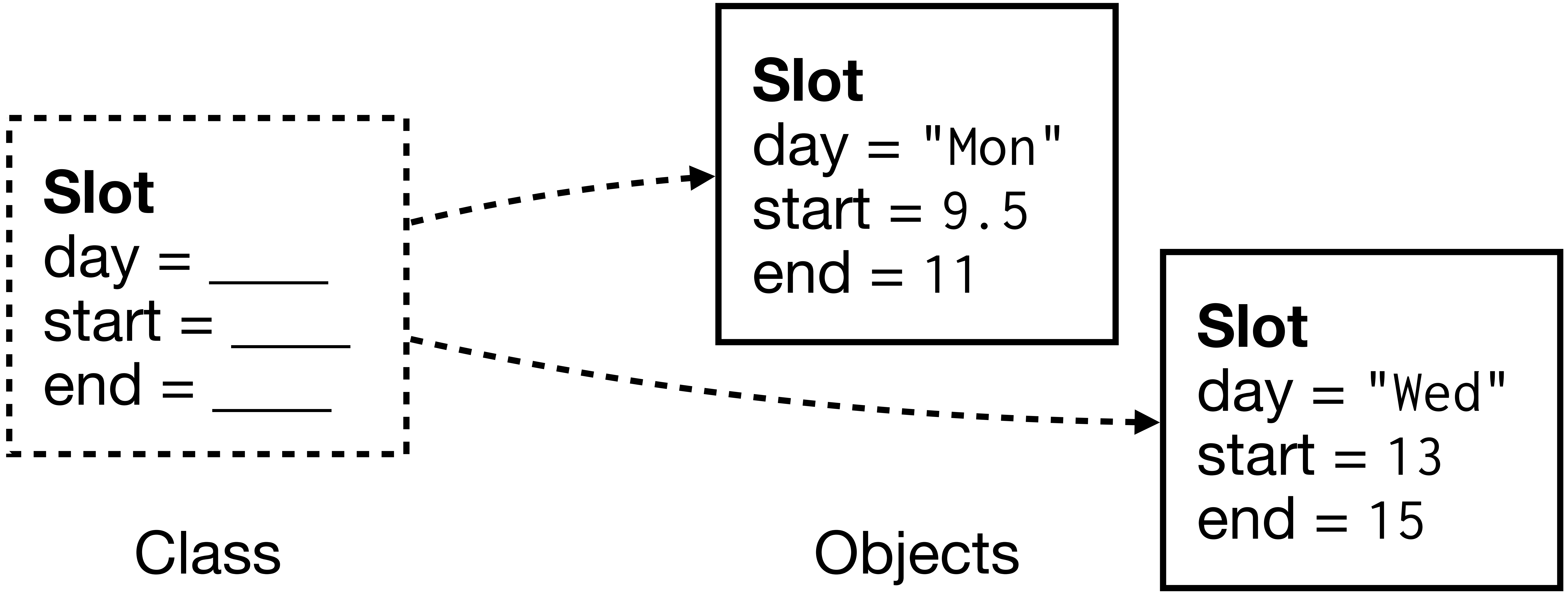
COL100: Introduction to Computer Science

9: Object-oriented programming

Classes and objects

A **class** is a programmer-defined type which specifies certain **attributes**. Using a class we can create **objects**, which are **instances** of the class.

Attributes may be data (a.k.a. **fields**) or functions (a.k.a. **methods**)

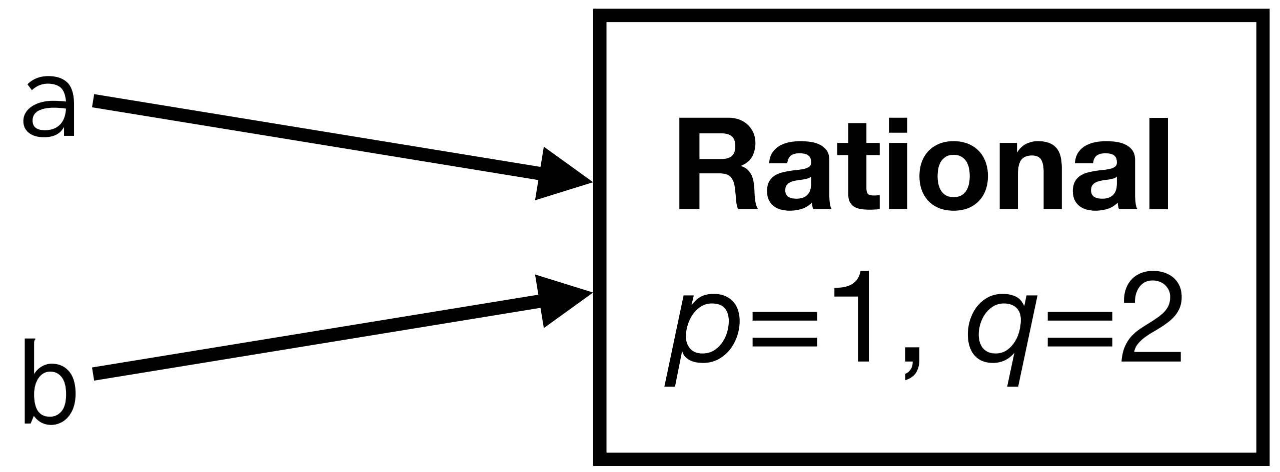


Objects and identity

Just like arrays, objects have **identity**.

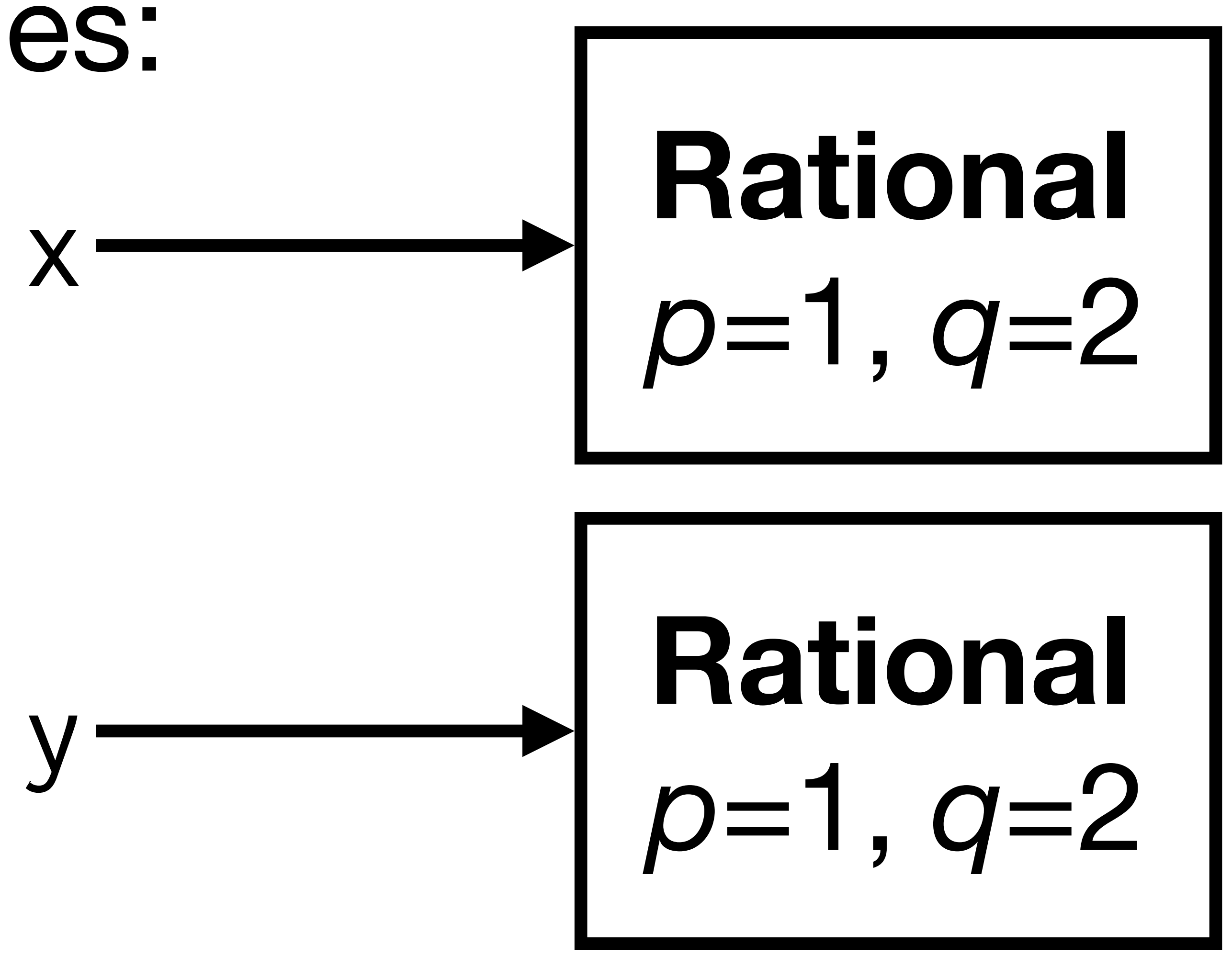
- Different variables may refer to the same object:

```
a = Rational(1,2)
b = a
```



- Objects that appear the same may have different identities:

```
x = Rational(1,2)
y = Rational(1,2)
x == y # true (if __eq__ implemented correctly)
```



This matters if you are going to **mutate** the object (modify its attributes' values)

What's the difference between these two functions?

```
def addOne(r):
  r.p += r.q
  return r

def addOne(r):
  return Rational(r.p + r.q, r.q)
```

What are `a`, `b`, `c` after the following?

```
a = Rational(1,2)
b = a
c = addOne(b)
```

To check object identity, use the `id` function or the `is` keyword:

```
a = Rational(1,2)
b = a
id(a)  # 4426107824
id(b)  # 4426107824
a is b # true

x = Rational(1,2)
y = Rational(1,2)
x == y # true
id(x)  # 4426107440
id(y)  # 4426105376
x is y # false
```

Example: Sets as sorted lists

Set
elements
cardinality()
add(*elem*)
remove(*elem*)
contains(*elem*)
union(*set*)
intersection(*set*)

Recall that we can represent sets as sorted lists.

```
class Set:
  def __init__(self, elems):
    self.elements = elems # ?
    sort(self.elements)   # ?
  def cardinality(self):
    return len(self.elements)
  def add(self, elem):
    insert(self.elements, elem)
  …
```

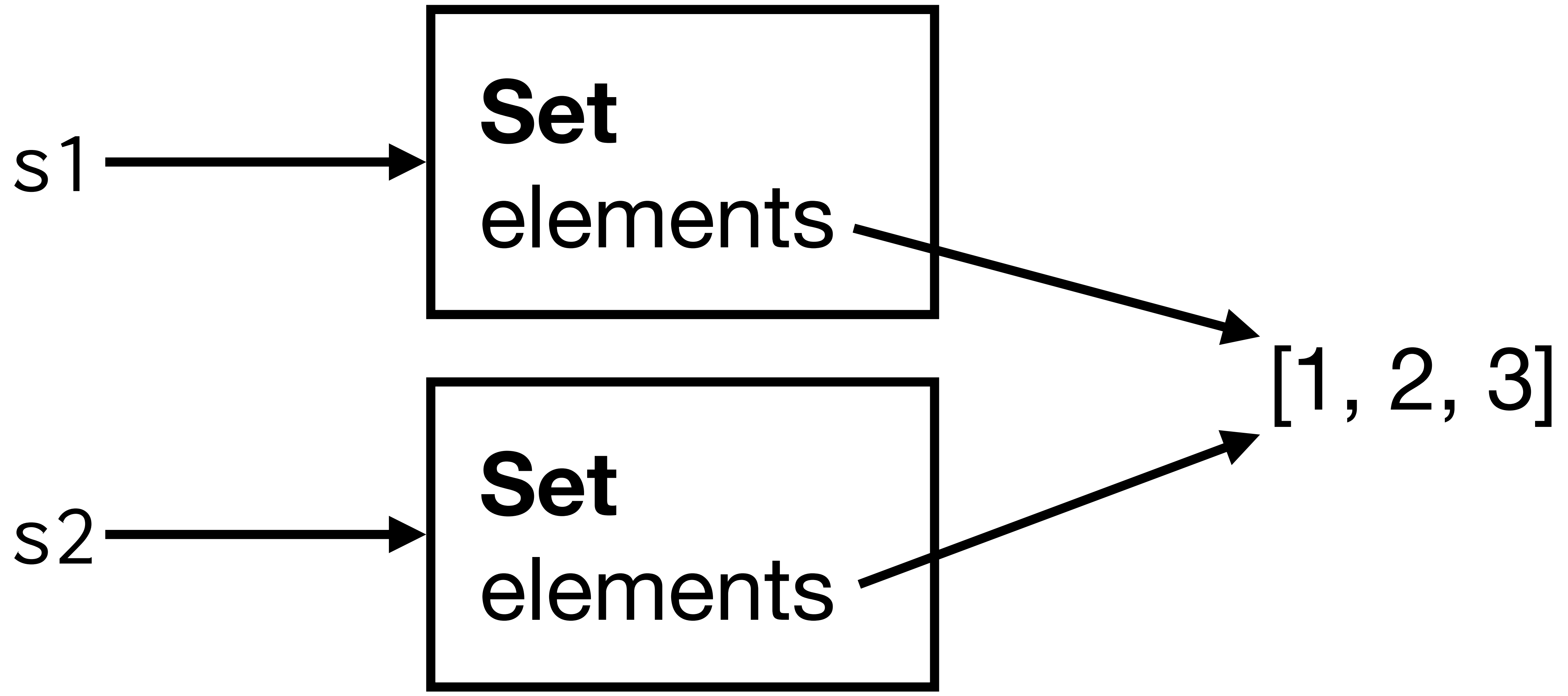
Then, outside the class we don't need to worry about how it works internally, we can just do the familiar set operations.

Be careful:

```
class Set:
  def __init__(self, elems):
    self.elements = elems # ?
    sort(self.elements)   # ?
  …

a = [3,2,1]
s1 = Set(a)
s2 = s1
```

But also:

```
a = [3,2,1]
s1 = Set(a)
s2 = Set(a)
```



Set `self.elements` to a *copy* of `elems` instead. (Do it yourself or use `elems.copy()`)

Exercises

- Complete the implementation of `Set`, and test it on some simple set theory problems.
- Union and intersection could *mutate* the current set, or they could return a new set. Which one does your implementation do? What are the pros and cons?

Encapsulation

Set
elements
cardinality()
add(*elem*)
remove(*elem*)
contains(*elem*)
union(*set*)
intersection(*set*)

Our `Set` object maintains the invariant that `elements` is always a sorted list. Every method (`__init__`, `add`, `remove`, `union`, `intersection`) keeps it sorted.

But what if…

```
s = Set([1,2,3])
s.elements.append(-5)
```

Code outside the object shouldn't have direct access to components that could break its invariants!

Public and private

Set
– elements
+ cardinality()
+ add(*elem*)
+ remove(*elem*)
+ contains(*elem*)
– merge(…)
+ union(*set*)
+ intersection(*set*)

Sensitive “internal” attributes should be **private**, others can remain **public**

- Private attributes can only be accessed by the object itself

- Public attributes can be accessed by outside code

Invariants need to be maintained only at the end of each public method

Public and non-public in Python

An attribute whose name starts with _ is considered non-public. Such attributes are not supposed to be accessed by outside code.

```
class Set:
  def __init__(self, elems):
    self._elements = elems.copy()
    sort(self._elements)
  def cardinality(self):
    return len(self._elements)
  def add(self, elem):
    insert(self._elements, elem)
  def _merge(…):
    …
  …
```

Exercise

- Suppose I am generating a sequence of millions of numbers in my code, and I want to collect their statistics (mean, min, max, etc.) without $O(n)$ space.

```
stats = Statistics()
while …:
  …
  aNumber = …
  stats.add(aNumber)
```

  Define the `Statistics` class so that at the end of the loop, I can get `stats.count()`, `stats.sum()`, `stats.mean()`, `stats.min()`, `stats.max()` in $O(1)$ time. (**Bonus:** also provide the standard deviation `stats.std()` in $O(1)$ time.)

Encapsulation and modularity

Set
– elements
+ cardinality()
+ add(*elem*)
+ remove(*elem*)
+ contains(*elem*)
– merge(…)
+ union(*set*)
+ intersection(*set*)

Seen from the outside, `Set` just provides the standard set-theoretic operations. Anyone using the class doesn't need to know that internally it is using a sorted list!

This **abstraction barrier** makes it possible to develop large programs in a modular fashion.

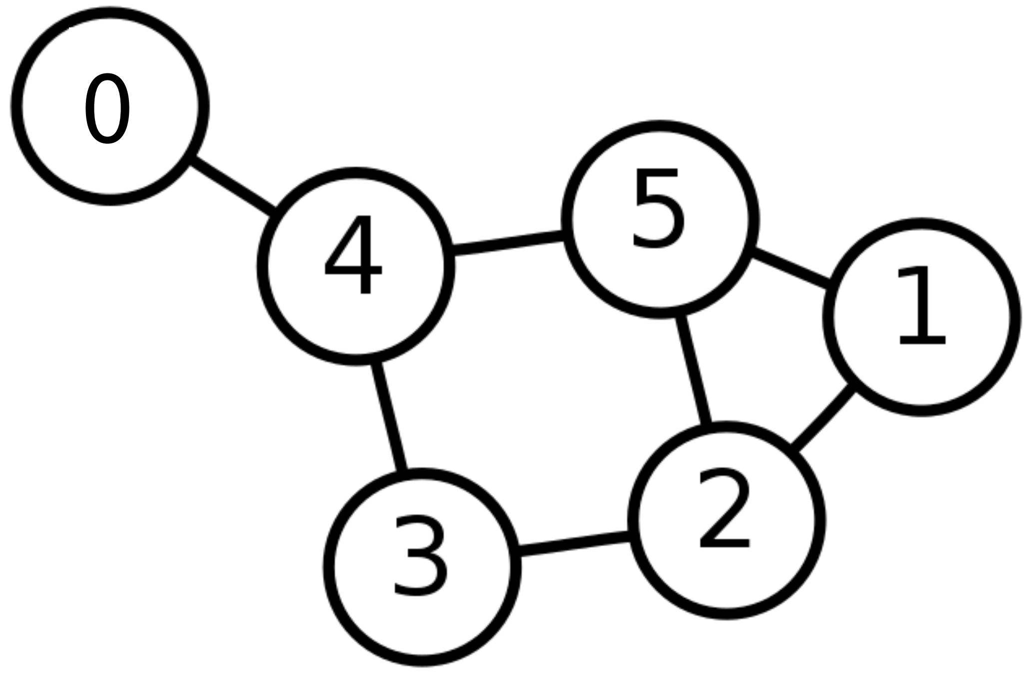
Exercise:

- Change the `Set` class to use a dictionary instead of a sorted list. All your previous tests should still work.
  (Look up the use of `for key in dict` and `if key in dict`.)

Example: Graphs and networks

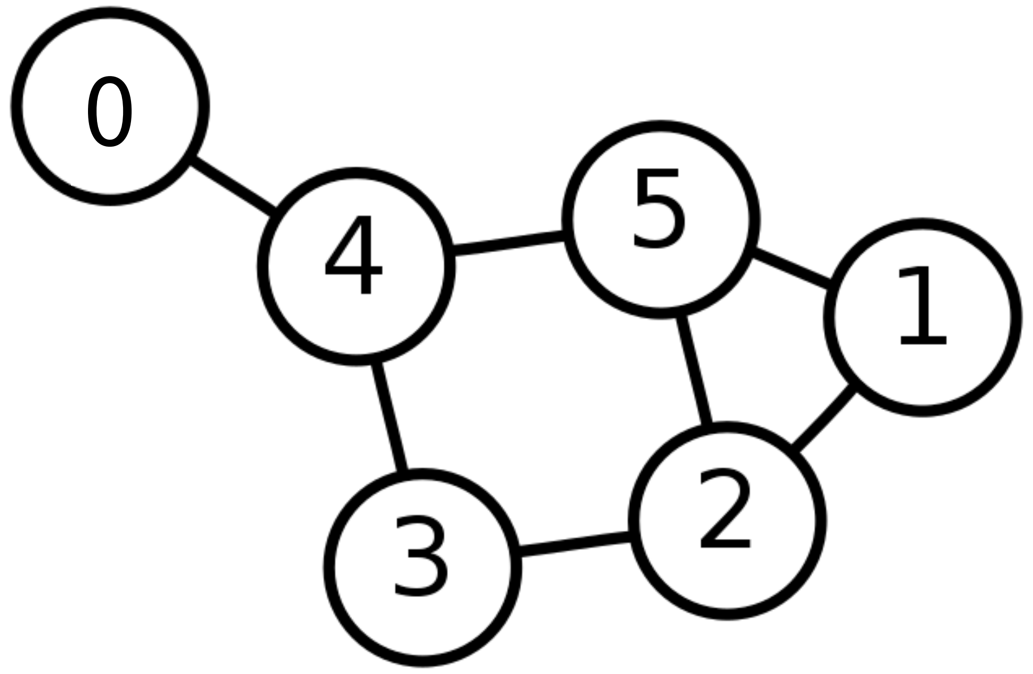
How should I represent a network of nodes (**vertices**) connected by **edges**?

Graph
+ Graph(*n*, *edges*)
+ countVerts()
+ adjacent(*u*, *v*)
+ neighbours(*v*)



- countVerts() → 6
- adjacent(2, 5) → true
- neighbours(2) → [1, 3, 5]

Example: Graphs



Graph
+ Graph(*n*, *edges*)
+ countVerts()
+ adjacent(*u*, *v*)
+ neighbours(*v*)

	0	**1**	**2**	**3**	**4**	**5**
0					✓	
1			✓			✓
2		✓		✓		✓
3			✓		✓	
4	✓			✓		✓
5		✓	✓		✓	

0	4		
1	2	5	
2	1	3	5
3	2	4	
4	0	3	5
5	1	2	4

Example: Graphs

```
class MatrixGraph:
  def __init__(self, n, edges):
    self._matrix = [[false for j in range(n)] for i in range(n)]
    for e in edges:
      self._matrix[e[0]][e[1]] = true
      self._matrix[e[1]][e[0]] = true

  def countVerts(self):
    return len(self._matrix)

  def adjacent(self, u, v):
    return self._matrix[u][v]

  def neighbours(self, v):
    …
```

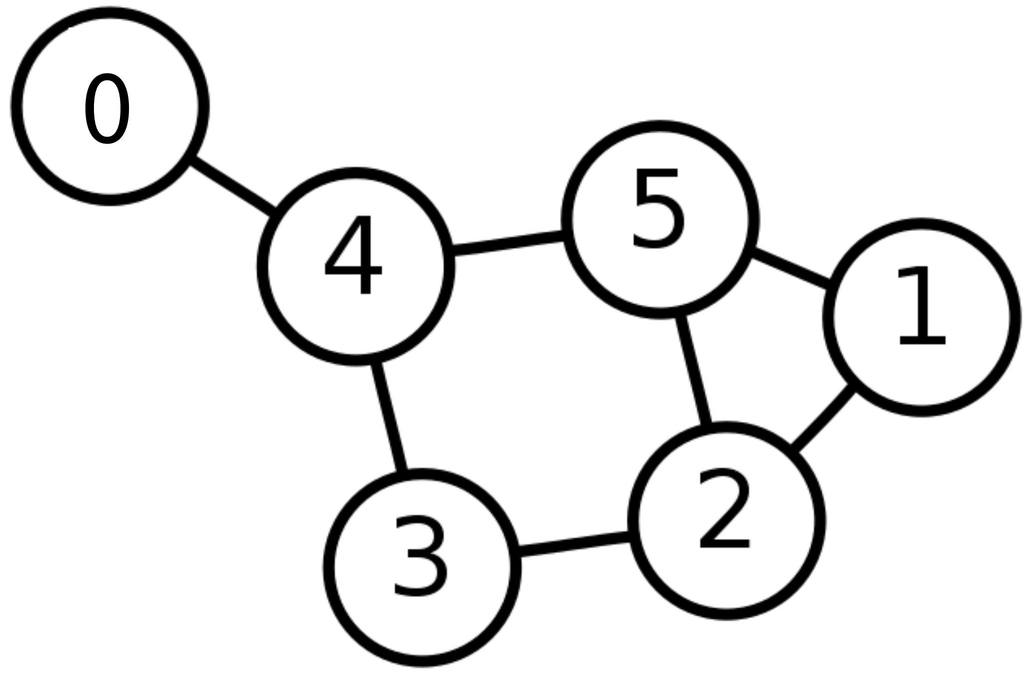
Graph
+ Graph(*n*, *edges*)
+ countVerts()
+ adjacent(*u*, *v*)
+ neighbours(*v*)

Example: Graphs

```
class ListGraph:
  def __init__(self, n, edges):
    self._list = [[] for i in range(n)]
    for e in edges:
      insert(self._list[e[0]], e[1])
      insert(self._list[e[1]], e[0])

  def countVerts(self):
    return len(self._list)

  def adjacent(self, u, v):
    …

  def neighbours(self, v):
    return self._list[v].copy()
```

Graph
+ Graph(*n*, *edges*)
+ countVerts()
+ adjacent(*u*, *v*)
+ neighbours(*v*)

Find a triangle in the given graph, e.g. (1, 2, 5).




```
def findTriangle(g):
  n = g.countVerts()
  for u in range(n):
    for v in g.neighbours(u):
      for w in g.neighbours(v):
        if g.adjacent(u, w):
          return (u, v, w)
  return None
```

This code works no matter whether `g` is a `MatrixGraph` or a `ListGraph`.

Graph
+ Graph(*n*, *edges*)
+ countVerts()
+ adjacent(*u*, *v*)
+ neighbours(*v*)

Exercises

- Finish the implementation of `MatrixGraph` and `ListGraph`.
- What are the time complexities of `adjacent(u,v)` and `neighbours(v)` in the two classes?
- Write a function which, given a graph, checks if every vertex is reachable from vertex 0. Your function should work they same way on both `MatrixGraph` and `ListGraph` objects.

Interfaces and polymorphism

Different objects with the same “interface” (set of public attributes) can be used the same way.

The same outer code can work on objects of different classes!

Example: Suppose you are writing a drawing program, so

- you need to store different shapes (circles, rectangles, triangles, etc.), and
- you need to do geometry calculations (area, perimeter, etc.) on all of them

Example: Shapes

- Circle: center $c = (x, y)$ and radius r
- Rectangle: x range (x_0, x_1) and y range (y_0, y_1)
- Triangle: vertices $a = (x_a, y_a)$, $b = (x_b, y_b)$, $c = (x_c, y_c)$
- …

```
class Circle:
  def __init__(self, c, r):
    self.c = c
    self.r = r
class Rectangle:
  …
class Triangle:
  …

def area(shape):
  if isinstance(shape, Circle):
    return pi * shape.r * shape.r
  elif isinstance(shape, Rectangle):
    …
  elif …
```

Circle
+ *c*, *r*

Rectangle
+ *xr*, *yr*

Triangle
+ *a*, *b*, *c*

Awkward and error-prone! Especially if you add new kinds of shapes later

```
class Circle:
  def __init__(self, c, r):
    self.c = c
    self.r = r
  def area(self):
    return pi * self.r * self.r
  def perimeter(self):
    return 2 * pi * self.r
  …

class Rectangle:
  …

class Triangle:
  …
```

Then you can just call `shape.area()`.

Circle
+ *c*, *r*
+ area()
+ perimeter()

Rectangle
+ *xr*, *yr*
+ area()
+ perimeter()

Triangle
+ *a*, *b*, *c*
+ area()
+ perimeter()

Exercises

- Implement the `Circle`, `Rectangle`, `Triangle` classes. Verify that it is easy to find the total area of a list of shapes even if they are of different types:

```
total = 0
for s in shapes:
  total += s.area()
```

- It is straightforward to add a new class that has a given interface. For example, add a `Polygon` class. (Look up the “shoelace formula” for the area.)

- It is more cumbersome to add a new method to an interface that many classes share. Add an `isInside(p)` method that checks if the given point is inside the shape. (You can raise a `NotImplementedError` in nontrivial cases.)