Current hp doesn't impact equivalence
assert(s == SpiderMan("Peter"))
```

What special methods is the code expecting you to implement ?

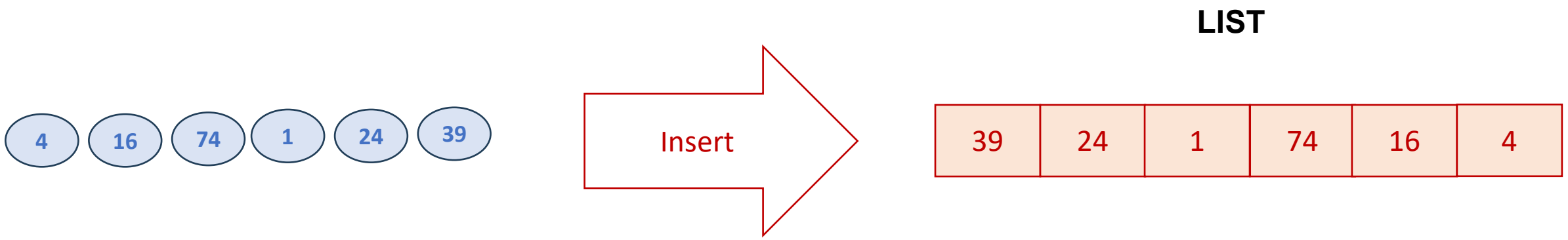
- What is the expected behavior or outcome of these methods?

Recap

- Class = Properties + Methods
- Class vs Object
- Functions vs Methods
- How to define a class
- How to instantiate a class
  - Using constructor (__init__) to preload instances with attribute values
- How to set/get properties of instances
- How to create and call methods on instances
- Special methods
  - __eq__(self, other)
  - __repr__(self)
  - __hash__(self)

Lists VS Sets Operations Efficiency - Review

Inserting Elements in a List



Inserting Elements in Sets & Hashing

Insertion Steps:

1) Hash (find the bucket)

2) Once in the bucket, perform Equality Testing

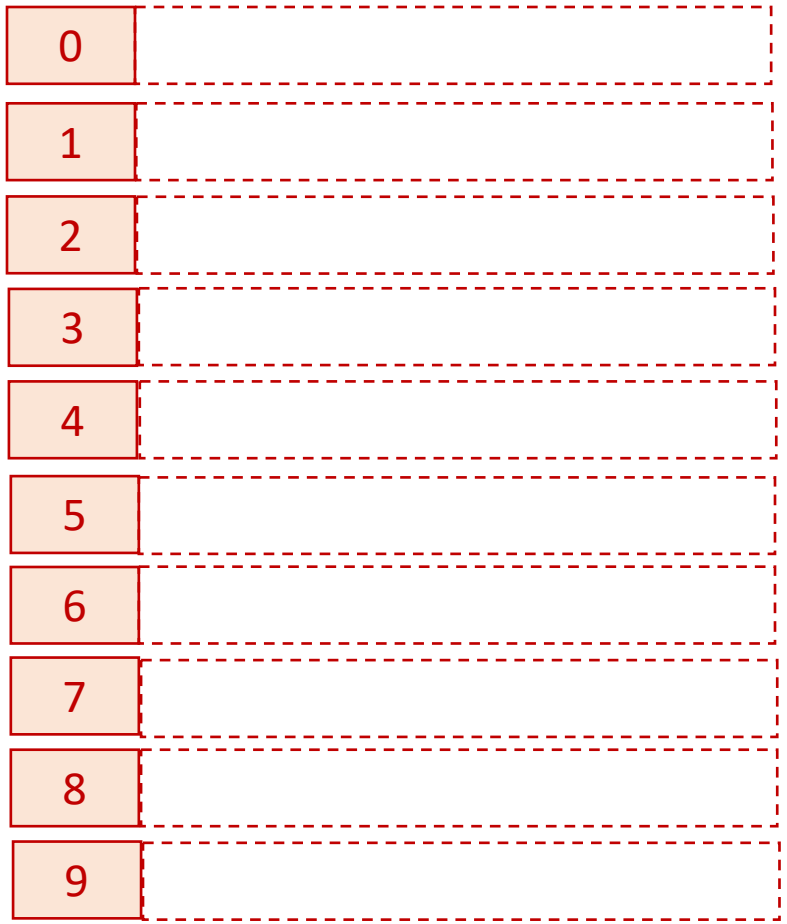
- Compare the element to each existing element in the bucket
- If it doesn't exist, add it

4 16 74 1 24 39

Hash Function

```
def myHash(n):
    return n%10
```

BUCKETS



If I try to Insert a duplicate

Insertion Steps:

1) Hash (find the bucket)

2) Once in the bucket, perform Equality Testing

- Compare the element to each existing element in the bucket
- If it doesn't exist, add it

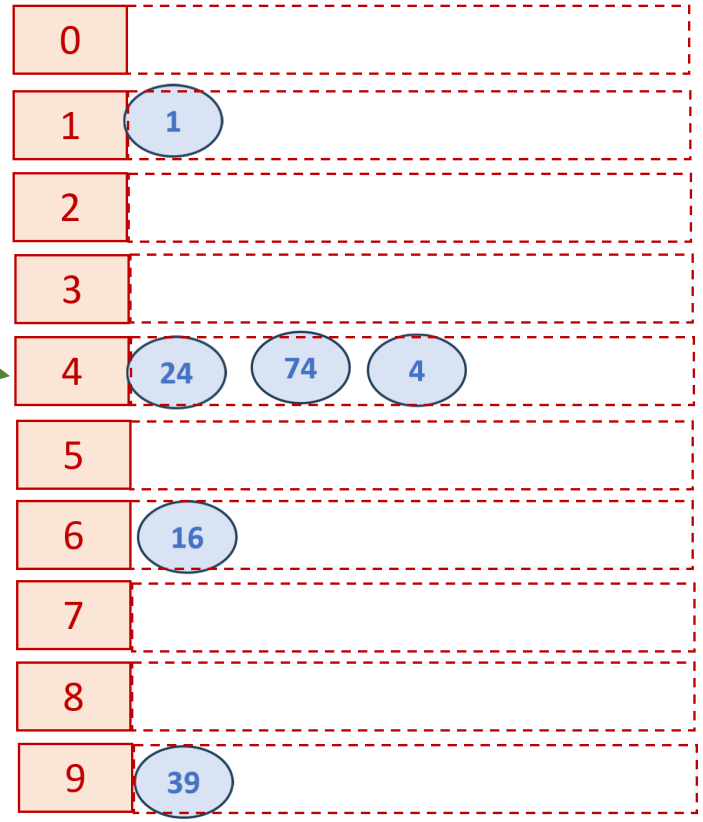
74

Insert

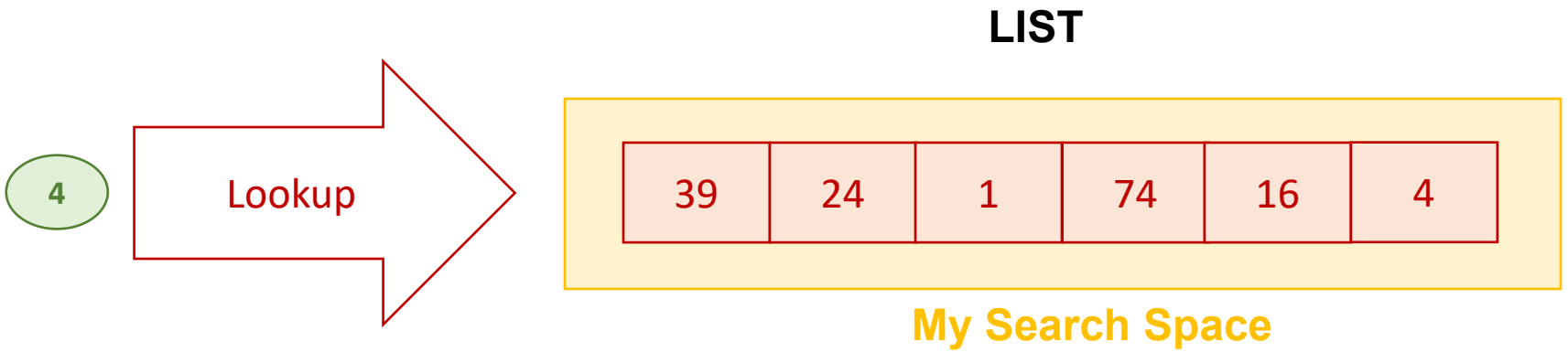
Hash Function

```
def myHash(n):
    return n%10
```

BUCKETS



Looking Up an Element in a List



Equality Testing

Compare the value to each existing element in the list. If it found, return it

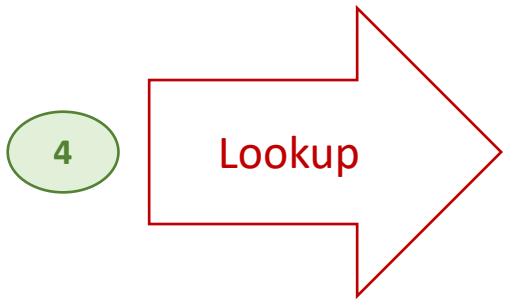
Looking Up an Element in a Set

Lookup Steps:

1) Hash (find the bucket)

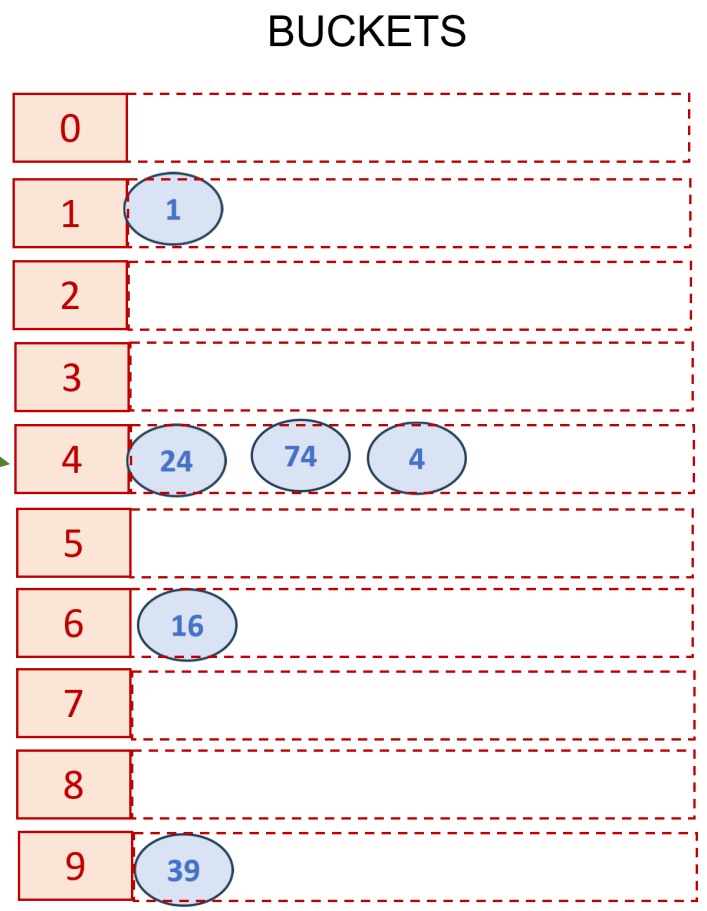
2) Once in the bucket, perform Equality Testing

- Compare the value to each existing element in the bucket
- If found, return it

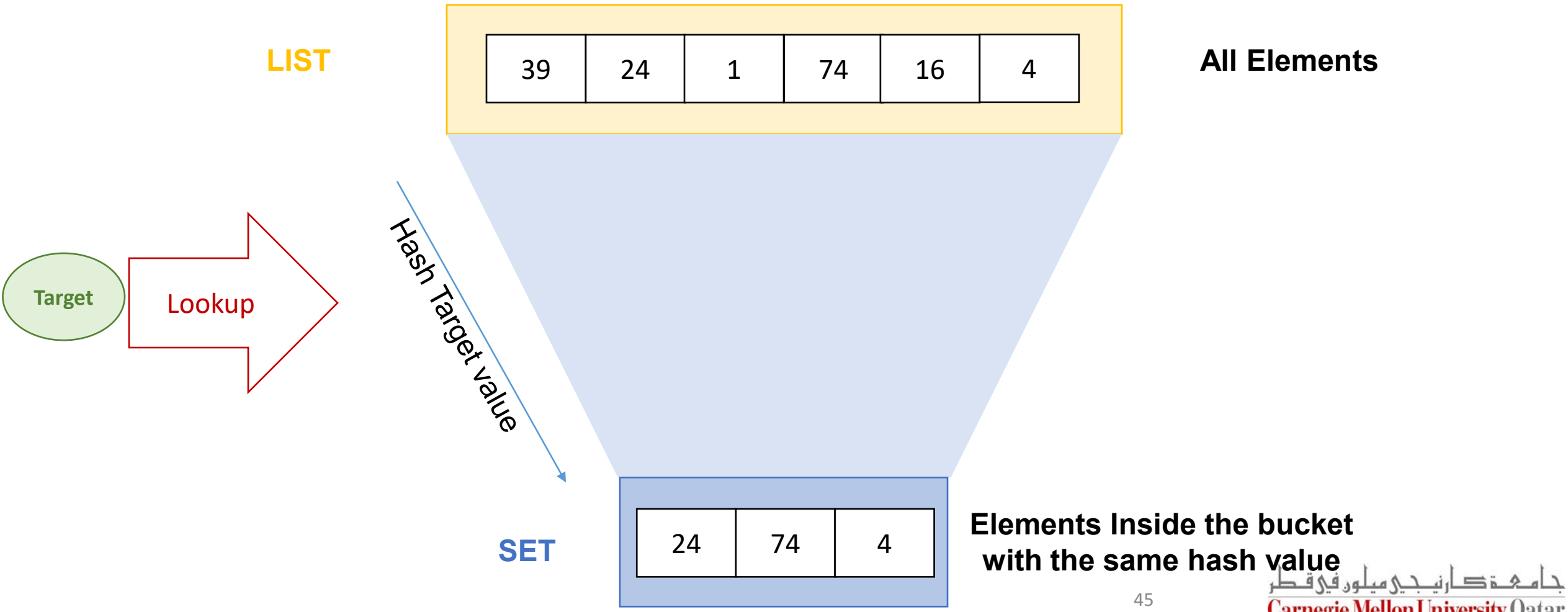


Hash Function

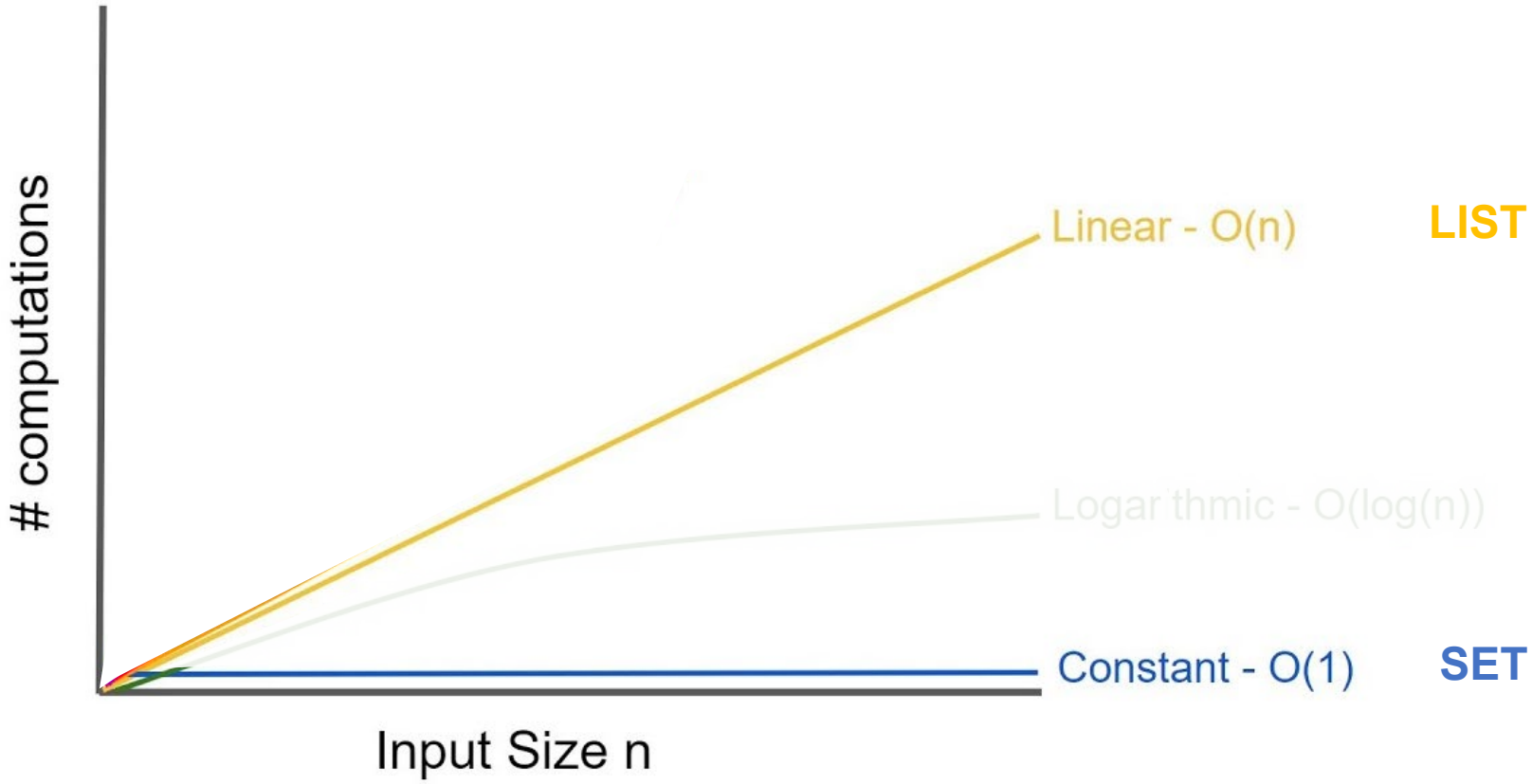
```
def myHash(n):
    return n%10
```



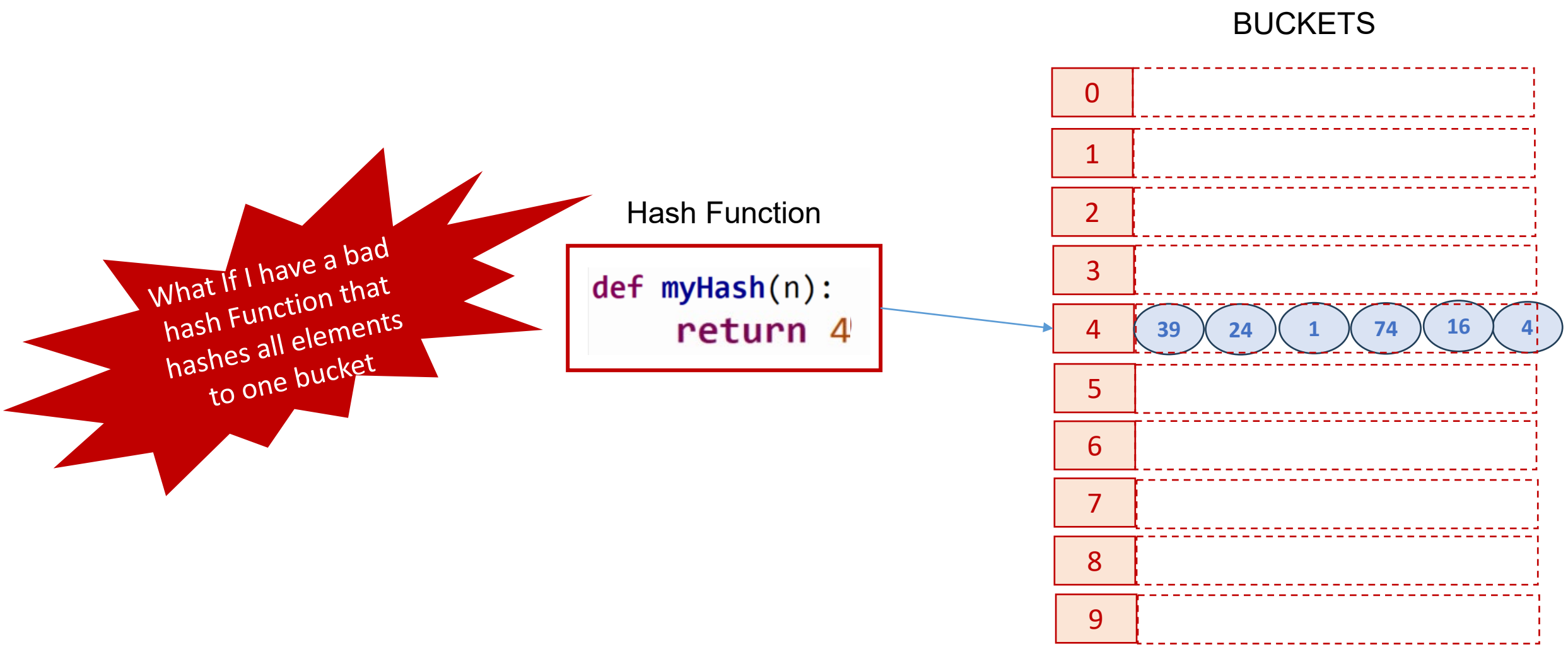
Looking Up in Lists Vs Sets (Search Space)



Looking Up in Lists Vs Sets (As Input Size Increases)



Sets & Bad Hash Functions



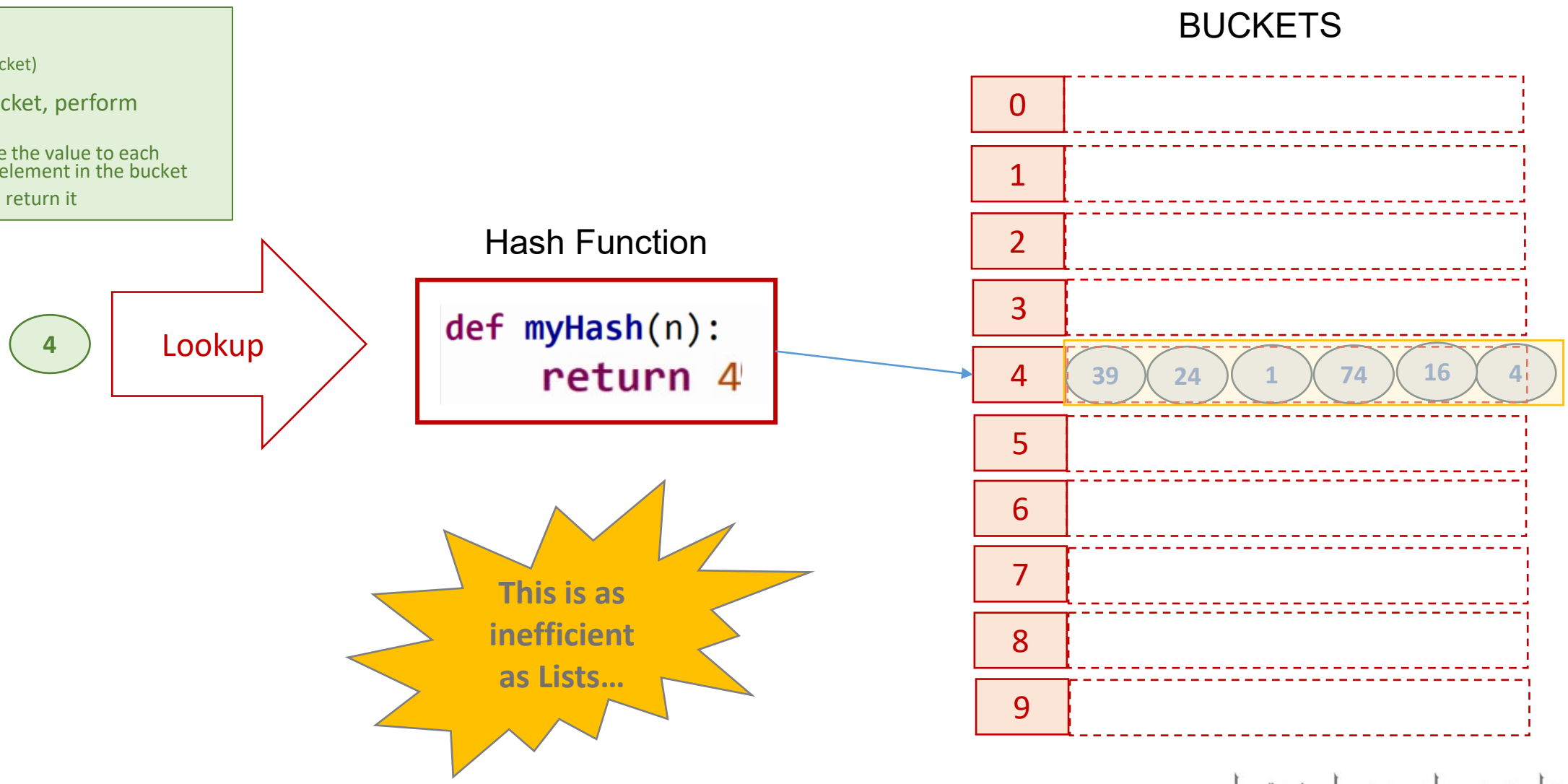
Sets & Bad Hash Functions

Lookup Steps:

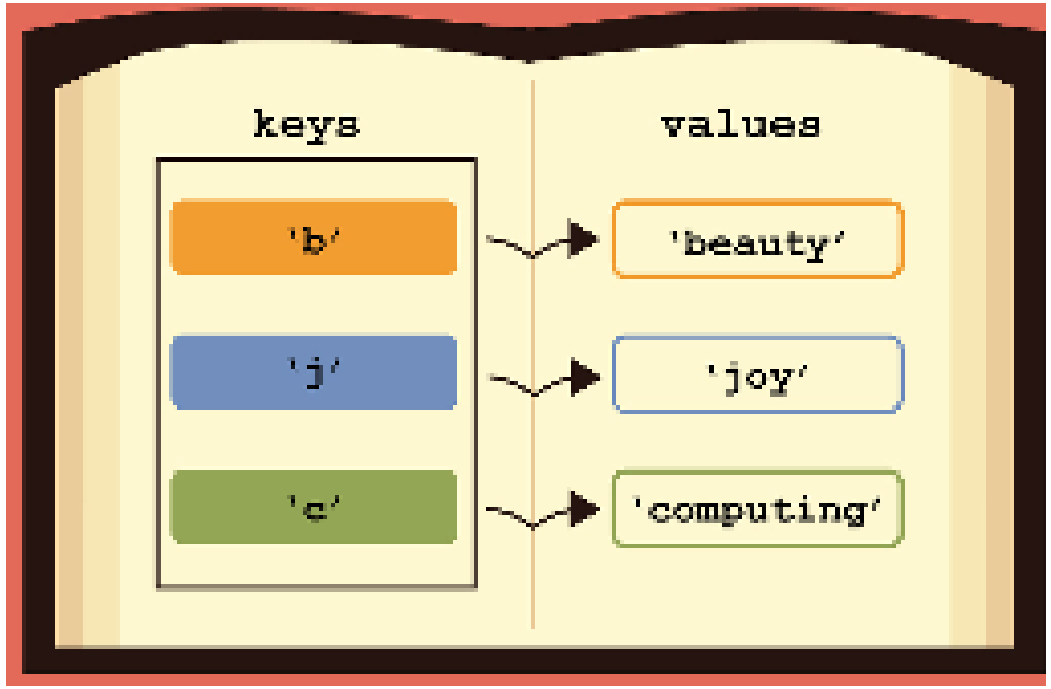
1) Hash (find the bucket)

2) Once in the bucket, perform Equality Testing

- Compare the value to each existing element in the bucket
- If found, return it



Using Objects w/ Sets & Dictionaries



Using Objects w/ Sets & Dictionaries

```
1  class A(object):
2      def __init__(self, x):
3          self.x = x
4
5  s = set()
6  s.add(A(5))
7  print(A(5) in s) # False
8
9  d = dict()
10 d[A(5)] = 42
11 print(d[A(5)]) # crashes
```

Objects do not seem to hash right by default

```
5  a = A(5)
6  b = A(5)
7
8  print(hash(a) == hash(b))       # False
```

Using Objects w/ Sets & Dictionaries
__hash__ and __eq__

The **__hash__ method** tells Python how to hash the object.

The properties you choose to hash on should be immutable types and should never change (so hash(obj) is immutable).

For sets and dictionaries to work properly, whenever you add hash, you need to add eq method

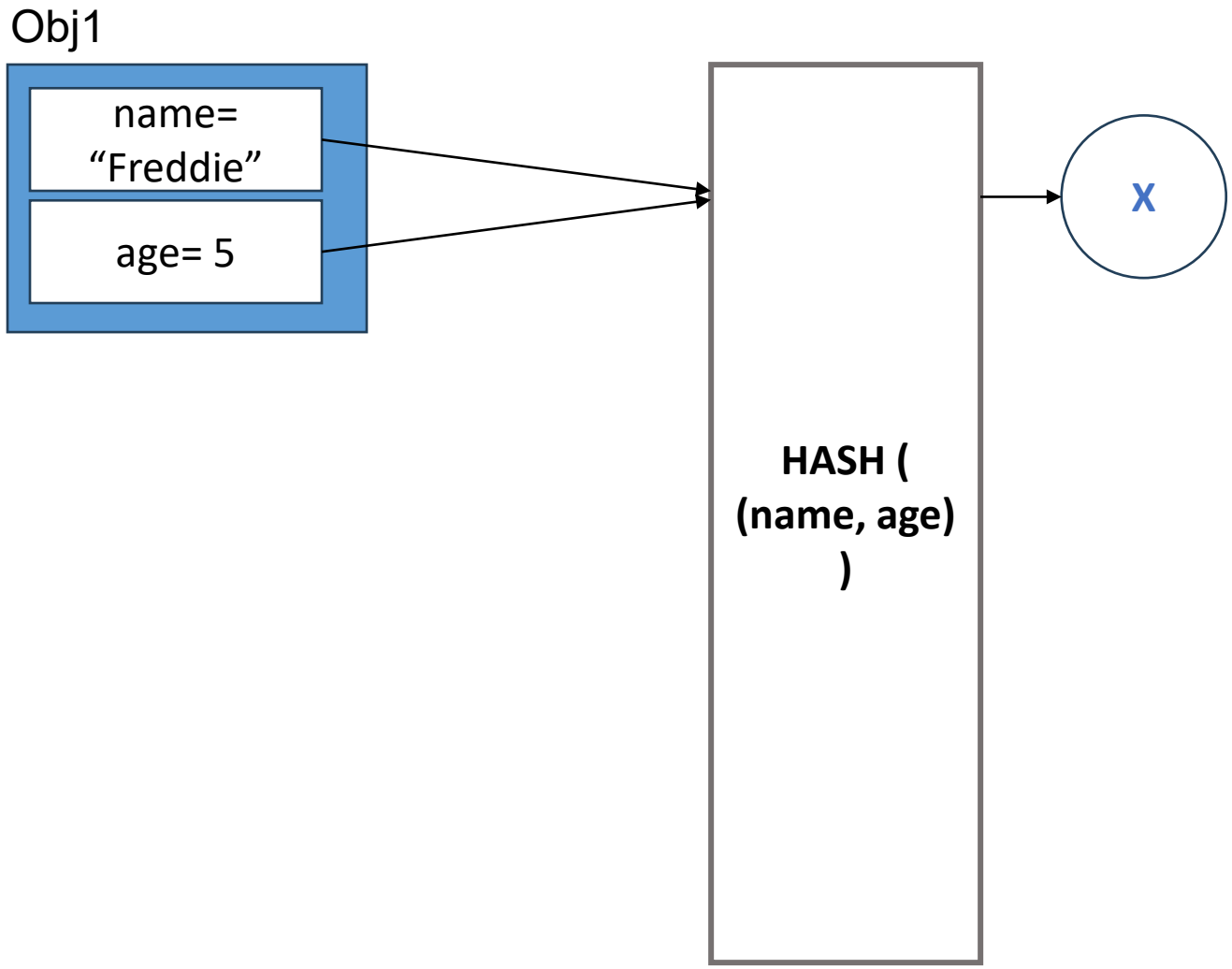
```
1  class A(object):
2      def __init__(self, x):
3          self.x = x
4      def __hash__(self):
5          return hash(self.x)
6      def __eq__(self, other):
7          return (isinstance(other, A) and (self.x == other.x))
8
9  s = set()
10 s.add(A(5))
11 print(A(5) in s) # True (whew!)
12
13 d = dict()
14 d[A(5)] = 42
15 print(d[A(5)]) # works!
```

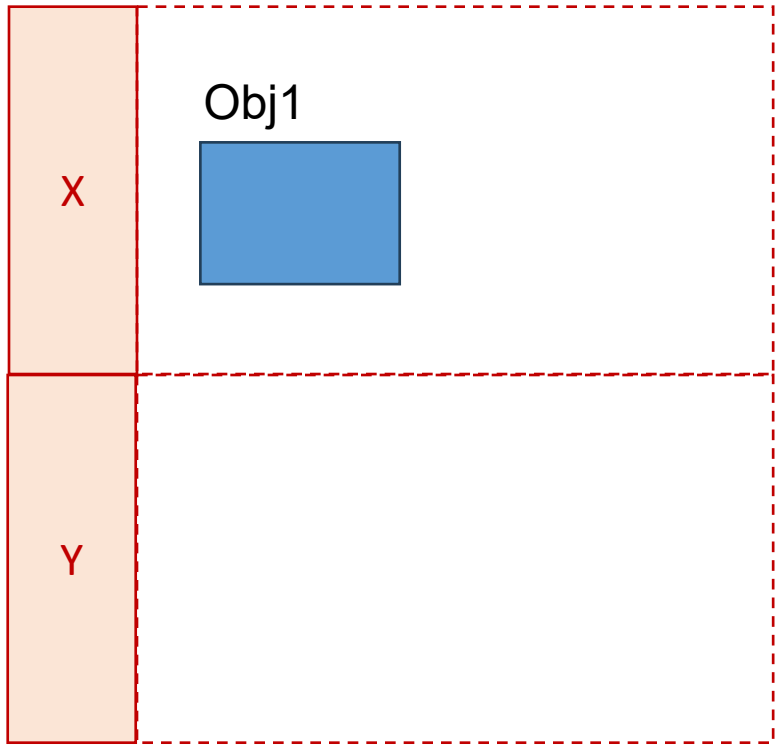
Using Objects w/ Sets & Dictionaries
A better (more generalized) approach

You can define the method `getHashables` that packages the things you want to hash into a tuple, and then you can use a more generic approach to `__hash__`

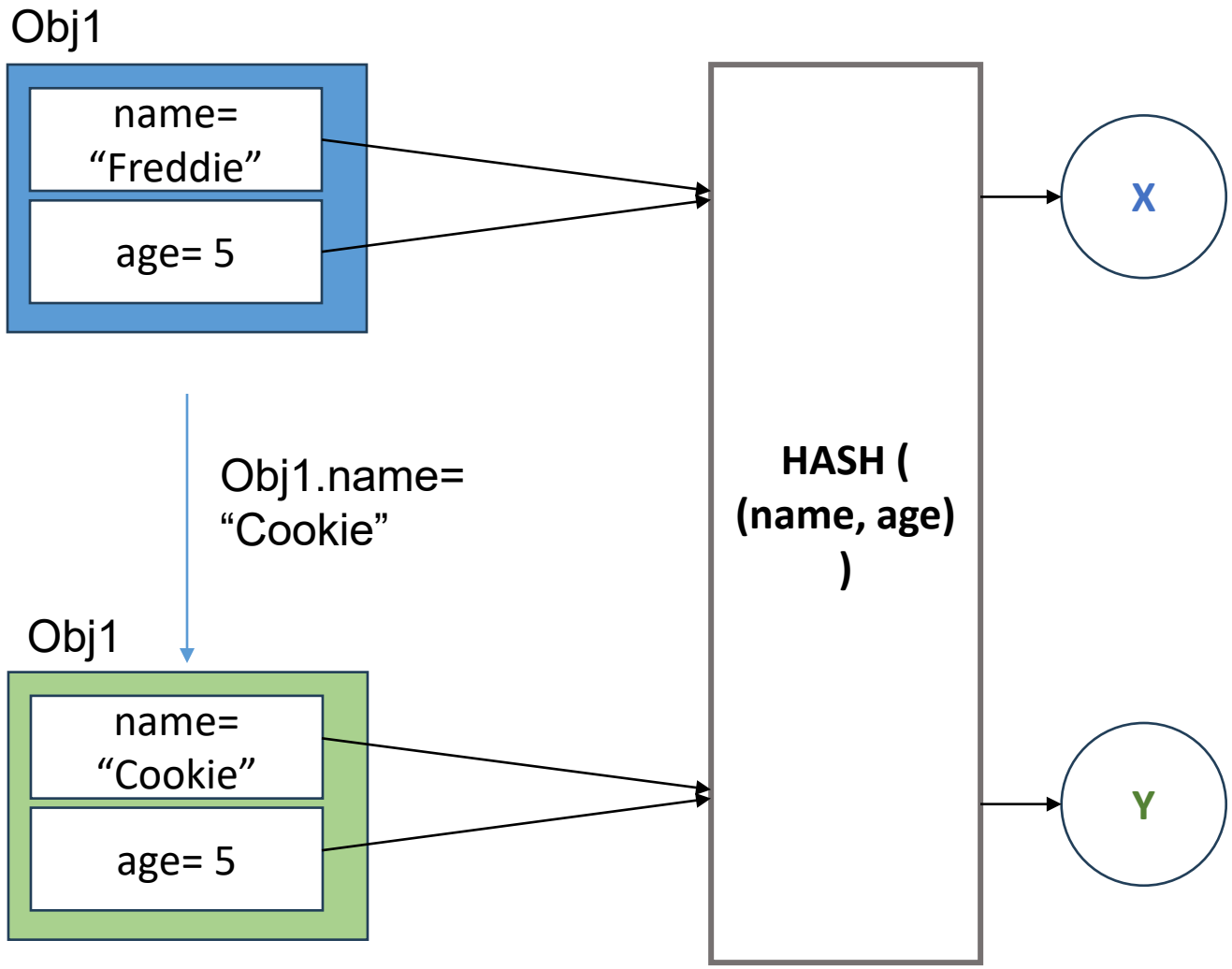
```
1  # Your getHashables method should return the values upon which
2  # your hash method depends, that is, the values that your __eq__
3  # method requires to test for equality.
4  # CAVEAT: a proper hash function should only test values that will not change!
5
6  class A(object):
7      def __init__(self, x):
8          self.x = x
9      def getHashables(self):
10         return (self.x, ) # return a tuple of hashables
11     def __hash__(self):
12         return hash(self.getHashables())
13     def __eq__(self, other):
14         return (isinstance(other, A) and (self.x == other.x))
15
16 s = set()
17 s.add(A(5))
18 print(A(5) in s) # True (still works!)
19
20 d = dict()
21 d[A(5)] = 42
22 print(d[A(5)]) # works!
```

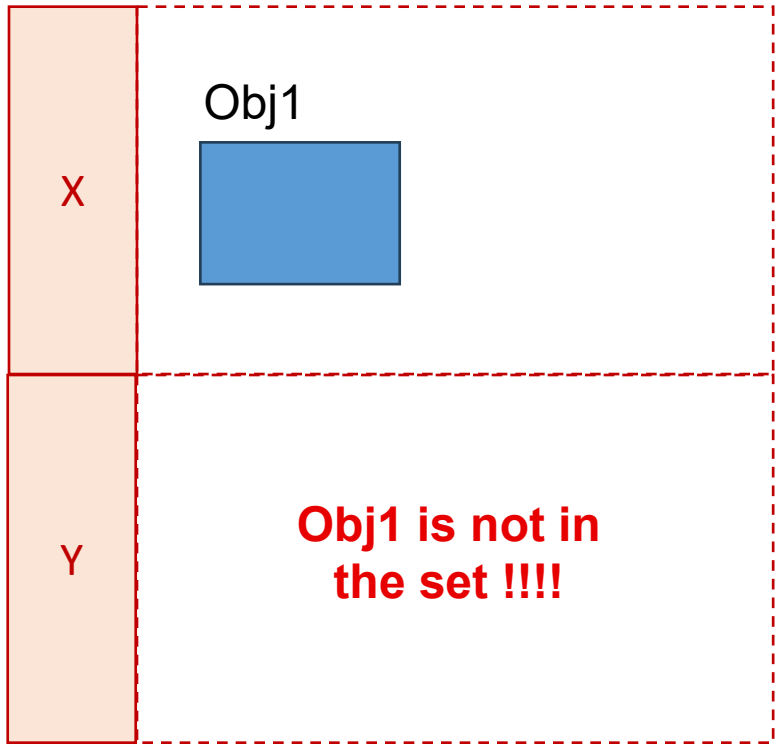
Hash attributes that will not change later in your program





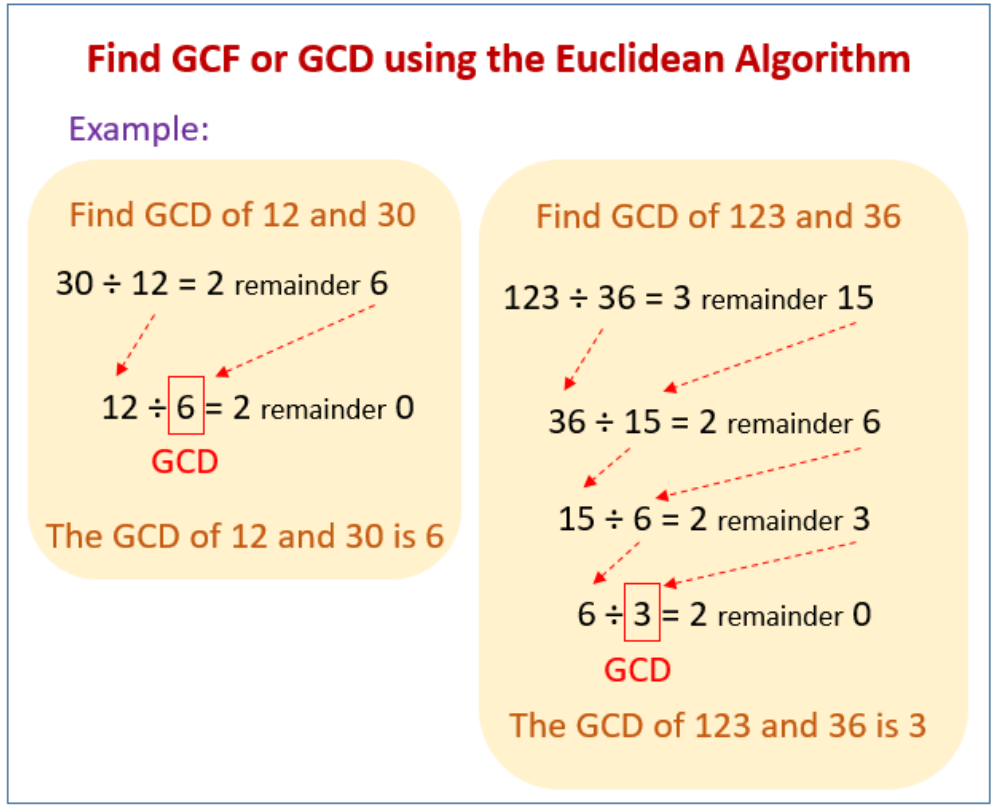
Hash attributes that will not change later in your program





Fraction Class Demo

GCD



The greatest common divisor is the largest number that will divide evenly into both the numerator and denominator.

- gcd(123, 36) #x, y
  - return gcd(36, 15) #y, x%y
    - return gcd(15, 6)
      - return gcd(6, 3)
        - return gcd(3, 0) # y=0
          - return 3 # x

Practice- Simplified Fraction Class

Write the class **Fraction** so that the test code runs as specified. Do not hardcode against the values used in the testcases, though you can assume that the testcases cover the needed functionality. You must use proper OOP design

//Note: You don't need to deal with 0 or negatives for now

You will need to reduce fractions (using gcd) upon creation

```
def testFractionClass():
    print('Testing Fraction class...', end='')
    assert(str(Fraction(2, 3)) == '2/3')
    assert(str([Fraction(2, 3)]) == '[2/3]')
    assert(Fraction(2,3) == Fraction(4,6))
    assert(Fraction(2,3) != Fraction(2,5))
    assert(Fraction(2,3) != "Don't crash here!")
    assert(Fraction(2,3).times(Fraction(3,4)) == Fraction(1,2))
    assert(Fraction(2,3).times(5) == Fraction(10,3))
    s = set()
    assert(Fraction(1, 2) not in s)
    s.add(Fraction(1, 2))
    assert(Fraction(1, 2) in s)
    s.remove(Fraction(1, 2))
    assert(Fraction(1, 2) not in s)
    print('Passed.')

if (__name__ == '__main__'):
    testFractionClass()
```